Twig: Profile-Guided BTB Prefetching for Data Center Applications

Tanvir Ahmed Khan
takh@umich.edu
University of Michigan, USA

Nathan Brown
nlbrow@umich.edu
University of Michigan, USA

Akshitha Sriraman
akshitha@umich.edu
University of Michigan, USA

Niranjan Soundararajan
niranjan.k.soundararajan@intel.com
Intel Labs, India

Rakesh Kumar
rakesh.kumar@ntnu.no
Norwegian University of Science and Technology, Norway

Joseph Devietti
devietti@cis.upenn.edu
University of Pennsylvania, USA

Sreenivas Subramoney
sreenivas.subramoney@intel.com
Intel Labs, India

Gilles Pokam
gilles.a.pokam@intel.com
Intel Labs, USA

Heiner Litz
hlitz@ucsc.edu
University of California, Santa Cruz, USA

Baris Kasikci
barisk@umich.edu
University of Michigan, USA

ABSTRACT

Modern data center applications have deep software stacks, with instruction footprints that are orders of magnitude larger than typical instruction cache (I-cache) sizes. To efficiently prefetch instructions into the I-cache despite large application footprints, modern server-class processors implement a decoupled frontend with Fetch Directed Instruction Prefetching (FDIP). In this work, we first characterize the limitations of a decoupled frontend processor with FDIP and find that FDIP suffers from significant Branch Target Buffer (BTB) misses. We also find that existing techniques (*e.g.*, stream prefetchers and predecoders) are unable to mitigate these misses, as they rely on an incomplete understanding of a program's branching behavior.

To address the shortcomings of existing BTB prefetching techniques, we propose *Twig*, a novel profile-guided BTB prefetching mechanism. *Twig* analyzes a production binary's execution profile to identify critical BTB misses and inject BTB prefetch instructions into code. Additionally, *Twig* coalesces multiple non-contiguous BTB prefetches to improve the BTB's locality. *Twig* exposes these techniques via new BTB prefetch instructions. Since *Twig* prefetches BTB entries without modifying the underlying BTB organization, it is easy to adopt in modern processors. We study *Twig*'s behavior across nine widely-used data center applications, and demonstrate that it achieves an average 20.86% (up to 145%) performance speedup over a baseline 8K-entry BTB, outperforming the state-of-the-art BTB prefetch mechanism by 19.82% (on average).

CCS CONCEPTS

• **Computer systems organization** → *Pipeline computing*.

KEYWORDS

Prefetching, frontend stalls, branch target buffer, data center

ACM Reference Format:
Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3466752.3480124

1 INTRODUCTION

Modern data center applications have deep software stacks that are composed of complex application logic [56], diverse libraries [38], and numerous kernel modules [16, 45, 46]. Such deep stacks result in multi-megabyte instruction footprints [16, 38, 59] that easily exhaust typical on-chip cache structures which are smaller than hundred kilobytes [14]. As a result, data center applications suffer from significant frontend stalls, when the processor frontend is unable to supply instructions to the processor backend. Such frontend stalls significantly hurt the Total Cost of Operation of a data center, as even single-digit performance improvements of frontend stalls can save millions of dollars and meaningfully reduce the global carbon footprint [79].

Processor architects attempt to address this overwhelming frontend stall problem by proposing numerous instruction prefetching mechanisms [25, 26, 39, 44–46, 60, 69, 76]. Fetch Directed Instruction Prefetching (FDIP) [69] is one such mechanism that is pervasively explored in academia [42, 45, 46] and industry [35, 36]. Between the branch prediction unit and the instruction fetch engine, FDIP introduces a queue containing the addresses of I-cache lines that will be accessed in the future [68]. FDIP prefetches I-cache lines based on the queue contents to avoid instruction fetch stalls. FDIP allows the branch prediction unit and the instruction fetch engine to operate independently with high efficiency. Prior work [35] has shown that FDIP provides comparable performance to aggressive I-cache prefetchers [54, 70, 74] used in recent instruction prefetching championships. Due to its success, FDIP has been widely implemented in modern processors [29, 61, 72, 80].

Given that data center applications still continue to face the frontend stall problem, we first ask the question: What limits FDIP from eliminating all frontend stalls? To this end, we comprehensively study FDIP in the context of frontend-bound data center applications and show that FDIP still falls significantly short of an ideal I-cache (by 24% on average). We also find that FDIP's effectiveness primarily depends on the efficacy of the Branch Target Buffer (BTB); therefore, the large number of BTB misses, which is typical for data center applications, hurts FDIP's effectiveness. We then investigate the reasons behind the large number of BTB misses for data center applications. We find that these applications contain a large number of unique branch instructions that cannot fit into moderately-sized BTBs. Furthermore, we show that the state-of-the-art BTB prefetching techniques, such as Shotgun [45] and Confluence [40], suffer from limited prefetching coverage and accuracy while introducing significant hardware modifications. For this reason, they have not been adopted in modern data center processors [16, 41].

In this paper, we propose *Twig*, a novel profile-guided BTB prefetching mechanism for data center applications. Unlike prior techniques [40, 45], *Twig* does not require any modifications to the typical BTB organization. Instead, *Twig* introduces a new BTB prefetching instruction that is directly injected into the program binary at link time. By inserting BTB prefetch instructions in software, *Twig* leverages the rich execution information available in a program profile, when collected using performance counters in modern data center environments [16, 22, 38, 58].

Twig introduces two key techniques: *software BTB prefetching* and *BTB prefetch coalescing*.

Software BTB prefetching. A BTB entry is composed of a branch instruction address and a corresponding branch target address. To prefetch a BTB entry, the processor has to decode the branch target of a given branch instruction. However, the branch instruction itself may not be present in the I-cache, rendering BTB prefetching impossible. *Twig* addresses this challenge by introducing an explicit *prefetch* instruction to prefetch BTB entries in advance, without bringing the required instructions into the I-cache. This *prefetch* instruction prefetches branch instruction address and target into the BTB. Unlike pure hardware techniques that rely on limited past run-time information [40, 45], *Twig* determines which branch instructions cause frequent BTB misses based on profiles collected from the entire program execution. *Twig*'s prefetch instruction takes as operands the address of the branch instruction and the address of the corresponding target instruction. *Twig* then ensures that the corresponding entry is inserted into the BTB even if the branch instruction is not in the I-cache.

Twig further leverages production execution profiles to identify program locations that can predict the future execution of a BTB-miss inducing branch instruction with high accuracy and timeliness. *Twig* then inserts prefetch instructions into these locations.

BTB prefetch coalescing. Inserting many BTB prefetch instructions with multiple parameters can increase the static and dynamic instruction footprint. To mitigate this code bloat, *Twig* proposes *BTB prefetch coalescing*, where multiple BTB entries are prefetched with a single instruction. *Twig* analyzes the program profile to identify consecutively-executed branches that incur repetitive BTB misses. Consequently, *Twig* uses the coalesced prefetching instruction to prefetch the BTB entries of all of these branch instructions simultaneously.

We evaluate *Twig* in the context of nine data center applications that suffer from frequent frontend stalls. *Twig* achieves an average 20.86% (2%-145%) speedup over a baseline 8K-entry BTB across all nine applications, while reducing 65.4% of all BTB misses. Compared to the state-of-the-art BTB prefetcher [45], *Twig* achieves an average 19.82% (up to 139.8%) greater speedup, while covering 57.4% more BTB misses. *Twig*'s average static and dynamic instruction increase overhead is 6% and 3% respectively.

In summary, we contribute:

- A detailed characterization of a decoupled frontend with FDIP that shows that a large number of BTB misses hurt FDIP's effectiveness.
- *Software BTB prefetching:* A technique to prefetch BTB entries that improves the decoupled frontend's performance by avoiding costly BTB misses.
- *BTB prefetch coalescing:* A profile-guided mechanism to coalesce multiple BTB prefetch operations that reduces prefetch instructions' static and dynamic overhead.
- An evaluation of *Twig* in the context of nine data center applications, showing its effectiveness in reducing BTB misses and achieving significant performance benefit.

2 LIMITATIONS OF PRIOR I-CACHE & BTB PREFETCHING TECHNIQUES

In this section, we comprehensively characterize existing I-cache and BTB prefetching mechanisms to understand why data center applications continue to suffer from frontend stalls. We first analyze FDIP [69], the state-of-the-art prefetching technique in processors with a decoupled frontend. We measure the unrealized performance potential of FDIP and find that its performance is mainly limited by BTB misses. We then analyze Shotgun [45] and Confluence [40], two recently proposed techniques that introduce BTB prefetching on top of FDIP. While these techniques reduce BTB misses for some applications, they fail to eliminate BTB misses that occur due to complex branch patterns faced by data center applications.

We characterize nine popular real-world data center applications [41] that face significant frontend stalls. In Fig. 1, we use Intel's Top-Down methodology [88] to show that these applications spend 24%-78% of the processor pipeline slots in waiting for the frontend to return. Two applications, `finagle-chirper` (a microblogging service) and `finagle-http` (an HTTP server) are from the

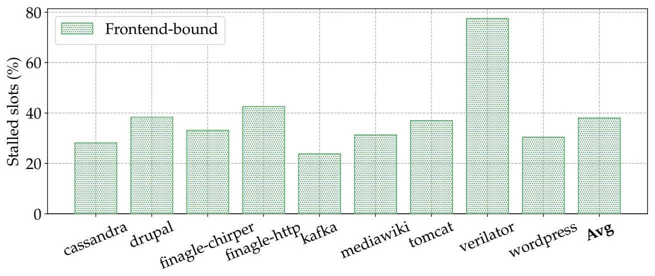


Figure 1: Many popular data center applications waste a large portion of their pipeline slots due to "frontend-bound" stalls [34], measured using the Top-down methodology [88].

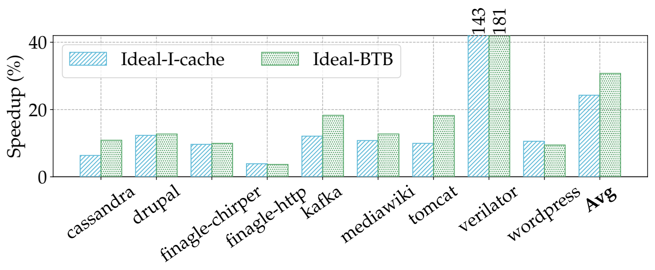


Figure 2: Limit study of FDIP: an ideal I-cache achieves an average 24% speedup, while an ideal BTB provides an average 31% speedup over the FDIP baseline.

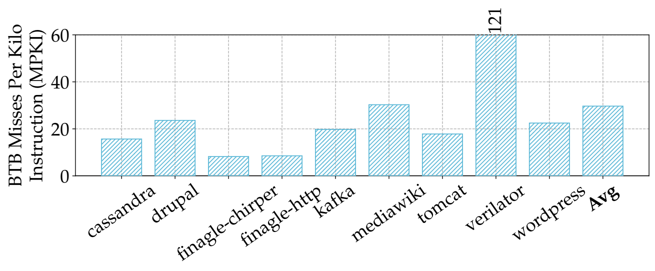


Figure 3: BTB Misses Per Kilo Instructions (MPKI) for nine data center applications: these applications experience an average BTB MPKI of 29.7 (8-121).

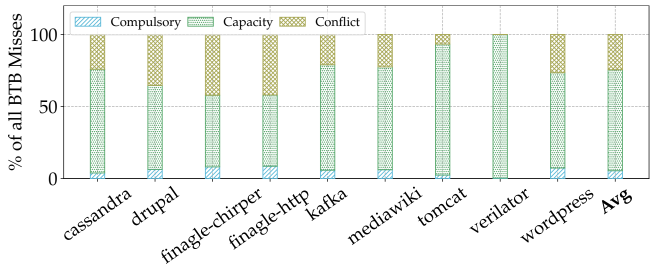


Figure 4: Breakdown of all BTB misses using 3C miss classification [33]: data center applications suffer BTB misses due to both capacity and conflict issues.

Java Renaissance [66] benchmark suite and use Twitter Finagle [7] which is a Remote Procedure Call (RPC) library. Three applications, `kafka` [84] (Apache stream-processing framework used by companies like Uber, Linkedin, and Airbnb [3]), `tomcat` [4] (open-source Java web server), and `cassandra` [2] (NoSQL DBMS used by companies like Uber, Netflix, and Grubhub [86]) are from the Java DaCapo [17] benchmark suite. We also study three HHVM [10, 55] applications (`drupal`, `wordpress`, and `mediawiki`) from Facebook's OSS-performance [9] benchmark suite. `verilator` [8] is a tool used by companies like Intel and ARM to evaluate custom hardware designs [85]. We detail our experimental setup, trace collection methodology, and simulation parameters in §4.

2.1 What stops FDIP from eliminating all frontend stalls?

Recent processor designs [29, 61, 72, 80] have adopted decoupled frontends with FDIP to reduce costly frontend stalls. Given FDIP's widespread adoption [35, 36], we ask the question: Does FDIP achieve performance comparable to an ideal/perfect frontend where pipeline slots are not stalled in the frontend? To this end, we analyze FDIP's limitations, characterizing why FDIP falls short for data center applications. Additionally, we determine how to address FDIP's limitations.

We perform two limit studies, measuring the Instructions Per Cycle (IPC) metric of nine data center applications running on an FDIP-enabled processor. In the first study, we analyze FDIP with an ideal I-cache (*i.e.*, every I-cache access is a hit), and in the second study, we analyze FDIP with an ideal BTB (*i.e.*, every branch target lookup is a hit). We assume a 75KB 8K-entry BTB and a 32KB I-cache. Fig. 2 shows an average IPC improvement of 24% with an ideal I-cache and a 31% improvement with an ideal BTB. FDIP with an ideal BTB offers greater performance benefits since (1) it eliminates almost all I-cache misses (due to FDIP prefetching) and (2) it reduces branch resteers (*i.e.*, pipeline flushes) triggered by BTB misses. Hence, we conclude that reducing BTB misses is critical to mitigating frontend stalls. Next, we investigate why data center applications suffer from poor BTB locality even with a relatively large, 75KB 8K-entry BTB.

2.2 Why is a large BTB insufficient for data center applications?

As an ideal BTB significantly improves FDIP's performance, we examine how we can improve the performance of the 75KB 8K-entry BTB that is implemented in today's FDIP-enabled processors.

Fig. 3 shows the BTB Misses Per Kilo Instructions (MPKI) across all nine data center applications. While measuring BTB MPKI, we only consider real BTB misses caused by direct branch instructions, *i.e.*, unconditional jumps, calls, and conditional jumps. We do not include non-control flow instructions or branch instructions where the branch target that the BTB returns is different from the actually taken branch target (*e.g.*, branch target changed due to just-in-time code compilation).

As shown in Fig. 3, data center applications experience MPKIs in the range of 8-121 (29.7 on average). To understand the reason behind significant BTB misses, in Fig. 4, we categorize whether these misses are *compulsory*, *capacity*, or *conflict* misses, *i.e.*, the 3C

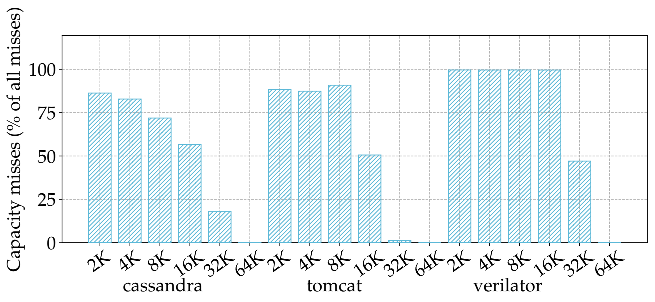


Figure 5: Percentage of capacity misses as BTB size increases from 2K to 64K entries: data center applications require large BTB with at least 32K entries to avoid all capacity misses. For brevity, we show results for only 3 applications, but the behavior is similar across all applications.

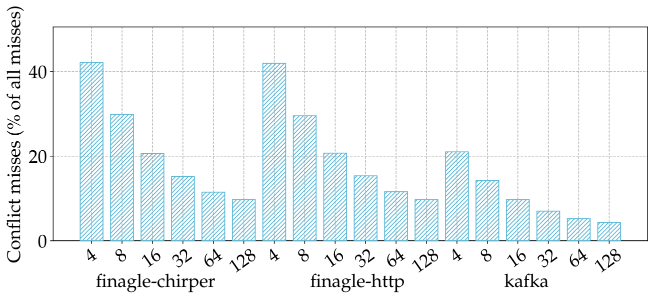


Figure 6: Percentage of conflict misses as BTB associativity increases from 4-way to 128-way: data center applications still suffer conflict BTB misses even with an 128-way set-associative BTB. For brevity, we show results for only 3 applications, the behavior is similar across all applications.

miss classification [33]. We find that the majority of these misses are capacity (on average 70%) and conflict (on average 24.48%) misses.

To investigate these capacity and conflict misses, we vary the BTB size (from 2K entries to 64K entries) and associativity (from 4-way to 128-way) and show the results in Fig. 5 and Fig. 6. We observe that these data center applications require a 64K-entry BTB to avoid most of the capacity misses. On the other hand, the BTB associativity needs to be at least 128 to cover the majority of conflict misses. Increasing BTB size and associativity to these levels will drastically increase on-chip storage and BTB lookup/update latency [20, 37]. Furthermore, future applications may require an even larger BTB size and associativity since data center applications' instruction footprints grow in an unprecedented manner [38]. Therefore, we conclude that BTB prefetching is a more future-proof solution as it can avoid latencies due to both types of BTB misses without requiring any change to the BTB organization.

Finally, in Fig. 7 and Fig. 8, we study the distribution of all BTB accesses and misses across different branch types to identify whether a specific branch type suffers from poor BTB locality. We note that unconditional direct branches and calls disproportionately face more BTB misses. Specifically, unconditional direct branches and calls are responsible for 20.75% of all dynamic branches, but

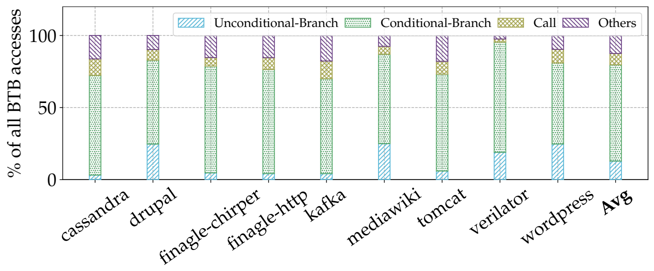


Figure 7: Breakdown of all BTB accesses into branch types: conditional branch instructions dominate the total number of BTB accesses

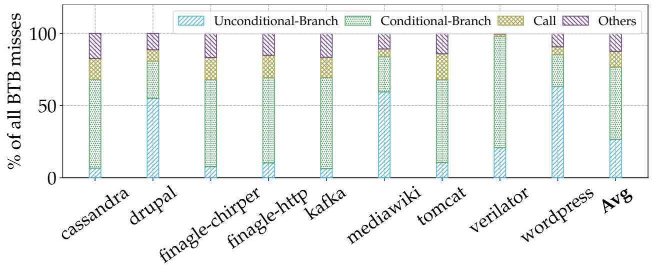


Figure 8: Breakdown of all BTB misses into different branch types: as conditional branch instructions are responsible for most BTB accesses, conditional branch instructions also experience the most number of BTB misses.

incur 37.5% of all BTB misses. This result justifies the design decisions of prior work [45] that partitions the BTB structure to prefetch conditional branch entries that follow unconditional branch executions.

2.3 Why do existing BTB prefetching mechanisms fall short?

Previously, we showed that an ideal BTB provides on average 31% speedup over the FDIP baseline. We now compare this ideal BTB speedup against speedups achieved by state-of-the-art BTB prefetchers, Confluence [40] and Shotgun [45].

Confluence observes that although the I-cache and the BTB operate at the granularity of a cache line and a branch instruction respectively, hardware prefetching mechanisms for I-cache lines and BTB entries require the same metadata. Using this insight, Confluence (1) modifies the BTB organization to match the I-cache granularity (cache line), (2) operates on the same prefetch metadata, and (3) utilizes the temporal streaming (also referred to as "record and replay" [25, 26, 39]) technique, to perform both I-cache and BTB prefetching. While Confluence was designed for a fixed-length instruction size (4B), we modify Confluence for variable-length instruction sizes since most data center applications operate on servers that use variable-length ISAs (*i.e.*, x86).

Shotgun observes that the working set size of unconditional branch instructions is significantly smaller than the working set size of all branch instructions. Hence, Shotgun statically partitions the BTB among unconditional and conditional branch entries to

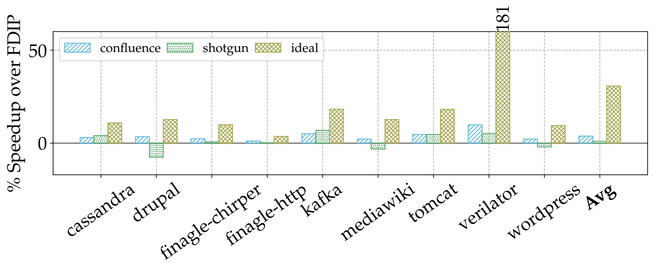


Figure 9: Speedups from Shotgun and Confluence over FDIP.

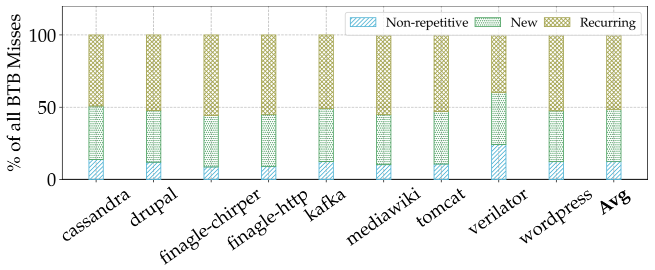


Figure 10: Fraction of BTB misses in temporal streams [81]

ensure that a certain type of branch entry does not cause evictions of the other type. Moreover, Shotgun leverages dynamic execution information to record the I-cache footprint for all unconditional branches. The next time the program executes the same unconditional branch, Shotgun prefetches the recorded I-cache lines (if not present in the I-cache) and predecodes the corresponding conditional branch entries. In our evaluation, Shotgun consists of 5120-entry unconditional BTB (63.125KB), 1536-entry conditional BTB (12.1875KB), and 1536-entry return address stack (7.5KB). All other methodological details are in §4.

Fig. 9 shows the speedup provided by Confluence and Shotgun over FDIP across all nine applications. Confluence and Shotgun offer only a fraction of an ideal BTB's speedup as they are unable to cover a significant portion of all BTB misses.

We investigate the performance of these prior BTB prefetching techniques to understand why they fail to cover so many BTB misses. Since both Confluence and Shotgun leverage temporal stream prefetching to avoid BTB misses, we categorize all BTB misses into three types of temporal streams [81]: *non-repetitive*, *new*, and *recurring* streams. Temporal stream prefetching can inherently cover only recurring miss streams. As shown in Fig. 10, while recurring miss streams constitute the majority of all BTB misses (on average 52%), new and non-repetitive streams still include a large fraction of the remaining BTB misses (on average 36% and 12% respectively) that Confluence and Shotgun do not cover. Recording access patterns at the granularity of I-cache lines instead of at the granularity of branch instructions helps Shotgun cover more BTB misses than Confluence, as Shotgun predecodes all branch instructions corresponding to a single I-cache line. Still, Shotgun falls significantly short of the ideal BTB, which we explain next.

Shotgun requires the unconditional branch footprint of the application to be small enough to fit into the BTB partition reserved for unconditional branches. Unfortunately, different applications have different unconditional branch working set sizes as we portray in Fig. 11. As a result, Shotgun's BTB partition for unconditional branches is too large for some applications and too small for others. Moreover, irrespective of whether an unconditional branch correlates with conditional branches, Shotgun reserves precious BTB storage bits as prefetch metadata for unconditional branches. Consequently, Shotgun wastes critical on-chip storage for some applications (*e.g.*, `drupal`, `mediawiki`, and `wordpress`) where the number of unconditional branches are much smaller than Shotgun's unconditional BTB partition size.

Shotgun incurs additional BTB misses due to one of its design constraints: the spatial range of conditional branches. Shotgun prefetches conditional branch entries based on the execution of unconditional branches. While doing so, Shotgun can only prefetch conditional branches that are within a spatial range of up to 8 cache lines of the last executed unconditional branch target. In other words, if a conditional branch resides outside this 8 cache line range, Shotgun will not be able to prefetch the corresponding BTB entry. However, as we show in Fig. 12, a significant portion (26-45%) of all conditional branches falls outside this spatial range. Hence, Shotgun cannot cover a large portion of all BTB misses.

Based on our characterization's insights, we next present *Twig*, a profile-guided solution to avoid costly BTB misses.

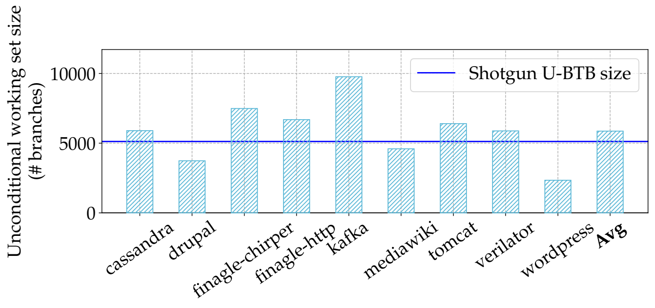


Figure 11: Working set size of unconditional branches and calls. Shotgun's U-BTB of 5120 entries is shown in blue.

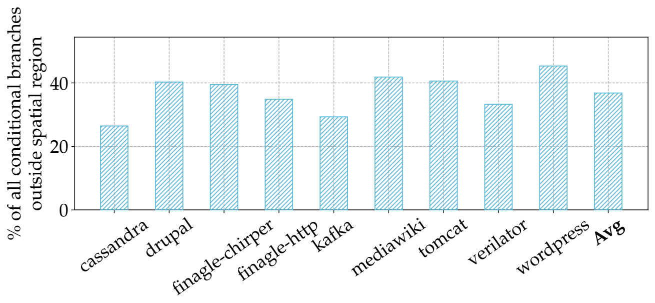


Figure 12: Percentage of all conditional branches that are outside the range (8 cache lines) of the last executed unconditional branch target. Shotgun cannot prefetch BTB entries for these conditional branches.

3 TWIG

Modern data center application binaries are large and contain numerous unique branch instructions. These applications suffer from frequent BTB misses. Prior work addresses this issue with BTB prefetchers that require significant hardware modification and yet fail to cover a large fraction of BTB misses. We propose *Twig*, a profile-guided solution to prefetch BTB entries. Specifically, *Twig* introduces two novel techniques to avoid BTB misses. First, *Twig* uses a novel profile-guided mechanism to prefetch BTB entries. Second, *Twig* coalesces prefetch operations of multiple BTB entries into a single instruction to reduce the code bloat.

3.1 Software BTB Prefetching

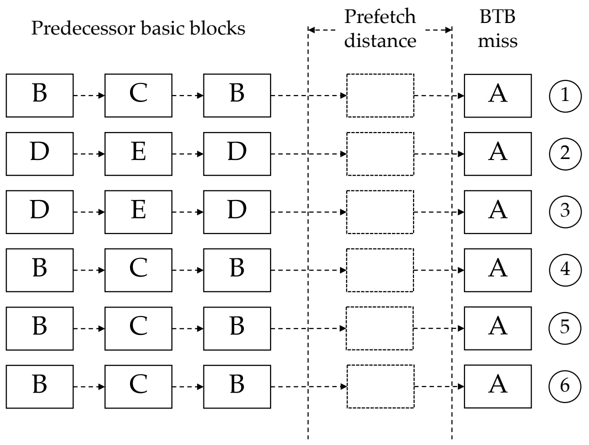
Determining branch Program Counter (PC) and target for populating the BTB requires the processor to decode (potentially variable-length) instructions. Hardware-based BTB prefetchers such as Shotgun [45] hence need to prefetch the instructions and decode them before filling the BTB, introducing significant hardware overheads for implementing the additional pre-decoders. Additionally, the prefetch latency deteriorates if the instruction being prefetched into the BTB is not present in the processor's I-cache. *Twig* addresses both of these challenges. First, *Twig* identifies the PC and target of every direct branch instruction for an application by examining its binary. Then, *Twig* leverages the program's dynamic execution profile to find the branch PCs causing a large number of BTB misses. Finally, *Twig* modifies the application binary to prefetch corresponding BTB entries in a timely manner.

To realize *Twig*, we introduce a new instruction, `brprefetch` to prefetch BTB entries. The `brprefetch` instruction uses two parameters—the branch PC and the target, to insert the corresponding branch entry into the BTB. Both these fields represent instruction pointers and can be as large as 48-bit signed integers [87]. Moreover, *Twig* must schedule the `brprefetch` instruction early enough so that it updates the BTB before the corresponding branch target lookup occurs. We now explain how *Twig* meets these requirements by finding the appropriate program location to insert the `brprefetch` instruction and by storing only the address difference between the branch instruction and the target.

Prefetch injection location. *Twig* must insert the `brprefetch` instruction in a timely manner, *i.e.*, the `brprefetch` instruction must retire before the corresponding branch is looked up in the BTB to avoid a BTB miss. Hence, it is critical to precisely identify the appropriate program location for inserting the `brprefetch` instruction. *Twig* must also emit accurate `brprefetch` instructions to avoid polluting the BTB with unnecessary entries. Since many different program paths can lead to a particular BTB miss, *Twig* must find the right program location to satisfy the accuracy constraint.

Twig leverages execution information to identify the appropriate program path that satisfies both the timeliness and accuracy constraint. With the help of Intel Last Branch Record (LBR) feature [5], *Twig* collects program execution profiles that lead to BTB misses. Intel LBR records a history of the last 32 basic blocks executed before a BTB miss along with their execution latency in cycles.

Fig. 13a portrays an example of such a profile for BTB misses at the branch instruction address, *A*, showing how *Twig* leverages this profile to find the injection site for the `brprefetch` instruction.



(a) An example of profile samples for BTB misses at branch instruction address, *A*, containing basic block executions that precede the miss.

Basic block	Total executed	# of unique BTB misses at A that can be timely covered by the basic block	P(BTB miss at A | Basic block)
B	16	4	0.25
C	8	4	0.5
D	6	2	0.33
E	3	2	0.66

(b) An example of the conditional probability calculation to predict the BTB miss at *A*, given the execution of a particular basic block.

Figure 13: An example of how *Twig* analyzes BTB miss profiles to find accurate and timely prefetch injection site

This example includes six different BTB misses for *A*. To satisfy the timeliness constraint, *Twig* considers basic blocks that precede the BTB miss by at least several cycles as candidate injection sites. We call this particular cycle count the *prefetch distance*, which is one of *Twig*'s design parameters. We use 20 cycles as the prefetch distance and evaluate *Twig*'s sensitivity to this parameter in §4 (Fig. 26). *Twig* only considers predecessor basic blocks before the prefetch distance as the prefetch injection candidates. As shown in Fig. 13a, predecessor basic blocks *B* and *C* are considered for the BTB miss ① as they precede the BTB miss by the prefetch distance.

To satisfy the accuracy constraint, *Twig* computes the conditional probability of a BTB miss at *A*, given the execution of each candidate basic block. We show an example of this computation in Fig. 13b. First, *Twig* calculates the execution count/frequency of each candidate block using the execution profile (including BTB misses at other branch instructions apart from *A*). Next, *Twig* counts how many BTB misses at *A* can be avoided by inserting a prefetch instruction at the candidate injection site. Then, *Twig* computes the ratio of these two counts as the conditional probability of a BTB miss at *A*, given the execution of each candidate basic block. Finally, *Twig* picks the candidate with the highest conditional probability for each BTB miss as the prefetch injection site. In case of this example, *Twig* selects *C* to cover BTB misses ①, ④, ⑤, and ⑥, while *Twig* chooses *E* to avoid BTB misses ② and ③.

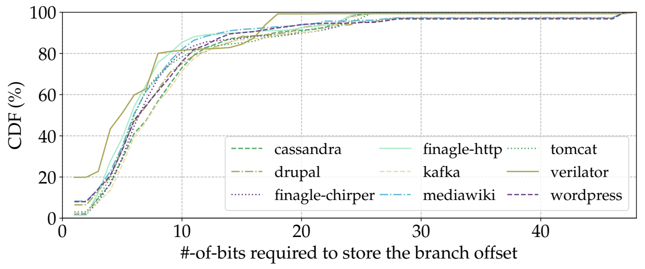


Figure 14: CDF of branch offset (from the prefetch injection site to the branch instruction) with variation in the number of bits required to store the offset: with just 12-bits *Twig* stores 80% of all branch offsets for all applications.

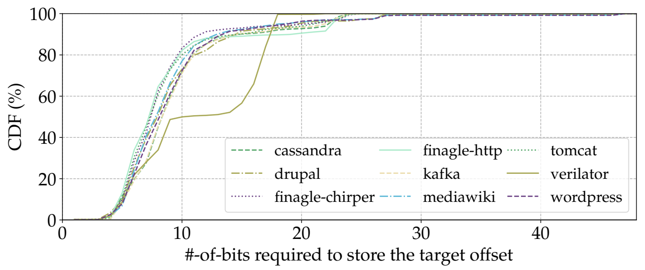


Figure 15: CDF of branch target offset with variation in the number of bits required to store the offset: with just 12-bits *Twig* stores 80% of all branch targets for most applications.

Prefetch target compression. The storage cost of large instruction pointers (branch PC and target) is a significant challenge for software BTB prefetching. *Twig* reduces this storage overhead by storing the *prefetch-to-branch-offset* instead of the entire absolute address. We define the prefetch-to-branch-offset as the delta between the prefetch instruction PC and the prefetched branch PC. Fig. 14 shows the quantitative insight behind this optimization. On the X-axis, we show the number of bits required to encode the prefetch-to-branch-offset, while on the Y-axis, we show the Cumulative Distribution Function (CDF) for all BTB misses. We find that *Twig* covers more than 80% of all BTB misses using just 12-bits to encode the prefetch-to-branch-offset. *Twig* uses the same technique to also compress the branch target. Fig. 15 plots the *branch-to-target-offset* on the X-axis and the CDF of all BTB misses on the Y-axis. We note that *Twig* again covers 80% of all BTB misses for most applications using just 12-bits. Only for `verilator`, covering more than 80% of all BTB misses requires larger than 12-bit signed integers. To cover the remaining BTB misses and to optimize the storage overhead even further, *Twig* proposes BTB prefetch coalescing that we describe next.

3.2 BTB Prefetch Coalescing

Branch instructions with large address differences cannot directly be encoded using the prefetch instruction introduced in §3.1. For these too-large-to-encode branch instructions, *Twig* stores the addresses of the branch instruction and the target as key-value pairs in memory. *Twig* stores these pairs in sorted order based on the branch instruction address. Storing branch entries in sorted order helps *Twig* leverage spatial locality among different entries. The key-value pairs are generated at compile time and added to the instruction binary as part of the text segment.

Twig introduces the `brcoalesce` instruction that takes the address of a key-value pair as a parameter and prefetches the corresponding entry to the BTB. To improve its efficiency, `brcoalesce` includes an *n*-bit bitmask as an additional parameter to prefetch multiple consecutive entries (for this reason, the key-value pairs are sorted in memory). Coalescing enables prefetching of multiple too-large-to-encode BTB entries with a minimal increase in the instruction footprint.

The size of the bitmask, *n*, is another design parameter. With a smaller bitmask, *Twig* would be able to prefetch only a small number of correlated BTB entries. With a larger bitmask, *Twig* can coalesce more prefetch operations. We investigate the impact of the bitmask size on the effectiveness of BTB prefetch coalescing in §4 and show that *Twig* achieves a majority of the performance benefit with just an 8-bit bitmask (Fig. 27).

Table 1: Simulator Parameters

Parameter	Value
CPU	3.2GHz, 6-wide OOO, 24-entry FTQ, 224-entry ROB, 97-entry RS
Branch prediction unit	64KB TAGE-SC-L [73] (up to 12-instruction), 8192-entry 4-way BTB, 32-entry RAS, 4096-entry 4-way IBTB
Memory hierarchy	32KB 8-way L1i, 32KB 8-way L1d, 1MB 16-way unified L2, 10MB 20-way shared L3 per socket

4 EVALUATION

In this section, we first describe (1) our experimental setup to collect execution profiles for our target data center applications, (2) different application input configurations, and (3) our simulation infrastructure. Then, we evaluate *Twig* using several key performance metrics.

4.1 Methodology

Data center applications and inputs. We evaluate *Twig* in the context of nine popular data center applications (as described in §2). We evaluate these applications with different input configurations such as the input data size, the webpage requested by the client, the number of client requests per second, random number seeds, and the number of server threads. Since *Twig*'s profile-guided optimizations depend on the application input, we optimize each of these applications using the profile from one input and test the performance of the optimization on a different input.

Profile collection. We leverage Intel's "`baclears.any`" hardware performance event along with LBR [5] to collect the application execution context profiles that lead to a BTB miss.

Simulation and trace collection. We evaluate *Twig* using Scarab [6]. In Scarab, we implement support for the BTB prefetch instructions (`brprefetch` and `brcoalesce`) and also add implementations for FDIP, Shotgun, and Confluence. We list different simulation parameters that resemble a recent state-of-the-art industry baseline [35, 36] in Table 1. Both trace-driven and execution-driven

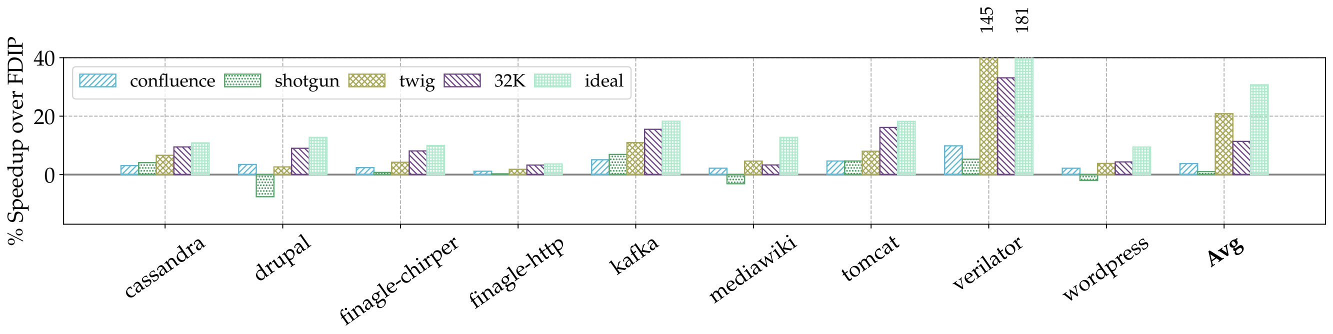


Figure 16: Percentage speedup over the FDIP baseline: 32K is for a 32K-entry BTB compared to the 8K-entry baseline BTB. *Twig* outperforms even the 32K-entry BTB on average with just an 8K-entry BTB with prefetching.

Scarab modes use Intel PIN [49], which cannot instrument kernel mode instructions. To support kernel mode instruction simulations, we collect application traces using Intel Processor Trace [1] and modify Scarab to support simulating such traces as well. We simulate traces of 100 million representative, steady-state instructions for each data center application.

4.2 Performance Analysis

We now validate *Twig*'s effectiveness using key performance metrics. First, we compare *Twig*'s speedup to the speedup offered by an ideal BTB and the state-of-the-art BTB prefetcher, Shotgun [45]. Then, we evaluate the individual speedup contributions of software BTB prefetching and BTB prefetch coalescing. We also compare *Twig* against Shotgun in terms of BTB miss coverage and BTB prefetch accuracy. Furthermore, we compare speedups achieved by *Twig* and Shotgun across different application inputs. Finally, we measure *Twig*'s static and dynamic overhead due to the additional BTB prefetch instructions.

Speedup. We show *Twig*'s speedup (brown bars) for nine data center applications in Fig. 16. For comparison, we also show speedup offered by an ideal BTB (purple bars) and state-of-the-art BTB prefetcher, Shotgun (green bars). As shown, *Twig* achieves on average 20.86% speedup compared to 31% mean speedup achieved by an ideal BTB and 1% mean speedup achieved by Shotgun. On average, *Twig* achieves 48% (and up to 80%) of the speedup achieved by an ideal BTB that incurs no BTB misses. *Twig* cannot provide the entire benefit (100%) of an ideal BTB for a number of reasons. First, some BTB misses do not have a predecessor basic block that can predict the potential BTB miss with high accuracy. Second, BTB prefetch instructions injected by *Twig* incur both static and dynamic instruction overheads (we quantify this overhead later in this section). Finally, *Twig* cannot cover some previously unobserved BTB misses due to the use of different inputs in profiling and testing (we also quantify this later in the section). Still, *Twig* advances the state-of-the-art by outperforming Shotgun by 19.82% on average (and up to 139.8%) as *Twig* covers more BTB misses than Shotgun.

BTB miss coverage. Fig. 17 shows the BTB miss coverage comparison between *Twig* and Shotgun. As shown, *Twig* covers on average 65.4% (and up to 95.84%) of all BTB misses. Additionally, *Twig* covers on average 57.4% (and up to 94%) more BTB misses than

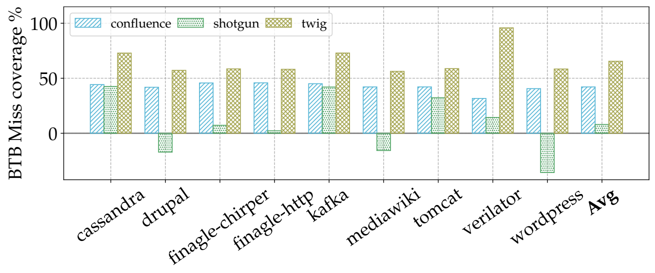


Figure 17: BTB miss coverage of *Twig*, Confluence, and Shotgun: on average *Twig* covers 65.4% of all BTB misses.

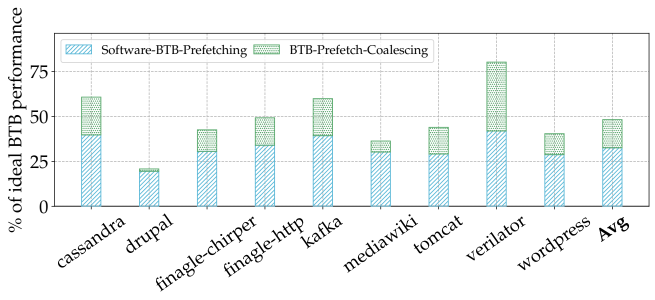


Figure 18: Contribution of software BTB prefetching and BTB prefetch coalescing toward *Twig* performance of an ideal BTB: software BTB prefetching provides greater benefits than BTB prefetch coalescing across applications.

the state-of-the-art prefetcher, Shotgun. *Twig* outperforms Shotgun to cover 57.4% more BTB misses primarily because of the reasons we describe in §2.3. In contrast to Shotgun's ability to prefetch only conditional branch entries within a limited spatial range, *Twig* can prefetch BTB entries irrespective of branch type or distance.

Performance of software BTB prefetching and BTB prefetch coalescing. Fig. 18 shows the individual contributions of software BTB prefetching and BTB prefetch coalescing to *Twig*'s overall speedup. As shown, software BTB prefetching without any coalescing provides on average 32.6% speedup (70.9% of overall performance gains) across different applications. On top of this,

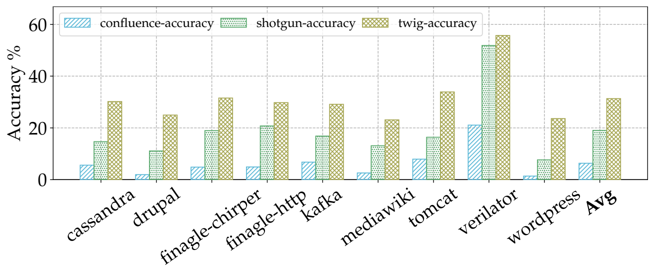


Figure 19: Prefetch accuracy of *Twig*, Confluence, and Shotgun: on average *Twig* provides 31.3% BTB prefetch accuracy across nine data center applications.

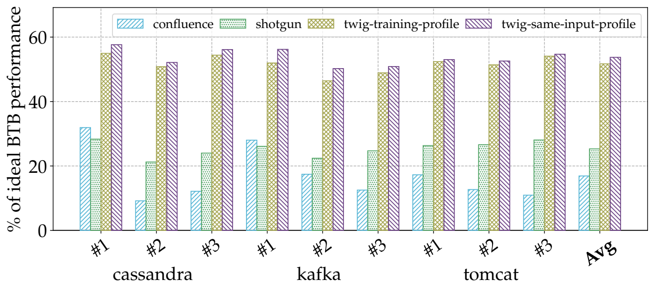


Figure 20: *Twig*'s speedup across different application inputs as the percentage of an ideal BTB performance: *Twig* trained on a different input provides performance benefits comparable to *Twig* trained on the same input and outperforms existing BTB prefetching mechanisms.

prefetch coalescing provides on average 15.7% speedup (29.1% of overall benefits) by reducing the static and dynamic instruction overhead.

Prefetch accuracy. We show *Twig*'s prefetch accuracy in Fig. 19 and compare it against Shotgun's prefetch accuracy. As shown, *Twig* provides 31.3% average accuracy. Moreover, *Twig* achieves 12.3% higher prefetch accuracy than Shotgun due to the fundamental limitation of hardware temporal stream prefetching. Like most prior hardware techniques on temporal memory streaming [20, 26, 77, 81–83], Shotgun remembers the spatial footprint seen during the last execution and prefetches the corresponding BTB entries. While prefetching the most recently executed footprint is efficient in terms of metadata storage (compared to most frequently executed footprint), it incurs many inaccurate BTB prefetches. *Twig*, on the other hand, leverages a large amount of execution information from the collected profile to identify the most accurate prefetch predecessor and achieves higher prefetch accuracy.

Performance across different application inputs. The effectiveness of profile-guided optimizations usually depends on the corresponding application input. To investigate how this dependence affects *Twig*'s performance, we compare the speedups achieved by *Twig* across different application inputs in Fig. 20. For each application, we use the profile from input `'#0'` to optimize BTB performance using *Twig* and measure the speedups for other inputs, `'#1, #2, #3'`. For comparison, we also measure speedups achieved by

Table 2: *Twig*'s average speedup across different application inputs with standard deviations.

Application	% of ideal BTB performance			
	Same input profile		Training profile	
	Average	Standard deviation	Average	Standard deviation
cassandra	49.31	10.04	45.93	15.53
drupal	36.77	14.31	43.15	9.84
finagle-chirper	38.30	9.13	31.99	10.29
finagle-http	34.03	7.73	32.66	5.62
kafka	52.35	2.17	49.93	2.26
mediawiki	38.78	10.95	43.78	5.11
tomcat	51.25	4.02	45.77	15.84
verilator	80.33	0.39	79.19	0.33
wordpress	45.15	14.69	49.71	12.85

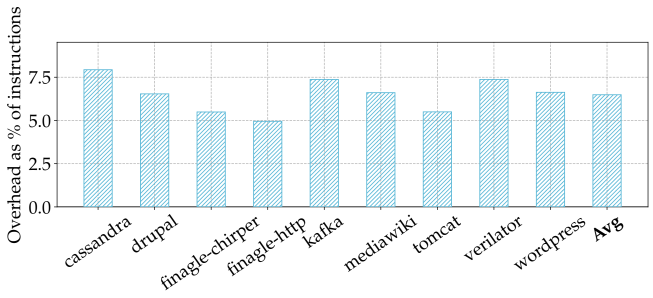


Figure 21: Static overhead of *Twig*, measured in % of additional instructions in the binary for a given workload: on average *Twig* inserts 6% extra static instructions.

Table 3: Instruction working set size overhead of *Twig*.

Application	Instruction working set size (MB)	Additional instruction size (MB)	Overhead (%)
cassandra	4.23	0.26	6.08
drupal	1.75	0.05	2.93
finagle-chirper	2.05	0.07	3.54
finagle-http	5.29	0.42	7.97
kafka	3.28	0.16	4.78
mediawiki	2.24	0.08	3.70
tomcat	2.40	0.10	4.10
verilator	13.56	1.34	9.86
wordpress	1.93	0.06	3.09

Twig when optimized with the profile from the same input. Finally, we compare *Twig* against Confluence and Shotgun for different application inputs. For each configuration, we normalize the overall speedup by expressing it in terms of ideal BTB performance.

As shown in Fig. 20, *Twig* provides significantly more benefit than state-of-the-art mechanisms [40, 45] even while using profiles from a different application input. *Twig* provides a greater speedup when optimized using input-specific profiles (as shown in Table 2) for 6 out of 9 applications. However, for the remaining three applications, *Twig* can achieve even better speedup with profiles from a different application input. Nonetheless, *Twig* achieves comparable speedups with profiles from both same and different inputs.

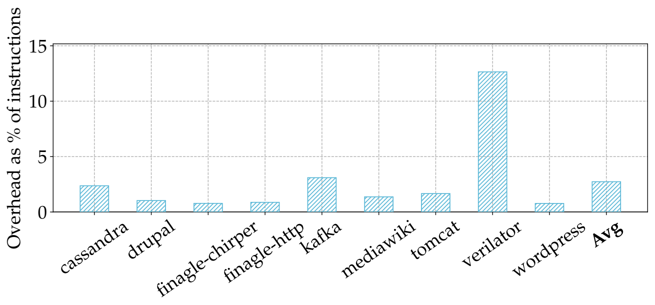


Figure 22: Dynamic overhead of *Twig*, measured in % of additional executed instructions for a given workload: on average *Twig* incurs only 3% extra dynamic instructions.

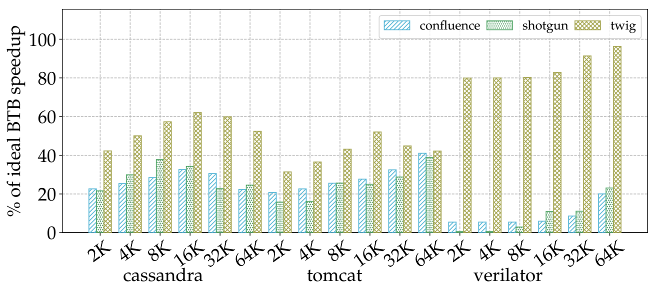


Figure 23: % of speedup obtained by *Twig* compared to an ideal BTB for BTB capacities ranging from 2048 entries to 65536 entries

Prefetch overhead. *Twig* does not introduce any extra metadata storage. Therefore, instructions added to perform BTB prefetching are the only overhead *Twig* introduces. We quantify the static and dynamic overhead of these prefetch instructions in Fig. 21 and 22. In Table 3, we quantify the combined overhead of static and dynamic instruction increase based on working set size increase in terms of the number of added bytes. As shown, *Twig* introduces less than 8% static and 12.6% dynamic instruction overhead for all cases. Specifically, *Twig* incurs the highest dynamic overhead for `verilator` to cover the large number of BTB misses incurred by the application (BTB MPKI of 121).

4.3 Sensitivity Analysis

We investigate the sensitivity of different design parameters on *Twig*'s effectiveness. First, we compare the speedup achieved by *Twig* and Shotgun for different BTB storage budgets (size and associativity) and prefetch buffer sizes. Additionally, we evaluate the effect of changing the prefetch distance and FDIP run-ahead on *Twig*'s effectiveness.

BTB storage budget. In Fig. 23, we evaluate how sensitive *Twig* is to the storage budget allocated to the BTB by varying the number of BTB entries. We fix all other parameters and vary the number of BTB entries between 2048 (2K) and 65536 (64K). As Fig. 23 shows, *Twig* achieves more speedup than either Shotgun or Confluence across all BTB sizes. We also vary BTB's associativity from 4 ways per set to 128 ways per set. Fig. 24 shows how *Twig* outperforms both Shotgun and Confluence for any associativity.

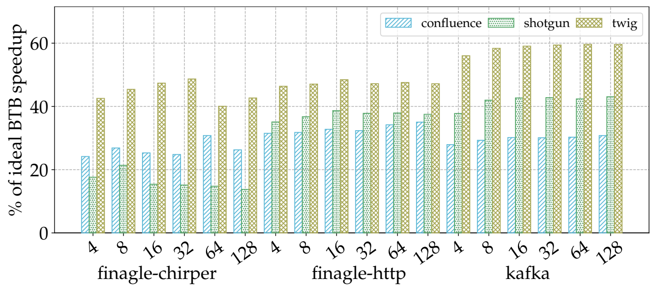


Figure 24: % of speedup obtained by *Twig* compared to an ideal BTB for BTB associativity ranging from 4 to 128

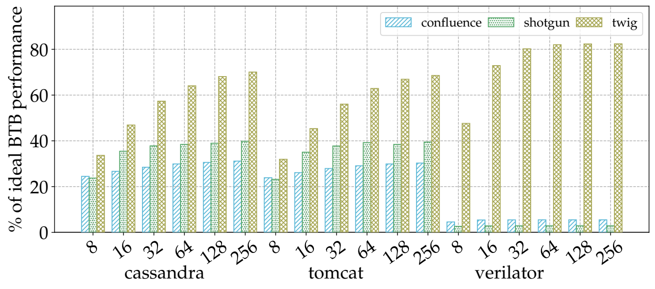


Figure 25: Percent of speedup obtained by *Twig* compared to an ideal BTB for the size of the prefetch buffer, ranging from 8 to 256

Prefetch buffer size. We next vary the size of the BTB prefetch buffer. This enables us to hold additional BTB entry candidates at any given time, enabling *Twig* prefetches to not evict each other. As shown in Fig. 25, *Twig*'s performance scales from from 8 to about 128 entries before it begins to experience diminishing returns. Shotgun and Confluence do not experience this same scaling, indicating that *Twig* provides greater benefits than prior works irrespective of the prefetch buffer size.

Prefetch distance. Fig. 26 shows how *Twig*'s effectiveness varies in response to variation in prefetch distance. We vary the prefetch distance from 0 to 50 cycles and measure *Twig*'s average performance as a percentage of ideal BTB performance across applications. As shown, *Twig* provides only a portion of the potential speedup when the prefetch distance is too small to complete the prefetch before the BTB lookup. On the other hand, *Twig* cannot find an appropriate prefetch injection site when the prefetch distance is too large to ignore accurate predecessors. Consequently, *Twig* provides the greatest benefit with 15-25 cycles of prefetch distances across different applications.

Coalescing size. We investigate the effectiveness of *Twig*'s BTB prefetch coalescing with an increase in coalescing bitmask size. Fig. 27 shows the average performance gains of BTB prefetch coalescing as the percentage of ideal BTB performance for different bitmask sizes (1-bit to 64-bit) across nine data center applications. As shown, *Twig* realizes a large fraction of the potential speedup

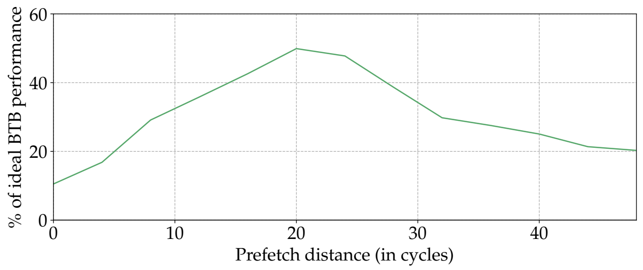


Figure 26: *Twig*'s average performance variation in response to increasing the prefetch distance. Across different applications, *Twig* provides greatest benefit with prefetch distance 15-25 cycles.

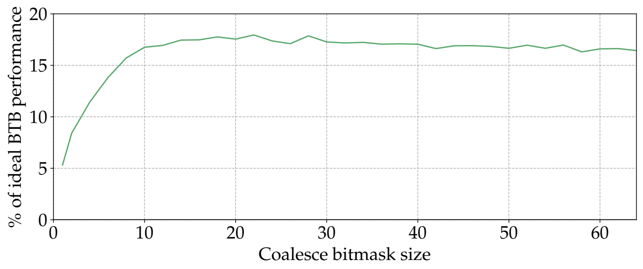


Figure 27: *Twig*'s average performance variation in response to changes in the coalesce bitmask size. *Twig* achieves a majority of the potential performance gains with a 8-bit coalesce bitmask.

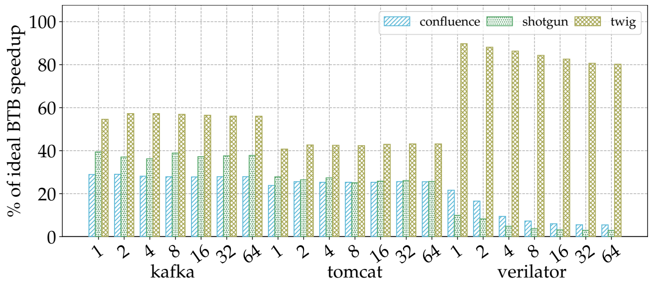


Figure 28: % of speedup obtained by *Twig* compared to an ideal BTB for the size of the FTQ, or maximum distance the decoupled frontend can run ahead, varied between 1 and 64

with an 8-bit bitmask. Consequently, we use 8-bits to coalesce BTB prefetch instructions.

FDIP Run-ahead. Finally, we vary the size of the Fetch Target Queue (FTQ), which determines how far ahead the decoupled frontend can run. We vary the FTQ size from 1 to 64 branches. Fig. 28 shows that *Twig* achieves a similar performance relative to ideal at every measured FTQ length. Since a longer FTQ has been shown to improve performance by reducing frontend stalls [35], this result implies that *Twig* scales well to frontends that run far ahead of the fetch unit.

5 RELATED WORK

Preventing I-cache misses. Many prior works focus on reducing frontend stalls via eliminating I-cache misses. These techniques can be summarized in three distinct categories: software only, hardware only, and a hybrid software/hardware approach. Software techniques include improving instruction locality via basic block/function reordering [57, 65], hot/cold splitting [23], and other Profile-Guided Optimizations (PGO) [15, 19, 22, 31, 32, 48, 50, 51, 58, 63, 67, 90]. Improved layout techniques are only able to eliminate a subset of all I-cache misses and finding the optimal code layout for I-cache performance is intractable in practice [16, 64]. Hardware-only techniques tend to have one of two limitations. Either the techniques have prohibitive on-chip storage costs [25, 26], or they end up being significantly more complex [39, 40, 44] than prefetching techniques implemented in real hardware [69, 71]. Hybrid hardware and software approaches [16, 41] attempt to avoid the pitfalls of software only or hardware only approaches by performing the complicated software analysis ahead of time and executing simple prefetch instructions at runtime. However, prior approaches either make assumptions that are too simplistic, limiting prefetching accuracy, or execute too many dynamic instructions which exacerbate the application's code footprint [16, 53]. State-of-the-art I-cache prefetchers include the SN4L+Dis+BTB design [12] and the contenders of the first instruction cache prefetching competition (IPC-1) [11, 27, 28, 30, 52, 54, 70, 74]. Of the above proposals, FDIP has a desirable trade-off between metadata cost and prefetching effectiveness [45, 46]. Even with significantly smaller metadata storage costs, FDIP provides comparable performance benefits to state-of-the-art I-cache prefetchers [35, 36]. Moreover, recent commercial CPU designs adopted some FDIP variants to reduce frontend stalls [29, 61, 72, 80]. Therefore, in this work, we focus on improving FDIP effectiveness by introducing software BTB prefetching that provides 20.86% average speedup without requiring any extra metadata storage.

BTB redesign / compression. The design and usage of the storage allocated to the BTB has long been debated. BTB entries commonly hold some combination of a tag, prediction information, and target address [47, 62]. The basic-block style BTB also contains the address of the fall-through basic block [89]. Compressing BTB entry size is common to enable the BTB to host more entries in the same storage budget. Such techniques [21, 24, 37, 43, 62, 68, 75] include using fewer bits for the tag, removing the page number from the tag, encoding the branch target as a small delta from the branch PC, and adding a larger second level BTB for which the first level BTB acts as a small cache. BTB-X [13] and PDede [78] apply several of these compression techniques, including partitioning the BTB into segments to enable aggressive compression and deduplication. All of these techniques enable the underlying BTB to have a larger capacity for a given storage budget. Since *Twig* prefetches entries into the BTB, it is independent of the underlying BTB and should be just as effective with the above techniques.

BTB prefetching. Phantom-BTB (PBTB) [20] virtualized predictor metadata into the shared L2 cache, and used entries in the virtualized table to prefetch BTB entries. PBTB suffers from a relatively high cost of metadata storage and a longer latency access time for important branch prediction metadata. Two-level bulk preload [18] maintains two BTB levels per-core, with a mechanism to fetch a group of BTB entries for a fixed-size region to the first level on a miss to any branch in that region. This is limited to exploiting the available spatial locality of a branch, and thus is similar to the next-line prefetchers. Confluence [40] keeps the I-cache and BTB contents in sync via their AirBTB design, with the ability to predecode branches and BTB entries for a given I-cache block. Locking the I-cache and BTB contents limits the runahead ability of the branch predictor unit. Moreover, Confluence relied on a metadata-expensive temporal prefetcher, SHIFT [39, 40, 46]. Boomerang [46] modifies FDIP to predecode fetched I-cache blocks and insert the corresponding BTB entries. However, the ability for these entries to be timely is largely dependent upon the frontend to run far enough ahead, and miss coverage suffers when there are many BTB misses [45]. Shotgun [45] partitions the BTB into the Unconditional BTB (U-BTB) and much smaller Conditional BTB (C-BTB), with a way to prefetch entries into the C-BTB when the U-BTB is hit. As such, Shotgun relies on a high U-BTB hit rate to keep the C-BTB full of useful entries [12]. This reliance limits Shotgun's ability to scale. Additionally, any fixed partitioning scheme, as in U-BTB vs. C-BTB sizes, need the workload's distribution of branches to match, and results in underutilized space when the application deviates from the fixed partitioning scheme. See §2.3 for a in-depth investigation on the impact of the limitations of each approach, and why they cannot cover all BTB misses. In this work, we investigate the reasons behind their limitation and address such limitations by proposing profile-guided BTB prefetch mechanisms that outperform prior techniques.

6 CONCLUSION

Large branch footprints of data center applications cause frequent BTB misses, resulting in significant frontend stalls. We showed that existing BTB prefetching techniques fail to overcome these stalls due to inadequate understanding of the applications' branch access patterns. To address this limitation, we proposed *Twig*, a profile-guided BTB prefetching mechanism. *Twig* presents two BTB prefetching techniques: software BTB prefetching and BTB prefetch coalescing. We evaluated *Twig* in the context of nine popular data center applications. Across these applications, *Twig* achieves an average of 20.86% (2%-145%) performance speedup and outperforms the state-of-the-art BTB prefetching technique by 19.82%.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback and suggestions. This work was supported by the Intel Corporation, NSF grants #1823559, #2011168, #2010810, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We thank Maksim Panchenko and Guilherme Ottoni from Facebook for helpful discussions about HHVM control-flow behavior.

REFERENCES

[1] [n. d.]. Adding Processor Trace support to Linux. https://lwn.net/Articles/648154/.
[2] [n. d.]. Apache Cassandra. http://cassandra.apache.org/.
[3] [n. d.]. Apache kafka. https://kafka.apache.org/powered-by.
[4] [n. d.]. Apache Tomcat. https://tomcat.apache.org/.
[5] [n. d.]. An Introduction to Last Branch Records. https://lwn.net/Articles/680985/.
[6] [n. d.]. Scarab. https://github.com/hpsresearchgroup/scarab.
[7] [n. d.]. Twitter Finagle. https://twitter.github.io/finagle/.
[8] [n. d.]. Verilator. https://www.veripool.org/wiki/verilator.
[9] 2019. facebookarchive/oss-performance: Scripts for benchmarking various php implementations when running open source software. https://github.com/facebookarchive/oss-performance. (Online; last accessed 15-November-2019).
[10] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 777–790.
[11] Ali Ansari, Fatemeh Golshan, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2020. MANA: Microarchitecting an instruction prefetcher. *The First Instruction Prefetching Championship* (2020).
[12] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2020. Divide and Conquer Frontend Bottleneck. In *Proceedings of the 47th Annual International Symposium on Computer Architecture*.
[13] Truls Asheim, Boris Grot, and Rakesh Kumar. 2021. BTB-X: A Storage-Effective BTB Organization. *IEEE Computer Architecture Letters* (2021).
[14] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 643–656.
[15] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 513–526.
[16] 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.
[17] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 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*. 169–190.
[18] James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, and Anthony Saporito. 2013. Two level bulk preload branch prediction. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture*. IEEE, 71–82.
[19] Peter Braun and Heiner Litz. 2019. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*.
[20] Ioana Burcea and Andreas Moshovos. 2009. Phantom-BTB: a virtualized branch target buffer design. *Acm Sigplan Notices* 44, 3 (2009), 313–324.
[21] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. 2011. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro* 31, 2 (2011), 6–15.
[22] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *CGO*.
[23] Robert Cohn and P Geoffrey Lowney. 1996. Hot cold optimization of large Windows/NT applications. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 80–89.
[24] Barry Fagin. 1997. Partial resolution in branch target buffers. *IEEE Trans. Comput.* 46, 10 (1997), 1142–1145.
[25] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. In *International Symposium on Microarchitecture*.
[26] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *International Symposium on Microarchitecture*.
[27] Nathan Gober, Gino Chacon, Daniel Jiménez, and Paul V Gratz. [n. d.]. The Temporal Ancestry Prefetcher. ([n. d.]).
[28] Daniel A Jiménez Paul V Gratz and Gino Chacon Nathan Gober. [n. d.]. BARCa: Branch Agnostic Region Searching Algorithm. ([n. d.]).
[29] 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*. IEEE, 40–51.

[30] Vishal Gupta, Neelu Shivprakash Kalani, and Biswabandan Panda. [n. d.]. Run-Jump-Run: Bouquet of Instruction Pointer Jumpers for High Performance Instruction Prefetching. ([n. d.]).

[31] Stavros Harizopoulos and Anastassia Ailamaki. 2004. STEPS towards cache-resident transaction processing. In *International conference on Very large data bases*.

[32] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. *arXiv preprint arXiv:1803.02329* (2018).

[33] Mark D Hill and Alan Jay Smith. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput.* 38, 12 (1989), 1612–1630.

[34] Intel. 2021. Front-End Bound. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/front-end-bound.html.

[35] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2020. Rebasing Instruction Prefetching: An Industry Perspective. *IEEE Computer Architecture Letters* (2020).

[36] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective. *IEEE International Symposium on Performance Analysis of Systems and Software* (2021).

[37] Daniel A Jiménez, Stephen W Keckler, and Calvin Lin. 2000. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 67–76.

[38] 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.

[39] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. Shift: Shared history instruction fetch for lean-core server processors. In *International Symposium on Microarchitecture*.

[40] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*. 166–177.

[41] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 146–159.

[42] 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 *Proceedings of the 48th International Symposium on Computer Architecture*.

[43] Ryotaro Kobayashi, Yuji Yamada, Hideki Ando, and Toshio Shimada. 1999. A cost-effective branch target buffer with a two-level table organization. In *Proceedings of the 2nd International Symposium of Low-Power and High-Speed Chips (COOL Chips II)*.

[44] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. 2013. RDIP: return-address-stack directed instruction prefetching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 260–271.

[45] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the Front-End Bottleneck with Shotgun. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 30–42. https://doi.org/10.1145/3173162.3173178

[46] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 493–504.

[47] Lee and Smith. 1984. Branch Prediction Strategies and Branch Target Buffer Design. *Computer* 17, 1 (1984), 6–22. https://doi.org/10.1109/MC.1984.1658927

[48] David Xinliang Li, Raksit Ashok, and Robert Hundt. 2010. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 53–61.

[49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[50] Chi-Keung Luk and Todd C Mowry. 1998. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *International Symposium on Microarchitecture*.

[51] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. 2004. Ispike: a post-link optimizer for the Intel/spl reg/Itanium/spl reg/architecture. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 15–26.

[52] Pierre Michaud. 2020. PIPS: Prefetching Instructions with Probabilistic Scouts. In *The 1st Instruction Prefetching Championship*.

[53] Nayana Prasad Nagendra, Grant Ayers, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2020. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. *IEEE Micro* 40, 3 (2020), 56–63.

[54] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. [n. d.]. D-JOLT: Distant Jolt Prefetcher. ([n. d.]).

[55] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–165.

[56] Guilherme Ottoni and Bin Liu. [n. d.]. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 340–350.

[57] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 233–244.

[58] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[59] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 119–130.

[60] Reena Panda, Paul V Gratz, and Daniel A Jiménez. 2011. B-fetch: Branch prediction directed prefetching for in-order processors. *IEEE Computer Architecture Letters* 11, 2 (2011), 41–44.

[61] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, et al. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* 40, 2 (2020), 53–62.

[62] Chris H Perleberg and Alan Jay Smith. 1993. Branch target buffer design and optimization. *IEEE transactions on computers* 42, 4 (1993), 396–412.

[63] Larry L Peterson. 2001. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Transactions on Computer Systems* (2001).

[64] Erez Petrank and Dror Rawitz. 2002. The Hardness of Cache Conscious Data Placement. In *POPL*.

[65] Karl Pettis and Robert C Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 16–27.

[66] 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 *Programming Language Design and Implementation*.

[67] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P Geoffrey Lowney, and Mateo Valero. 2001. Code layout optimizations for transaction processing workloads. *ACM SIGARCH Computer Architecture News* (2001).

[68] Glenn Reinman, Todd Austin, and Brad Calder. 1999. A scalable front-end architecture for fast instruction delivery. *ACM SIGARCH Computer Architecture News* 27, 2 (1999), 234–245.

[69] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 16–27.

[70] Alberto Ros and Alexandra Jimborean. 2020. The entangling instruction prefetcher. *IEEE Computer Architecture Letters* 19, 2 (2020), 84–87.

[71] Eric Rotenberg, Steve Bennett, and James E Smith. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 24–34.

[72] J Rupley. 2018. Samsung Exynos M3 Processor. *IEEE Hot Chips* 30 (2018).

[73] André Seznec. 2014. Tage-sc-l branch predictors. In *JILP-Championship Branch Prediction*.

[74] André Seznec. 2020. The FNL+ MMA Instruction Cache Prefetcher. In *IPC-1-First Instruction Prefetching Championship*.

[75] S Seznec. 1996. Don't use the page number, but a pointer to it. In *23rd Annual International Symposium on Computer Architecture*. IEEE, 104–104.

[76] Alan Jay Smith. 1978. Sequential program prefetching in memory hierarchies. *Computer* 12 (1978), 7–21.

[77] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 69–80.

[78] Niranjan Soundararajan, Peter Braun, Tanvir Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. PDede: Partitioned, Deduplicated, Delta Branch Target Buffer. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*.

[79] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[80] David Suggs, Mahesh Subramony, and Dan Bouvier. 2020. The AMD "Zen 2" Processor. *IEEE Micro* 40, 2 (2020), 45–52.

[81] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal streams in commercial server applications. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 99–108.

[82] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 79–90.

[83] Thomas F Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. 2005. Temporal streaming of shared memory. In *32nd International Symposium on Computer Architecture*. IEEE, 222–233.

[84] Wikipedia contributors. 2020. Apache Kafka — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_Kafka&oldid=988898935. [Online; accessed 23-November-2020].

[85] Wikipedia contributors. 2020. Verilator — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Verilator&oldid=989046249. [Online; accessed 8-April-2021].

[86] Wikipedia contributors. 2021. Apache Cassandra — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_Cassandra&oldid=1010524207. [Online; accessed 7-April-2021].

[87] Wikipedia contributors. 2021. X86-64 — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=X86-64&oldid=1016690406. [Online; accessed 10-April-2021].

[88] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *ISPASS*.

[89] Tse-Yu Yeh and Yale N Patt. 1992. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *ACM SIGMICRO Newsletter* 23, 1-2 (1992), 129–139.

[90] Jingren Zhou and Kenneth A Ross. 2004. Buffering databse operations for enhanced instruction cache performance. In *International conference on Management of data*.