ICARUS: Criticality and Reuse based Instruction Caching for Datacenter Applications

Vedant Kalbande*
Indian Institute of Technology Bombay
Mumbai, India
vedantk@iitb.ac.in

Hrishikesh Jedhe Deshmukh*
Indian Institute of Technology Bombay
Mumbai, India
hrishijedhe06@gmail.com

Alberto Ros
University of Murcia
Murcia, Spain
aros@ditec.um.es

Biswabandan Panda
Indian Institute of Technology Bombay
Mumbai, India
biswa@cse.iitb.ac.in

Abstract

Datacenter applications with huge code footprints suffer from front-end CPU bottlenecks even with a decoupled front-end. These applications are composed of complex system stacks with subtle interdependencies. One of the primary contributors to the front-end bottleneck is instruction misses at L2, which cause decode starvation. State-of-the-art L2 cache replacement policies, such as EMISSARY, utilize front-end criticality to identify instruction lines that cause decode starvation and attempt to keep those critical lines in L2. We observe that only 28.32% of the critical lines retain their criticality behavior, and a significant fraction of the critical instruction lines show dynamic behavior.

We propose ICARUS, an L2 replacement policy that incorporates branch history as context information to improve critical instruction line detection. We observe that the reuse distance of instruction lines varies based on the branch history that led to the instruction fetch. Next, we enhance the L2 replacement policy by considering both criticality and the reuse of instruction lines at L2, as we observe that the reuse behavior of critical lines differs from that of non-critical lines. On average, across 12 datacenter applications, ICARUS outperforms Tree-based Pseudo LRU (TPLRU) by 5.6% and as high as 51%. The state-of-the-art replacement policy, EMISSARY, on the other hand, provides an improvement of 2.2% over TPLRU. We demonstrate the robustness of ICARUS across various L1I and L2 cache sizes, as well as its effectiveness in the presence of hardware prefetchers.

CCS Concepts: • **Computer systems organization → Superscalar architectures**.

*Authors have contributed equally.

Keywords: Cache Microarchitecture; Cache Replacement Policy; Instruction Caching

ACM Reference Format:
Vedant Kalbande, Hrishikesh Jedhe Deshmukh, Alberto Ros, and Biswabandan Panda. 2026. ICARUS: Criticality and Reuse based Instruction Caching for Datacenter Applications. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* March 22-26, Pittsburgh, USA. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779212.3790175

1 Introduction

The code footprint of datacenter applications grows by 30% every year [17, 31], creating enormous pressure on the cache hierarchy. One of the primary reasons for the increase in the code footprint is that these applications interact with the system stack, with multiple layers of kernel modules [17, 49], libraries [31], and language runtimes [5]. Most first-level instruction cache (L1I) miss latencies are tolerated thanks to a decoupled front-end that performs fetch-directed instruction prefetching [27, 43]. This is evident from the industry trend, as the L1I sizes of commercial datacenter processors are not increasing significantly. AMD uses 32KB of L1I [13], while Intel and ARM servers use 64KB of L1I [8, 15].

However, second-level cache (L2) misses for instruction accesses cause significant front-end bottlenecks, resulting in decode starvation and eventually leading to an empty issue queue, which stalls the CPU completely. Replacement policies for L2 that prioritize instructions [29, 39] can be essential to mitigate front-end stalls. A recent L2 replacement policy, EMISSARY [39], preserves *critical instruction lines* that cause decode starvation and empty issue queue, improving the overall front-end performance. Yet, a significant performance uplift is still possible.

The opportunity. We quantify the performance boost that we can achieve if we can have a perfect L2 only for instructions (100% hit rate only for instructions). Figure 1 shows the performance uplift of different L2 replacement policies and the perfect L2 for instructions over a baseline L2

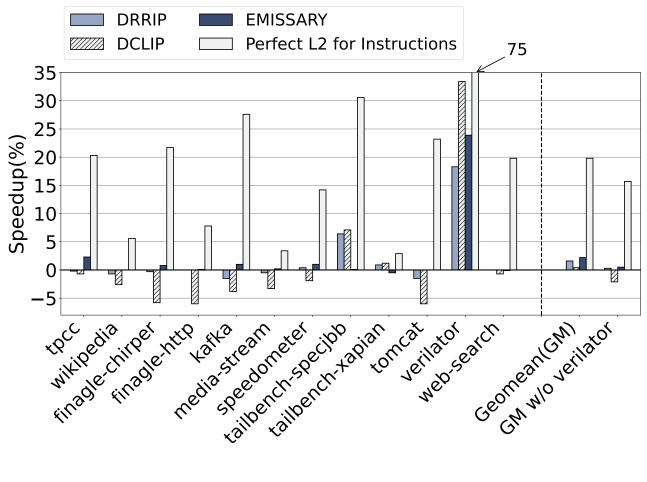


Figure 1. Speedup of different L2 replacement policies normalized to TPLRU across 12 datacenter applications

that implements a Tree-based Pseudo Least Recently Used (TPLRU) policy. For a set of datacenter applications (Table 4), the state-of-the-art EMISSARY provides an average speedup of 2.2%, whereas a perfect L2 for instructions provides an average speedup of 19.80%, with a maximum of 75%. Since Verilator is an outlier, we also report the geomean excluding it, which is 15.7%. A well-established replacement policy like Dynamic Re-Reference Interval Prediction (DRRIP) [30] improves performance by 1.6%. Dynamic Code Line Preservation (DCLIP) [29] speeds up a few datacenter applications with an average performance improvement of 0.4%. This happens primarily because DCLIP tries to keep the instruction lines at the cost of the data lines at the L2, even if all the instruction lines do not cause front-end stalls.

State-of-the-art data replacement policies such as Hawkeye [28] and Mockingjay [47] that use the program counter (PC) for reuse prediction fail to improve the performance of instruction caches, and these techniques predict that a large fraction of PCs (more than 99%) are cache friendly [33]. EMISSARY [39], the state-of-the-art L2 replacement policy for instructions, observes that a small percentage of instruction lines cause the majority of decode starvation, and preserves them in L2.

Observation I: We observe that with EMISSARY, once an L2 line is marked critical, it assumes it to be critical in future instruction fetches, too. On average, we find that 3.49% of instruction fetches cause decode starvation, which we call *critical fetches*, and the cache lines responding to these critical fetches are critical lines. A small fraction (28.32%) of critical lines persist with their criticality behavior (we call these lines *static*, or s-critical lines). In contrast, a large fraction of critical lines exhibit *dynamic* behavior (we call these lines d-critical lines). We observe this trend for the instruction lines accessed by the user and the operating system (OS). One pertinent reason for dynamic behavior is the dynamic control flow in user accesses, kernel accesses involving OS and network stacks, and remote procedure calls. These affect the front-end criticality behavior in terms of instruction fetches that cause decode starvation. EMISSARY is agnostic of dynamic behavior while preserving instruction lines in L2.

Observation II: EMISSARY evicts non-critical lines and preserves critical lines unless the number of critical lines reaches a certain threshold (N). For a given L2 set, if the critical lines are more than N, it evicts them using TPLRU. For example, for a 16-way associative cache, EMISSARY uses a threshold of eight (N=eight is optimal) for critical lines. On an eviction, EMISSARY prioritizes non-critical lines over critical lines as long as the number of critical lines within a set is less than N. Once it crosses N, EMISSARY starts evicting critical lines, leading to the thrashing of critical lines as the effective cache associativity for critical lines reduces from 16 to eight. Overall, EMISSARY fails to provide hits to critical lines with *long reuse*.

Observation III: We observe that EMISSARY calculates the reuse distance, keeping the entire L2 capacity (global reuse distance) and not limited to a particular cache set (local reuse distance) in mind even though replacement policy decisions are per cache set. As a consequence, instruction lines with a global reuse distance of more than 5000 accesses to L2 are indeed *short reuse* lines with a reuse distance of around five ($\frac{5000}{1024}$) if we calculate the local reuse distance of a particular L2 set for an L2 with 1024 sets, and these accesses are distributed equally among all sets.

Our goal in this work is to consider the dynamic nature of front-end criticality and reuse behavior of critical and non-critical lines while designing criticality-driven cache replacement policies for instructions. We propose Instruction Criticality And ReUSe (ICARUS), a new replacement policy for L2 that considers the above pertinent observations.

The key insight behind ICARUS is that the criticality of the instruction is driven by the recent control flow behavior as the reuse of instruction lines differs based on the control flow. Datacenter applications use user-level code, libraries, virtual machines (like JVM), and OS, and the criticality of an instruction depends on the control flow jumps. Our second insight is that criticality alone is not effective enough in keeping instruction cache lines of interest, as the reuse of critical and non-critical lines varies. Some critical lines exhibit short reuse behavior, and some exhibit long reuse behavior. So, there is a need for a replacement policy that uses criticality and reuse to maximize the benefits of an L2 replacement policy. We create four bins of L2 lines based on criticality and reuse and allocate the L2 space appropriately among all the bins so that the overall utility of L2 will improve in terms of L2 hits to critical instruction fetches.

Our approach and contributions. First, we propose a new way to identify critical instruction lines that lead to decode starvation. We use the branch history coupled with the instruction cache line address to indicate the criticality of future instruction accesses. Our approach improves the detection of critical lines. Second, we propose different replacement priority chains for critical and non-critical instruction lines based on their reuse behavior.

Overall, we make the following contributions in the form of ICARUS: we present a case for a context-based critical instruction line detection (BHC), where we utilize the coupling of branch history with the instruction cache line address as the context. With context-based critical instruction detection, we detect the dynamic behavior of critical lines (Sections 3.1, 3.2, and 3.3). We propose a reuse and criticality based bins for L2 replacement policy (BRC), which uses the context-based critical instruction line detection and reuse of critical and non-critical lines for L2 replacement policy (Section 3.4 and Section 3.5). BRC enhances the residency of critical lines with long reuse, thereby improving the hit rates of critical instruction lines at L2. On average, for 12 datacenter applications evaluated using the gem5 simulator [18], ICARUS (BHC+BRC) provides a speedup of 5.6% compared to the baseline TPLRU policy which is 2.2% excluding verilator. The EMISSARY replacement policy provides a speedup of 2.2% which becomes 0.5% excluding verilator (Section 4.2).

2 Background

2.1 Decoupled front-end

Modern datacenter processors use a decoupled front-end [27, 43] with a Fetch Target Queue (FTQ) between the branch predictor and the L1I. The FTQ allows the branch predictor to run ahead with respect to the instruction address sent to the L1I. The FTQ stores the fetch streams generated by the branch predictor until these predictions are consumed by the L1I or squashed from the FTQ. This decoupling enables Fetch-Directed Instruction Prefetching (FDIP), an instruction prefetching mechanism that uses the FTQ to fetch instructions into L1I well ahead of time. With a decoupled front-end and FDIP, most L1I misses do not contribute to the front-end bottleneck. However, an L2 miss for an instruction fetch causes decode starvation if the decode queue is empty, leading to an empty issue queue and stalling the processor.

2.2 Replacement policies for instruction lines

An L2 replacement policy can play a significant role in mitigating front-end stalls. Dynamic Code Line Preservation (DCLIP) [29] preserves instruction lines over data lines in L2. DCLIP extends the 2-bit Re-Reference Interval Prediction (RRIP) [30] policy and inserts instructions with priority value two and data requests with priority value three. On future L2 hits, DCLIP updates the priority of instruction lines to zero but does not change re-reference predictions for data requests. DCLIP uses sampled sets to decide whether to prioritize instruction lines over data lines. EMISSARY [39] tries to keep in L2 the instruction lines that cause decode starvation. On decode starvation leading to an empty issue queue, the next instruction fetch is marked as critical. EMISSARY prioritizes critical lines over non-critical lines. Global History Reuse Predictor (GHRP) [16] is a replacement policy targeting L1I. For eviction, GHRP prioritizes lines that are likely to be *dead.* GHRP is expensive to implement, as it incurs a storage overhead of approximately 40 KB. Ripple [33] is a profile-guided software-only instruction cache replacement policy that uses profiling and program context.

3 ICARUS L2 Cache Replacement Policy

In this section, we first characterize the datacenter applications and then propose an effective mechanism for identifying critical instruction fetches (instruction lines) with the usage of branch history as an additional context (BHC). We also make a case for a bin-based replacement policy that uses *reuse* in conjunction with *criticality* (BRC), as we observe that the reuse behavior of critical lines differs from that of non-critical lines. In a nutshell, ICARUS (BHC+BRC) utilizes branch history to identify critical fetches and employs a bin-based replacement policy based on criticality and reuse.

Table 1 characterizes the evaluated benchmarks based on instruction footprint, branch predictor accuracy, critical instruction fetches, CPU stalls caused by critical fetches, and cache misses per kilo instructions (MPKI), separated by instructions and data. We use the gem5 full system simulator with FDIP [9, 18].

The analysis presented in the introduction (Figure 1) shows an average speedup of 19.80% when all instruction accesses hit in L2. The state-of-the-art L2 replacement policy EMISSARY advocates for a front-end criticality-based replacement policy for instructions in L2, which is a promising direction. Although EMISSARY improves performance, we observe some missed opportunities. Table 1 shows that a large fraction of instruction fetches do not cause decode starvation and an empty issue queue and are not critical. On average, only 3.49% of the instruction fetches are critical. According to Table 1, these 3.49% of the instruction fetches cause an average of 23.18% front-end stalls due to decode starvation. Ideally, a replacement policy should provide L2 hits to these critical instruction fetches. However, identifying critical instruction fetches (and critical lines) is not straightforward, as we observe the dynamic nature of criticality behavior (a particular instruction line sometimes behaves as a critical line and sometimes does not).

3.1 Identifying the dynamic behavior of critical fetches

In this Section, we argue for new ways to detect critical instruction fetches that show dynamic behavior in terms of criticality (d-critical). Figure 2 shows the percentage of s-critical and d-critical instruction lines (based on all fetched

Table 1. Characteristics of datacenter applications simulated for 200M instructions in their respective region of interest for a Granite Rapids-like cache hierarchy [15]: 64KB L1I, 2MB L2, and 3MB L3 per core. L2 uses TPLRU replacement policy.

Benchmarks	Instruction Footprint (MB)	Branch Predictor Accuracy (%)	Critical Fetches (%)	Stalls due to Critical Fetches (%)	L1I MPKI	L2 Instruction MPKI	L2 Data MPKI	L3 MPKI
tpcc	2.75	96.70	2.48	23.75	33.46	1.05	1.85	1.88
wikipedia	1.14	96.74	1.58	16.14	26.53	0.28	2.20	2.35
finagle-chirper	2.91	94.74	2.96	27.68	60.39	4.87	4.60	3.29
finagle-http	1.29	94.60	3.59	30.38	70.18	0.55	1.70	1.17
kafka	1.09	98.66	1.32	8.99	8.50	1.08	4.37	4.50
media-stream	0.54	98.59	2.19	10.68	14.64	1.04	13.92	14.00
speedometer	1.45	98.40	0.92	16.26	17.38	0.45	1.58	1.82
tailbench-specjbb	0.48	99.11	1.74	5.13	12.96	5.49	6.75	7.31
tailbench-xapian	0.37	99.14	0.44	7.16	1.29	0.33	1.51	1.11
tomcat	4.15	95.48	4.44	32.89	43.82	2.75	2.59	3.51
verilator	2.22	99.44	19.89	90.69	90.23	38.50	0.16	1.07
web-search	1.50	99.33	0.34	8.35	0.72	0.26	0.39	0.65
Average	**1.66**	**97.58**	**3.49**	**23.18**	**31.68**	**4.72**	**3.47**	**3.57**

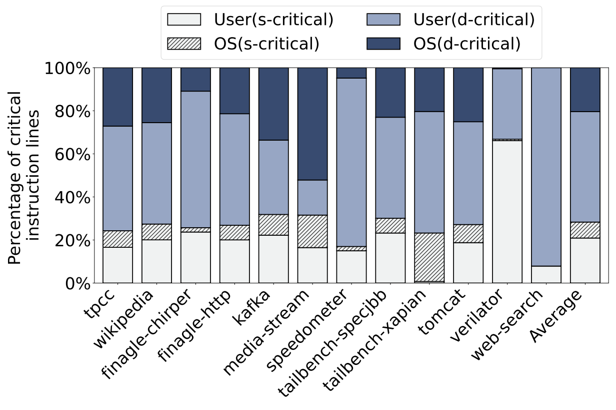


Figure 2. Distribution of critical instruction lines (s-critical and d-critical)

instructions that have been critical at least once). An instruction line that behaves critically for more than 95% of the instruction fetches is termed s-critical. The figure shows that 71.68% of the critical instruction lines are d-critical. A small fraction (28.32%) of instruction lines persist in their criticality behavior, which means that only 28.32% of the lines cause decode starvation the majority of times they are fetched, and a significant fraction of critical lines may or may not cause decode starvation in their future fetches. Interestingly, the dynamic behavior is visible across OS and user-level instruction lines.

3.2 An additional context (branch history) for critical fetch detection

We observe that the control flow has a significant impact on the criticality behavior of the instruction fetch. One way to capture control flow is to examine the branch history. We

Table 2. Effect of context (branch history) on decode starvation behavior for `finagle-chirper`

PC	Branch history	Decode starvation?
0xFFFFE8C8A240	**1**1000100	Always
	01000100	Never

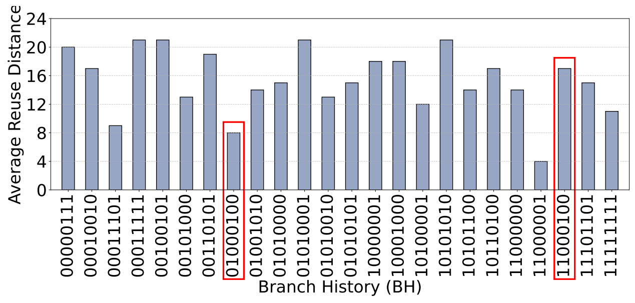


Figure 3. Variations in local reuse distance (set-based) at 16-way associative L2 with different branch histories for PC `0xFFFFE8C8A240` of `finagle-chirper`. Branch histories used in Table 2 are highlighted.

look at the last k predicted branches and encode the history into a string of 1s (taken) and 0s (not taken). Table 2 shows an example for the benchmark `finagle-chirper` for a cache line aligned program counter (PC) `0xFFFFE8C8A240` with two different branch histories (rightmost bit denotes the recent branch prediction) and its corresponding decode starvation behavior due to an L2 miss. As we can see, the exact cache line address causes decode starvation if the branch history is "11000100"; however, for the history of "01000100",

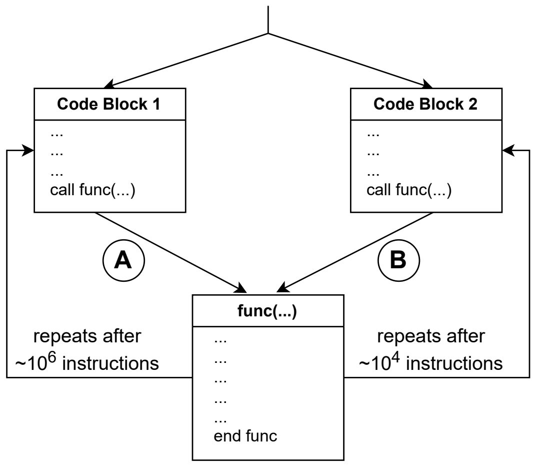


Figure 4. Control flow diagram of `finagle-chirper`.

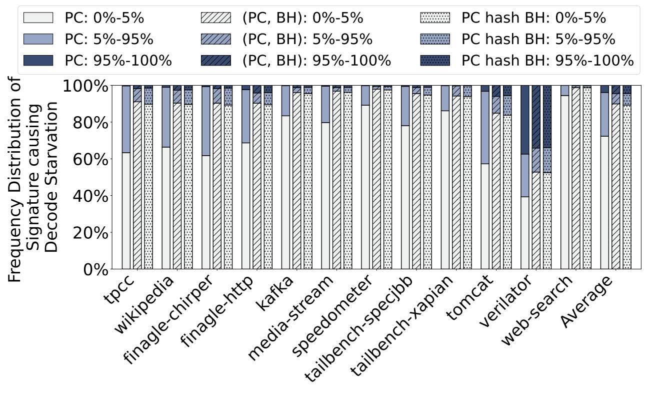


Figure 5. Distribution of PCs, tuple of PC and branch history (BH) of size nine, and PC hashed with BH categorized based on the percentage of occurrences causing decode starvation

it never leads to decode starvation. This motivates the inclusion of branch history in identifying critical instruction cache lines.

Why branch history affects criticality behavior? We observe that an instruction fetch shows different criticality behavior based on different branch histories as the reuse distance of an instruction cache line at L2 varies based on the branch history (Figure 3). Specifically, longer local reuse distances (greater than 16) tend to result in an L2 miss, leading to decode starvation. In contrast, branch histories associated with shorter local reuse distances result in L2 hits, thereby reducing the likelihood of decode starvation. Combining Table 2 and Figure 3, we can see that for PC `0xFFFFE8C8A240`, branch histories **1**1000100 and **0**1000100 have local reuse distances of 17 (L2 miss) and 8 (L2 hit), respectively. Figure 4 illustrates this behavior using a simplified control flow diagram for `finagle-chirper`. Depending on the branch outcome, execution flows through either block one (A) or block two (B), both of which lead to a common function `func`. When execution follows Block 1, `func` is invoked after approximately 10^6 instructions. However, when the execution flows through Block 2, `func` is invoked after every 10^4 instructions. This variation in control flow results in differences in branch histories, leading to significant differences in reuse distance for the same PC, thereby contributing to different criticality behavior. Please note that the branch history is affected by both user-level and OS-level instructions, as well as the interdependencies within the system stack.

Effect of branch history on decode starvation. Next, we quantify the effect of branch history and analyze the decode starvation behavior based on PCs and their branch history in Figure 5, where the first bar represents PCs and the second represents a tuple of PC and its branch history segregated based on the percentage of times that decode starvation is caused across their accesses. So, each benchmark in the X-axis has three signatures: PC and two different hash functions that are used to detect the criticality of an instruction line. There are three bins in the bar graph: 0%-5%, 5%-95%, and 95%-100%. If a PC almost always causes decode starvation, it appears in the bin 95%-100%. Similarly, if a PC rarely causes decode starvation, it appears in the 0%-5% bin. The PCs that exhibit dynamic behavior regarding decode starvation are located in the middle bin: 5%-95%. Ideally, a detection mechanism that detects critical fetches with high accuracy should have fewer PCs in the 5%-95% bin and more PCs in the two extreme bins.

Figure 5 shows the effect of using just the PC as context information (signature) to identify critical fetches (similar to EMISSARY). Next, we show the impact of a combination of PC with branch history as a signature to detect whether a particular instance of an instruction fetch with a particular branch history will lead to decode starvation. This is the ideal signature (hash function), although it is challenging to implement because of storage overheads. Finally, we use a hash of PC and branch history and show that we are closer to the ideal detector. The combination of PC with branch history is a well-known hash from initial branch predictor designs like GShare [37]. We observe that, on average, with PC as the signature, only 3.87% of PCs cause decode starvation more than 95% of the time, and almost 24% of PCs cause decode starvation between 5% to 95% of the time. This distribution becomes more concentrated if we take branch history into account with the PCs, as 4.6% of PCs cause decode starvation more than 95% of the time, while between 5%-95%, there is a frequency of only 5.5% (a drop from 24%). From this figure, it is evident that the decode starvation behavior of an instruction line significantly depends on the branch history that precedes the fetch of that instruction line.

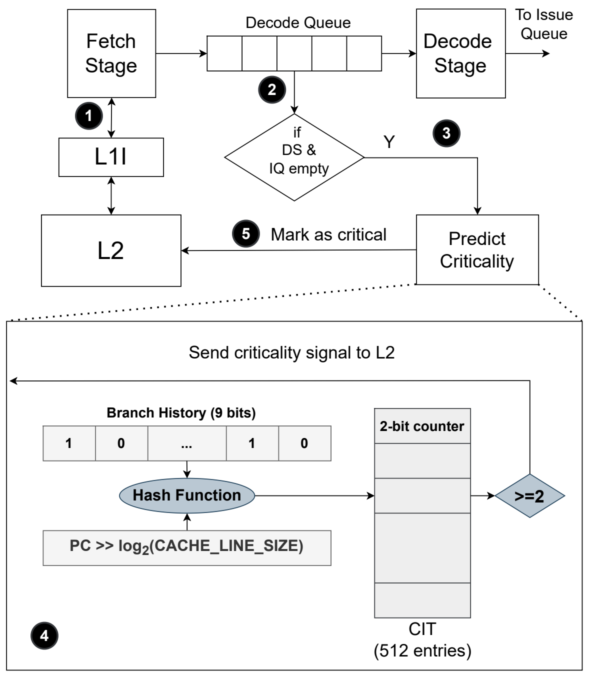


Figure 6. Critical instruction fetch detection with CIT. DS: Decode starvation and IQ empty: Issue queue empty.

3.3 Critical fetch detection

To detect critical instruction fetches, we utilize a critical instruction fetch identification table (CIT), a table with n entries where each entry uses a k-bit saturating counter. The cache line address of every fetch that sees a decode starvation is hashed with the branch history. We index the CIT with the hash, and for that particular index, we increment the counter every time we see a critical instruction fetch coming from L2, L3, or DRAM (❶❷❸ of Figure 6). If the counter value is greater than a threshold (two), then we send a criticality signal to the L2, marking that L2 line as critical (❹❺ of Figure 6).

Please note that with this threshold of two on the counter, we filter out dead L2 lines that are critical. We use the term "dead" for lines with no reuse (dead-on-arrival). The threshold of two ensures that a line causes decode starvation at least twice before it is marked as critical. Thus, dead-on-arrival decode-starvation-causing lines are never marked as critical. As we filter them, all critical lines see at least one reuse.

Next, as we do not decrement the counters of the critical fetch identification table, it is possible that we will end up over-predicting critical lines. To mitigate that, we flush the table once in x million cycles and restart the learning from scratch. For our implementation, we use a table of 512 entries, a branch history of nine bits, and a saturating counter of two bits with a threshold of two, and we flush the criticality table once in one million cycles. We do not use any replacement policy as CIT is tagless, which allows aliasing. We use the following hash function to index the CIT.

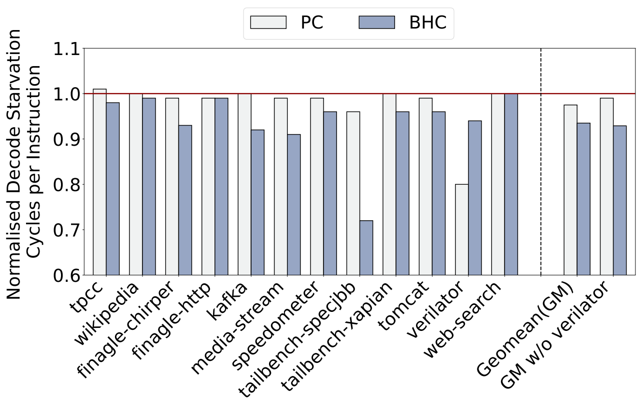


Figure 7. Decode starvation cycles per instruction normalized to TPLRU, for EMISSARY with PC-based signature and PC with branch history(BHC) based signature

$$line\_addr = pc \gg \log_2(CACHE\_LINE\_SIZE) \quad (1)$$

$$\text{signature} = (line\_addr \oplus (line\_addr \gg 3) \oplus bh) \;\&\; (\text{CIT\_SIZE} - 1) \quad (2)$$

Effect of BHC on decode starvation. Figure 7 shows the effect of using PC with additional context information in the form of branch history (BHC) for identifying critical fetch with the underlying L2 replacement policy as EMISSARY. Compared to TPLRU, PC with branch history-based EMISSARY reduces the decode starvation cycles by 6.5%, which used to be 2.5% with EMISSARY without context. Compared to the baseline TPLRU policy, if we exclude `Verilator`, then EMISSARY with the context information reduces decode starvation cycles by 7.1%, which was 1% with EMISSARY without the context.

3.4 Criticality, but also reuse

Cache replacement policies primarily work based on *reuse*. So, when designing a front-end criticality-aware replacement

As per Figure 8, once a line becomes critical, it shows that it will get reused (irrespective of the local reuse distance) as we filter out dead-on-arrival critical lines. Please note that the local reuse distance of a line refers to the reuse distance calculated within its cache set. Please refer to C:Short, C:Mid, C:Long in Figure 8. Also, with our context-based hash and the threshold used for the counter, our indexing in the criticality indicator table filters out dead critical lines (C:No reuse is nil in Figure 8). On the other hand, around 19% of non-critical lines are *dead*, and they get no reuse. So, EMISSARY rightly decides to isolate critical lines from non-critical lines so that non-critical lines cannot evict critical lines that will get reused. EMISSARY evicts non-critical lines unless the critical

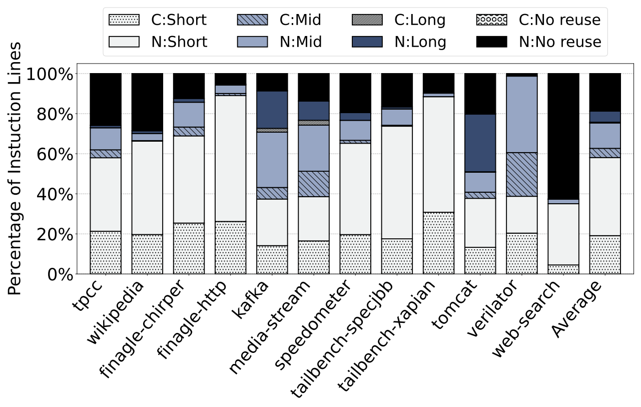
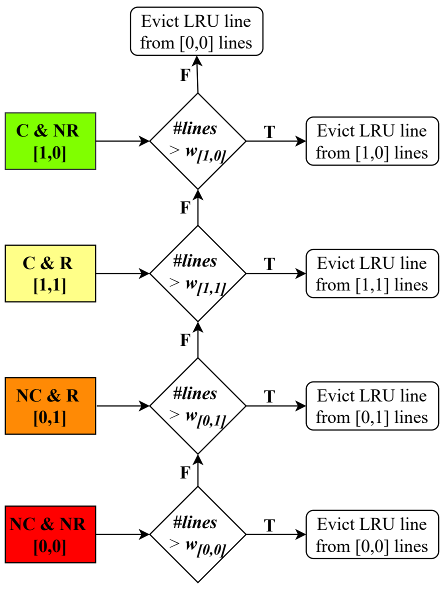


Figure 8. Local reuse distance of instruction lines at L2: short [0-16), mid [16-32), long 32+, and no reuse for both critical (C) and non-critical (N) lines. We use the hash of PC with branch history for detecting critical instruction fetches. Note that this study is independent of an L2 replacement policy and is conducted over a simulation of 50M instructions.

lines reach a certain threshold (N). If the number of critical lines is greater than N, then it evicts critical lines.

As per Figure 8, on average, 19.10% of the critical lines have a short reuse distance of less than 16 (C:Short in Figure 8), and as EMISSARY isolates critical lines from non-critical ones, on average, EMISSARY provides hits for most of the critical lines with short reuse distance even with reduced associativity of eight. However, because of its limited associativity, EMISSARY fails to provide additional cache hits to some mid-reuse critical lines. The same trend continues for long reuse cache lines with a reuse distance of more than 32. Please note that the number of critical lines that belong to mid and long reuse distances is extremely small compared to the number of non-critical lines, which corroborates our observation mentioned in Section 1, showing only 3.49% of instruction fetches lead to decode starvation. Finally, there are a few non-critical lines with mid and long reuse distances that, if kept in the cache for a longer duration, will start providing L2 hits. However, the utility of these L2 hits is marginal as they do not cause decode starvation. Please note that a non-critical cache line can become critical the next time it comes to the L2, thanks to the dynamic behavior of criticality.

In summary, based on our definition of criticality, a critical cache line always gets a reuse, irrespective of the local reuse distance. So, a critical line, which is yet to get a reuse should be preserved at the L2. EMISSARY does not consider the effect of reuse and treat all critical lines, equally. As an effect, it misses the opportunity of providing L2 hits to mid and long reuse critical lines.



Evict LRU line from [0,0] lines
F
C & NR [1,0]
#lines > $w_{[1,0]}$
T
Evict LRU line from [1,0] lines
F
C & R [1,1]
#lines > $w_{[1,1]}$
T
Evict LRU line from [1,1] lines
F
NC & R [0,1]
#lines > $w_{[0,1]}$
T
Evict LRU line from [0,1] lines
F
NC & NR [0,0]
#lines > $w_{[0,0]}$
T
Evict LRU line from [0,0] lines

Figure 9. BRC eviction policy. The L2 lines are segregated into four bins: C&NR, C&R, NC&R, NC&NR, where each bin has its watermark (w[x,y]). The search for an eviction candidate follows the priority order from red to green based on the respective watermarks.

3.5 BRC: Reuse and criticality based bins for L2 replacement policy

Segregation of L2 lines. We segregate L2 lines with a local reuse distance of 0-16 from cache lines having a local reuse distance of [16-32). We keep the lines having a local reuse distance of [16-32) a bit longer in the cache so that they get a hit on the subsequent access. We achieve this segregation using a straightforward method. We associate each L2 line with an additional bit, the *reuse* bit. Initially, the bit is set to 0 when the cache line is filled in for the first time. If the cache line gets a hit in its lifetime, we set the bit to one. Based on the reuse bit, we try to improve the eviction policy to improve hits to mid and long reuse L2 lines.

Eviction policy. We propose the Reuse and Criticality Bin-based replacement policy (BRC), where the eviction decisions are made based on two bits (criticality and reuse) per L2 line. To allow critical instruction lines with high local reuse distance (32+) to remain long enough in the cache, we assign the highest priority to the cache lines that are critical and not reused (critical bit is one, but reuse bit is zero) to preserve them in L2. As per Figure 8, all critical lines get a reuse, which means critical lines with the reuse bit zero, will get a reuse in the future. However, this is not true for non-critical lines. Next, we try to keep critical lines that are reused (critical bit is one, and reuse bit is one). Next, we prioritize non-critical lines that are reused (critical bit is zero and reuse bit is one) and, finally, non-critical lines that are yet to be reused (critical and reuse bits are zero).

In summary, we classify cache lines of an L2 set into four bins: critical and not reused [1, 0]; critical and reused [1, 1]; non-critical and reused [0, 1]; and non-critical and not reused [0, 0]. Based on these bins, the ICARUS eviction policy decides which eviction bin to choose and then which cache line to choose from that selected bin. Please note that we use the Tree-based Pseudo LRU (TPLRU) bit apart from the critical and reuse bits. As the L2 is shared by data and instruction lines, additional protection to instruction lines at the cost of data lines will lead to a performance drop. So, we use TPLRU bits that maintain the TPLRU chain among all lines within a set, irrespective of data and instructions. All the data lines participate in two bins: non-critical and reused, and non-critical and yet to be reused, along with their respective PLRU bits, eventually becoming a 2-bit RRIP policy among data lines. If we remove the *reuse* bit from our policy, our policy will behave the same as EMISSARY but with BHC.

Figure 9 shows the eviction policy that evicts lines from low priority [0,0]. Overall, the eviction sequence among four bins is [0, 0], [0, 1], [1, 1], and [1, 0]. We use watermarks for each bin that set the minimum threshold for each bin before we start evicting lines from that bin. Based on the criticality and reuse, we use the watermarks of two, four, six, and four for four bins: [0,0],[0,1],[1,1], and [1,0]. Note that bin [1,1] needs more space than [1,0] because most of the [1,0] lines will eventually become [1,1], and on top of that, [0,1] lines can also become [1,1]. In case of a tie among multiple bins, our eviction policy evicts lines from the low-priority bin. However, on average, less than 0.10% of L2 evictions lead to a tie. With BRC, more than 99.6% of L2 evictions are non-critical instruction lines. With these watermarks, we ensure an appropriate L2 space for each bin without causing misses for other bins.

Insertion and promotion policy. Figure 10 shows the promotion policy with BRC. An L2 line is inserted in the MRU position with the criticality bit set to zero and the reuse bit set to zero. On a reuse or a criticality signal from the CIT, the cache line moves from one bin to another.

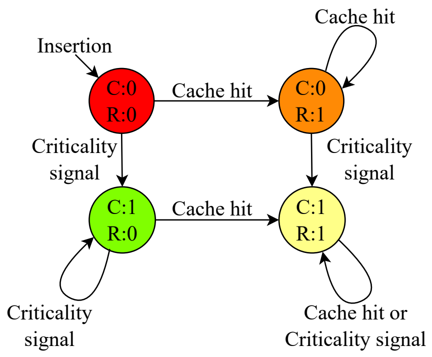


Figure 10. State transitions of L2 lines with ICARUS. C denotes the criticality bit, and R denotes the reuse bit.

3.6 How to choose watermarks?

For the 12 applications that we studied, watermarks of two, four, six, and four for bins [0,0],[0,1], [1,1], and [1,0] perform the best. We perform extensive experiments and sweep through all possible values for these watermarks. Based on our experiments, we conclude the following: (i) for bin [0,0], if we use more than two as the watermark, it starts occupying L2 space, affecting critical instruction lines. For applications like `Verilator`, for bin [0,0], a watermark of zero performs the best as the code footprint is huge and 19.89% of the instruction fetches are critical. So, decreasing the watermark for bin [0,0] ensures the eviction of non-critical and non-reused lines immediately. However, this is not the case with other applications. (ii) for bin [0,1], the watermark of four provides enough time to ensure a transition (if there is a possibility of a transition) from [0,1] to [1,1] as 1.2% of all transitions are from [0,1] to [1,1]. Increasing this watermark does not improve performance further. However, if we decrease it, applications like `finagle-chirper` and `tomcat` get affected. On average, less than 6.32% of transitions are from non-critical L2 lines to critical L2 lines. (iii) For bins [1,1] and [1,0], watermarks of six and four provide sufficient time for providing hits to long-reuse critical lines. We keep [1,1] lines for more time compared to [1,0] as we observe that once a line goes to [1,1], it gets atleast one more reuse. When we increase these watermarks further, there is no utility. Decreasing these watermarks further degrades performance marginally. However, if some of the watermarks are less than four (e.g., 4-6-3-3 and 4-8-2-2), especially for both [1,1] and [1,0], performance drops by more than 2%.

Storage overhead. Table 3 shows the storage budget of ICARUS, which is 8.13KB for a 2MB L2. Compared with EMISSARY, which takes 4KB of storage, ICARUS incurs an additional storage of 4.13KB and delivers significant performance improvement. We observe that, on average, across all benchmarks, we see 272 unique instruction cache lines getting accessed within 1M cycles (reset-interval), with a maximum of about 680 (`tomcat`). CIT of more than 512 provides no additional utility.

4 Evaluation

4.1 Simulation methodology

We use the gem5, a widely used cycle-accurate simulator with FDIP [9, 18], to run a detailed CPU model in Full System

Table 3. Storage overhead with ICARUS

Structures	Description	Storage
CIT	512 entries, 2-bit saturating counter per entry	128B
Branch history register	For storing branch history	9 bits
Flags	Two one-bit flags for criticality and reuse for each cache line at the L2 of size 2 MB	8 KB
Total		**8.13 KB**

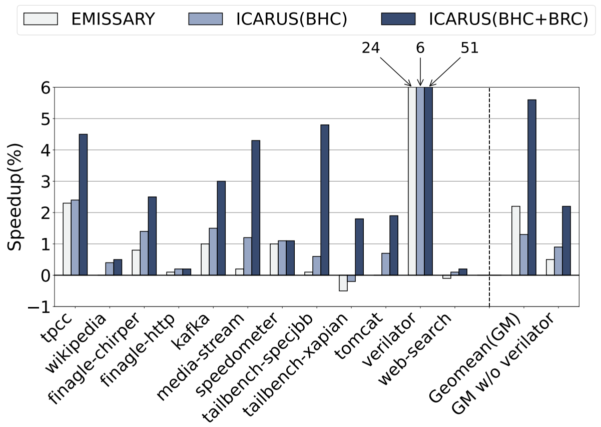


Figure 11. Speedup relative to TPLRU

Table 4. Description of benchmarks

Benchmarks	Description
tpcc	Online Transaction Processing application[10] from OLTP-Bench suite[22]
wikipedia	MediaWiki application on Wikipedia dataset[3] from OLTP-Bench suite
finagle-chirper	A microblogging service by Twitter[4] from Renaissance[42] based on JVM
finagle-http	Twitter's HTTP server from Renaissance
kafka	Apache's distributed event streaming application from Dacapo benchmark suite[19]
media-stream	Simulates video traffic from Cloudsuite V4[23]
speedometer	A JavaScript benchmark that runs on a web browser and tests the number of threads spawned in a minute[2]
specjbb	A SPEC benchmark to test Java application features[1] from Tailbench suite[32]
xapian	A web-search application from the Tailbench suite
tomcat	Apache's implementation of Jakarta Servlet, Jakarta Expression Language, and WebSocket[7] from Dacapo benchmark suite
verilator	Simulates the RTL design of Rocket Chip[11] simulating quick sort code
web-search	Apache Solr search engine application[6] from Cloudsuite V4

(FS) mode with Ubuntu 18.04 (Linux kernel version 4.15). We use the simulation infrastructure developed by the authors of EMISSARY and the checkpoints shared by the authors of EMISSARY as part of the artifact [24]. Table 4 provides the details of 12 datacenter applications that we use for our evaluation. Table 5 details the simulated processor and memory hierarchy. Our simulation parameters are similar to Intel's Granite Rapids [15] cache hierarchy, configured with the TPLRU baseline as the L2 replacement policy. We use the existing TPLRU bits without adding new state or bits. Insertion and promotion of blocks at L2 follow the standard TPLRU behaviour; only the eviction decision is modified based on the ICARUS eviction policy. We use the FDIP-based instruction prefetcher [43] in the baseline, and the cache hierarchy is non-inclusive. We simulate each datacenter benchmark for 200 million instructions after a warm-up of 50 million instructions.

4.2 Performance evaluation

Performance. We use improvement in runtime as the metric for performance. As a baseline L2 replacement policy, we use the TPLRU policy. We compare ICARUS with EMISSARY, as EMISSARY is the state-of-the-art replacement policy. We use the authors' version of EMISSARY, available in the public domain [9]. Figure 11 shows the performance improvement across all applications with EMISSARY, ICARUS having branch history criticality (BHC), and ICARUS with both branch history criticality and criticality and reuse bin-based eviction policy (BHC+BRC). On average, ICARUS (BHC+BRC), our final contribution, improves performance by 5.6% as high as 51% for `Verilator` while the performance excluding `Verilator` is 2.2%. EMISSARY, on the other hand, delivers a performance improvement of 2.2%, which becomes 1% without `Verilator`. For `tailbench-xapian`, there is a marginal performance drop (less than 0.2%) with ICARUS (BHC) as the data misses increase by 5.06%. However, with ICARUS (BHC+BRC), we mitigate this performance drop and improve performance by 1.8%.

Table 5. Simulated parameters

Fetch Target Queue	24 entry 192-instructions, FDIP [43]
Branch Predictor	TAGE [46], ITTAGE [45]
BTB	16K entries
Fetch/Decode/ Commit width	6 / 6 / 8
ROB	512 entries
Issue/Load/Store Queue	240 / 192 / 114 entries
Int/FP Registers	280 / 332
Granite Rapids-like cache hierarchy	
Private L1I/L1D	64KB(I), 48KB(D), 8/12 way, 64B line, TPLRU, 6 and 5 cycles, 16 MSHRs
Private unified L2	2MB, 16-way, 64B line size, TPLRU, 16 cycles, 32 MSHRs
Shared L3	3MB/core, 12-way, 64B line, TPLRU, 33 cycles, 64 MSHRs, non-inclusive

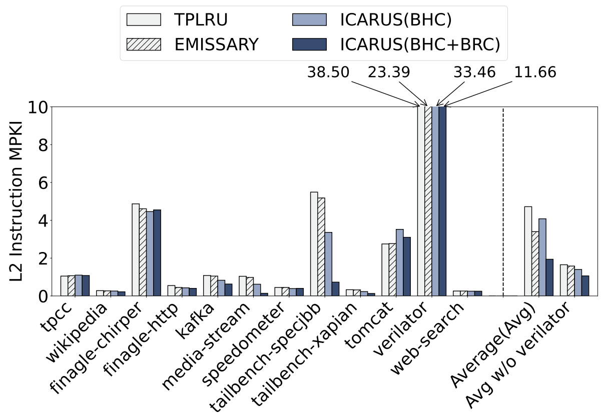


Figure 12. L2 instruction MPKI

Instruction MPKI and decode starvation. Figure 12 shows the reduction in instruction MPKI at the L2. On average, ICARUS reduces instruction MPKI from 4.72 to 1.94. The reduction in MPKI results in a reduction in decode starvation cycles (because of an L2 miss) per instruction. Figure 13 shows decode starvation cycles per instruction normalized to TPLRU. ICARUS outperforms EMISSARY in all benchmarks, reducing the front-end bottleneck because of an L2 miss, significantly (an average reduction in decode starvation cycle per instruction from 0.97 to 0.86). ICARUS improves the instruction MPKI but not at the cost of a significant increase in data MPKI. We observe an interesting trend for `tomcat`, where there is an increase in instruction MPKI with ICARUS (Figure 12). However, as per Figure 13, there is a reduction

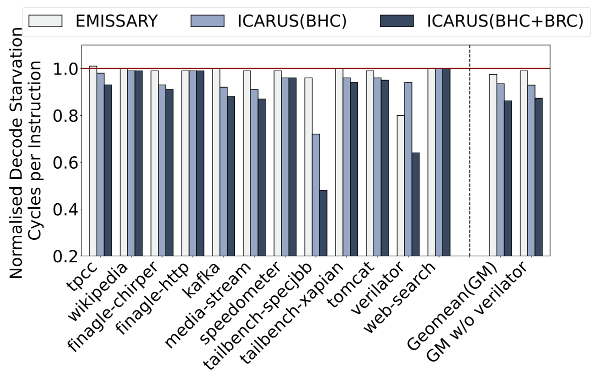


Figure 13. Decode starvation cycles per instruction normalized to TPLRU

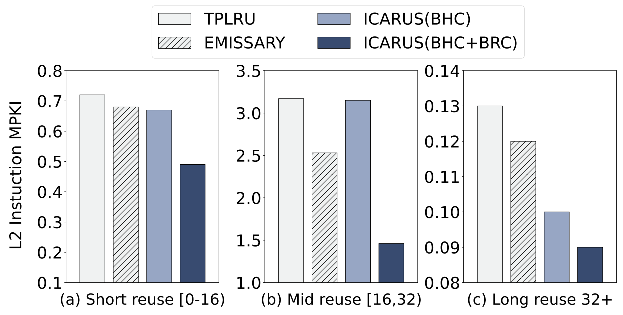


Figure 14. L2 instruction MPKI segregated based on reuse

in decode starvation cycles per instruction. So, the increase in MPKI is actually for the non-critical lines, and because of this, an increase in MPKI does not affect performance. This trend is also a side effect of our replacement policy, prioritizing critical lines over non-critical lines.

Instruction MPKI based on their reuse. One of the key contributions of ICARUS (BHC+BRC) is to keep the critical instruction lines that will be reused (mid and long local reuse distance). Figure 14 shows the decrease in misses per kilo instructions (MPKI) of instruction lines at the L2. As per Figure 14, ICARUS (BHC+BRC) reduces the MPKI of short reuse lines marginally as compared to ICARUS (BHC). However, the primary difference comes from the ICARUS (BHC+BRC) when it comes to mid reuse lines, as it significantly reduces the MPKI of instruction lines that are critical and have high reuse distance. For long reuse instruction lines, the figure shows the average trend, where the MPKI drop appears marginal, which is 0.1 to 0.09. However, certain workloads, such as `Kafka`, benefit significantly, where the MPKI of long reuse lines drops from 0.8 to 0.02 with ICARUS (BHC+BRC). Regarding the data lines at the L2, ICARUS (BHC+BRC) reduces the MPKI of both mid- and long-reuse data lines, with a slight increase in MPKI for short-reuse data lines. Note that criticality plays no role among data lines, and the effect in terms of MPKI is solely based on reuse (Figure 15).

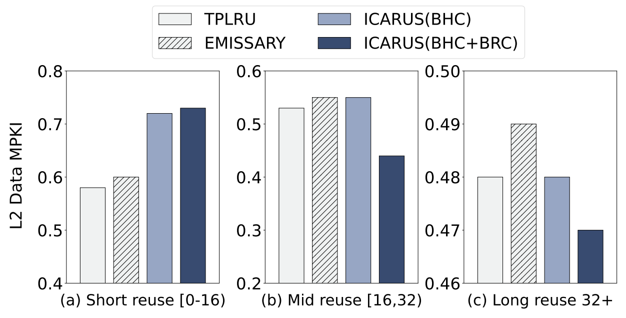


Figure 15. L2 data MPKI segregated based on reuse

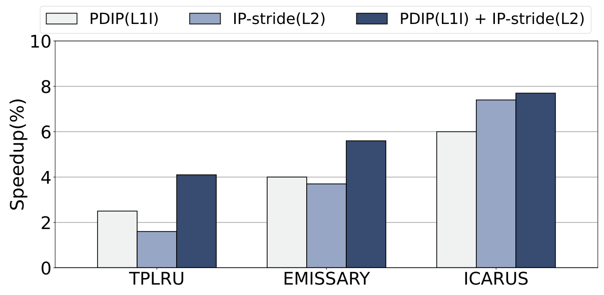


Figure 16. Speedup with different prefetching techniques normalized to no data prefetching and FDIP.

4.3 Interaction with instruction and data prefetchers

Interaction with instruction prefetcher. Priority Directed Instruction Prefetching (PDIP) [25] is a recent instruction prefetching technique that provides a good performance boost on top of the EMISSARY replacement policy. PDIP is a front-end criticality-based instruction prefetcher that considers instruction fetches missed in the L1I and caused decode starvation. We quantify the effect of ICARUS in the presence of PDIP as an instruction prefetcher on top of FDIP. Figure 16 shows the effect of PDIP without EMISSARY and ICARUS (best combination of ICARUS among all proposed techniques), where PDIP alone improves performance by 2.5% with TPLRU, and the combination of PDIP+EMISSARY improves performance by 4% compared to the baseline with TPLRU as the replacement policy and FDIP as the instruction prefetcher. PDIP with ICARUS improves performance by 6%. The improvement comes from additional L1I hits because of PDIP. PDIP and ICARUS work synergistically as ICARUS keeps critical lines longer, and PDIP prefetches these lines into L1I. As some of the critical instruction lines from L2 are also causing decode starvation even on an L2 hit, L1 hits to those instruction fetches mitigate decode starvation cycles further.

Interaction with data prefetcher. Next, we quantify the effect of the IP-stride data prefetcher at L2 that is commercially implemented in AMD and Intel processors. Figure 16 shows the effect of IP-stride without EMISSARY and ICARUS, where IP-stride at the L2 improves performance by 1.6% with TPLRU, and the combination of IP-stride+EMISSARY improves performance by 3.7% compared to the baseline with TPLRU as the replacement policy and FDIP as the instruction prefetcher. IP-stride with ICARUS improves performance by 7.4%. The improvement comes from additional L2 data hits. Next, we quantify the interaction of PDIP and IP-stride with EMISSARY and ICARUS. PDIP+IP-stride improves the average performance of EMISSARY by 5.6%, whereas with ICARUS, it goes up to 7.7%. Overall, ICARUS works synergistically with instruction and data prefetchers.

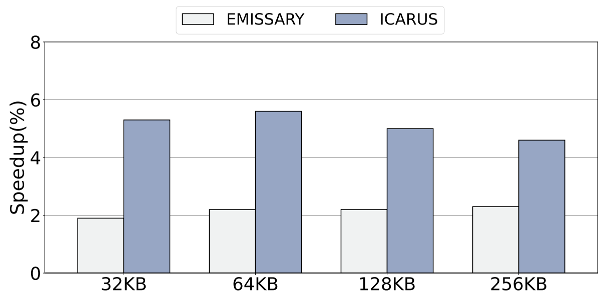


Figure 17. Sensitivity study: effect of L1I size

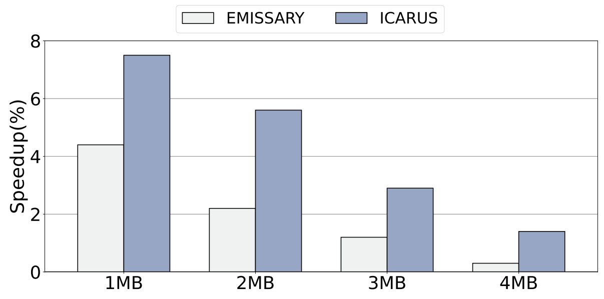


Figure 18. Sensitivity study: effect of L2 size on performance

4.4 Sensitivity studies

L1I size. Figure 17 shows the effect of L1I size on EMISSARY and ICARUS. There is a marginal decrease in speedup with the increase in L1I size. However, even for a large L1I like 256KB, ICARUS provides a significant speedup of 4.6% as compared to TPLRU, whereas EMISSARY provides a speedup of 2.2%. Overall, ICARUS outperforms EMISSARY across all L1I sizes.

L2 size. Figure 18 shows the effect of L2 size on EMISSARY and ICARUS. There is a drop in performance with the increase in L2 sizes from 1MB to 4MB. For a large L2 like 4MB, ICARUS provides a speedup of less than 1.4% as compared to TPLRU, whereas EMISSARY provides a speedup of 0.3%. This is primarily because the code working set of most applications fits within the large L2. As a result, the L2 instruction

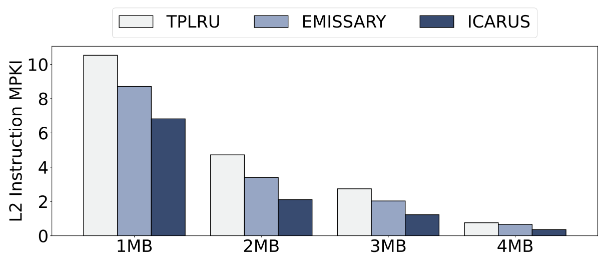


Figure 19. L2 instruction MPKI for different L2 sizes

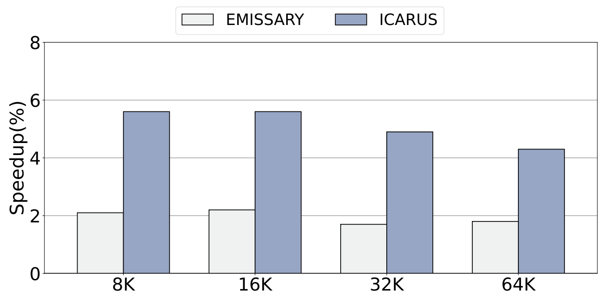


Figure 20. Sensitivity study: effect of BTB size

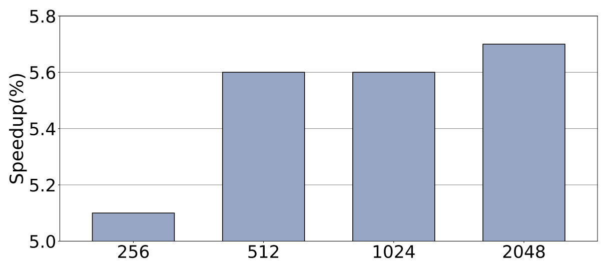


Figure 21. Speedup with ICARUS with different criticality indicator table sizes normalised to TPLRU.

MPKI is significantly reduced—falling below 1 for a 4MB L2, as shown in Figure 19.

BTB size. Figure 20 shows the effect of BTB size on EMISSARY and ICARUS. There is a marginal decrease in speedup with the increase in BTB size. For a large BTB of 64K entries, ICARUS provides a significant speedup of 4.3% as compared to TPLRU, whereas EMISSARY provides a speedup of 1.8%. Overall, ICARUS outperforms EMISSARY across all BTB sizes.

CIT size. For our evaluation, we use a 512-entry table to identify critical instruction fetches. We perform a sweep across different table sizes, starting from 256 to 2048 entries, and quantify ICARUS speedup as shown in Figure 21. Beyond 2048 entries, we observe marginal improvement (less than 0.01%) in overall performance, and below 512, with 256 entries, there is a performance drop.

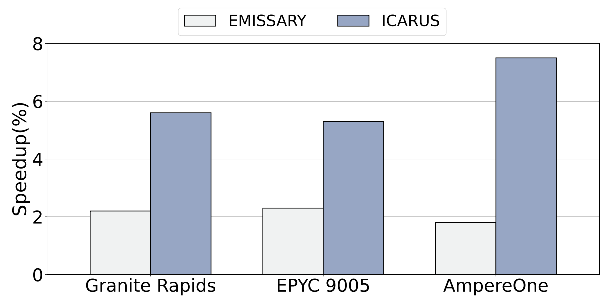


Figure 22. Sensitivity study: effect of cache hierarchies

Cache hierarchies. So far, we have evaluated ICARUS for Intel Xeon Granite Rapids like cache hierarchy. Now, we evaluate it for AMD Zen 5 [13] based EPYC 9005 (Turin) [12] and ARM-based AmpereOne [14] cache hierarchies. The parameters used for them are shown in Table 6 and Table 7. Figure 22 shows the effectiveness of EMISSARY and ICARUS across three different cache hierarchies. ICARUS outperforms EMISSARY across all cache hierarchies.

4.5 ICARUS for applications with high critical instruction fetches and small L2 sizes

We observe that a large number of instruction fetches are critical for applications like `Verilator` with a smaller L2 such as 1MB L2 (same as AMD's Zen 5 [13]). Through empirical analysis, we observe that with `Verilator`, more than 50% of instruction fetches are critical with an extremely high L2 instruction MPKI of 61.52 if we use an L2 of 1MB. On the other hand, the data MPKI at the L2 is just 0.24. We observe that, with high L2 instruction MPKI, `Verilator` sees hits at L3, causing decode starvation for more than 40 cycles, which is a significant front-end bottleneck. With ICARUS for a 1MB L2 [13], a large fraction of L2 instruction lines becomes critical. However, there are a few L2 instruction lines that cause decode starvation for more than 40 cycles. So, for `Verilator`, there is a need to preserve highly costly critical instruction lines. To accommodate the same, we redefine the definition of critical fetch (critical instruction line) as any instruction fetch that causes decode starvation for more than 40 cycles. In this way, ICARUS preserves costly critical instructions. We use a threshold based on the fraction of critical fetches to trigger ICARUS that preserves only the costly critical lines. In our case, the threshold is 8% (more than the average). Depending on the application behavior, both the cost and the threshold to identify applications with costly fetches can be tuned.

5 Related work

Cache management policies for instructions. GHRP[16] is an instruction cache replacement policy focused on minimizing the number of misses by identifying dead blocks at the L1I cache, which is orthogonal to ours. Prior works show

Table 6. EPYC 9005-like cache hierarchy

Field/Model	AMD 9005-like cache hierarchy
L1I/L1D caches	32KB(I)/48KB(D), 8/12 way, 64B cache line, 4 cycles
Unified L2	1MB, 16-way, 64B line size, 14 cycles
Shared L3	4MB, 16-way, 64B line size, 40 cycles

Table 7. AmpereOne-like cache hierarchy

Field/Model	AmpereOne-like cache hierarchy
L1I/L1D caches	16KB(I)/64KB(D), 4-way, 64B cache line, 3/4 cycles
Unified L2	2MB, 8-way, 64B line size, 11 cycles
Shared L3	1MB, 8-way, 64B line size, 10 cycles

that GHRP fails to achieve significant performance gain for datacenter applications [33]. Ripple[33] is a software-only profile-guided cache management technique to improve the performance of the instruction cache. A cache line that is no longer required is identified in the offline analysis, and a cache line eviction instruction is inserted in the binary after the last access along all execution paths. Ripple can be applied on top of ICARUS for better L1I performance as these are orthogonal techniques. Admission-Controlled Instruction Cache (ACIC) [50] incorporates a small i-Filter as done in prior works [26, 35] to separate spatial from temporal instruction accesses. However, a simple separation does not suffice, so it additionally predicts whether the block will continue to have temporal locality after the burst of spatial locality. ACIC can be used on top of ICARUS to address instruction cache pollution. A recent work [20] proposes cooperative last-level TLB (STLB) and L2 replacement policies. The policy improves the instruction translation hit rates at the STLB at the cost of data translation misses. To compensate for that, it prioritizes data translations at the L2. The applications that we use do not have significant instruction translation misses at STLB. In the case of high-instruction translation MPKI, ICARUS can be used at the STLB too, preserving critical translation lines.

Instruction prefetching. ICARUS preserves instruction lines at the L2. However, instruction prefetching techniques go one step further and prefetch instructions into L1I. Priority directed instruction prefetching (PDIP)[25] is an instruction prefetching technique that complements FDIP by issuing prefetches for only targets where FDIP struggles. As shown in Figure 16, ICARUS complements PDIP well. Utility-Driven fetch-directed instruction Prefetching (UDP) [40] improves accuracy and timeliness of improve the accuracy of FDIP while prefetching instructions in the wrong path. UDP shows that dynamically tuning the FTQ depth can improve the accuracy and timeliness of instruction prefetch requests. Similar to PDIP, UDP will complement well with ICARUS, as ICARUS improves L2 performance and UDP improves L1I performance. Entangled Instruction Prefetching (EIP) [44] proposed entangling-based prefetching that associates a cache miss causing line of a variable latency L with an access that is accessed L cycles prior.

Cache replacement for other front-end structures. A profile-guided BTB replacement policy, Thermometer [48], focuses on both transient and holistic branch behavior to reduce BTB misses for datacenter applications. FURBYS [51], a practical profile-guided policy, reduces micro-op cache misses and improves performance-per-watt in datacenter applications. Ajorpaz et al. [38] use global history reuse prediction to eliminate dead entries both at the BTB and L1I. These techniques are orthogonal to ICARUS and can be used to enhance the front-end performance further.

Recent criticality and context-based microarchitecture optimizations. Load criticality-based data prefetching [41] highlights that conditional branches and branch history affect the loads and their criticality, resulting in dynamic-criticality behavior. Critical slice prefetching [36] performs data prefetching by identifying critical branch slices and tries to reduce branch misprediction penalties. The criticality-driven fetch [21] provides an execution paradigm that fetches, allocates, and executes instructions on the program's critical path. PHAST [34], a memory dependence predictor, uses the information about the path taken from the store to its dependent load and the store distance. ICARUS is orthogonal to these proposals as none target L2 instruction caching.

6 Conclusion

We proposed ICARUS, a front-end criticality and reuse-based L2 replacement policy prioritizing instruction lines for datacenter applications. We argued for context-based critical fetch detection and branch history to detect critical fetches in the presence of the dynamic behavior of critical fetches. Next, we proposed a criticality and reuse-based L2 replacement policy that preserves critical instruction lines with long reuse. On average, across 12 datacenter applications, ICARUS outperforms the baseline Tree-based Pseudo LRU policy by 5.6% with a maximum improvement of 51%, which is a significant improvement compared to the state-of-the-art replacement policy EMISSARY that improves performance by 2.2% over the Tree-based Pseudo LRU policy. ICARUS provides this performance improvement with a storage overhead of 8.13KB, which is marginal compared to a 2MB L2.

Acknowledgments

We thank the members of the CASPER group for their contributions to improving the final draft. This work is supported by an AMD research gift.

References

[1] 2015. SPECjbb 2015. https://www.spec.org/jbb2015/.
[2] 2018. Speedometer 2.0. https://browserbench.org/Speedometer2.0/.
[3] 2020. MediaWiki. https://www.mediawiki.org/wiki/MediaWiki.
[4] 2021. Twitter finagle. https://twitter.github.io/finagle/.

[5] 2023. Apache kafka. https://kafka.apache.org/.
[6] 2023. Apache Solr. https://solr.apache.org/.
[7] 2023. Apache tomcat. https://tomcat.apache.org/.
[8] 2023. ARM Neoverse V2. https://chipsandcheese.com/p/hot-chips-2023-arms-neoverse-v2/.
[9] 2023. Gem5-EMISSARY. https://github.com/PrincetonUniversity/gem5_FDIP.git.
[10] 2023. TPC-C. http://www.tpc.org/tpcc/.
[11] 2023. Verilator. https://www.veripool.org/wiki/verilator.
[12] 2024. AMD EPYC 9005 Turin. https://en.wikichip.org/wiki/amd/epyc#9005_Series_.28Zen_5.29.
[13] 2024. AMD Zen5. https://en.wikichip.org/wiki/amd/microarchitectures/zen_5.
[14] 2024. AmpereOne. https://chipsandcheese.com/p/ampereone-at-hot-chips-2024-maximizing-density.
[15] 2024. Intel Granite Rapids. https://en.wikipedia.org/wiki/Granite_Rapids.
[16] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A Jiménez. 2018. Exploring predictive replacement policies for instruction cache and branch target buffer. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 519–532. https://doi.org/10.1109/ISCA.2018.00050
[17] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*. 462–473. https://doi.org/10.1145/3307650.3322234
[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718
[19] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. 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* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488
[20] Dimitrios Chasapis, Georgios Vavouliotis, Daniel A. Jiménez, and Marc Casas. 2025. Instruction-Aware Cooperative TLB and Cache Replacement Policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*. Association for Computing Machinery, 619–636. https://doi.org/10.1145/3669940.3707247
[21] Aniket Deshmukh and Yale N. Patt. 2021. Criticality Driven Fetch. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, 380–391. https://doi.org/10.1145/3466752.3480115
[22] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288. https://doi.org/10.14778/2732240.2732246
[23] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices* 47, 4 (2012), 37–48. https://doi.org/10.1145/2150976.2150982
[24] Bhargav Reddy Godala. 2024. ARM 64-bit datacenter workloads used in EMISSARY paper. https://drive.google.com/file/d/1ac60R-nuENQjw-rRBR-0S9rYQEEuCvyp/view. Accessed: 2024-12-07.
[25] Bhargav Reddy Godala, Sankara Prasad Ramesh, Gilles A Pokam, Jared Stark, Andre Seznec, Dean Tullsen, and David I August. 2024. PDIP: Priority Directed Instruction Prefetching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 846–861. https://doi.org/10.1145/3620665.3640394
[26] Stephen Hines, David Whalley, and Gary Tyson. 2007. Guaranteeing hits to improve the efficiency of a small instruction cache. In *40th annual IEEE/ACM international symposium on microarchitecture (MICRO 2007)*. IEEE, 433–444. https://doi.org/10.1109/MICRO.2007.28
[27] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing fetch-directed instruction prefetching: An industry perspective. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 172–182. https://doi.org/10.1109/ISPASS51385.2021.00034
[28] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, June 18-22, 2016*. IEEE Computer Society, 78–89. https://doi.org/10.1109/ISCA.2016.17
[29] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely, and Joel Emer. 2015. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 343–353. https://doi.org/10.1109/HPCA.2015.7056045
[30] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH computer architecture news* 38, 3 (2010), 60–71. https://doi.org/10.1145/1815961.1815971
[31] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169. https://doi.org/10.1145/2749469.2750392
[32] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10. https://doi.org/10.1109/IISWC.2016.7581261
[33] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-guided instruction cache replacement for data center applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 734–747. https://doi.org/10.1109/ISCA52012.2021.00063
[34] Sebastian S Kim and Alberto Ros. 2024. Effective Context-Sensitive Memory Dependence Prediction. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 515–527. https://doi.org/10.1109/HPCA57654.2024.00045
[35] Johnson Kin, Munish Gupta, and William H Mangione-Smith. 1997. The filter cache: An energy efficient memory structure. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 184–193. https://doi.org/10.1109/MICRO.1997.645809
[36] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: critical slice prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, 300–313. https://doi.org/10.1145/3503222.3507745
[37] Scott McFarling. 1993. Combining branch predictors. In *Vol. 49. Technical Report TN-36, Digital Western Research Laboratory, 1993.*

[38] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A. Jiménez. 2018. Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 519–532. https://doi.org/10.1109/ISCA.2018.00050

[39] Nayana Prasad Nagendra, Bhargav Reddy Godala, Ishita Chaturvedi, Atmn Patel, Svilen Kanev, Tipp Moseley, Jared Stark, Gilles A Pokam, Simone Campanoni, and David I August. 2023. EMISSARY: Enhanced Miss Awareness Replacement Policy for L2 Instruction Caching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13. https://doi.org/10.1145/3579371.3589097

[40] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. 2024. UDP: Utility-Driven Fetch Directed Instruction Prefetching . In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 1188–1201. https://doi.org/10.1109/ISCA59077.2024.00089

[41] Biswabandan Panda. 2023. CLIP: Load criticality based data prefetching for bandwidth-constrained many-core systems. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 714–727. https://doi.org/10.1145/3613424.3614245

[42] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[43] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 16–27. https://doi.org/10.1109/MICRO.1999.809439

[44] Alberto Ros and Alexandra Jimborean. 2021. A cost-effective entangling prefetcher for instructions. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 99–111. https://doi.org/10.1109/ISCA52012.2021.00017

[45] André Seznec. 2011. A 64-Kbytes ITTAGE indirect branch predictor. In *JWAC-2: Championship Branch Prediction*. JILP, San Jose, United States. https://jilp.org/jwac-2/program/cbp3_07_seznec.pdf

[46] André Seznec. 2016. TAGE-SC-L Branch Predictors Again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. https://jilp.org/jwac-2/program/cbp3_07_seznec.pdf

[47] Ishan Shah, Akanksha Jain, and Calvin Lin. 2022. Effective mimicry of belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 558–572. https://doi.org/10.1109/MM.2023.3275079

[48] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: profile-guided btb replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 742–756. https://doi.org/10.1145/3470496.3527430

[49] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci. 2022. One Profile Fits All: Profile-Guided Linux Kernel Optimizations for Data Center Applications. *SIGOPS Oper. Syst. Rev.* 56, 1 (June 2022), 26–33. https://doi.org/10.1145/3544497.3544502

[50] Yunjin Wang, Chia-Hao Chang, Anand Sivasubramaniam, and Niranjan Soundararajan. 2023. ACIC: Admission-controlled instruction cache. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 165–178. https://doi.org/10.1109/HPCA56546.2023.10071033

[51] Kan Zhu, Yilong Zhao, Yufei Gao, Peter Braun, Tanvir Ahmed Khan, Heiner Litz, Baris Kasikci, and Shuwen Deng. 2025. From Optimal to Practical: Efficient Micro-op Cache Replacement Policies for Data Center Applications. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 716–731. https://doi.org/10.1109/HPCA61900.2025.00060