EMISSARY: Enhanced Miss Awareness Replacement Policy for L2 Instruction Caching

Nayana Prasad Nagendra*†
Arm
USA
nayanaprasad.nagendra@arm.com

Bhargav Reddy Godala*
Princeton University
USA
bgodala@princeton.edu

Ishita Chaturvedi
Princeton University
USA
ishitac@princeton.edu

Atmn Patel
Northwestern University
USA
atmn@u.northwestern.edu

Svilen Kanev
Google
USA
skanev@google.com

Tipp Moseley
Google
USA
tipp@google.com

Jared Stark
Intel Corporation
USA
jared.w.stark@intel.com

Gilles A. Pokam
Intel Corporation
USA
gilles.a.pokam@intel.com

Simone Campanoni
Northwestern University
USA
simone.campanoni@northwestern.edu

David I. August
Princeton University
USA
august@princeton.edu

ABSTRACT

For decades, architects have designed cache replacement policies to reduce cache misses. Since not all cache misses affect processor performance equally, researchers have also proposed cache replacement policies focused on reducing the total miss cost rather than the total miss count. However, all prior cost-aware replacement policies have been proposed specifically for data caching and are either inappropriate or unnecessarily complex for instruction caching. This paper presents EMISSARY, the first cost-aware cache replacement family of policies specifically designed for instruction caching. Observing that modern architectures entirely tolerate many instruction cache misses, EMISSARY resists evicting those cache lines whose misses cause costly decode starvations. In the context of a modern processor with fetch-directed instruction prefetching and other aggressive front-end features, EMISSARY applied to L2 cache instructions delivers an impressive 3.24% geomean speedup (up to 23.7%) and a geomean energy savings of 2.1% (up to 17.7%) when evaluated on widely used server applications with large code footprints. This speedup is 21.6% of the total speedup obtained by an unrealizable L2 cache with a zero-cycle miss latency for all capacity and conflict instruction misses.

*Authors have contributed equally.
†Work was performed while the author was at Princeton University.

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**.

KEYWORDS

Cache Microarchitecture, Cache Replacement Policy, Cost-Aware Replacement Policy, Instruction Caching

1 INTRODUCTION

Caches play a vital role in improving processor performance. For many decades, research has focused on improving processor performance by reducing cache misses [12, 14, 27, 29, 37, 38, 41, 45, 49, 52]. One important way to reduce cache misses for a given cache size, line size, and associativity is to improve the cache replacement policy. The classic LRU (least recently used) replacement policy exploits temporal locality by evicting the line least recently accessed [48, 56]. Bélády's OPT algorithm achieves the minimum number of misses through ideal cache replacement, but it is not realizable because it requires perfect knowledge of future references [15]. While theoretical in nature, OPT informs the design

of many practical cache replacement policies [27, 29, 41, 49]. Almost all previously proposed cache replacement policies use *only* reference information.

Architects have long recognized that not all cache misses have identical costs [30, 31, 35, 44, 50, 55]. Cost-aware cache replacement policies recognize this and attempt to increase performance even at the cost of increased cache misses. For modern architectures, many misses do not impact performance at all. For example, an aggressive out-of-order processor can entirely tolerate many first-level data cache (L1D) misses without any negative performance impact [39]. Likewise, many modern processors have decoupled front-ends with early fetch engines that can tolerate certain first-level instruction cache (L1I) misses without causing decode starvation, a state in which the head instruction of the queue feeding the decode stage is not yet available [24, 25, 51].

The optimal cost-aware replacement policy (CSOPT) is the perfect-knowledge cost-aware cache replacement algorithm [31]. Unfortunately, CSOPT, like OPT, is not realizable in practice. Prior research has produced realizable cost-aware data cache replacement policies [35, 44, 50, 55]. For example, the MLP-aware Linear (LIN) policy and related techniques prioritize lines that miss with lower memory-level parallelism (MLP) [50]. Other techniques consider load criticality as approximated by a variety of methods [35, 44, 55]. Since instruction fetch exhibits a different behavior, known cost-aware data cache replacement policies cannot be effectively applied to instruction caching.

This paper presents EMISSARY (Enhanced MISS-Awareness Replacement Policy) to address the lack of cost-aware cache replacement policies for instructions. Observing that modern architectures entirely tolerate many L1I misses, EMISSARY prioritizes instruction lines whose miss caused a decode starvation. These higher-priority lines are preserved in L2 upon eviction from L1I. By avoiding decode starvation, EMISSARY consistently improves performance and saves energy. EMISSARY has a minimal hardware footprint because it does not track history, coordinate with prefetchers, make predictions, or perform complex calculations.

A simple EMISSARY configuration on a baseline model with a fetch-directed instruction prefetcher (FDIP) front-end, yields a geomean speedup of 3.24% and proportional energy savings on 13 front-end-bound data center workloads. EMISSARY achieves 15% of the speedup obtainable by an unrealizable model with a zero-cycle miss latency for all capacity and conflict L2 instruction cache misses. Furthermore, these results are significant in the context of the aggressive front-end found in modern processors that are designed to tolerate instruction cache misses. The aggressive FDIP model used as the baseline for EMISSARY on its own provides an impressive 33.1% geomean speedup, creating a challenging environment for any further improvements. Prior work shows that achieving performance gains over FDIP's highly effective baseline is difficult [25, 26]. In fact, a non-realizable **perfect** prefetcher implemented over an FDIP baseline boosts performance by only 5.4% [25, 26].

Section 2 gives a high-level overview of the family of EMISSARY techniques. Since EMISSARY is a bimodal technique (misses are treated in one of two ways), this overview is contextualized with prior bimodal techniques. That section also highlights the value of persisting this bimodality over a line's **entire** lifetime in the cache,

a feature the authors believe to be an EMISSARY first. Section 3 quantifies the extent to which different instruction cache lines tend to have different decode starvation behaviors, an important characteristic contributing to EMISSARY's success. Using the observation that bimodal selection and treatment are orthogonal, Section 4 introduces a notation covering the space of EMISSARY techniques and related prior works. Using this notation, it describes the EMISSARY algorithm in detail. Section 5 outlines how and by how much these policies outperform prior work in both speed and energy. Since EMISSARY preserves instruction lines in the unified L2 cache, Section 6 explains how it moderates its use to leave sufficient space for data caching. Section 7 reviews the related work in-depth, and Section 8 concludes.

2 AN OVERVIEW TOUR OF EMISSARY

This paper argues for treating instructions bimodally (i.e., treating instruction cache lines whose misses impact performance differently from those that do not). Crafting a performant, low-complexity, and energy-efficient cost-aware bimodal policy requires choosing both a suitable *bimodal selection policy* (i.e., how to set the mode) and a *bimodal treatment policy* (i.e., how to treat the modes differently). This section provides an overview tour of these elements using Figure 1 to illustrate their impact on the performance, L2 instruction MPKI, commit-path decode rate, L2 data MPKI, and commit-path issue rate for the tomcat [8] benchmark. (Note: While §4 defines the notation in parentheses and §5 defines the experimental environment; they are not necessary to understand this section's discussion.)

All policies apply only to L2 instruction lines rather than L1I because L1I misses are generally tolerated well in modern processors with aggressive front-ends. Also, the longer average reuse intervals in large programs make the L2 more appropriate (§3). All policies may use decode starvation (i.e., the decode stage stalls waiting for the head instruction in the queue to become available), and the issue queue empty condition to make decisions.

In Figure 1, traditional LRU is labeled **MRU Insert:Always** as insertion is always into MRU (most recently used) position without any bimodality. Contrary to traditional LRU, a previously proposed mechanism, called LIP, is an *insertion policy* in an LRU *replacement policy* cache that always inserts lines in the LRU position instead [49]. The BIP prior work adds bimodality to LIP using a 1/32 probability random signal [49]. To add bimodality based on miss cost (i.e., decode starvation) to BIP, Figure 1 includes the **MRU Insert:Starvation Decode Only** policy. Unfortunately, as shown in Figure 1, **MRU Insert:Starvation Decode Only** performs slightly worse than LRU. While this result suggests that decode starvation is not a valuable bimodal selection signal, we observe that the problem is not the bimodality but that the bimodal treatment of a line is short-lived, as it takes only a few subsequent references to remove any MRU/LRU position differentiation.

To preserve the effect of the starvation signal longer, EMISSARY cache replacement policies are *persistently* bimodal. Instead of differentiating solely at insertion, EMISSARY replacement policies treat high-priority lines (e.g., those whose misses caused decode starvation) differently for their *entire lifetime in the cache*. This is partly done by not allowing insertions of low-priority lines (e.g.,
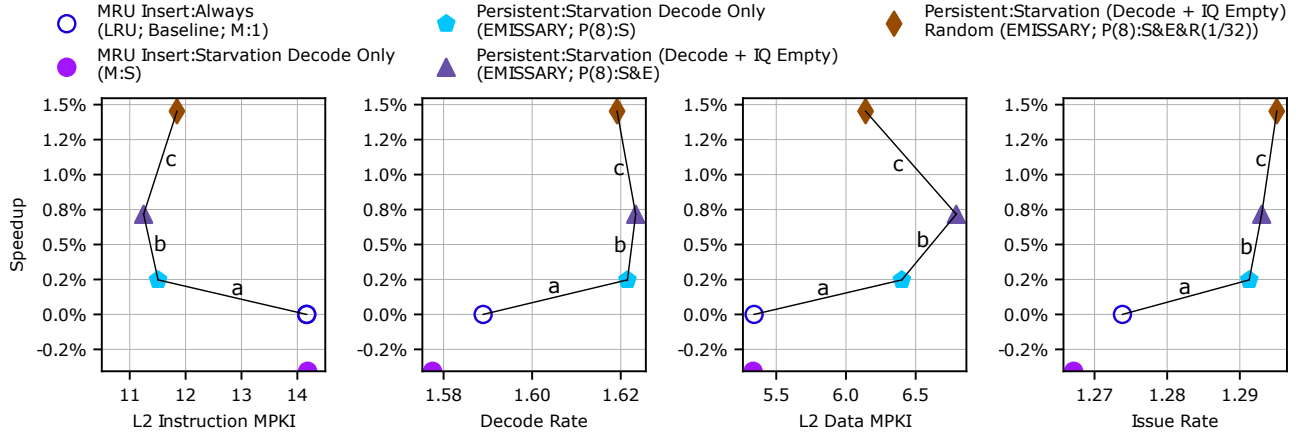
Figure 1: Speedup vs L2 Instruction MPKI, Decode Rate, L2 Data MPKI, and Issue Rate of various cache replacement policies for `tomcat` benchmark on a 1M 16-way L2 cache(with true LRU and no prefetchers).

from a miss without decode starvation) to evict any of up to N protected high-priority lines in the set. Only high-priority lines may be protected in this way, but not all high-priority lines are protected (i.e., when more than N lines in a set are high-priority). EMISSARY marks each line in the L1I cache with a priority bit, setting $P = 1$ if high-priority. This bit is preserved in L2 upon eviction from L1I and does not change as long as the line remains in either cache. The EMISSARY configurations in Figure 1 support up to eight protected lines ($N = 8$) per set in the L2 cache, leaving eight ways in the 16-way cache available for low-priority instruction and data lines. Having low-priority lines bypass the cache was not found to be effective, so all misses in EMISSARY result in an insertion. The performance and decode rate impact of persistence is highlighted by line **a** in Figure 1, which shows the power of persistence (i.e., **Persistent:Starvation Decode Only** over **MRU Insert:Always**).

Since available instructions in the instruction queue can hide decode starvation, requiring both the decode starvation and the empty issue queue signals to be true for bimodal selection will likely result in a more judicious use of persistence. Line **b** in Figure 1 confirms this by highlighting the difference between the **Persistent:Starvation (Decode + IQ Empty)** and **Persistent:Starvation Decode Only** EMISSARY policies.

Preserving high-priority lines over longer periods keeps them available for future accesses. Lines used only once, regardless of miss cost, do not benefit from persistence and instead, unnecessarily consume valuable cache resources. Thus, to filter out single-use but high-priority cache lines from being selected for bimodal treatment, the **Persistent: Starvation (Decode + IQ Empty) Random** EMISSARY policy only marks misses with starvation conditions as high-priority with random probability 1/32. This highly selective EMISSARY policy further improves performance, as evidenced by line **c**. Interestingly, with this policy, the instruction decode rate is less than **Persistent:Starvation (Decode + IQ Empty)** since not all high-priority instruction lines are selected for bimodal treatment. However, higher performance and an increased issue rate result from reduced data cache misses from the better allocation of L2 resources between instructions and data.

3 DECODE STARVATION BEHAVIOR

Instruction fetch is responsible for keeping the decode stage fed. If the processor could perfectly predict the target of every control-flow instruction, instruction fetch could issue all of its memory requests early enough to tolerate the latency to main memory without starving decode. Unfortunately, even the best branch predictors are not perfect. They are, however, quite good. Modern processor front-ends incorporate decoupled, aggressive fetch engines guided by excellent branch predictors, large BTBs, and pre-decoders [25]. Such front-ends accurately fetch several tens or even hundreds of instructions early. Instruction decode queues filled this way can often tolerate L1I misses before emptying and leading to decode starvation; this is especially true when an L1I miss leads to an L2 hit. From the perspective of cost-aware cache replacement policies, keeping lines with tolerated L1I misses in the cache has less utility than keeping lines whose misses cause decode starvation. The key to making this work involves differentiating tolerated L1I misses from those that cause starvation.

Branch mispredictions invalidate early fetch work, requiring a flush of the processor pipeline. Re-steering the front-end takes time, and more time is necessary for fetch to run far enough ahead of decode to fill the instruction decode queue enough to tolerate L1I misses. This concept suggests a cache replacement policy based on proximity to poorly predicted branch targets. The authors' early explorations in this direction considered cache replacement policies that were either too complex, ineffective, or both. Partially, this was because not all branch mispredictions lead to decode starvation. Often the lines necessary after re-steer are in L1I despite the branch mispredict. For example, this is the case for near-target branches in which the mispredicted path and the committed path share the same L1I cache lines. The mispredicted path fetch (or prefetch) acts as a prefetch in such a scenario.

Beyond branch prediction, many factors interact to determine whether or not an L1I miss will cause decode starvation. For example, a *stalled* decode cannot *starve*. Decode stalls occur when the processor back-end cannot accept more instructions. When
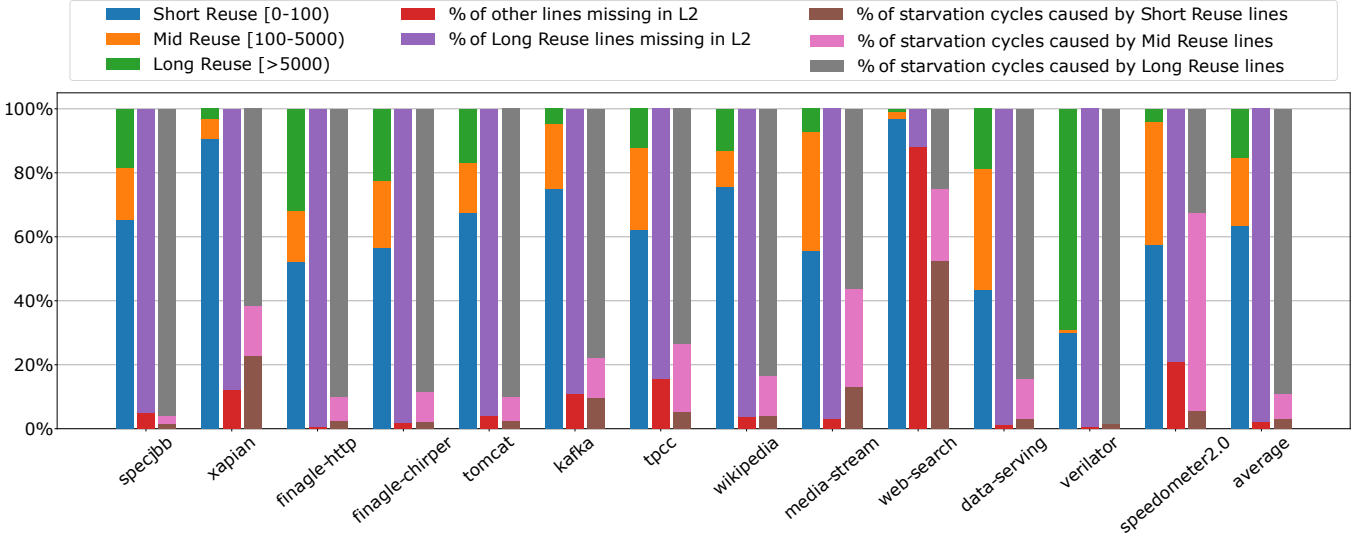
Figure 2: First bar: distribution of Short, Mid, and Long Reuse access lines. Second Bar: fraction of L2 Instruction Misses by Long Reuse lines. Third Bar: distribution of starvation cycles caused by Short, Mid, and Long Reuse lines.

decode stalls, it does not attempt to pull from the instruction queue, a necessary condition for starvation.

While predicting starvation by its component factors is hard, observing starvation during execution is easy. Existing signals assert when starvation occurs (likewise for when the issue queue is empty). Furthermore, when decode starvation occurs, the address for which the decode is waiting is also known. All of this information is known many cycles before the line that missed under these circumstances is inserted into the cache. Knowing this information in advance does not mean that it is necessarily of value to a cost-aware cache replacement policy.

An instruction fetch that causes starvation must be a miss in L1I cache. These L1I misses could be served either from L2 cache, L3 cache, or main memory. Lines served from the L2 cache introduce fewer starvation cycles than the ones served from more remote levels. The first bar in Figure 2 depicts the distribution of reuse distance based on observed cache line accesses in the committed path across the datacenter workloads used in this study. Reuse distance is measured as the number of unique lines accessed between two access to the same line. The same line accessed consecutively is not counted. Reuse distances are categorized into three buckets - Short [0-100), Mid [100-5000), and Long [>5000) Reuse. Short Reuse lines are likely to hit in L1I, Mid Reuse lines are likely to miss in L1I and hit in L2, and Long Reuse lines are likely to miss in L2. This predicted behavior is confirmed in the second bar showing the % of Long Reuse accesses that miss in L2. Overall, more than 90% of L2 misses are from Long Reuse lines.

The third bar in Figure 2 shows the interplay between these different categories of reuse lines and decode starvation. Interestingly, more than 90% of the starvation cycles are caused by Long Reuse lines, which account for less than 20% of all accesses. Thus, a small number of accesses contribute to the majority of starvation cycles – a property that can be utilized by a replacement policy. Consequently, EMISSARY at the L1I cache may have little value as

the lines that cause the majority of starvations are Long Reuse lines, lines which the L1I cannot realistically preserve. Since the majority of starvations are caused by L2 miss lines, in this work, EMISSARY policies are applied only to instructions cached in L2 cache. The L1I does play a role in that only L1I misses causing starvation are treated as high-priority. **A line's priority is only communicated to L2 cache once it is evicted from the L1I cache.** Instruction lines cached in L2 are then guided by the EMISSARY replacement policy. In this way, Long Reuse lines that have caused starvations are now cached in L2 cache for longer. EMISSARY uses 2 bits per line to record priority and for TPLRU access in the L1I and L2 caches. It is 1024 bits in L1I (32kB cache with 64B line size) and 32,768 bits in L2 (1MB cache with 64B line size), a little over 4kB total.

4 THE EMISSARY POLICIES

EMISSARY instruction cache replacement policies build on the lessons of §3, namely that only lines that caused starvations will likely cause further more starvations in the future. An EMISSARY cache leverages this by holding on to starvation-causing lines for longer, even if they have been less recently accessed than other starvation-free lines. Thus, EMISSARY cache replacement policies are bimodal. Bimodal techniques have two orthogonal aspects: *mode selection* and *mode treatment*. These aspects are discussed in this section.

4.1 Mode Selection

The two modes of a bimodal cache replacement policy are referred to as *high* and *low* priority, respectively. Table 1 shows the mode selection options for the space of realizable cache replacement algorithms referenced in this paper. These mode selection options are combined in Boolean equations. For example, S&R(1/32) requires a missed line to have caused starvation (S) during the miss AND to have been the lucky one of 32 chosen by pseudo-random selection

Notation	Description
1	Always High-Priority
0	Never High-Priority
R(r)	High-Priority with random probability r
S	High-Priority, line miss causes decode starvation
E	High-Priority, line miss occurs with an empty issue queue

Table 1: Mode Selection Options

Notation	Description
M	Insert High-Priority lines in MRU position, otherwise LRU
P(N)	Protect up to N MRU High-Priority lines/set from eviction

Table 2: Mode Treatment Options

Notation	Description
M:1	Always insert as MRU; Classic LRU; Baseline
M:0	Never insert as MRU (only as LRU); LRU Insertion Policy (LIP) [49]
M:R(r)	MRU insert with probability r; Bimodal Insertion Policy (BIP) [49]
M:S&E	MRU insert when starvation occurs and issue queue is empty
M:S&E&R(r)	MRU insert when starvation occurs, issue queue is empty and with probability r
P(N):R(r)	EMISSARY bimodal behavior only; high-priority lines selected with probability r
P(N):S	EMISSARY: high-priority on starvation
P(N):S&E	EMISSARY: high-priority on starvation and empty issue queue
P(N):S&E&R(r)	EMISSARY: high-priority on starvation, empty issue queue, and with probability r
SRRIP	Static re-reference interval prediction [29]
BRRIP	Bimodal re-reference interval prediction with probability (1/32)[29]
DRRIP	Dynamic re-reference interval prediction [29]
PDP	Static protective distance policy [20]
DCLIP	Dynamic Code Line Preservation [28]

Table 3: Cache replacement policies explored

(R(1/32)). EMISSARY policies all contain S in their mode selection equations. For all policies in this paper, the mode selection is determined once during cache line insertion. LRU can be thought of as a bimodal predictor degenerated to treat all inserted lines as high-priority by MRU position placement.

4.2 Mode Treatment

A meaningful bimodal cache replacement policy must treat lines differently based on the selected mode. Thus, the next aspect determines how high-priority lines are treated differently from low-priority lines. All realizable cache replacement policies discussed here use one of the two bimodal behaviors shown in Table 2.

In the first, M, bimodality comes from inserting high-priority lines into the cache's MRU position while placing low-priority lines into the cache's LRU position [49]. In the second, P(N), is the EMISSARY behavior. It is described by Algorithm 1. P(N) techniques do not act on priority at insertion. Instead, the priority is recorded as a priority bit (P) associated with each line that impacts eviction. High-priority lines have $P = 1$, while low-priority lines have $P = 0$. When inserting a line L into a P(N) cache, if the number of high-priority lines in the set is less than or equal to the maximum N; if it is then, the line L to be inserted (regardless of priority) replaces the LRU of the low-priority lines. Thus, the step in line 2 may increase the number of high-priority lines in the cache but cannot reduce it. For insertions where the number of high-priority lines in the set is greater than the maximum N, the cache evicts the LRU among the high-priority lines. Note that the number of high-priority lines is not reduced less than N at any point without a separate reset mechanism.

Algorithm 1 The EMISSARY Eviction Policy

1: **if** number of high-priority ($P = 1$) lines <= N **then**
2:     Evict the LRU among the low-priority ($P = 0$) lines
3: **else**
4:     Evict the LRU among high-priority lines
5: **end if**

The EMISSARY treatment option is orthogonal to the specific LRU algorithm used. For lines 2 and 4 of Algorithm 1, finding the LRU among the low-priority or the high-priority lines can be calculated precisely from a true LRU algorithm. With a pseudo-LRU (PLRU) algorithm, however, keeping separate PLRU's for low- and high-priority lines limits the imprecision. The PLRU-based EMISSARY uses the Tree Pseudo-LRU (TPLRU) algorithms with separate trees for low- and high-priority lines. When a high-priority line is accessed, only the high-priority tree is updated. Likewise, for a low-priority line and tree. For eviction, the appropriate tree is used to find the line to replace, skipping any lines that do not match the priority criteria. TPLRU requires $ways - 1$ bits per tree. Section 2 explored EMISSARY with true LRU. The evaluations use the TPLRU implementation.

4.3 Cache Replacement Policies

The top of Table 3 shows the prior work and proposed bimodal cache replacement policies used in this work. Each policy is a combination of a mode selection option (individually or by combination with a Boolean expression) and a mode treatment option described earlier. The rest of Table 3 lists other advanced policies used in the experimental comparison.

5 EXPERIMENTAL EXPLORATION

This section describes the simulation infrastructure, machine model, and benchmarks used to evaluate EMISSARY in various ways.

5.1 Simulation Infrastructure and Machine Model

This study uses gem5, a popular cycle-accurate simulator [17], running a detailed CPU model in FS (Full System) mode, with a full Operating System (OS). Gem5 supports a checkpointing mechanism that creates reusable snapshots for later restarts. For datacenter workloads, collecting gem5 checkpoints itself can be a significant bottleneck in evaluating microarchitectural changes. To reduce this, we used the QEMU [16] emulator and built a tool, FS_Lapidary, to create gem5-compatible snapshots. Snapshots consist of a dump of the physical memory, disk image, device state, CPU architectural state, and a checkpoint file compatible with gem5. Typically, checkpoint files can be only ported to another environment with the same system board configuration. We extended gem5 to support a hardware board configuration called virt_machine for snapshots created with QEMU. This enables the use of QEMU fast emulation features, like hardware acceleration with KVM [23]. We used an ARM64 system running Ubuntu 18.04 with Linux kernel 4.15, and an Apple M1 Mac mini to accelerate QEMU emulation.

As shown in Table 4, we used an Intel's Alderlake-like model for all experiments with the TPLRU config. The next line prefetcher (NLP) is enabled in the Adlerlake-like model for the L1D, L2, and L3 caches. The baseline for all experiments has FDIP enabled. L3 is an exclusive cache with DRRIP. L2 uses an SFL (Served From Last-level) bit to track each line's origin (i.e., L3 or memory). When a line with its SFL bit set is evicted from L2, it is placed at the MRU position in the L3. Each simulation includes a warm-up period of 5 million instructions from the resumed state followed by a measurement period of 100 million instructions in the detailed simulation model.

5.2 Decoupled Fetch Engine

We extended the fetch engine of the gem5 simulator to model the aggressive front-end found in modern processors with state-of-the-art FDIP prefetchers [52]. FDIP includes a Fetch Target Queue (FTQ) to decouple the fetch pipeline from the rest of the processor, enabling the fetch pipeline to run ahead of the rest of the processor pipeline [25, 52, 53]. The fetch pipeline, including the BTB and FTQ, has been modified to operate at the dynamic basic block granularity.

We modified the BTB such that each entry corresponds to a basic block. In addition to the target, entries contain details pertaining to the basic block - starting address, size, and the type of control-flow instruction that ends the basic block. This enabled the BTB to be indexed based on the branch target or the basic block's starting address instead of the branch PC. We used ARM binaries in this work. ARM's fixed-length encoding made it easier to model the BTB. Specifically, given the starting address and size of the basic block in terms of the number of instructions, it is straightforward to find the address of the control-transfer instruction that ends the respective block. This flexibility helped in reducing the otherwise necessary changes to the branch predictor. Variable-width instructions can be supported with additional hardware.

The branch predictor and BTB enqueue up to one basic block prediction per cycle to the FTQ. As in the BTB, each entry in the FTQ contains the starting address and size of the dynamic basic block. Naturally, enqueuing stalls on BTB misses. The next two fall-through lines are prefetched in the event of a BTB miss. As an

Field \ Model	Alderlake-like
ISA	Aarch64 (64-bit ARM)
Private L1I, L2D Caches	32kB (I), 64kB (D) NLP, 8-way 64B line size, 2 cycle hit TPLRU
Unified L2 Cache	1MB, 16-way, 64B line size 12 cycle hit, Inclusive NLP
Shared L3 Cache	2MB, 16-way, 64B line size 32 cycle hit latency Exclusive Victim Cache NLP DRRIP + SFL
Branch Predictor	TAGE, ITTAGE
BTB size	16K entries
Fetch Target Queue	24 entry 192-instruction
Fetch/Decode/ Issue/Commit	8 wide
ROB Entries	512
Issue/Load/Store Queue	240 / 128/ 72
Int/FP Registers	280 / 224

Table 4: Processor configurations

enhancement, the modeled front-end also includes a pre-decoder to update the BTB and minimize such enqueue stalls proactively. Branch re-steers flush the FTQ before resuming predictions on the corrected path. The FTQ along with basic-block level fetch enabled the front-end to run-ahead from the processor pipeline soon after every flush operation.

The FTQ enhancements allowed for the memory requests to be issued early. This work includes an FTQ size of 24 entries with a 192-instruction buffer. This offered the right balance by having sufficient starvation tolerance for hiding many L1I misses (see §3) while keeping the front-end from becoming overly aggressive in the presence of branch mispredictions. The extended run-ahead front-end requires the instruction buffer to be able to receive memory responses out-of-order. Overall, our optimized FDIP provides a ge-omean speedup of 33.1% over a no FDIP model for the 13 datacenter benchmarks as described in Section 5.3.

5.3 Benchmarks

To evaluate EMISSARY, we used 13 popular server applications with large code footprints from various benchmark suites: tomcat (Apache's implementation of Jakarta Servlet, Jakarta Expression Language, and WebSocket [8], from Dacapo benchmark suite [18]); kafka (Apache's distributed event streaming application used by companies like LinkedIn [6], from Dacapo benchmark suite); tpcc (On-Line Transaction Processing workload [9], from OLTP-Bench suite [19]); wikipedia (MediaWiki application on Wikipedia dataset [3], from OLTP-Bench suite); data-serving (Cassandra NoSQL database application [5], from Cloudsuite V4 [21]); media-streaming (Simulates video traffic, from Cloudsuite V4); web-search (Apache Solr search engine application [7], from Cloudsuite V4); xapian (a web-search application, Tailbench suite [32]); specjbb (A SPEC benchmark to test Java application features [1], from Tailbench); finagle-http (Twitter's HTTP
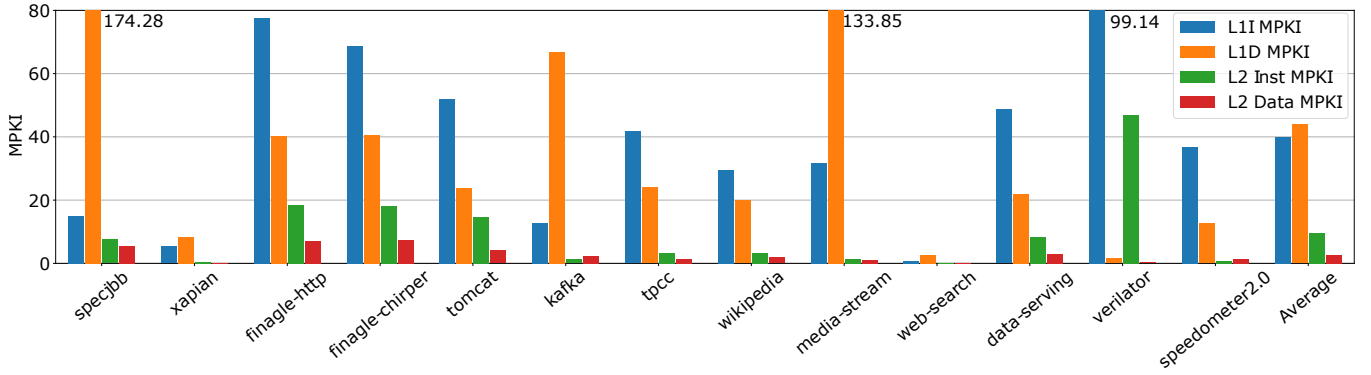
Figure 3: Average L1I, L1D, L2 Instruction, and L2 Data Cache MPKI of the benchmarks on the TPLRU + FDIP baseline.
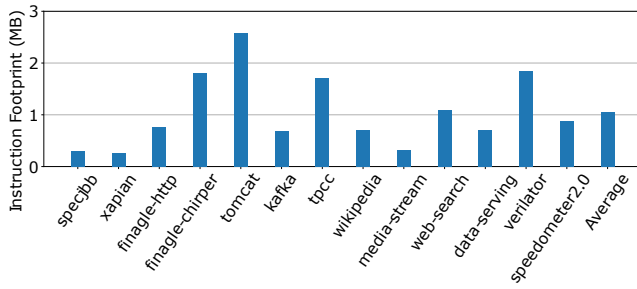

Figure 4: Instruction footprint of all benchmarks

server [4], from Renaissance [47]); finagle-chirper (A microblogging service by Twitter, from Renaissance); verilator [10] (simulates the RTL design of Rocket Chip [13] simulating quick sort code); and speedometer2.0 (a Java Script benchmark runs on a web browser benchmark which tests for the number of threads spawned in a minute [2]).

Benchmarks from the Tailbench suite are compiled using the default flags provided by the suite. verilator benchmark was built from the code provided and also optimized further using Facebook's BOLT [46] binary optimization tool. All benchmarks were built on the emulated environment described in Section 5.1. speedometer2.0 benchmark is simulated on a Chromium web browser.

Since all benchmarks except verilator are multithreaded, we scaled them to a single core ($N = 1$) for the evaluation on the simulator. To ensure that this simulation of a multithreaded workload is meaningful, we looked for performance trend differences between single core ($N = 1$) and multicore ($N > 1$) thread scalings on a real x86 Linux host machine using hardware performance monitoring counters. We examined the data at the feature level (e.g., branch misprediction rate), at the overall performance level, and according to the methodology outlined in [57]. We determined with confidence that the single-core scaling of these applications had the same workload characteristics as the $N = 4$ and $N = 8$ scalings. Software thread scheduling during simulation is handled by the Linux thread scheduler in Full System mode.

The benchmarks used exhibit various characteristics, as shown in Figure 3. specjbb, kafka, and media-stream have very high

L1D MPKI when compared to L1I MPKI. The average L1D MPKI is higher than the average L1I MPKI. media-stream and kafka benchmarks additionally have a higher L2 Data MPKI than L2 Instruction MPKI. However, the average L2 Instruction MPKI (9.63) is much larger than the Data counterpart (2.69). Figure 4 shows the instruction footprints of all benchmarks. Instruction footprints are measured based on the total number of unique cache lines accessed by the application during the simulation times the cache line size. tomcat has the highest footprint of 2.57 MB and xapian has the lowest footprint of 0.29 MB. The average footprint of the selected workloads is 1.05 MB. The chosen workloads were selected over the more traditional SPEC CPU workloads because they have larger code footprints and do not easily fit into the larger L2 caches of modern processors. Also, these benchmarks have been used in the most related works as well [33, 34].

5.4 Policy Selection and Parameterization

Section 4 outlines a large space of possible cache replacement policies. To narrow the design space to a small and meaningful set of policies, using an initial exploration, we first select a small set of desirable policy types and then find a reasonable set of configuration parameters for these policy types. The useful representative policy types chosen are the ones listed in Table 3. Ideally, we would like to find a single value of r and N that works well across all policies. Based on prior work, we expect the best r to be from 1/2 to 1/64 [49]. For a 16-way cache, useful values of N are from 2 to 14.

Table 5 shows the geometric mean speedup across all programs for a ranch of r and N values. The "#Best" row (or column) indicates the number of best configurations found in each column (or row). An r value of 1/32 consistently gives the best results in many cases. Prior work (M:R(r), BIP [49]) also suggests 1/32 or 1/64 as the value for r. Generally, benchmarks reach peak performance when N is 8, except verilator which continues to improve as N goes to 14. Hence, we set $N = 8$ and $r =$ 1/32 for the evaluation.

5.5 Performance

Figure 5 shows the speedup versus MPKI (odd rows) and speedup versus change in starvation cycles (Decode + IQ Empty) for committed instructions (even rows). For space reasons, tpcc is omitted as its L2 instruction MPKI is quite low. Values of N shown range

P(N)	S&E	R(1/2)	R(1/8)	R(1/16)	R(1/32)	R(1/64)	S&E&R(1/2)	S&E&R(1/8)	S&E&R(1/16)	S&E&R(1/32)	S&E&R(1/64)	# Best
2	-0.350	-1.511	-0.460	**0.216**	0.053	0.166	0.116	0.947	0.969	1.245	**1.548**	0
4	1.946	-0.433	0.736	**1.337**	1.171	0.897	1.621	1.767	2.025	**2.379**	1.634	0
6	1.995	0.7	1.406	1.571	**2.023**	1.813	1.656	2.261	2.486	**2.546**	1.906	5
8	1.294	0.579	0.906	1.112	**1.354**	1.193	**2.576**	2.301	2.419	2.490	2.005	2
10	-0.020	-0.995	-0.247	-0.287	-0.173	**-0.066**	0.125	1.47	**2.507**	3.15	2.018	1
12	-3.275	-4.926	-4.072	-3.751	-2.927	**-1.834**	-2.378	2.07	2.153	**3.063**	2.235	0
14	-7.698	-10.941	-8.710	-7.269	-5.472	**-3.628**	-5.039	-0.087	1.878	**3.241**	2.385	2
# Best	-	0	0	2	2	3	1	0	1	4	1	-

Table 5: Geomean speedup with respect to a LRU + FDIP baseline (Alderlake model) across all configurations for various values of r and N when run on a system with EMISSARY Policy at L2 Cache

from 0 to 14 by 2. An N of 0 is equivalent to the baseline. Lines connect P(N) to P($N + 2$) for each N from 0 to 12.

Generally, when MPKI is greater than 1.0, performance increases and starvations reduce as N increases to 8 (i.e., half of the 16-way cache is preserved for high-priority instruction lines). As N increases further, the performance gains decrease despite the consistent starvation reduction. This is because the L2 cache is shared by instructions and data. As more ways get used by high-priority instruction lines, resources are constrained to data lines, leading to more back-end stalls. See §5.8.

The results show that higher performance can come without much change in MPKI. This is the central observation of all cost-aware cache replacement policy proposals and this observation is confirmed for the EMISSARY techniques. Not all cache misses in modern out-of-order processors have the same cost. A significant portion of the misses can be tolerated without degrading processor performance. Similarly, a significant portion of the addresses that are latency-sensitive and cause decode starvations do so every time they are accessed. EMISSARY handles both of these categories very efficiently. It assigns higher priority to starvation-prone addresses, keeping them in the cache longer even if they are not accessed frequently. EMISSARY gives lower priority to latency-tolerant addresses, but it does cache them long enough to capture as much of their (belated) temporal locality.

The speedup and energy reduction of EMISSARY compared to other techniques over the TPLRU baseline is shown in Figure 7. Overall, P(8):S&E&R(1/32), the preferred EMISSARY configuration, yields a geomean speedup of 2.49% on all benchmarks, with gains as high as 11.7% (verilator) and as low as -1% (finagle-chirper). Unlike others, EMISSARY does not show any significant slowdowns.

Figure 7 also shows that EMISSARY policies outperform all of the others in terms of speedup and energy savings. The preferred configuration, P(8):S&E&R(1/32), performs consistently better than P(8):S&E. This is because the random filter tends to require lines to prove themselves with multiple starvations before being marked high-priority. This filters single reference lines very effectively, but it also filters single decode starvation lines just as well. This is important because high-priority reservations should be allocated to lines that starve with high probability and frequency, especially since an early single starvation is possible due to branch mispredictions and warm-up as new regions of code are executed.

Replacement policies such as SRRIP [29], BRRIP [29], and DRRIP [29] are designed to keep reused lines longer in the cache than the ones that are either used less frequently or have no reuse. Figure 3 shows that the hit rate at L2 is very high compared to the miss rate. In such a scenario, reused lines reach the highest priority state very quickly, and very often, this is the case at L2 in the datacenter workloads studied. When all ways in a cache set reach a high priority state, then the state of all lines is reset to a low priority state. In SRRIP policy, a newly inserted line would stay in the cache longer than in BRRIP policy, where only 3% of lines stay longer. BRRIP policy is detrimental when the newly inserted are evicted before they can be promoted to a higher priority state. A dynamic policy DRRIP is designed to reduce the negative effects of BRRIP and SRRIP. A dynamic policy dedicates a small sample (32 sets) to each policy and decides the winning policy based on the policy that contributes to fewer misses. Since the hit rate is much higher than the miss rate at L2, deciding the winner based on the miss rate is detrimental in the datacenter workloads studied. In a scenario where L2 capacity is limited EMISSARY identifies a small fraction of long reuse instruction lines that caused starvations in the processor pipeline and preserves them in the L2 cache longer.

5.6 Contextualizing EMISSARY's Benefits

EMISSARY's impact is significant given how often the modeled architectures tolerate L1I misses [25]. Prior work suggests that increasing the front-end performance of a modern processor with a *properly tuned* FDIP front-end is extremely difficult [25, 26]. These works show that FDIP alone outperforms the latest stand-alone prefetching policies such as EIP (one of the top prefetchers in IPC-1) by 2.2% [26]. The authors further claim that a non-realizable Perfect prefetcher provides just 5.4% of the performance gains [25, 26, 54]. The EIP-128KB prefetcher improves FDIP performance by 4.3% [25, 26, 54]. It does this with a significant hardware storage cost of 128KB. In contrast, EMISSARY provides up to 3.24% with only 4KB of additional storage.

To further contextualize EMISSARY's performance, we compare EMISSARY to a perfect and unattainable model with zero cycle miss latency for all capacity and conflict instruction cache misses in the L2. The aforementioned zero cycle miss latency model achieves a geomean speedup of 15% over the FDIP baseline. EMISSARY achieves 21.6% of this unrealizable gain with only 4KB of additional state.

Finally, we also compare EMISSARY to DCLIP, DRRIP, and PDP. These techniques achieve geomean speedups of −2.48%, −2.9%, and −3.36%, respectively, when implemented on top of the FDIP baseline for the workloads studied in this work.
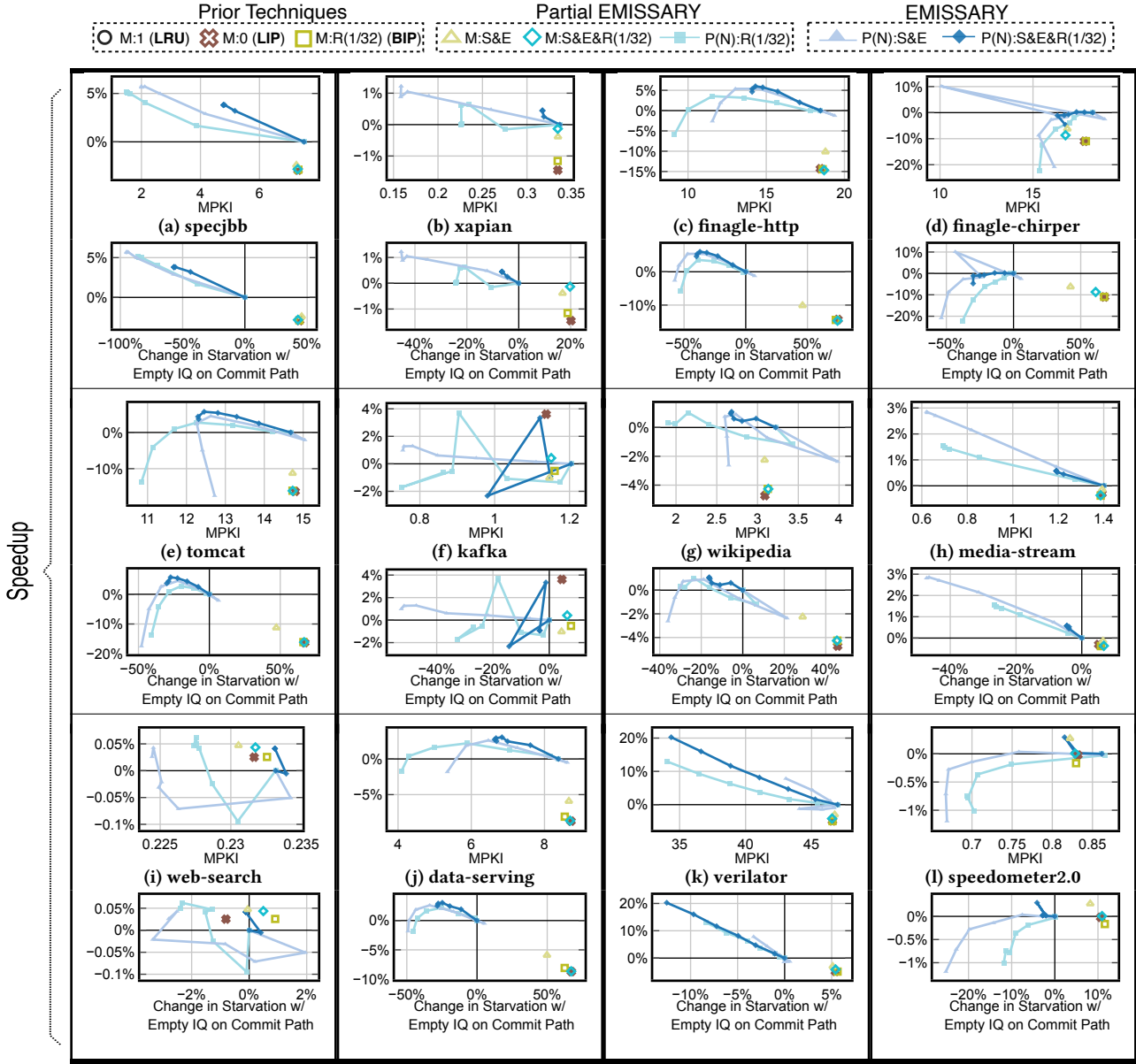
Figure 5: Speedup vs. L2 Instruction MPKI and Speedup vs. Change in Decode Starvation cycles when issue queue is empty for instructions on the committed path. P(N) techniques are shown as line segments with points corresponding to values of N from 0 to 14 in increments of 2. Lines connect P(N) to P(N + 2). TPLRU (N = 0) serves as the baseline.

5.7 Persistence, By Itself, Improves Hit Rate

Figure 5 shows that, in a majority of the programs, to a point, as N increases, L2 Instruction MPKI proportionately reduces. In other words, EMISSARY techniques not only reduce starvation but MPKI as well. Even as the number of ways available to a significant fraction of low-priority addresses is reduced, misses decline as well. This was observed previously with the BIP technique [49] as well. With the prevalence of single reference (or extremely long time between reference) addresses, dedicating fewer cache resources to such lines makes way for lines that would otherwise miss. In

this aspect, starvation acts as a filter, increasing the probability of isolating such lines by assigning them low-priority. Put another way, it helps reduce the extent to which these types of single reference lines can pollute the cache.

5.8 Impact on Back-end Stalls

EMISSARY's impact on commit path front-end and back-end stall cycles is shown in Figure 6. Specifically, it depicts the reduction in stall cycles of the preferred P(8):S&E&R(1/32) policy when compared to the TPLRU baseline. Across benchmarks, EMISSARY's
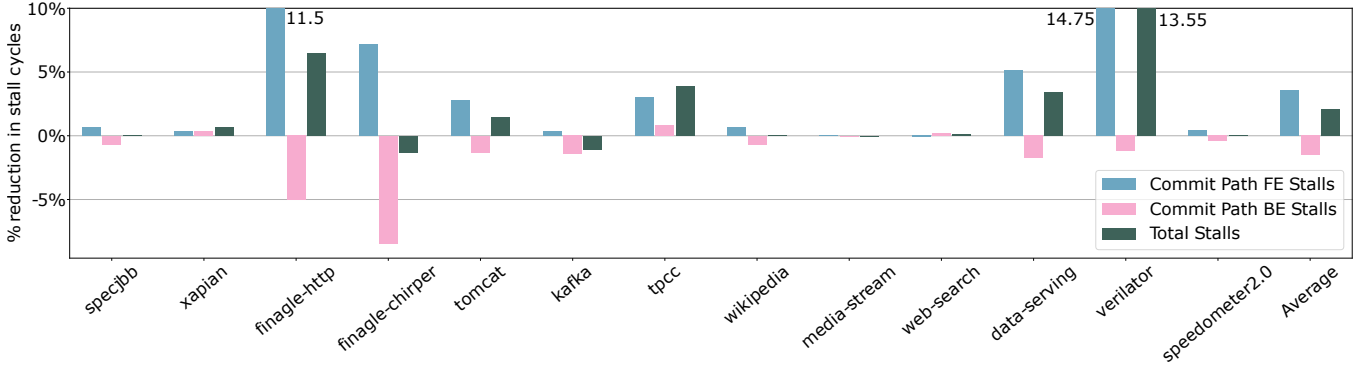
Figure 6: Reduction in various stall types of P(8):S&E&R(1/32) with respect to the TPLRU + FDIP baseline policy

impact on front-end stalls is more evident than its impact on back-end stalls. This is expected as EMISSARY is applied specifically to instruction lines. Interestingly, many benchmarks show an increase in back-end stalls but there is still an overall reduction in total stalls.

5.9 Energy Savings

We used McPAT [40] to model the energy consumption of different cache replacement policies explored. Fig. 7 shows energy savings for each benchmark and configuration. The energy savings are strongly correlated with the speedups achieved because of the relatively small amount of hardware added. In EMISSARY, there are only two bits added per cache line, one to mark the priority set once on insert and an additional one for TPLRU set on access. The EMISSARY P(8):S&E&R(1/32) configuration achieves a geomean reduction in the overall energy of 2.12% (up to 17.67%).

6 BALANCING DATA LINES

The EMISSARY policy advocated for in this work has $N = 8$ maximum ways reserved for high-priority instruction lines. In a 16-way L2 cache, this policy reserves up to half of the ways for instructions. As mentioned in §4, once a cache with an EMISSARY policy reaches N high-priority lines in a set, the number of high-priority lines can never be reduced. In this section, we study the number of sets in the L2 cache that get saturated by high-priority instruction lines (i.e., 8 lines in a set are dedicated to instructions) and propose methods to minimize their impact on caching data lines.

Figure 8 shows the distribution of the number of cache lines occupied by high-priority lines when using the P(8):S&E and P(8):S&E&R (1/32) policies among all sets in the L2 cache at the end of the simulation averaged over all programs. With P(8):S&E, `finagle-chirper`, `tomcat`, `tpcc`, and `verilator` saturate (reaches N) for all sets. Less than 25% of all sets observe saturation with the highly selective (and more desirable) P(8):S&E&R(1/32) policy. In simulations of 1B instructions, resetting all $P = 1$ bits every 128M instructions has a negligible impact on performance.

7 RELATED WORK

Cache replacement algorithms have been of interest to academia and industry for decades. This section describes several cache replacement policies related to EMISSARY.

7.1 Cost-Aware Cache Replacement Policies

Architects have long recognized that not all cache misses have the same performance cost. In light of this observation, several prior works have proposed cost-aware cache replacement policies that give deference to lines with higher miss costs while selecting a line to evict [35, 44, 50, 55]. All of these techniques target either data or shared caches. We are not aware of any proposed cost-aware replacement algorithms designed specifically for instruction caches.

CSOPT [31] is the ideal cost-aware cache replacement algorithm. It is essentially Bélády's OPT augmented with cost awareness. Like OPT, CSOPT is also unrealizable.

Among the realizable cost-aware policies, MLP-aware replacement policies [50] identify costly misses by observing memory-level parallelism. These techniques reduce the overall miss cost by attempting to reduce the number of isolated misses (i.e., misses that do not have MLP). The MLP LIN policy utilizes the miss status handling register (MSHR) as input to cost calculation hardware that uses fixed-point calculations. In contrast, there is no cost calculation in EMISSARY other than obtaining the already-existing starvation signal, which contributes to its energy efficiency. MLP LIN does not always outperform its baseline, LRU.

LACS [35] is a cost-aware cache replacement policy for last-level caches. It calculates the cost by counting the number of instructions that the processor can execute while the miss is being serviced. After insertion, LACS uses reference information to adjust the priority. Like EMISSARY, LACS requires two bits per line. However, LACS requires a history table, counters, and enhancements to the MSHRs. Such additional design complexity is not required in EMISSARY .

Critical Cache [55] proposes the use of a victim cache to give performance-critical loads a second chance. Critical loads are defined with heuristics related to the types of instructions executing near the load. Despite the additional hardware, a Critical Cache does not report performance gains over LRU.

7.2 Policies Challenging LRU

Ignoring cost awareness, many techniques have been designed to keep important lines in the cache, such as the Least Frequently Used (LFU) replacement policy, various cache bypassing techniques [36, 42], and others. In prior work, lines not meeting a threshold may still be inserted but in a way that has them closer to eviction [49].
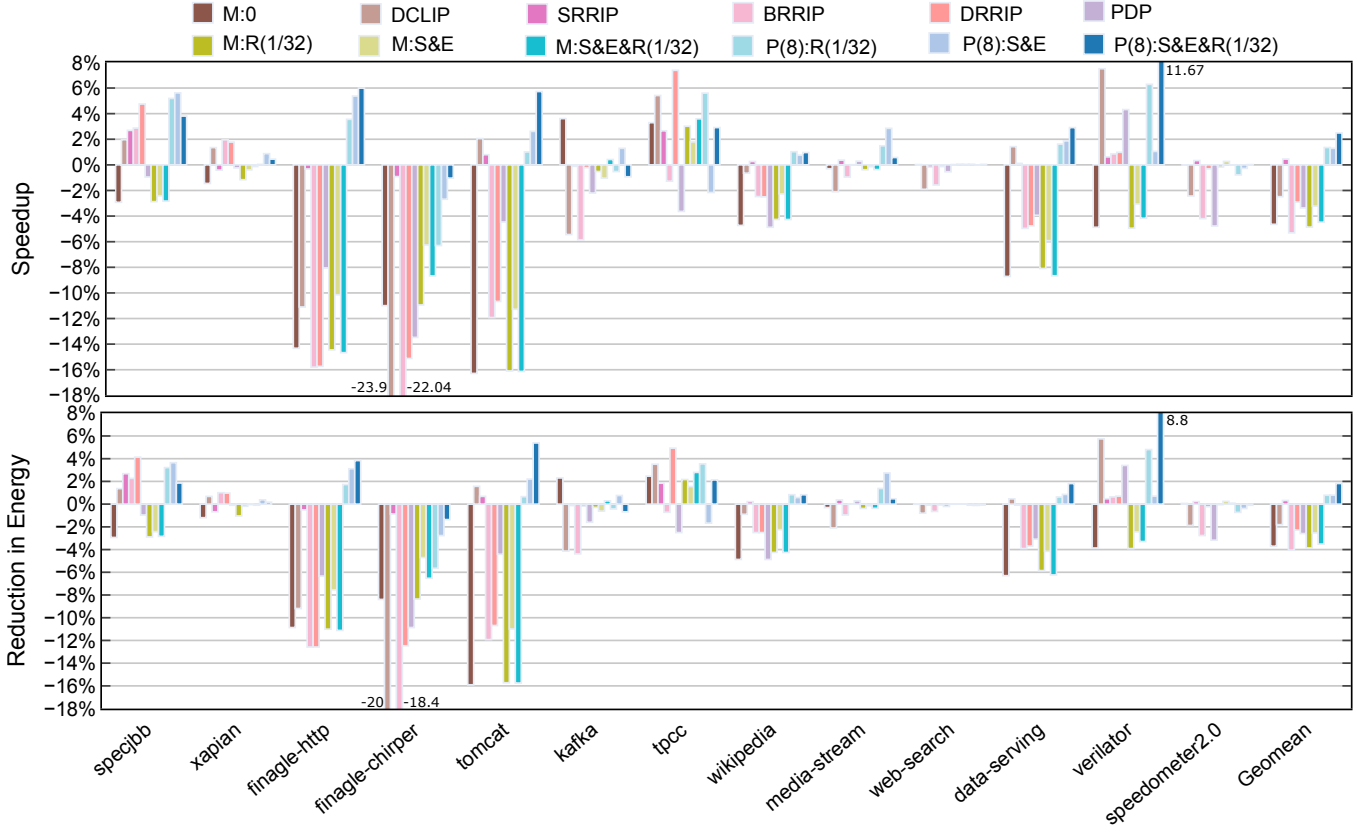
Figure 7: Speedup and Energy Reduction of a range of techniques relative to TPLRU + FDIP baseline policy
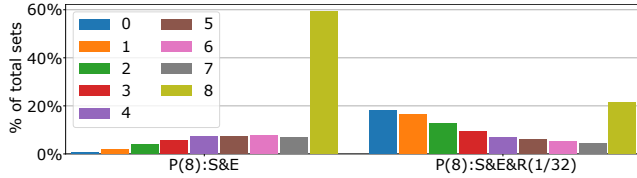

Figure 8: Distribution of high-priority lines across all sets when guided by P(8):S&E and P(8):S&E&R(1/32) policies, averaged across all benchmarks at the end of simulation.

These techniques proposed inserting the most recently used lines into the LRU position to improve performance even when working sets are thrashed. The policy was further improved by a schema that only inserted new lines into the MRU slot with a low, random probability. The final modification was a dynamic method, which chose between traditional LRU and the aforementioned random insertion policy, depending on which resulted in fewer misses for a given workload. The parallel between this cache replacement policy and EMISSARY is that both decide to prefer some lines over others in cache replacement based on a factor other than recency. Of course, EMISSARY prefers lines that are cost-effective to prefer in addition to randomness. The key difference, though, is that EMISSARY sets the preference with a relevant cost signal in addition to randomness. Thus, it not only benefits from requiring lines to prove themselves as commonly used before being deemed important to keep in the

cache but also from making this determination based on a specific cost factor, consistently effective for many workloads.

GHRP [11] is an instruction cache replacement policy focused on minimizing the number of misses by identifying dead blocks, which is orthogonal to ours. It uses the access history to identify if a block needs to be bypassed upon insertion. Else, a dead-block predictor is used to select the candidate for eviction. GHRP's dead-block prediction mechanism could be combined with EMISSARY to identify the low-priority dead blocks for eviction. Doing so might further improve the performance of EMISSARY .

Ripple [34] is a software-only profile-guided technique to improve the performance of 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. Unlike EMISSARY, Ripple identifies lines that are no longer required and ensures they do not waste cache space. Ripple is complementary to EMISSARY and could be used alongside EMISSARY to get the best of both techniques.

In [28], the authors propose CLIP, a cache replacement policy for instruction lines. By modifying the re-reference predictions of instruction and data lines separately, they dynamically prioritize instructions in a cache when the instructions cause L2 cache contention. Unlike EMISSARY, CLIP prioritizes all instruction lines blindly, without confirming that a future miss would cause front-end stalls. Furthermore, by restricting the number of ways used for

instruction lines, EMISSARY prevents contention between data and instruction in the L2.

7.3 Special Caching Solutions

The high-priority lines identified by EMISSARY could be due to branches with a high misprediction rate. A special cache has been proposed (MRC [43]) to mitigate this cost. It has also been adapted in commercial processors as a Misprediction Recovery Buffer (MRB) [22]. These solutions were specifically designed to mitigate branch recovery cost on misprediction. These structures need to be small to keep up with the timing constraints as they sit on a critical path. These solutions are applicable when the reuse distance is short but in large code footprints reuse distance is long which cannot be fit into small structures. An MRC/MRB and EMISSARY can likely be used together with success as they address orthogonal problems (short vs. long reuse intervals).

8 CONCLUSION

This paper presents EMISSARY, a new family of cost-aware cache-replacement policies for instructions that are well-suited for the L2 cache. Observing that modern architectures completely tolerate many instruction cache misses, EMISSARY prioritizes, with persistence, inserted lines whose misses cause decode starvation over those whose misses did not. Without the need to track history, coordinate with prefetchers, make predictions, or perform complex calculations, EMISSARY consistently improves performance and saves energy while remaining simple to implement. This allows EMISSARY to achieve a geomean performance gain of 3.24% (up to 23.7%), and a geomean energy savings of 2.12% (up to 17.7%) over TPLRU on top of a state-of-the-art FDIP prefetcher to model the aggressive front-ends found in modern processors. This speedup is 21.6% of the total speedup obtained by an unrealizable model with an ideal L2 instruction cache, with a mere 4KB hardware budget.

ACKNOWLEDGMENTS

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 cassandra. http://cassandra.apache.org/
[6] 2023. Apache kafka. https://kafka.apache.org/
[7] 2023. Apache Solr. https://solr.apache.org/
[8] 2023. Apache tomcat. https://tomcat.apache.org/
[9] 2023. TPC-C. http://www.tpc.org/tpcc/
[10] 2023. Verilator. https://www.veripool.org/wiki/verilator
[11] 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. https://doi.org/10.1109/ISCA.2018.00050

[12] Erik R Altman, Vinod K Agarwal, and Guang R Gao. 1993. A Novel Methodology Using Genetic Algorithms for the Design of Caches and Cache Replacement Policy.. In *ICGA*. 392–399. https://doi.org/10.1109/WorldS450073.2020.9210347
[13] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[14] Grant Ayers, Nayana P. 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 *Computer Architecture (ISCA)*. https://doi.org/10.1145/3307650.3322234
[15] L.A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. In *IBM Systems journal, pages 78–101*. https://doi.org/10.1147/sj.52.0078
[16] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.
[17] 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* (2011). https://doi.org/10.1145/2024716.2024718
[18] 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
[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (dec 2013), 277–288. https://doi.org/10.14778/2732240.2732246
[20] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *2012 45Th annual IEEE/ACM international symposium on microarchitecture*. IEEE, 389–400.
[21] 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. *SIGPLAN Not.* 47, 4 (mar 2012), 37–48. https://doi.org/10.1145/2248487.2150982
[22] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, et al. 2020. Evolution of the samsung exynos cpu microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 40–51.
[23] Irfan Habib. 2008. Virtualization with KVM. *Linux J.* 2008, 166, Article 8 (feb 2008).
[24] Glenn J Hinton and Robert M Riches Jr. 1995. Instruction fetch unit with early instruction fetch mechanism. https://patents.google.com/patent/US5423014A/en US Patent 5,423,014.
[25] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2020. Rebasing Instruction Prefetching: An Industry Perspective. *IEEE Computer Architecture Letters* 19, 2 (2020), 147–150. https://doi.org/10.1109/LCA.2020.3035068
[26] 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)*. 172–182. https://doi.org/10.1109/ISPASS51385.2021.00034
[27] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *43rd International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2016.17
[28] 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)*. 343–353. https://doi.org/10.1109/HPCA.2015.7056045
[29] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *37th International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/1815961.1815971

[30] Jaeheon Jeong and Michel Dubois. 2003. Cost-sensitive cache replacement algorithms. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 327–337. https://doi.org/10.1109/HPCA.2003.1183550

[31] Jaeheon Jeong and Michel Dubois. 2006. Cache replacement algorithms with nonuniform miss costs. *IEEE Trans. Comput.* 55, 4 (2006), 353–365. https://doi.org/10.1109/TC.2006.50

[32] Harshad Kasture and Daniel Sanchez. 2016. TailBench: A benchmark suite and evaluation methodology for latency-critical applications. In *Workload Characterization (IISWC)*.

[33] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 816–829.

[34] 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)*. 734–747. https://doi.org/10.1109/ISCA52012.2021.00063

[35] Mazen Kharbutli and Rami Sheikh. 2014. LACS: A locality-aware cost-sensitive cache replacement algorithm. *IEEE Trans. Comput.* 63, 8 (2014), 1975–1987. https://doi.org/10.1109/TC.2013.61

[36] Mazen Kharbutli and Yan Solihin. 2008. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput.* 57, 4 (2008), 433–447. https://doi.org/10.1109/TC.2007.70816

[37] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the front-end bottleneck with shotgun. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3173162.3173178

[38] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A metadata-free architecture for control flow delivery. In *High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2017.53

[39] Alvin R Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. 2002. A large, fast instruction window for tolerating cache misses. *ACM SIGARCH Computer Architecture News* 30, 2 (2002), 59–70. https://doi.org/10.1145/545214.545223

[40] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480. https://doi.org/10.1145/1669112.1669172

[41] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwan Ahn. 2020. An Imitation Learning Approach for Cache Replacement. In *arXiv:2006.16239*. https://arxiv.org/abs/2006.16239

[42] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–33. https://doi.org/10.1145/2893356

[43] Ashwini K Nanda, James O Bondi, and Simonjit Dutta. 1998. The misprediction recovery cache. *International journal of parallel programming* 26, 4 (1998), 383–415.

[44] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. 2018. Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 96–109. https://doi.org/10.1109/ISCA.2018.00019

[45] Mark Palmer and Stanley B Zdonik. 1991. *Fido: A cache that learns to fetch.* Brown University, Department of Computer Science. http://vldb.org/conf/1991/P255.PDF

[46] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 2–14.

[47] 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

[48] Thomas Roberts Puzak. 1986. ANALYSIS OF CACHE REPLACEMENT-ALGORITHMS. (1986). https://scholarworks.umass.edu/dissertations/AAI8509594/

[49] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *34th International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/1273440.1250709

[50] Moinuddin K Qureshi, Daniel N Lynch, Onur Mutlu, and Yale N Patt. 2006. A case for MLP-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE, 167–178. https://doi.org/10.1109/ISCA.2006.5

[51] Glenn Reinman, Todd Austin, and Brad Calder. 1999. A Scalable Front-End Architecture for Fast Instruction Delivery. *SIGARCH Comput. Archit. News* 27, 2 (May 1999), 234–245. https://doi.org/10.1145/307338.300999

[52] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.1999.809439

[53] Glenn Reinman, Brad Calder, and Todd Austin. 2001. Optimizations enabled by a decoupled front-end architecture. *Computers, IEEE Transactions on* 50 (05 2001), 338 – 355. https://doi.org/10.1109/12.919279

[54] Alberto Ros and Alexandra Jimborean. 2020. The Entangling Instruction Prefetcher. *IEEE Computer Architecture Letters* 19, 2 (2020), 84–87. https://doi.org/10.1109/LCA.2020.3002947

[55] Srikanth T. Srinivasan, Roy Dz-ching Ju, Alvin R. Lebeck, and Chris Wilkerson. 2001. Locality vs. criticality. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 132–143. https://doi.org/10.1145/379240.379258

[56] Wing Shing Wong and Robert J. T. Morris. 1988. Benchmark synthesis using the LRU cache hit function. *IEEE Trans. Comput.* 37, 6 (1988), 637–645. https://doi.org/10.1109/12.2202

[57] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 35–44.