Shooting Down The Server Front-End Bottleneck

RAKESH KUMAR, Norwegian University of Science and Technology (NTNU), Norway
BORIS GROT, University of Edinburgh, United Kingdom

The front-end bottleneck is a well-established problem in server workloads owing to their deep software stacks and large instruction footprints. Despite years of research into effective L1-I and BTB prefetching, state-of-the-art techniques force a trade-off between metadata storage cost and performance. Temporal Stream prefetchers deliver high performance but require a prohibitive amount of metadata to accommodate the temporal history. Meanwhile, BTB-directed prefetchers incur low cost by using the existing in-core branch prediction structures, but fall short on performance due to BTB's inability to capture the massive control flow working set of server applications. This work overcomes the fundamental limitation of BTB-directed prefetchers, which is capturing a large control flow working set within an affordable BTB storage budget. We re-envision the BTB organization to maximize its control flow coverage by observing that an application's instruction footprint can be mapped as a combination of its unconditional branch working set and, for each unconditional branch, a spatial encoding of the cache blocks around the branch target. Effectively capturing a map of the application's instruction footprint in the BTB enables highly effective BTB-directed prefetching that outperforms the state-of-the-art prefetchers by up to 10% for equivalent storage budget.

CCS Concepts: • **Computer systems organization** → **Architectures**;

Additional Key Words and Phrases: Server, Microarchitecture, Prefeteching, Instruction Cache, Branch Target Buffer (BTB)

ACM Reference Format:
Rakesh Kumar and Boris Grot. 2021. Shooting Down The Server Front-End Bottleneck. *ACM Trans. Comput. Syst.* 1, 1, Article 1 (January 2021), 30 pages. https://doi.org/10.1145/3484492

Authors' addresses: Rakesh Kumar, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, rakesh.kumar@ntnu.no; Boris Grot, University of Edinburgh, Edinburgh, United Kingdom, boris.grot@ed.ac.uk.

The work presented in this manuscript is an extension of our conference paper entitled "Blasting Through The Front-End Bottleneck With Shotgun", which was published in the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) March, 2018 [1]. This manuscript extends the conference paper by proposing a new design optimization, significantly expanding the evaluation, and identifying future research directions. Concretely, the new contributions of this manuscript are as follows:

- This work proposes a new mechanism for spatial region prefetching, called "Bimodal 5-blocks", which offers a significant improvement in storage efficiency compared to the original Shotgun design with minimal performance impact. (Section 6.3)
- Shotgun features two components for each of L1-I prefetching (FTQ and Spatial prefetching) and BTB prefilling (Proactive and Reactive prefilling). This work quantifies the contribution of each of these components towards overall performance and shows that while FTQ and Spatial L1-I prefetching complement each other, the reactive BTB prefilling provides the majority of the benefits by itself. (Section 6.4)
- This work identifies the factors that prevent Shotgun from matching the performance of an ideal server front-end. Based on these factors, we suggest the future research directions to erase the performance difference between an ideal and practical front-end. (Section 6.7)
- This work qualitatively compares Shotgun's FTQ (fetch target queue) filling mechanism to that of Boomerang (a state-of-the-art prefetcher) and details how it enables continuous prefetching under a BTB miss, whereas Boomerang is unable to do so. (Details in the "Discussion" at the end of Section 4)

Manuscript submitted to ACM

1 INTRODUCTION

Traditional and emerging server workloads are characterized by large instruction working sets stemming from deep software stacks. A user request hitting a modern server stack may go through a web server, database, custom scripts, logging and monitoring code, and storage and network I/O paths in the kernel. Depending on the service, even simple requests may take tens of milliseconds to complete while touching MBs of code.

The deep stacks and their large code footprints can easily overwhelm private instruction caches (L1-I) and branch prediction structures, diminishing server performance due to the so-called *front-end bottleneck*. Specifically, instruction cache misses may expose the core to tens of cycles of stall time if filled from the last-level cache (LLC). Meanwhile, branch target buffer (BTB) misses may lead to unpredicted control flow transfers, triggering a pipeline flush when misspeculation is discovered.

The front-end bottleneck in servers is a well-established problem, first characterized in the late 90s [2–4]. Over the years, the problem has persisted; in fact, according to a recent study from Google [5], it is getting worse due to continuing expansion in instruction working set sizes in commercial server stacks. As one example of this trend, the Google study examined the Web Search workload whose multi-MB instruction footprint had been expanding at an annualized rate of 27%, doubling over the course of their study [5].

Microarchitecture researchers have proposed a number of instruction [6–10] and BTB [11, 12] prefetchers over the years to combat the front-end bottleneck in servers. State-of-the-art prefetchers rely on *temporal streaming* [7] to record and replay instruction cache or BTB access streams. While highly effective, each prefetcher requires hundreds of kilobytes of metadata storage per core. Recent temporal streaming research has focused on lowering the storage costs [8, 13, 14]; however, even with optimizations, for a many-core CMP running several consolidated workloads, the total storage requirements can reach into megabytes.

To overcome the overwhelming metadata storage costs of temporal streaming, the latest work in relieving the front-end bottleneck leverages *fetch-directed instruction prefetching* (FDIP) [9] and extends it with unified prefetching into the BTB [15]. The scheme, called Boomerang, discovers BTB misses on the prefetch path and fills them by fetching the appropriate cache blocks and extracting the necessary branch target metadata.

While Boomerang reduces the prefetcher costs to near zero by leveraging existing in-core structures (BTB and branch direction predictor), it has limited effectiveness on workloads with very large instruction working sets. Such workloads result in frequent BTB misses that reduce Boomerang's effectiveness, because instruction prefetching must stall whenever a BTB miss is being resolved to uncover subsequent control flow. As a result, Boomerang captures less than 50% of the opportunity of an ideal front-end prefetcher on workloads with the largest instruction working sets.

This work addresses the key limitation of Boomerang, which is that a limited-capacity BTB simply cannot track a sufficiently large control flow working set to guarantee effective instruction prefetching. Our solution is guided by software behavior. Specifically, we observe that contemporary software is structured as a collection of small functions; within each function, there is high spatial locality for the constituent instruction cache blocks. Short-offset conditional branches steer the *local control flow* between these blocks, while long-offset unconditional branches (e.g., calls, returns), drive the *global control flow* from one function to another.

Using this intuitive understanding, we make a critical insight that an application's instruction footprint can be mapped as a combination of its unconditional branch working set and, for each unconditional branch, a spatial encoding of the cache blocks around the branch target. The combination of unconditional branches and their corresponding

spatial footprints effectively encode the application's control flow across functions and the instruction cache working sets within each function.

Based on these insights, this work introduces *Shotgun*, a BTB-directed front-end prefetcher powered by a new BTB organization specialized for effective prefetching. Shotgun devotes the bulk of its BTB capacity to unconditional branches and their targets' spatial footprints. Using this information, Shotgun is able to track the application's instruction working set at a cache block granularity, enabling accurate and timely BTB-directed prefetching. Moreover, because the unconditional branches comprise just a small fraction of the application's entire branch working set, they can be effectively captured in a practical-sized BTB. Meanwhile, conditional branches are maintained in a separate small-capacity BTB. By exploiting prior observations on control flow commonality in instruction and BTB working sets [14], Shotgun prefetches into the conditional branch BTB by predecoding cache lines brought into the L1-I through the use of spatial footprints. In doing so, Shotgun achieves a high hit rate in the conditional branch BTB despite its small size.

Using a diverse set of server workloads, we make the following contributions:

- Demonstrate that limited BTB capacity inhibits timely instruction prefetching in existing BTB-directed prefetchers. This calls for BTB organizations that can map a larger portion of an application's instruction working set within a limited storage budget.
- Show that local control flow has high spatial locality and a small cache footprint. Given the target of an unconditional branch, on average, over 80% of subsequent accesses (prior to the next unconditional branch) are to cache blocks within 10 blocks of the target. This observation enables a compact spatial encoding of code regions.
- Propose a new BTB organization in which most of the capacity is dedicated to unconditional branches, which steer the global control flow, and spatially-encoded footprints of their target regions. By compactly encoding footprints of entire code regions, the proposed organization avoids the need to track a large number of conditional branches inside these regions to discover their instruction cache working set.
- Introduce Shotgun, a unified instruction cache and BTB prefetcher powered by the proposed BTB organization. By tracking a much larger fraction of an application's instruction footprint within a fixed BTB storage budget, Shotgun outperforms the state-of-the-art BTB-directed front-end prefetcher (Boomerang) by up to 10%.
- To further improve Shotgun's storage efficiency, we explore different alternatives for spatial encoding of target regions and evaluate their performance/storage trade-offs.
- Identify the factors that prevent Shotgun from matching the performance of an ideal server front-end. Based on these factors, we suggest the future research directions to erase the performance difference between an ideal and practical front-end.

2 BACKGROUND

2.1 Temporal streaming prefetching

Over the past decade, *temporal streaming [7]* has been the dominant technique for front-end prefetching for servers. The key principle behind temporal streaming is to record control flow access or miss sequences and subsequently replay them to prefetch the necessary state. The general concept has been applied to both instruction cache [16] and BTB [12] prefetching, and shown to be highly effective in eliminating misses in these structures.

The principal shortcoming of temporal streaming is the need to store large amounts of metadata (hundreds of kilobytes per core) for capturing control flow history [12, 16]. To mitigate the cost, two complementary techniques

have been proposed. The first is sharing the metadata across all cores executing a common workload [13]. The second is using one set of *unified* metadata for both instruction cache and BTB prefetching, thus avoiding the cost and complexity of maintaining two separate control flow histories [14]. The key insight behind unified front-end prefetching is that the metadata necessary for populating the BTB can be extracted from cache blocks containing the associated branch instructions. Thus, history needs to be maintained only for instruction prefetching, while BTB prefetching happens "for free", storage-wise.

The state-of-the-art in temporal streaming combines the two ideas into a unified front-end prefetcher called Confluence [14]. Confluence maintains only the L1-I history metadata for both instruction and BTB prefetching, virtualizes it into the LLC and shares it across the cores executing a common workload. While effective, Confluence introduces a significant degree of cost and complexity into a processor. LLC virtualization requires invasive LLC modifications, incurs extra traffic for metadata movement and necessitates system software support to pin the cache lines containing the history metadata in the LLC. Moreover, the effectiveness of metadata sharing diminishes when workloads are colocated, in which case each workload requires its own metadata, reducing the effective LLC capacity in proportion to the number of colocated workloads.

2.2 BTB-directed prefetching

To mitigate the exorbitant overheads incurred by temporal streaming prefetchers, recent research has revived the idea of BTB-directed (also called fetch-directed) instruction prefetching [9]. The basic idea is to leverage the BTB to discover future branches, predict the conditional ones using the branch direction predictor, and generate a stream of future instruction addresses used for prefetching into the L1-I. The key advantage of BTB-directed prefetching is that it does not require any metadata storage beyond the BTB and branch direction predictor, both of which are already present in a modern server core.

The original work on BTB-directed prefetching was limited to prefetching of instructions. Recent work has addressed this limitation by adding a BTB prefetch capability in a technique called Boomerang [15]. Boomerang uses a basic-block-oriented BTB to detect BTB misses, which it then fills by fetching and decoding the necessary cache lines from the memory hierarchy. By adding a BTB prefetch capability without introducing new storage, Boomerang enables a unified front-end prefetcher at near-zero hardware cost compared to a baseline core.

While highly effective on workloads with smaller instruction working sets, Boomerang's effectiveness is reduced when instruction working sets are especially large. The branch footprint in such workloads can easily exceed the capacity of a typical BTB by an order of magnitude, resulting in frequent BTB misses. Whenever each BTB miss occurs, Boomerang must stall instruction prefetching to resolve the miss and uncover subsequent control flow. When the active branch working set is much larger than the BTB capacity, the BTB will thrash, resulting in a chain of misses whenever control flow transfers to a region of code not in the BTB. Such a cascade of BTB misses impedes Boomerang's ability to issue instruction cache prefetches due to frequently unresolved control flow. Thus, Boomerang's effectiveness is tightly coupled to its ability to capture the control flow in the BTB.

2.3 Competitive Analysis

Figure 1 compares the performance of the state-of-the-art temporal streaming (Confluence) and BTB-directed (Boomerang) prefetchers. Complete workload and simulation parameters can be found in Section 5. As the figure shows, on workloads with smaller instruction working sets, such as Nutch and Zeus, Boomerang matches or outperforms Confluence by avoiding the latter's reliance on the LLC for metadata accesses. In Confluence, the latency of these

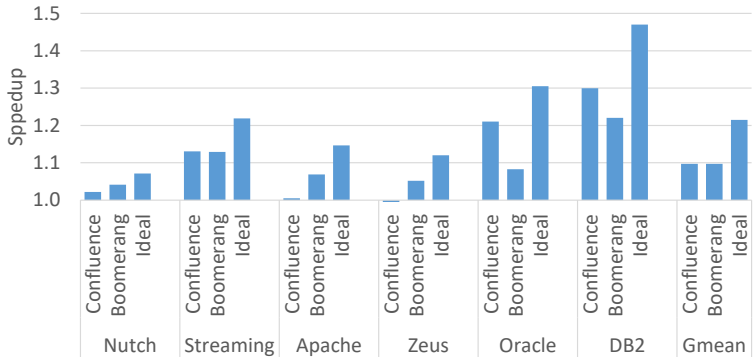

Fig. 1. Comparison of state-of-the-art unified front-end prefetchers to the ideal front-end on server workloads. The baseline core features a fetch-directed instruction prefetcher (FDIP) and a 2K-entry BTB.

accesses is exposed on each L1-I miss, which resets the prefetcher and incurs a round-trip to the LLC to fetch new history before prefetching can resume.

In contrast, on workloads with larger instruction working sets, such as Oracle and DB2, Confluence handily outperforms Boomerang by 13% and 8%, respectively. On these workloads, Boomerang experiences the highest BTB miss rates of any in the evaluation suite (see Table 1), which diminishes prefetch effectiveness as explained in the previous section.

Given that software trends point in the direction of larger code bases and deeper call stacks [5], there is a need for a better control flow delivery architecture that can enable prefetching for even the largest instruction working sets without incurring prohibitive storage and complexity costs.

3 BTB: CODE MEETS HARDWARE

To maximize the effectiveness of BTB-directed prefetching, we next study the interplay between software behavior and the BTB.

3.1 Understanding Control Flow

Application code is typically organized as a collection of functions to increase code reusability and productivity. The function body itself can be thought of as a contiguous region of code that spans a small number of adjacent cache blocks, as small functions are favored by modular design and software engineering principles. To achieve the desired functionality, execution is steered between different code regions through function calls, system calls and the corresponding return instructions; collectively, we refer to these as *global control flow*. Meanwhile, *local control flow*

Workload	MPKI
Nutch	2.5
Streaming	14.5
Apache	23.7
Zeus	14.6
Oracle	45.1
DB2	40.2

Table 1. Miss rate of a 2K-entry BTB without prefetching.

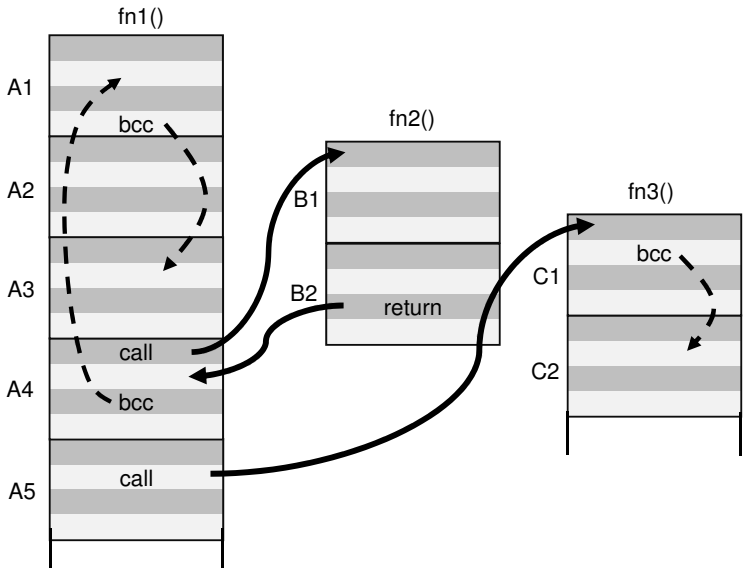

Fig. 2. Program control flow example. The solid arrows represent *global control flow* and dotted arrows depict *local control flow*. A1, B1, etc denote cache block addresses.

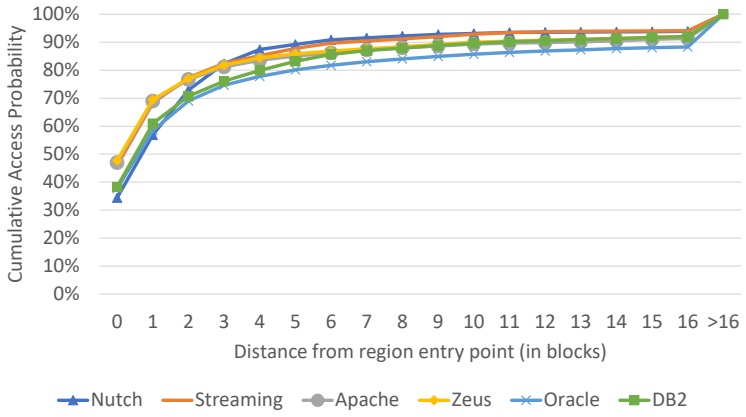

Fig. 3. Instruction cache block access distribution inside code regions.

guides the execution *within* a code region using a combination of conditional branches and fall-through (next sequential instruction) execution.

Figure 2 shows an example of three code regions and the two types of control flow. Global control flow that transfers execution between the regions is depicted by solid arrows, which correspond to *call* and *return* instructions. Meanwhile, *local control flow* transfers due to conditional branches within the code regions are shown with dashed arrows.

Local control flow tends to have high spatial locality as instructions inside a code region are generally stored in adjacent cache blocks. Furthermore, conditional branches that guide local control flow tend to have very short displacements,

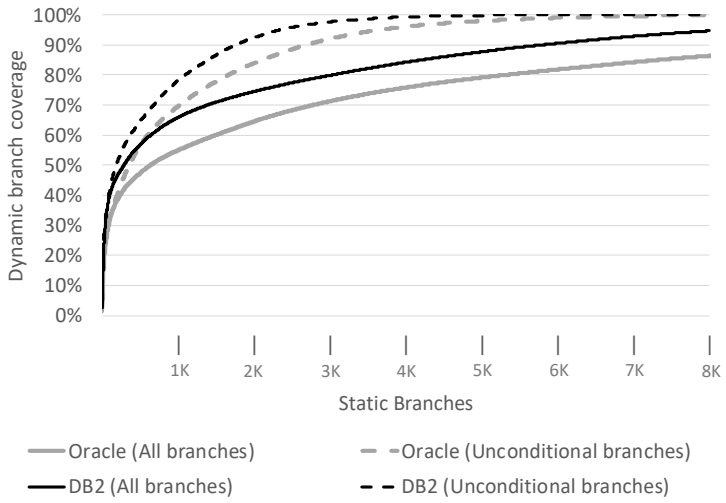

Fig. 4. Contribution of static branches towards dynamic branch execution for Oracle and DB2.

typically within a few cache blocks [15], as shown by dashed arrows in Figure 2. Thus, even for larger functions, there is high spatial locality in the set of instruction cache blocks being accessed within the function.

Figure 3 quantifies the spatial locality for a set of server workloads. The figure shows the probability of an access to a cache block in relation to its distance from an entry point to a code region, where a code region is defined as a set of cache blocks spanning two unconditional branches (region entry and exit points) in dynamic program order. As the figure shows, regions tend to be small and with high spatial locality: 90% of all accesses occur within 10 cache blocks of the region entry point.

Finally, we demonstrate that the total branch working set of server workloads is large but the unconditional branch working set is relatively small. As shown in Figure 4, for Oracle, accommodating 90% of all dynamic branches is not possible even by tracking 8K hottest static branches. With a practical-sized BTB of 2K entries, only 65% of Oracle's dynamic branches can be covered. Meanwhile, the unconditional branch working set, responsible for the global control flow, is rather modest because conditional branches that guide application logic within code regions dominate. On Oracle, a 2K-entry BTB can capture 84% of all dynamically-occurring unconditional branches; increasing the capacity to 2.75K can cover 90% of dynamic unconditional branch executions. The trend is similar on the DB2 workload, for which 2K hottest static branches can cover only 75% of the total dynamic branches, whereas the same number of hottest unconditional branches cover 92% of the unconditional dynamic branches.

3.2 Implications for BTB-directed Prefetching

BTB-directed prefetchers rely on the BTB to discover control flow transfer points between otherwise sequential code sections. Correctly identifying these transfer points is essential for accurate and timely prefetching. Unfortunately, large branch working sets in server workloads cause frequent BTB misses. Existing BTB-directed prefetchers handle BTB misses in one of two ways:

- The original FDIP technique [9] speculates through the misses, effectively fetching straight line code when a branch goes undetected; this, however, is ineffective if the missing branch is a global control flow transfer that redirects execution to a new code region.
- The state-of-the-art proposal, Boomerang, stalls prefetching and resolves the BTB miss by probing the cache hierarchy. While effective for avoiding pipeline flushes induced by the BTB miss, Boomerang is limited in its ability to issue instruction prefetches when faced with a cascade of BTB misses inside a code region as explained in Sec 2.2.

We thus conclude that effective BTB-directed prefetching requires two elements: (1) identifying global control flow transfer points, and (2) racing through local code regions unimpeded. Existing BTB-directed prefetchers are able to achieve only one of these goals at the expense of the other. The next section will describe a new BTB organization that facilitates both of these objectives.

4 SHOTGUN

Shotgun is a unified BTB-directed instruction cache and BTB prefetcher. Its key innovation is using the BTB to maintain a logical map of the program's instruction footprint using software insights from Sec 3. The map allows Shotgun to incur fewer BTB-related stalls while staying on the correct prefetch path, thus overcoming a key limitation of prior BTB-directed prefetchers.

Shotgun devotes the bulk of its BTB capacity to tracking the *global control flow*; this is captured through unconditional branches that pinpoint the inter-region control flow transfers. For each unconditional branch, Shotgun maintains compact metadata to track the spatial footprint of the target region, which enables bulk prefetching of cache blocks within the region. In contrast, prior BTB-directed prefetchers had to discover intra-region control flow by querying the BTB one branch at a time. Because unconditional branches represent a small fraction of the dynamic branch working set and because the spatial footprints summarize locations of entire cache blocks (which are few) and not individual branches (which are many), Shotgun is able to track a much larger instruction footprint than a traditional BTB with the same storage budget.

4.1 Design Overview

Shotgun relies on a specialized BTB organization that judiciously uses the limited BTB capacity to maximize the effectiveness of BTB-directed prefetching. Shotgun splits the overall BTB storage budget into dedicated BTBs for capturing *global* and *local control flow*. *Global control flow* is primarily maintained in the U-BTB, which tracks the unconditional branch working set and also stores the spatial footprints around the targets of these branches. The U-BTB is the heart of Shotgun and drives the instruction prefetch engine. Conditional branches are maintained in the C-BTB, which is comprised of just a few hundred entries to track the *local control flow* within the currently-active code regions. Finally, Shotgun uses a third structure, called Return Instruction Buffer (RIB), to track *return* instructions; while technically part of the global (unconditional) branch working set, *returns* require significantly less BTB metadata than other unconditional branches, so allocating them to a separate structure allows for a judicious usage of the limited BTB storage budget. Figure 5 shows the three BTBs and the per-entry metadata in each of them.

For L1-I prefetching, Shotgun extends Boomerang to leverage the separate BTBs and the spatial footprints as follows: whenever Shotgun encounters an unconditional branch, it reads the spatial footprint of the target region from the U-BTB and issues prefetch probes for the corresponding cache blocks. For filling the BTBs, Shotgun takes a hybrid approach by incorporating the features from both Boomerang [15] and Confluence [14]. Specifically, while prefetching

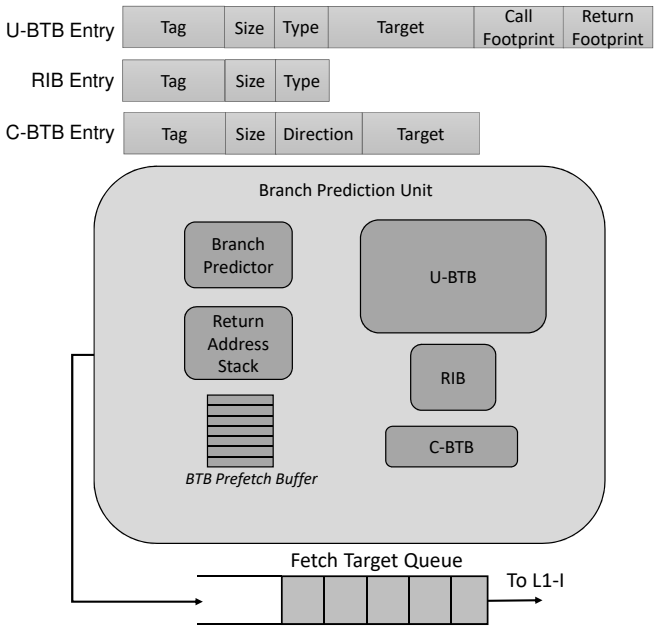

Fig. 5. Shotgun BTB organization.

instruction blocks from LLC, Shotgun leverages the *proactive* BTB fill mechanism of Confluence to predecode the prefetched blocks and fill the BTB before the entries are accessed. Should a BTB miss be encountered by the front-end despite the proactive fill mechanism, it is resolved using the *reactive* BTB fill mechanism of Boomerang that fetches the associated cache block from the memory hierarchy and extracts the necessary branch metadata.

4.2 Design Details

4.2.1 BTB organization. We now detail the microarchitecture of Shotgun's three BTBs shown in Figure 5.

Unconditional branch BTB (U-BTB). The U-BTB tracks the unconditional branch working set, the spatial footprints for the target and, when applicable, return regions of these branches. Because unconditional branches and their spatial footprints are critical for prefetching, Shotgun devotes the bulk of total BTB storage budget to U-BTB.

Each U-BTB entry, as shown in Figure 5, is comprised of the following fields:

Tag: the branch identity.
Size: the size of the basic block containing the branch (like Boomerang, Shotgun uses a basic-block-oriented BTB [17])[1].
Type: the type of branch instruction (call, jump, etc.).
Target: the target address of the branch instruction.
Call Footprint: the spatial footprint for the target region of a call or unconditional jump instruction.
Return Footprint: the spatial footprint for the target region of a return instruction as explained next.

[1]Here, a basic block means a sequence of straight-line instructions ending with a branch instruction; slightly different from a conventional definition of single-entry single-exit straight-line code

Because a function may be called from different sites, the footprint associated with a *return* instruction is call-site-dependent. Meanwhile, tracking potentially many footprints for each *return* instruction is impractical. To resolve this conundrum, Shotgun leverages a simple observation that the target region of a particular instance of a *return* is, in fact, the fall-through region of the preceding *call* (static code region immediately following the *call*). Therefore, Shotgun associates the spatial footprint of the return region with the entry of the corresponding *call* instruction in the U-BTB. To support this design, each U-BTB entry must maintain two spatial footprints; one for the target region of the *call* and the other for the return region.

Return Instruction Buffer (RIB). Shotgun employs a dedicated storage structure, RIB, to track *return* instructions corresponding to function and trap returns. Storing *returns* in the U-BTB along with other unconditional branches would result in severe storage under-utilization because the majority of U-BTB entry space is not needed for *returns*. For example, *returns* read their target address from Return Address Stack (RAS) instead of the Target field of U-BTB entry. Similarly, as discussed above, the spatial footprint for return target region is stored along with the corresponding *call*. Together, these fields (Target, Call Footprint, and Return Footprint) account for more than 50% of a U-BTB entry storage. The impact of such space under-utilization is significant because *returns* occupy a significant fraction of U-BTB entries. Indeed, our studies show that 25% of U-BTB entries are occupied by *return* instructions, hence resulting in storage inefficiency. Note that with a conventional BTB, allocating the *return* instructions into the BTB does not lead to a high inefficiency because over 70% of BTB entries are occupied by conditional branches, while *returns* are responsible for fewer than 10% of all entries.

These observations motivate Shotgun's use of a dedicated RIB structure to track *return* instructions. As shown in Fig 5, each RIB entry contains only (1) Tag, (2) Type, and (3) Size fields. Compared to a U-BTB entry, there are no Target, Call Footprint, and Return Footprint fields in a RIB entry. Thus, by storing only the necessary and sufficient metadata to track *return* instructions, RIB avoids wasting U-BTB capacity.

Conditional branch BTB (C-BTB). Shotgun incorporates a small C-BTB to track the *local control flow* (conditional branches) of currently active code regions. As shown in Fig 5, a C-BTB entry is composed of (1) Tag, (2) Size, (3) Direction, and (4) Target fields. A C-BTB entry does not contain branch Type field as all the branches are conditional. As explained in Section 4.2.3, Shotgun aggressively prefetches into the C-BTB by exploiting spatial footprints, which affords a high hit rate in the C-BTB with a capacity of only a few hundred entries.

4.2.2 Recording spatial footprints. Shotgun monitors the retire instruction stream to record the spatial footprints. As an unconditional branch represents the entry point of a code region, Shotgun starts recording a new spatial footprint on encountering an unconditional branch in the retire stream. Subsequently, it tracks the cache block addresses of the following instructions and adds them to the footprint if not already present. The spatial footprint recording for a code region terminates on encountering a subsequent unconditional branch, which indicates entry to a different code region. Once the recording terminates, Shotgun stores the footprint in the U-BTB entry corresponding to the unconditional branch that triggered the recording.

Spatial footprint format: A naive approach to record a spatial footprint would be to record the full addresses of all the cache blocks accessed inside a code region. Clearly, this approach would result in excessive storage overhead due to the space requirements of storing full cache block addresses. A storage efficient alternative would be to record only the entry and exit points of the region and later prefetch all the cache blocks between these points. However, as not all

the blocks in a region are accessed during execution, prefetching the entire region would result in over prefetching, potentially leading to on-chip network congestion and cache pollution.

To achieve both precision and storage-efficiency, Shotgun leverages the insight that the accesses inside a code region are centered around the target block (first block accessed in the region) as discussed in Sec 3. To exploit the high spatial locality around the target block, Shotgun uses a short bit-vector, where each bit corresponds to a cache block, to record spatial footprints. The bit positions in the vector represent the relative distance from the target block and the bit value (1 or 0) indicates whether the corresponding block was accessed or not during the last execution of the region. Thus, by using a single bit per cache block, Shotgun dramatically reduces storage requirements while avoiding over prefetching.

4.2.3 Prefetching with Shotgun. Similar to FDIP [9], Shotgun also employs a Fetch Target Queue(FTQ), as shown in Figure 5, to hold the fetch addresses generated by the branch prediction unit. These addresses are later consumed by the fetch-engine to fetch and feed the corresponding instructions to core back-end. To fill the FTQ, the branch prediction unit of Shotgun queries all three BTBs (U-BTB, C-BTB, and RIB) in parallel. If there is a hit in any of the BTBs, the appropriate fetch addresses are inserted in to the FTQ. As these addresses are eventually going to be used for fetching instructions from L1-I, they represent natural prefetching candidates. Therefore, like FDIP, Shotgun capitalizes on this opportunity by scanning through the fetch addresses, as they are inserted into the FTQ, and issuing prefetch probes for corresponding L1-I blocks.

On a U-BTB or RIB hit, Shotgun also reads the spatial footprint of the target code region to issue L1-I prefetch probes for appropriate cache blocks. Accessing the spatial footprint is simple for U-BTB hits because it is directly read from the Call Footprint field of the corresponding U-BTB entry. However, the mechanism is slightly more involved on RIB hits because the required spatial footprint is not stored in RIB, rather in the U-BTB entry of the corresponding *call*. To find this U-BTB entry, we extend the RAS such that on a *call*, in addition to the return address that normally gets pushed on the RAS, the address of basic block containing the *call* is also pushed[2]. Because the RAS typically contains a small number of entries (8-32 is common), the additional RAS storage cost to support Shotgun is negligible. On a RIB hit for a *return* instruction, Shotgun pops the basic block address of the associated *call* from the RAS to index the U-BTB and retrieve the spatial footprint from the Return Footprint field.

In addition to using the spatial footprint to prefetch instructions into the L1-I, Shotgun exploits control flow commonality [14] to prefetch into the C-BTB as well. Thus, when the prefetched blocks arrive at the L1-I, Shotgun uses a set of predecoders to extract branch metadata from them and uses it to populate the C-BTB ahead of the access stream. By anticipating the upcoming instruction working set via the spatial footprints and prefetching its associated branch working set into the C-BTB via predecoding, Shotgun affords a very small yet highly effective C-BTB.

Figure 6 shows an example of using a spatial footprint for L1-I and C-BTB prefetching on a U-BTB hit. Shotgun first reads the target address *A* and the call footprint *01001000* from the U-BTB entry. It then generates prefetch probes to the L1-I for the target block *A* and, based on the call footprint in the U-BTB entry, for cache blocks *A+2* and *A+5* (step ①). If any of these blocks are not found in the L1-I, Shotgun issues prefetch request(s) to the LLC (step ②). Once prefetched blocks arrive from the LLC, they are installed in the L1-I (step ③) and are also forwarded to a predecoder (step ④). The predecoder extracts the conditional branches from the prefetched blocks and inserts them into the C-BTB (step ⑤).

If Shotgun detects a miss in all three BTBs, it invokes Boomerang's BTB fill mechanism to resolve the miss in the following manner: first, the instruction block corresponding to the missed branch is accessed from L1-I or from lower

[2]Because Shotgun uses a basic-block oriented BTB, it is the basic block address, and not the PC, corresponding to the *call* instruction that is stored on the RAS.

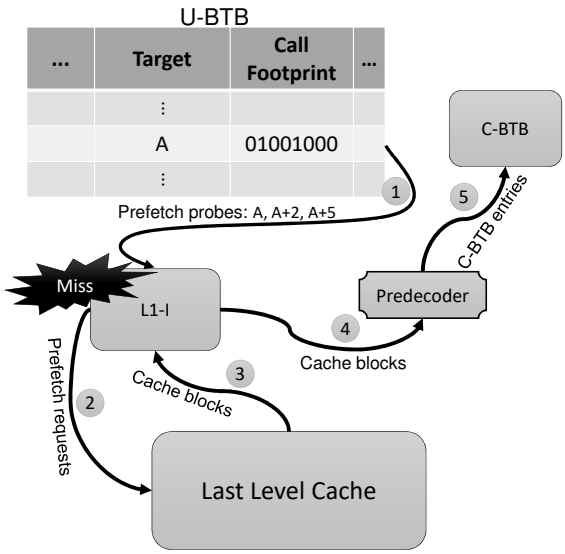

Fig. 6. Shotgun prefetching using spatial footprints.

cache levels if not present in the L1-I. The block is then fed to the predecoder that extracts the missing branch and stores it into one the BTBs depending on branch type. The rest of the predecoded branches are stored in the BTB Prefetch Buffer [15]. On a hit to the BTB Prefetch Buffer, the accessed branch is moved to the appropriate BTB based on the branch type.

Discussion:

Populating FTQ entries - Shotgun vs Boomerang: On a BTB hit, both Shotgun and Boomerang insert the fetch addresses indicated by the BTB entry into the fetch target queue (FTQ). However, their functionality differs on BTB misses. Though it is possible to speculatively insert fall-through addresses to FTQ on a BTB miss, Boomerang stops populating the FTQ until the BTB miss is resolved. This helps to avoid pipeline squashes in case the missed branch is eventually taken. However, as L1-I prefetching also stops when no new FTQ entries are being populated, Boomerang misses potential prefetching opportunities especially if the missed branch turns out to be not taken. To compensate for this missed opportunity, Boomerang prefetches next two sequential blocks on BTB misses that are filled from LLC or memory.

Shotgun, in contrast, speculatively inserts a single fall-though address to FTQ every cycle until the next BTB hit and then starts running at basic block granularity again. Such an approach not only continues to provide instruction prefetch targets under a BTB miss, as it keeps inserting new addresses to FTQ, but also provides two additional advantages. First, Shotgun can potentially discover and fill multiple BTB misses in parallel. Second, and more important, it increases the likelihood of getting the next U-BTB hit sooner; hence, prefetching the next spatial region early. To avoid pipeline squashes whenever fall-through is the wrong path, Shotgun marks the speculatively inserted addresses and does not allow the fetch engine to read them until their corresponding BTB miss is resolved. In other words, instead of stalling fetch address insertion into the FTQ, Shotgun stalls instruction fetch. After BTB miss resolution, the branch is predicted and all addresses after the speculatively inserted address are flushed if the predicted direction is *taken*; otherwise the addresses become non-speculative and the fetch-engine is allowed to read them for fetching instructions from L1-I. To

Web Search	
Nutch	Apache Nutch v1.2 230 clients, 1.4 GB index, 15 GB data segment
Media Streaming	
Darwin	Darwin Streaming Server 6.0.3 7500 clients, 60GB dataset, high bitratez
Web Frontend (SPECweb99)	
Apache	Apache HTTP Server v2.0 16K connections, fastCGI, worker threading model
Zeus	Zeus Web Server 16K connections, fastCGI
OLTP - Online Transaction Processing (TPC-C)	
Oracle	Oracle 10g Enterprise Database Server 100 warehouses (10GB), 1.4 GB SGA
DB2	IBM DB2 v8 ESE Database Server 100 warehouses (10GB), 2GB buffer pool

Table 2. Workloads

Processor	16-core, 2GHz, 3-way OoO 128 ROB, 32 LSQ
Branch Predictor	TAGE [20] (8KB storage budget)
Branch Target Buffer	2K-entry
L1 I/D	32KB/2way, 2-cycle, private 64-entry prefetch buffer
L2 NUCA cache	shared, 512KB per core, 16-way, 5-cycle
Interconnect	4x4 2D mesh, 3 cycles/hop
Memory latency	45 ns

Table 3. Microarchitectural parameters

summarize, speculatively inserting fetch addresses to FTQ under a BTB miss ensures continuous generation of prefetch targets, whereas restricting the fetch engine to fetch the corresponding instructions only after resolving the BTB miss minimizes the pipeline squashes.

5 METHODOLOGY

5.1 Simulation Infrastructure

We use Flexus [18], a full system multiprocessor simulator, to evaluate Shotgun on a set of enterprise and open-source scale-out applications listed in Table 2. Flexus, which models SPARC v9 ISA, extends the Simics functional simulator with out-of-order(OoO) cores, memory hierarchy, and on-chip interconnect. We use SMARTS [19] multiprocessor sampling methodology for sampled execution. Samples are drawn over 32 billion instructions (2 billion per core) for each application. At each sampling point, we start cycle accurate simulation from checkpoints that include full architectural and partial microarchitectural state consisting of caches, BTB, branch predictor, and prefetch history tables. We warm-up the system for 100K cycles and collect statistics over the next 50K cycles. We use the ratio of number of application instructions to the total number of cycles (including the cycles spent executing operating system core) to measure performance. This metric has been shown to be an accurate measure of server throughput [18].

Our modeled processor is a 16-core tiled CMP. Each core is 3-way out-of-order that microarchitecturally resembles an ARM Cortex-A57 core. The microarchitectural parameters of the modeled processor are listed in Table 3. We assume a 48-bit virtual address space.

5.2 Control Flow Delivery Mechanisms

We compare the efficacy and storage overhead of the following state-of-the-art control flow delivery mechanisms.

Confluence: Confluence is the state-of-the-art temporal streaming prefetcher that uses unified metadata to prefetch into both L1-I and BTB [14]. To further reduce metadata storage costs, Confluence virtualizes the history metadata into the LLC using SHIFT [13]. We model Confluence as SHIFT augmented with a 16K-entry BTB, which was shown to provide a generous upper bound on Confluence's performance [14]. To provide high L1-I and BTB miss coverage, Confluence requires at least a 32K-entry instruction history and an 8K-entry index table, resulting in high storage overhead. Furthermore, it adds significant complexity to the processor as it requires LLC tag extensions, reduction in effective LLC capacity, pinning of metadata cache lines in the LLC and the associated system software support, making it an expensive proposition as shown in prior work [15]. The LLC tag array extension, for storing index table, costs 240KB of storage overhead, whereas the history table for each colocated workload require 204KB of storage which is carved out from LLC capacity.

Boomerang: As described in Section 2.2, Boomerang employs FDIP for L1-I prefetching and augments it with BTB prefilling. Like FDIP, Boomerang employs a 32-entry fetch target queue (FTQ) to buffer the instruction addresses before they are consumed by the fetch engine. We evaluate Boomerang with a 2K entry basic-block oriented BTB. Each BTB entry consists of a 37-bit tag, 46-bit target address, 5 bits for basic-block size, 3 bits for branch type (conditional, unconditional, call, return, and trap return), and 2 bits for conditional branch direction prediction. In total, each BTB entry requires 93 bits leading to an overall BTB storage cost of 23.25KB. Also, our evaluated Boomerang design employs a 32-entry BTB prefetch buffer.

Shotgun: As described in Section 4.2, Shotgun uses dedicated BTBs for unconditional branches, conditional branches, and returns. For a fair comparison against Boomerang, we restrict the combined storage budget of all BTB components in Shotgun to be identical to the storage cost of Boomerang's 2K-entry BTB. Like Boomerang, Shotgun also employs a 32-entry FTQ and a 32-entry BTB prefetch buffer.

U-BTB storage cost: We evaluate a 1.5K (1536) entry U-BTB, which accounts for the bulk of Shotgun's BTB storage budget. Each U-BTB entry consists of a 38-bit tag, 46-bit target, 5 bits for basic-block size, and 1 bit for branch type (unconditional or call). Furthermore, each U-BTB entry also consists of two 8-bit vectors for storing spatial footprints. In each spatial footprint, 6 of the 8 bits are used to track the cache blocks after the target block and the other two bits for the blocks before the target block. Overall, each U-BTB entry costs 106 bits, resulting in a total storage of 19.87KB.

C-BTB storage cost: Since Shotgun fills C-BTB from L1-I blocks prefetched via U-BTB's spatial footprints, only a small fraction of overall BTB storage is allocated to C-BTB. We model a 128-entry C-BTB with each C-BTB entry consisting of a 41-bit tag, 22-bit target offset, 5 bits for basic-block size, and 2 bits for conditional branch direction prediction. Notice that only a 22-bit target offset is needed, instead of the complete 46-bit target address, as conditional branches always use PC relative offsets and SPARC v9 ISA limits the offset to 22-bits. Also, as C-BTB stores only the conditional branches, the branch type field is not needed. Overall, the 128-entry C-BTB requires 1.1KB of storage.

RIB storage cost: We model a 512-entry RIB, with each entry containing a 39-bit tag, 5 bits for basic-block size, and 1 bit for branch type (return or trap-return). Since *return* instructions get their target from the RAS, the RIB does not store target addresses (Section 4.2). With 45 bits per each RIB entry, a 512-entry RIB requires 2.8KB of storage.

Total: The combined storage cost of U-BTB, C-BTB and RIB is 23.77KB.

6 EVALUATION

In this section, we first evaluate Shotgun's effectiveness in eliminating front-end stall cycles, and the corresponding performance gains in comparison to temporal streaming (Confluence) and BTB-directed (Boomerang) control flow delivery mechanisms. Next, we evaluate the key design decisions taken in Shogun's microarchitectural design; we then

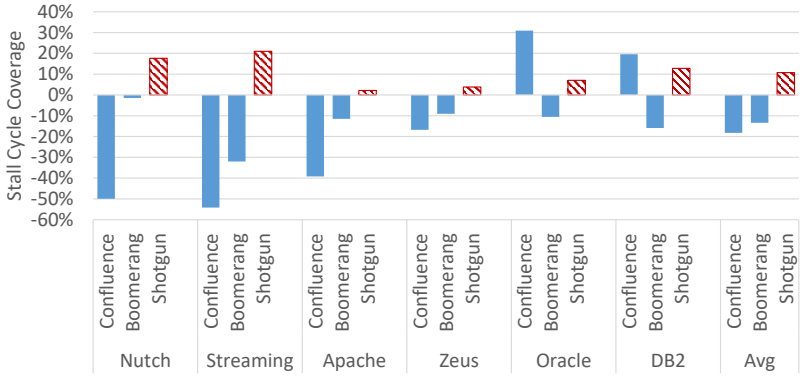

Fig. 7. Front-end stall cycles covered by different prefetching schemes over FDIP baseline.

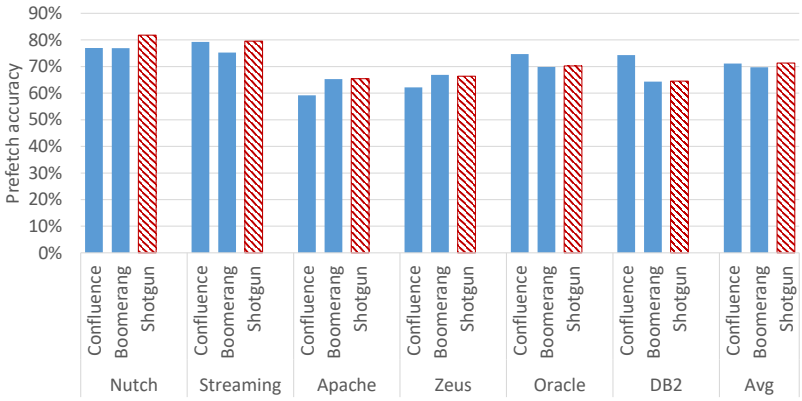

Fig. 8. Prefetch accuracy for different prefetching schemes.

analyze Shotgun's sensitivity to BTB storage budget. Finally, we study the factors that prevent Shotgun from matching the performance of an ideal server front-end.

6.1 Front-end stall cycle coverage and prefetch accuracy

To assess the efficacy of different prefetching mechanisms, we present the number of front-end stall cycles covered by each of them over the baseline fetch directed instruction prefetcher (FDIP) in Figure 7. Notice that instead of using the more common *misses covered* metric, we use *stall cycles covered*; that way, we can precisely capture the impact of *in-flight prefetches*: the ones that have been issued, but the requested block has not yet arrived in L1-I when needed by the fetch unit. Furthermore, we consider stall cycles only on the correct execution path, since wrong-path stalls do not affect performance.

On average, as shown in the Figure 7, Shotgun covers about 11% of the stall cycles experienced by the baseline FDIP prefetcher. Confluence and Boomerang, in contrast, incur about 18% and 14% more front-end stalls than FDIP, as also observed by prior work [15]. Though Boomerang builds on FDIP, the latter provides better stall coverage because it speculatively prefetches along the fall-through path on BTB misses, whereas Boomerang pauses prefetching while it

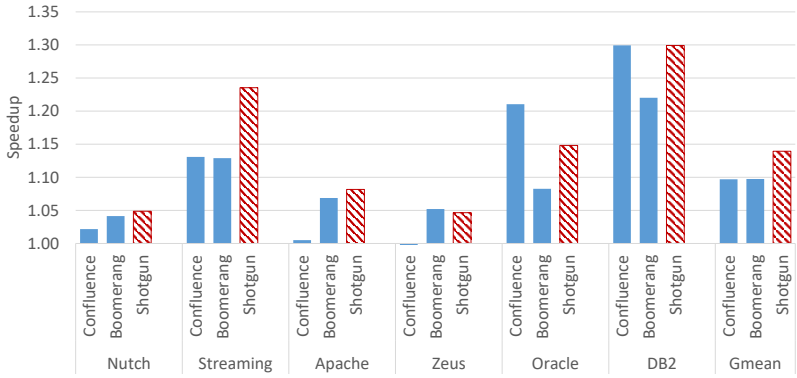

Fig. 9. Speedup of different prefetching schemes over FDIP baseline.

resolves the BTB misses. The speculative prefetching improves coverage if the fall-through path eventually turns out to be the correct path or if fall-through and taken paths converge quickly. However, the higher stall cycle coverage does not translate to better performance for FDIP as discussed in the next Section.

A closer inspection of Figure 7 reveals that Shotgun outperforms its direct rival Boomerang on all of the workloads by a significant margin. In particular, Shotgun provides nearly 18% and 29% more coverage than Boomerang on Oracle and DB2 – the workloads with a high BTB MPKI whose impact on front-end performance Shotgun aims to mitigate. Shotgun's improved coverage is a direct outcome of uninterrupted L1-I prefetching via U-BTB's spatial footprints; in contrast, Boomerang has to wait to resolve BTB misses.

Compared to Confluence, Shotgun provides better stall coverage on four out of six workloads. A closer inspection reveals that Shotgun comprehensively outperforms Confluence on Apache, Nutch, and Streaming with 41%-75% additional coverage. Confluence performs poorly on these applications, as also noted by Kumar et al. [15], owing to frequent LLC accesses for loading history metadata. On every misprediction in L1-I access sequence, Confluence needs to load the correct sequence from the LLC before starting issuing prefetches on the correct path. This start-up delay in issuing prefetches on each new sequence compromises Confluence's coverage.

On the workloads with the highest BTB MPKI (DB2 and Oracle), Shotgun is within 7% of Confluence on DB2, but is 24% behind on Oracle. As shown in Figure 4, Oracle's unconditional branch working set is much larger compared to other workloads. The most frequently executed 1.5K unconditional branches (equal to the number of Shotgun's U-BTB entries) cover only 78% of dynamic unconditional branch execution. Therefore, Shotgun often enters code regions not captured by U-BTB, which limits the coverage due to not having a spatial footprint to prefetch from.

Figure 8 presents the prefetch accuracy of the evaluated techniques. We define prefetch accuracy as the ratio of useful prefetches to total number of prefetches generated by a prefetcher. As the figure shows, all three prefetching techniques exhibit similar levels of accuracy with Confluence, Boomerang, and Shotgun being 71.10%, 69.75% and 71.32% accurate.

6.2 Performance Analysis

Figure 9 shows the performance improvements for different prefetching mechanisms over the baseline FDIP prefetching. The performance trends are similar to coverage trends (Figure 7) with Shotgun providing, on average, 14% performance improvement over the baseline and 4% improvement over each of Boomerang and Confluence. The speedup over

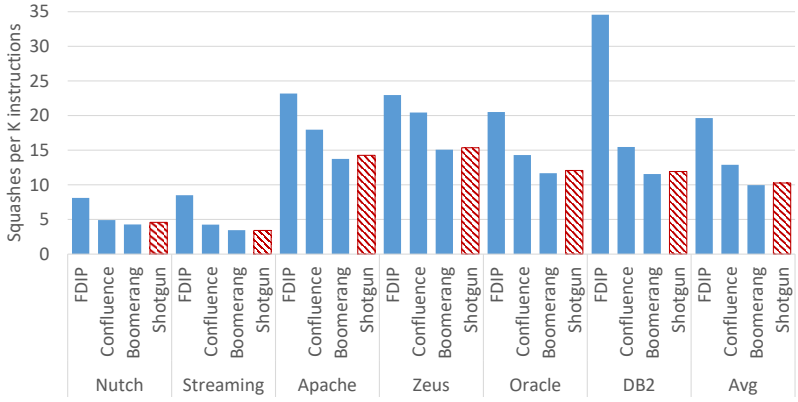

Fig. 10. Number of pipeline squashes per kilo instructions experienced by different techniques.

Boomerang is especially prominent on high BTB MPKI workloads, Oracle and DB2, where Shotgun achieves 7% and 8% improvement respectively.

Interestingly, Figure 9 shows that Shotgun attains a relatively modest performance gain over Boomerang on Nutch, Apache, and Zeus workloads, despite its noticeable coverage improvement. The reason behind this behavior is that these workloads have relatively low L1-I MPKI; therefore, the coverage improvement does not translate into proportional performance improvement. Similar to coverage results, Shotgun outperforms Confluence on Apache, Nutch, Streaming, and Zeus. Furthermore, it matches the performance gain of Confluence on DB2; however, due to lower stall cycle coverage, Shotgun falls behind Confluence on Oracle by 6%.

It is also interesting to note that both Confluence and Boomerang provide, on average, about 10% performance improvement over the baseline FDIP despite incurring more front-ends stalls as observed in Figure 7. This is because they both reduce pipeline squashes stemming from BTB misses in addition to covering L1-I miss induced stalls. FDIP, in contrast, is exclusively aimed at minimizing L1-I miss stalls but not the squashes caused by BTB misses. Consequently, as shown in Figure 10, it experiences about 20 squashes per thousand instructions compared to only 13 and 10 of Confluence and Boomerang respectively. By reducing the amount of pipeline squashes, Confluence and Boomerang are able to provide higher performance even with lower L1-I miss stall coverage.

Recall that FDIP, Boomerang, and Shotgun rely solely on BTB and branch predictor for prefetching, whereas Confluence needs dedicated storage for keeping the prefetch metadata. In our evaluation, we allocate similar BTB storage budgets for FDIP, Boomerang, and Shotgun (23.25KB, 23.25KB and 23.77KB respectively). Therefore, the comparison among these techniques is ISO-storage. We give Confluence a storage advantage in that its metadata storage (which is cache-resident) is not counted. For ISO-storage comparison with other techniques, we would need to drastically reduce Confluence's metadata and BTB budgets. However, this would greatly reduce Confluence's performance gain, which is already lower than Shotgun.

6.3 Quantifying the Impact of Spatial Footprints

As discussed in Sec 4.2.2, Shotgun stores the spatial region footprints in the form of a bit-vector to reduce the storage requirements while simultaneously avoiding over prefetching. This section evaluates the impact of spatial footprints and their storage format (bit-vector) on performance. We evaluate the following spatial region prefetching mechanisms:

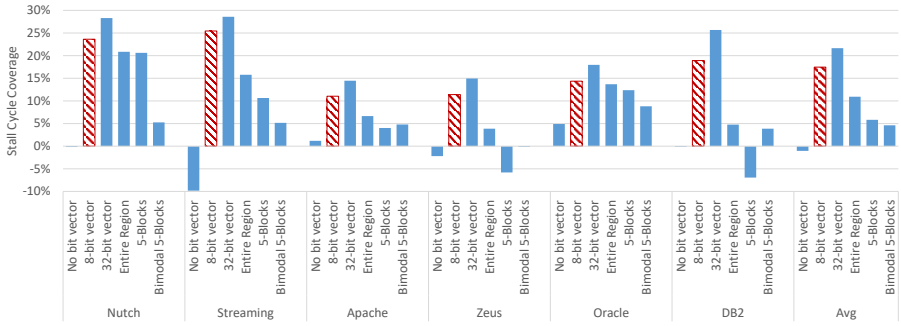

Fig. 11. Shotgun front-end stall cycle coverage over FDIP with different spatial region prefetching mechanisms.

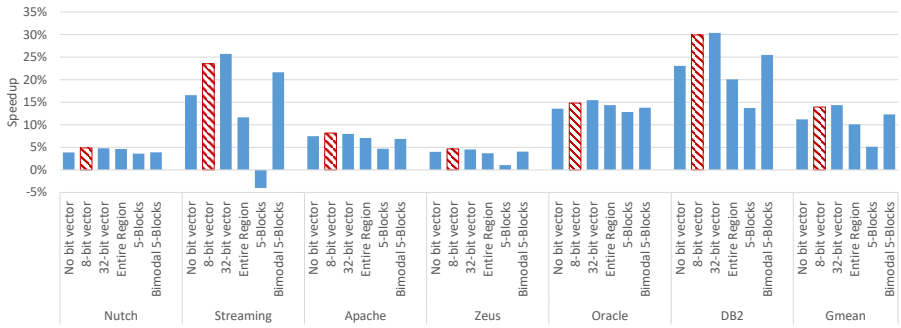

Fig. 12. Shotgun performance gain over FDIP with different spatial region prefetching mechanisms.

(1) No bit vector: does not perform any region prefetching; (2) 8-bit vector; (3) 32-bit vector; (4) Entire Region: prefetches all the cache blocks between entry and exit points of the target region; (5) 5-Blocks: prefetches five consecutive cache blocks in the target region starting with the target block. The "5-Blocks" design point is motivated by Figure 3, which shows that 80%-90% of the accessed blocks lie within this limit. The benefit of always prefetching a fixed number of blocks is that it completely avoids the need to store metadata for prefetching; and (6) Bimodal 5-blocks: prefetches five consecutive cache blocks in the target region only if at least three of these blocks were accessed during the last execution of the region; if fewer than three blocks were accessed, does not prefetch anything[3]. The advantage of the Bimodal approach is that it requires only a single bit to decide whether or not to prefetch. This bit is set/reset after executing a region based on the number of cache blocks accessed during the execution.

First, we focus on the stall cycle coverage and performance with different bit-vector lengths. For the No Bit Vector design, which performs no region prefetching, we increase the number of entries in the U-BTB up to the same storage budget as the 8-bit vector design. For the 32-bit vector, however, instead of reducing the number of U-BTB entries (to account for more bits in bit-vector), we simply provide additional storage to accommodate the larger bit-vector. Therefore, the results for 32-bit vector upper-bound the benefits of tracking a larger spatial region with the same global control flow coverage in the U-BTB as the 8-bit vector design.

As Figures 11 and 12 show, an 8-bit vector provides, on average, 18% coverage and 3% performance benefit compared to no spatial region prefetching. In fact, without spatial footprints, Shotgun provides only 1.5% better performance than

[3]We experimented with both the number of cache blocks to be prefetched in a region and the number required to enable prefetching. The chosen configuration provided the best performance.

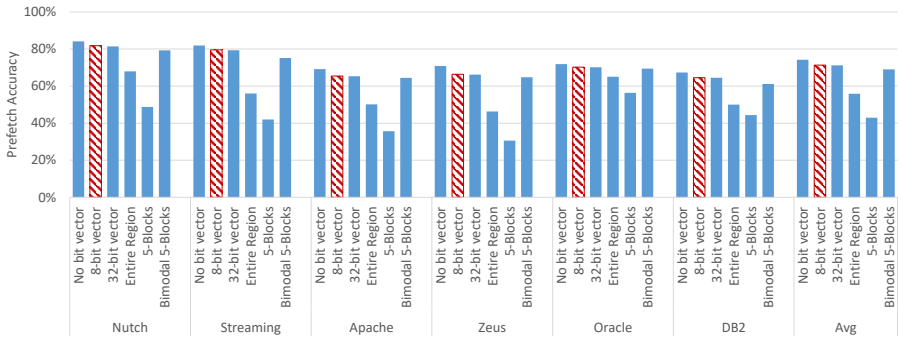

Fig. 13. Shotgun prefetch accuracy with different spatial region prefetching mechanisms.

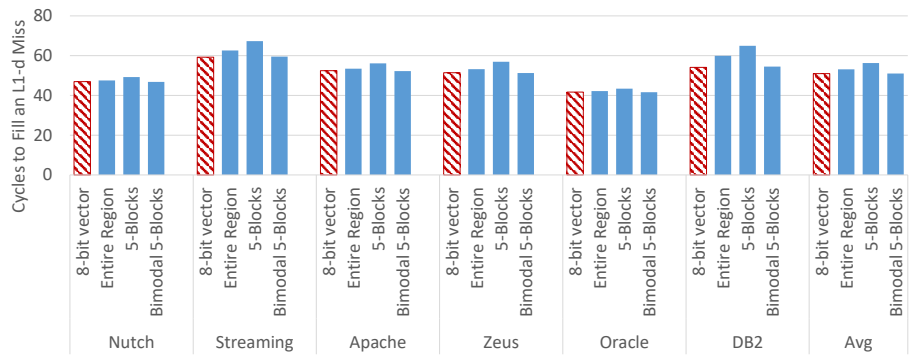

Fig. 14. Number of cycles required to fill an L1-D miss with different mechanisms for spatial region prefetching.

Boomerang. With an 8-bit vector, Shotgun improves the performance of every single workload, with the largest gain of 7% on Streaming and DB2, compared to No Bit Vector. Meanwhile, increasing the bit-vector length to 32 bits provides only 0.5% performance, on average, over an 8-bit vector. These results suggest that longer bit vectors do not offer a favorable cost/performance trade-off.

The next two spatial region prefetching mechanisms, Entire Region and 5-Blocks, lead to significant performance degradation compared to 8-bit vector as shown in Figure 12. The performance penalty is especially severe in two of the high opportunity workloads: DB2 and Streaming. This performance degradation results from over-prefetching, as these mechanisms lack the information about which blocks inside the target region should be prefetched. Always prefetching 5 blocks from the target region results in significant over prefetching and poor prefetch accuracy, as shown in Figure 13, because many regions are smaller than 5 blocks. The reduction in prefetch accuracy is especially severe in Streaming where it goes down to mere 42% with 5-Block prefetching compared to 80% with 8-bit vector. On average, 8-bit vector provides 71% accuracy whereas, Entire Region and 5-Blocks prefetching are only 56% and 43% accurate, respectively. Over-prefetching also increases pressure on the on-chip network, which in turn increases the effective LLC access latency, as shown in Figure 14. For example, as the figure shows, average latency to fill an L1-D miss increases from 54 cycles with 8-bit vector to 65 cycles with 5-Blocks prefetching for DB2. The combined effect of poor accuracy and

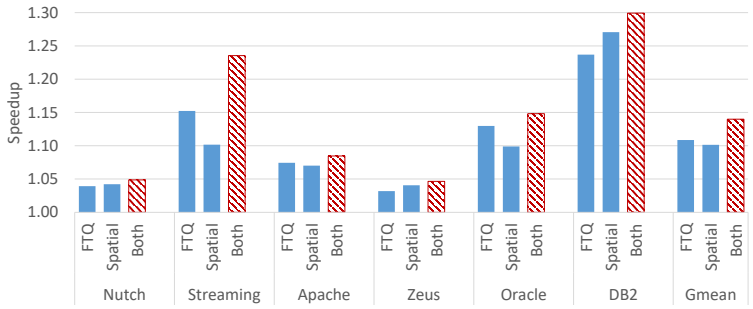

Fig. 15. Performance comparison of FTQ and Spatial Footprint prefetching.

increased LLC access latency due to over-prefetching makes indiscriminate region prefetching less effective than the 8-bit vector design.

To avoid the performance loss due to over prefetching, the final alternative, Bimodal 5-blocks, prefetches five consecutive cache blocks in the target region if at least three of these blocks were accessed during the last execution; otherwise, spatial prefetching is not triggered. As shown in Figure 13, the Bimodal design achieves good accuracy, approaching that of the 8-bit vector approach. However, disabling the spatial prefetching when only a few cache blocks are accessed in a region also limits coverage. As a result, Bimodal 5-blocks provides 12% lower coverage than 8-bit vector as shown in Figure 11. Due to the lower coverage, Bimodal 5-blocks delivers only 12.3% performance gain, in comparison to 14% of 8-bit vector, over the baseline. Compared to Entire Region and 5-Blocks prefetching, on average, Bimodal 5-blocks outperforms them, as shown in Figure 12, by virtue of its higher accuracy despite similar or lower coverage.

6.4 Breaking Down the Performance Contribution of Different Shotgun Components

Shotgun features two components for each of L1-I prefetching (FTQ and Spatial prefetching) and BTB prefilling (Proactive and Reactive prefilling). We next quantify the contribution of each of these components towards overall performance. To do so, we disable the component under consideration and measure the resulting performance loss.

6.4.1 L1-I Prefetching: FTQ vs Spatial prefetching. As detailed in Section 4.2.3 Shotgun employs two mechanisms for L1-I prefetching. First, Shotgun leverages fetch addresses inserted into the FTQ for L1-I prefetching. As these addresses are eventually going to be consumed by the fetch engine for fetching instructions, they represent natural prefetching candidates. Second, Shotgun also stores the spatial footprints of code regions in U-BTB so that it can later prefetch the whole region on U-BTB and RIB hits.

Figure 15 presents the performance of these individual prefetching mechanisms and how they fair when combined together. The results shows that individually, these techniques perform similar to each other, on average; however, using both together is more effective than using them individually. Specifically, Spatial and FTQ prefetching deliver 10% and 11% performance gain over the baseline, respectively. When combined, the performance gain increases to 14%. This is because, when combined, each technique helps capture the opportunities missed by the other. For example, if the spatial footprint misses a cache block, either because it was not accessed during the last execution or because the bit vector is too small to capture it, FTQ prefetching can potentially prefetch it once the corresponding fetch addresses are inserted into the FTQ. Meanwhile, spatial prefetching helps by bulk-prefetching of all the cache blocks in the spatial

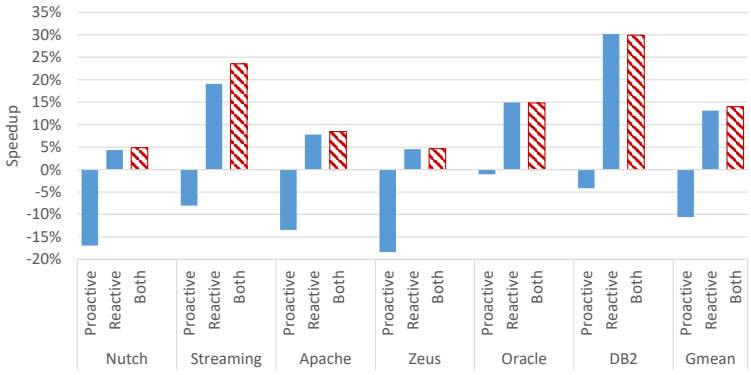

Fig. 16. Performance comparison of proactive and reactive BTB prefilling.

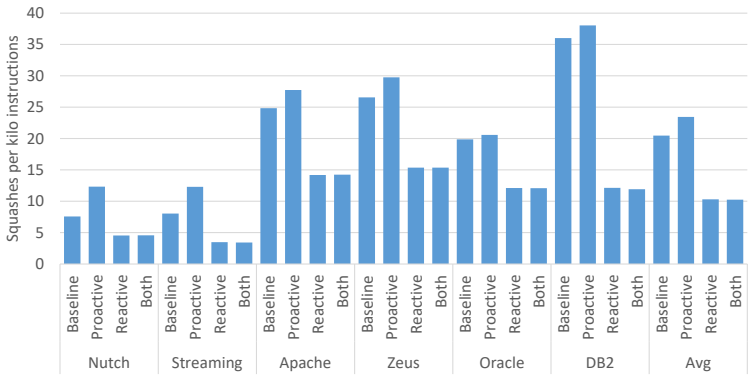

Fig. 17. Pipeline squashes per kilo instructions.

footprint at once, whereas FTQ prefetching would have had to insert each of the prefetch targets into the FTQ one by one, hence hurting timeliness compared to spatial prefetching.

In summary, both L1-I prefetching techniques are essential for Shotgun to maximize the performance due to their complementary nature.

6.4.2 BTB Prefilling: Proactive vs Reactive. As described in Section 4.1, Shotgun incorporates both proactive and reactive BTB prefilling mechanisms. Specifically, while prefetching instruction cache blocks from LLC, the proactive mechanism predecodes the prefetched blocks and fill the BTB before the entries are accessed. The reactive mechanism, in contrast, kicks in when a required entry is not found in BTB, i.e. on a BTB miss. It resolves the miss by fetching the associated cache block from the memory hierarchy and extracting the necessary branch metadata.

Figure 16 compares the performance of proactive and reactive BTB prefilling mechanisms. As the results show, the proactive prefilling performs very poorly by itself, resulting in a performance loss of 11% over the baseline FDIP. The performance degradation is especially severe on Nutch and Zeus that show 17% and 18% slowdown respectively. These results are in stark contrast to the ones presented in Confluence [14], where proactive BTB prefilling was shown to provide a significant performance boost. The reason for the difference in behavior is the very small size of the

C-BTB (128 entries) in Shotgun. The key principle behind the proactive BTB prefilling is to keep the active branches of L1-I resident cache blocks in the BTB. To achieve that, Confluence sizes its BTB in proportion to the L1-I capacity. However, the small C-BTB of Shotgun is incapable of holding the conditional branches corresponding to the entire L1-I. Therefore, in the absence of reactive BTB prefilling, C-BTB misses result in frequent pipeline squashes that lead to poor performance. Figure 17 validates this reasoning by showing that the proactive-only prefilling incurs much higher squashes per kilo-instructions than the other BTB filling mechanisms and even the baseline.

The reactive BTB prefilling, in contrast, performs very well by itself and its performance gain is within 1% of a design that features both proactive and reactive mechanisms as shown in Figure 16. This is because the reactive mechanism detects and resolves each BTB miss which minimizes the number of pipeline squashes[4] and improves the performance. Notice that the reactive BTB prefilling mechanism of Shotgun is exactly the same as used by Boomerang. However, Shotgun pays a much smaller BTB miss penalty, i.e. the number of cycles required to fill a missed BTB entry, than Boomerang. This is because Shotgun uses its spatial footprint to prefetch the required cache blocks within the spatial region before execution enters that region. As a result, BTB misses in Shotgun are likely to be filled from L1-I as the corresponding cache blocks have already been prefetched. Boomerang, in contrast, issues a prefetch only after detecting a BTB miss, which results in a larger latency penalty as the required cache block often resides in a lower cache level. Indeed, our results show that, on average, Shotgun needs only 10 cycles to fill a BTB miss compared to 15 cycles required by Boomerang. Thus, by reducing the BTB miss penalty, Shotgun with only reactive BTB prefilling significantly outperforms Boomerang.

To summarize, proactive BTB prefilling alone degrades performance, compared to baseline, due to frequent pipeline squashed caused by C-BTB thrashing. The reactive mechanism, in contrast, minimizes the squashes by resolving every BTB miss and thus provides high performance. Though the reactive BTB prefilling provides most of the performance benefits by itself, complementing it with a proactive mechanism comes for “free" in terms of required hardware. This is because the proactive prefilling leverages the same hardware (e.g., predecoder) as employed by the reactive one. Also, it delivers noticeable additional performance (nearly 5%) on Streaming.

6.5 Sensitivity to C-BTB Size

As discussed in Sec 4, Shotgun incorporates a small C-BTB and relies on both proactive and reactive mechanisms to fill it ahead of time. To measure Shotgun’s effectiveness in prefilling the C-BTB, Fig 18 presents performance sensitivity to the number of C-BTB entries. Any speedup with additional entries would highlight the opportunity missed by Shotgun.

To assess Shotgun’s effectiveness, we compare the performance of 128-entry verses 1K-entry C-BTBs. As the figure shows, despite an 8x increase in storage, the 1K entry C-BTB delivers, on average, only 0.7% improvement. This result validates our design choice, demonstrating that a larger C-BTB capacity is not useful.

On the other hand, reducing the number of entries to 64 results in noticeable performance loss especially on Streaming and DB2, with 3% lower performance compared to a 128-entry C-BTB. On average, the 128-entry C-BTB outperforms the 64-entry C-BTB by about 1.5% as shown in Figure 18.

6.6 Sensitivity to the BTB Storage Budget

We now investigate the impact of the BTB storage budget on the effectiveness of the evaluated BTB-directed prefetchers: Boomerang and Shotgun. We vary the BTB capacity from 512 entries to 8K entries for Boomerang, while using the

[4]The remaining squashes come from conditional branch direction misprediction and indirect branch target mispredictions.

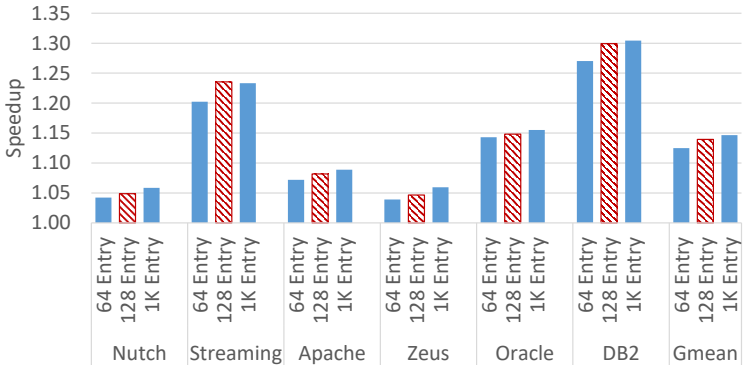

Fig. 18. Shotgun speedup over FDIP with different C-BTB sizes.

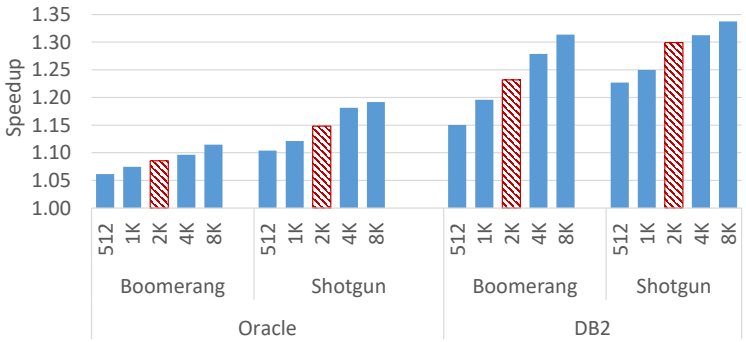

Fig. 19. Boomerang and Shotgun speedup over FDIP for different BTB sizes. The indicated BTB size is for FDIP and Boomerang; Shotgun uses the equivalent storage budget for its three BTBs.

equivalent storage budget for Shotgun. To match Boomerang's BTB storage budget in the 512- to 4K-entry range, we proportionately scale Shotgun's number of entries in U-BTB, RIB, and C-BTB from the values presented in Sec 5.2. However, scaling the number of U-BTB entries to match 8K-entry Boomerang BTB storage would lead to a 6K-entry U-BTB, which is an overkill, as 4K-entry U-BTB is sufficient to capture the entire unconditional branch working set as shown in Figure 4. Therefore, Shotgun limits the number of U-BTB entries to 4K and expands RIB and C-BTB to store 1K and 4K entries respectively, to utilize the remaining budget. Empirically, we found this to be the preferred Shotgun configuration for the 8K-entry storage budget.

Figure 19 shows the results for Oracle and DB2, the two workloads with the largest instruction footprints that are particularly challenging for BTB-based prefetchers. The striped bars highlight the results for the baseline 2K-entry BTB. As the figure shows, given an equivalent storage budget, Shotgun always outperforms Boomerang. On the Oracle workload, Shotgun, with a small storage budget equivalent to a 1K-entry conventional BTB outperforms Boomerang with an 8K-entry BTB (12% vs 11.5% performance improvement over no prefetch baseline). Similarly on DB2, Boomerang needs more than twice the BTB capacity to match Shotgun's performance. For instance, with a 2K-entry BTB, Shotgun delivers a 30% speedup, whereas Boomerang attains only a 28% speedup with a larger 4K-entry BTB. These results indicate that Shotgun's judicious use of BTB capacity translates to higher performance across a wide range of BTB sizes.

6.7 Analyzing the Remaining Front-end Stalls

Though, on average, Shotgun outperforms both Boomerang and Confluence, its performance lags that of an ideal front-end on high MPKI workloads: Oracle and DB2. We analyze the sources of these remaining front-end stalls and divide them into six categories. As prefetching via spatial footprints is the key enabler for Shotgun's performance gain over state-of-the-art prefetcher (Boomerang), these categories reflect the factors that prevent spatial prefetching from reaching the performance of an ideal front-end. The categories are as follows:

1. Late-arriving spatial prefetches: On a U-BTB hit, Shotgun leverages the spatial footprint to identify the cache blocks likely to be accessed in the next region and triggers their prefetch if not already present in L1-I. However, if Shotgun does not hit the U-BTB entry early enough, it would not get sufficient time to prefetch a block before it is accessed. As a result, the prefetch latency will be partially exposed and result in front-end stalls. We attribute such stalls to Shotgun's inability to hit the spatial footprint on time.

2. Mispredicted region: These stalls are caused by mispredicted unconditional branches that lead Shotgun to prefetch the wrong spatial region. When execution is redirected to the correct region after a pipeline flush, the latency of fetching any missed cache blocks will be exposed as Shotgun never prefetched the correct region. Such mispredictions are caused by indirect unconditional branches and calls corresponding to virtual function calls, switch statements, interrupts, exceptions, traps etc.

3. Out-of-range of bit vector: The results in Figure 3 imply that an 8-bit vector, as used by Shotgun to store the spatial footprint, is big enough to capture more than 85% of cache blocks in a region. However, 8 bits are not always sufficient to capture the whole spatial footprint. If a cache blocks is farther from the start of the region than can be accommodated in 8 bits, it would not be captured by a spatial footprint stored in an 8-bit vector. Consequently, it would not be prefetched via spatial region prefetching. The front-end stalls corresponding to such cache blocks are attributed to the limited range of the bit vector.

4. U-BTB miss: On a U-BTB (or RIB) miss, Shotgun cannot prefetch the next spatial region as the miss implies that the spatial footprint for the next region is also not present in U-BTB. Though Shotgun does prefill the unconditional branch metadata in U-BTB by predecoding the corresponding cache block, it cannot prefill the spatial footprint on a miss because the region must execute first in order to record the footprint. Any front-end stalls in such regions are attributed to U-BTB misses.

5. Wrong spatial footprint: If the cache blocks accessed inside a region change from one execution of the region to the next, the spatial footprint would not be able to prefetech all the required cache blocks for the current execution. We attribute the front-end stalls caused by the cache blocks that were not accessed during the last execution but are accessed in the current execution to the incorrect spatial footprint.

6. Uninitialized spatial footprint: If Shotgun encounters an uninitialized spatial footprint on a U-BTB (or RIB) hit, it cannot prefetch the next spatial region. There are two main reasons for the spatial footprint to be uninitialized on a U-BTB (or RIB) hit. First, the spatial footprint for return instructions might be uninitialized (or missing) as the returns are tracked in RIB whereas their spatial footprints are stored in U-BTB along with the call instructions (details in Section 4.2.1). So, if a call is evicted from U-BTB, the spatial footprint for return will be evicted alongside while the return instruction itself remains in RIB. Second, the spatial footprint for U-BTB misses filled on the wrong execution path will remain uninitialized because the wrong path will be detected before the region execution could complete and spatial footprint could be written to U-BTB. The next hit for such U-BTB entries on correct path will see uninitialized footprint. All front-end stalls in such regions are attributed to uninitialized spatial footprint.

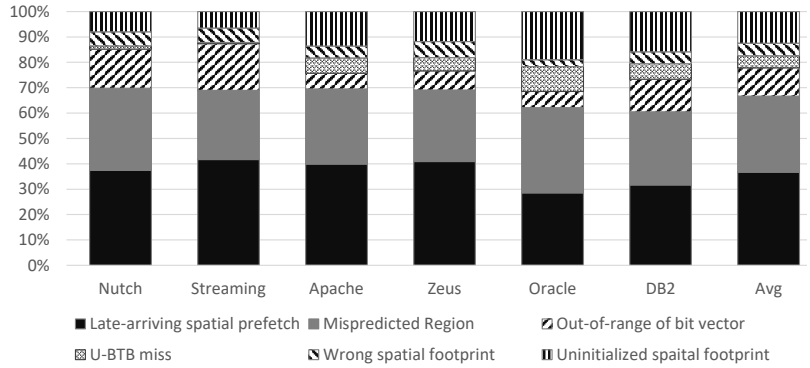

Fig. 20. Sources of the remaining front-end stalls.

Figure 20 presents the distribution of the remaining front-end stalls among these six categories. As the figure shows, the two major sources of the remaining stalls are the spatial prefetches not arriving on time and the mispredicted next regions. Together, these two sources constitute about 67% of all the remaining stalls on average, with 37% of the stalls coming from the late arriving spatial prefetches and 30% from mispredicted next spatial region.

The spatial prefetches arrive late when Shotgun is unable to reach a U-BTB entry, and issue prefetches, early enough. This happens primarily due to small basic block sizes and branch mispredictions. Shotgun strives to hide L1-I/BTB miss latency by accumulating enough fetch addresses in the FTQ so that the fetch engine remains busy while a missed block is being prefetched. The number of fetch addresses accumulated in the FTQ is often a function of the size of basic blocks because in each cycle Shotgun inserts fetch addresses corresponding to only a single basic block into the FTQ. In the case of small basic blocks, the address accumulation in the FTQ is slow or does not happen at all if the basic block have fewer instructions than the core fetch width (core consumes up to "fetch width" addresses from the FTQ in each cycle). In the absence of sufficient work in the FTQ, the queue drains quickly and miss latency is exposed. Similarly, branch mispredictions cause the FTQ to drain completely as all wrong path instructions, including the fetch addresses in the FTQ, have to be flushed. As a result, the latency of misses that occur before sufficient fetch addresses have been accumulated in FTQ after a flush is also exposed.

To alleviate the impact of these two factors, further investigation is required to keep the FTQ sufficiently occupied so that the miss latencies are not exposed. Multiple branch predictions per cycle can potentially address the small basic block size limitation by inserting fetch addresses from multiple basic blocks to the FTQ in one cycles. The branch prediction accuracy itself needs to be improved to avoid draining the FTQ on mispredictions. Furthermore, to reduce the stalls caused by mispredicted regions, we need to investigate better indirect branch target prediction mechanisms. We leave such explorations for the future work.

Of the remaining, *Out-of-range of bit vector* is responsible for about 11% of the stalls on average. A tempting solution to this problem is increasing the size of the bit vector; however, as Figure 3 implies, the missing cache blocks have a very diverse distribution. Therefore, any reasonable increase in bit vector size and storage cost is unlikely to capture these blocks. This hypothesis is also validated by the results in Figures 12 in Section 6.3, which shows that increasing the bit vector from 8-bit to 32-bit provides only 0.5% performance improvements.

U-BTB misses and *Wrong spatial footprint* each result in only about 5% stalls, on average. Stalls due to U-BTB misses are highest in Oracle (10%) because it has the largest branch working set as implied by Table 1. The low incidence of stalls attributed to wrong spatial footprint also implies that the cache blocks accessed inside a region mostly stay stable over executions, thus corroborating prior results [16]. The remaining 12% stalls come from the uninitialized spatial footprints.

7 RELATED WORK

As Shotgun aims to reduce L1-I miss induced stall cycles as well as the pipeline flushes cause by BTB misses, this section discusses related work in both of these domains.

L1-I prefetching: Stalls stemming from L1-I misses are a well known performance bottleneck especially in server workloads. Prior work has proposed both hardware and software mechanisms to mitigate this bottleneck. On the hardware side, simple next-line prefetchers [21] are widely deployed in commercial processors. These prefetchers are very efficient in prefetching sequential code; however, control flow discontinuities caused by function calls, taken branches, interrupts etc. render them of limited use for server applications. A number of prefetching techniques have been proposed to address the limitations of next-line prefetchers and these can be broadly categorized as either branch-predictor-directed prefetchers [9, 22–25] or temporal stream prefetchers [7, 8, 13, 16].

FDIP [9] is the most well known and widely used fetch-directed-instruction prefetcher. It decouples the branch prediction unit from the fetch unit by mean of a fetch target queue (FTQ). This decoupling enables FDIP to recursively query the branch predictor to generate future fetch addresses and insert them into the FTQ. As the FTQ entries are used for fetching instructions, FDIP checks whether the corresponding blocks are present in L1-I and issues prefetches if they are not. Branch history guided [23] and execution history guided [25] prefetchers associate discontinuity misses to earlier executed instructions which are then used to trigger prefetches in the next iterations.

Temporal stream prefetchers rely on the principle of record and replay. TIFS [7] records L1-I misses in a miss log and replays the log for prefetching during the next executions of the same control flow path. PIF [16] observes that recording the entire retire-order L1-I access stream, rather than only L1-I misses, provides better coverage and accuracy. Though these prefetchers are highly accurate, they incur huge, of the order of 100s KB, metadata storage overhead per core to record the L1-I miss or access stream history. SHIFT [13] reduces the metadata storage cost by sharing the metadata across multiple cores running the same workload.

While most of these techniques focus on improving L1-I miss coverage, a recent proposal, the entangling instruction prefetcher [26], aims to improve prefetch timing. Prefetch timing is important to ensure that a prefetched block arrives in L1-I before the demand fetch; however, not so early that it is evicted before being used. The key idea, similar to execution history guided prefetcher [25], is to find an earlier executed instruction that should trigger the prefetch of a later cache block. To ensure timeliness, it estimates the latency of cache miss operations and entangles them to prior instructions that are sufficiently far in the past to ensure timeliness of prefetches.

Similar to Shotgun, two previously proposed techniques, pTask [27] and RDIP [8], also leverage global control flow information for prefetching. pTask initiates prefetching only on OS context switches and requires software support. RDIP is closer to Shotgun as it also exploits global program context captured by RAS for prefetching. However, there are important differences between the two approaches. First, RDIP, for timely prefetching, predicts the future program context (next call/return instruction) solely based on the current context. This approach ignores local control flow in predicting the future execution path, which naturally limits accuracy. Shotgun, on the the other hand, predicts each and every branch to locate the upcoming code region. Therefore, Shotgun is more accurate in discovering future code

regions and L1-I accesses. Also, RDIP incurs a high storage cost, 64KB per core, as it has to maintain dedicated metadata for L1-I prefetching. Shotgun, in contrast, has no additional storage requirement, as it captures the *global control flow* and spatial footprints inside the storage budget of a conventional BTB.

On the software side, many approaches have been proposed to optimize the code layout at compile time [28], link time [29, 30], or post link time [31, 32]. There have also been proposals on inserting prefetch instructions in the binary at compile time [33] and exploiting recurring call-graph history [34]. Recent work in software prefetching, AsmDB [35], profiles thousands of production binaries in a data-center and proposes profile-driven injection of software prefetch instructions in to the application binary at compile time. In addition, it introduces hardware support for the execution of software prefetch instructions. I-SPY [36] improves over AsmDB by correlating L1-I misses to execution contexts and prefetching a block only if the same execution context is observed again. As these software techniques are mostly complementary to Shotgun, they are likely to aid each other.

All of the techniques discussed above target only one part of the overall front-end bottleneck as they prefetch only L1-I blocks but do not prefill BTB. Meanwhile, Shotgun offers a cohesive solution to the entire problem.

BTB prefetching: BTB plays a critical role in identifying control flow discontinuities and feeding the core with correct path instructions. The large instruction footprints of server workloads put immense pressure on BTB and the resulting BTB misses limit its ability to discover control flow discontinuities. To mitigate this limitation, prior work has suggested to augment a low latency first-level BTB with a large capacity second-level BTB. A dedicated prefetcher is then used to bring entries from second-level to first-level BTB.

IBM z-series processors use a technique called Two-level Bulk Preload [11] to prefetch branches from a 24K-entry second-level BTB to a 4K-entry first-level BTB. The prefetching mechanism uses spatial correlation to prefetch a set of spatially-proximate entries into the first-level BTB upon a miss. Another technique, called Phantom BTB [12], virtualizes temporal streams of BTB entries into the last level cache instead of using a dedicated second-level BTB. However, both of these techniques incur 100s of KB of storage overhead. In addition, as they trigger prefetches on a miss in first-level BTB, they expose the high latency of the second-level BTB or last level cache.

In addition to BTB prefetching, researchers have investigated BTB entry organizations to maximize the number of entries in a given storage budget [14, 37, 38].

Unified L1-I and BTB prefetching: While the techniques discussed above employ dedicated prefetchers for L1-I and BTB prefetching, recent work [14, 15, 39] proposes to unify them. Confluence [14] observes that L1-I blocks effectively embed the BTB metadata for the branches they contain. Therefore, it proposes to predecode the L1-I blocks as they are prefetched into L1-I to extract branch information and prefill it into the BTB. In doing so Confluence not only avoids a dedicate BTB prefetcher but also the second BTB level. However, its storage requirements still remain high as it requires a temporal stream based prefetcher for L1-I prefetching. SN4L+Dis+BTB [39] proposes to divide the front-end bottleneck in separate components and use a dedicated prefetcher for each component. Specifically, it uses a selective next-4-line (SN4L) prefetcher to prefetch sequential code and a modified discontinuity prefetcher (Dis) [10] to prefetch control flow discontinuities. Finally, it takes Confluence's approach of predecoding the prefetched L1-I blocks to prefill the BTB. In doing so, it eliminates the storage overhead of temporal stream prefetching.

Boomerang [15] relies on FDIP for L1-I prefetching and extends it with BTB prefetch capability to mitigate BTB miss related pipeline flushes. For BTB prefilling, it first detects BTB misses, fetches the cache block containing the corresponding branch instruction, predecodes the cache blocks to extract branch metadata, and finally inserts this

metadata into the BTB. As it relies solely on the existing branch prediction structures for L1-I prefetching and BTB prefilling, it incurs near zero hardware cost. However, Boomerang's complete reliance on BTB to discover L1-I prefetch candidates limits its prefetching opportunities. This is because back-to-back BTB misses severely limit its ability to run ahead of the fetch unit and discover prefetch candidates as it does not know which control flow path to follow on a BTB miss. Shotgun, in contrast, partially decouples L1-I prefetching from BTB, at it leverages spatial region footprints to prefetch the target code regions.

8 CONCLUSION

The front-end bottleneck in server workloads is a well-established problem due to frequent misses in the L1-I and the BTB. Prefetching can be effective at mitigating the misses; however, existing front-end prefetchers force a trade-off between coverage and storage overhead.

This paper introduces Shotgun, a front-end prefetcher powered by a new BTB organization and design philosophy. The main observation behind Shotgun is that an application's instruction footprint can be summarized as a combination of its unconditional branch working set and a spatial footprint around the target of each unconditional branch. The former captures the global control flow (mostly function calls and returns), while the latter summarizes the local (intra-function) instruction cache working set. Based on this insight, Shotgun devotes the bulk of its BTB capacity to unconditional branches and their spatial footprints. Meanwhile, conditional branches are maintained in a small-capacity dedicated BTB that is filled from the prefetched instruction cache blocks. By effectively summarizing the application's instruction footprint in the BTB, Shotgun enables a highly effective BTB-directed prefetcher that largely erases the gap between metadata-free and metadata-rich state-of-the-art prefetchers.

Finally, we identify the factors that prevent Shotgun from matching the performance of an ideal server front-end. Based on these factors, we suggest the future research directions to erase the performance difference between an ideal and practical front-end.

9 ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. This work is partially supported through the Research Council of Norway (NFR) grant 302279 to NTNU.

REFERENCES

[1] R. Kumar, B. Grot, and V. Nagarajan, "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*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 30–42. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173178

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?" in *International Conference on Very Large Data Bases*, 1999, pp. 266–277.

[3] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, "Performance characterization of a quad pentium pro SMP using OLTP workloads," in *International Symposium on Computer Architecture*, 1998, pp. 15–26.

[4] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of database workloads on shared-memory systems with out-of-order processors," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 307–318.

[5] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer," in *International Symposium on Computer Architecture*, 2015, pp. 158–169.

[6] I.-C. K. Chen, C.-C. Lee, and T. N. Mudge, "Instruction Prefetching Using Branch Prediction Information," in *International Conference on Computer Design*, 1997, pp. 593–601.

[7] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal Instruction Fetch Streaming," in *International Symposium on Microarchitecture*, 2008, pp. 1–10.

[8] A. Kolli, A. G. Saidi, and T. F. Wenisch, "RDIP: return-address-stack directed instruction prefetching," in *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*, 2013, pp. 260–271.

[9] G. Reinman, B. Calder, and T. Austin, "Fetch Directed Instruction Prefetching," in *International Symposium on Microarchitecture.* IEEE, 1999, pp. 16–27.

[10] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 225–236.

[11] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. Prasky, and A. Saporito, "Two Level Bulk Preload Branch Prediction," in *International Symposium on High-Performance Computer Architecture*, 2013, pp. 71–82.

[12] I. Burcea and A. Moshovos, "Phantom-btb: a virtualized branch target buffer design," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, 2009, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508281

[13] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: Shared History Instruction Fetch for Lean-core Server Processors," in *International Symposium on Microarchitecture*, 2013, pp. 272–283.

[14] C. Kaynak, B. Grot, and B. Falsafi., "Confluence: Unified Instruction Supply for Scale-Out Servers," in *International Symposium on Microarchitecture*, 2015, pp. 166–177.

[15] R. Kumar, C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 493–504. [Online]. Available: https://doi.org/10.1109/HPCA.2017.53

[16] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive Instruction Fetch," in *International Symposium on Microarchitecture*, 2011, pp. 152–162.

[17] T. Yeh and Y. N. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," in *International Symposium on Microarchitecture*, 1992, pp. 129–139.

[18] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[19] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *International Symposium on Computer Architecture*, 2003, pp. 84–95.

[20] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *J. Instruction-Level Parallelism*, vol. 8, 2006.

[21] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, p. 7–21, Dec. 1978. [Online]. Available: https://doi.org/10.1109/C-M.1978.218016

[22] I.-C. K. Chen, C.-C. Lee, and T. Mudge, "Instruction prefetching using branch prediction information," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, 1997, pp. 593–601.

[23] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, ser. HPCA '01. USA: IEEE Computer Society, 2001, p. 291.

[24] A. V. VEIDENBAUM, Q. ZHAO, and A. SHAMEER, "Non-sequential instruction cache prefetching for multiple–issue processors," *International Journal of High Speed Computing*, vol. 10, no. 01, pp. 115–140, 1999.

[25] Y. Zhang, S. Haga, and R. Barua, "Execution history guided instruction prefetching," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 199–208. [Online]. Available: https://doi.org/10.1145/514191.514220

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

[27] P. Kallurkar and S. R. Sarangi, "ptask: A smart prefetching scheme for os intensive applications," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[28] D. Chen, T. Moseley, and D. X. Li, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2016, pp. 12–23.

[29] D. X. Li, R. Ashok, and R. Hundt, "Lightweight feedback-directed cross-module optimization," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 53–61. [Online]. Available: https://doi.org/10.1145/1772954.1772964

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

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

[32] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: A practical binary optimizer for data centers and beyond," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 2–14.

[33] C.-K. Luk and T. Mowry, "Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998, pp. 182–193.

[34] M. Annavaram, J. Patel, and E. Davidson, "Call graph prefetching for database applications," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 281–290.

[35] N. P. Nagendra, G. Ayers, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers," *IEEE Micro*, vol. 40, no. 3, pp. 56–63, 2020.

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

[37] T. Asheim, B. Grot, and R. Kumar, "BTB-X: A storage-effective BTB organization," *IEEE Computer Architecture Letters*, 2021.

[38] "AMD software optimization guide. Section 2.8.1.2," https://www.amd.com/system/files/TechDocs/56665.zip.

[39] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 65–78.