Blasting Through The Front-End Bottleneck With Shotgun

Rakesh Kumar[*]
Uppsala University

Boris Grot
University of Edinburgh

Vijay Nagarajan
University of Edinburgh

Abstract

The front-end bottleneck is a well-established problem in server workloads owing to their deep software stacks and large instruction working sets. Despite years of research into effective L1-I and BTB prefetching, state-of-the-art techniques force a trade-off between performance and metadata storage costs. This work introduces Shotgun, a BTB-directed front-end prefetcher powered by a new BTB organization that maintains a logical map of an application's instruction footprint, which enables high-efficacy prefetching at low storage cost. To map active code regions, Shotgun precisely tracks an application's global control flow (e.g., function and trap routine entry points) and summarizes local control flow within each code region. Because the local control flow enjoys high spatial locality, with most functions comprised of a handful of instruction cache blocks, it lends itself to a compact region-based encoding. Meanwhile, the global control flow is naturally captured by the application's unconditional branch working set (calls, returns, traps). Based on these insights, Shotgun devotes the bulk of its BTB capacity to branches responsible for the global control flow and a spatial encoding of their target regions. By effectively capturing a map of the application's instruction footprint in the BTB, Shotgun enables highly effective BTB-directed prefetching. Using a storage budget equivalent to a conventional BTB, Shotgun outperforms the state-of-the-art BTB-directed front-end prefetcher by up to 14% on a set of varied commercial workloads.

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

Keywords Servers, Prefeteching, Instruction Cache, Branch Target Buffer (BTB), Control Flow.

[*]This work was done while the author was at University of Edinburgh.

ASPLOS'18, March 24-28, 2018, Williamsburg, VA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173178

ACM Reference Format:
Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting Through The Front-End Bottleneck With Shotgun. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3173162.3173178

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 [1, 11, 14]. Over the years, the problem has persisted; in fact, according to a recent study from Google [8], 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 [8].

Microarchitecture researchers have proposed a number of instruction [4, 6, 12, 15, 17] and BTB [2, 3] prefetchers over the years to combat the front-end bottleneck in servers. State-of-the-art prefetchers rely on *temporal streaming* [6] 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 [9, 10, 12]; 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) [15] and extends it with unified prefetching into the BTB [13]. 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 [10], 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 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 14%.

2 Background

2.1 Temporal streaming prefetching

Over the past decade, *temporal streaming [6]* 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 [5] and BTB [3] 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 [3, 5]. To mitigate the cost, two complementary techniques have been proposed. The first is sharing the metadata across all cores executing a common workload [9]. 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 [10]. 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

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.

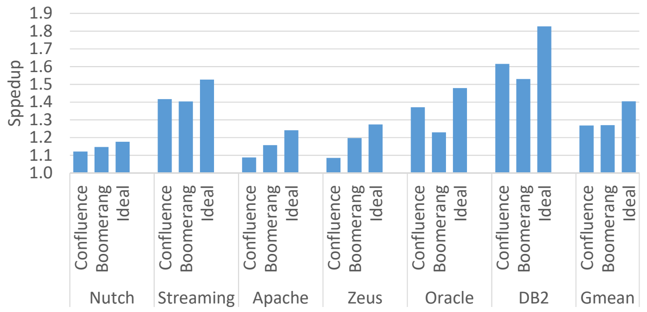

Figure 1. Comparison of state-of-the-art unified front-end prefetchers to the ideal front-end on server workloads.

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 [10]. 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 [15]. 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 [13]. 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 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 14% and 9%, 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 [8], 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.

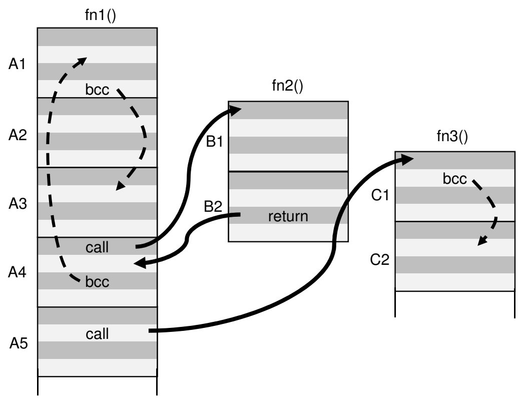

Figure 2. Program control flow example. The solid arrows represent *global control flow* and dotted arrows depict *local control flow*. A1, B1, etc denote cache block addresses.

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

Figure 2 shows a cartoon 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, typically within a few cache blocks [13], 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

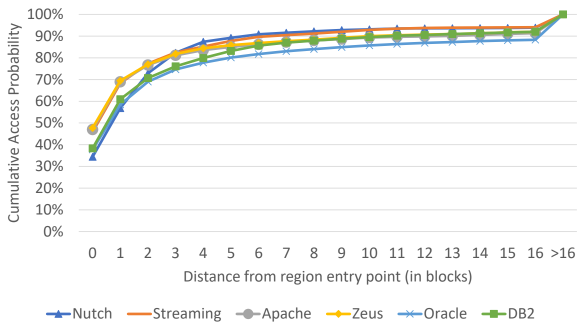

Figure 3. Instruction cache block access distribution inside code regions.

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 [15] 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.

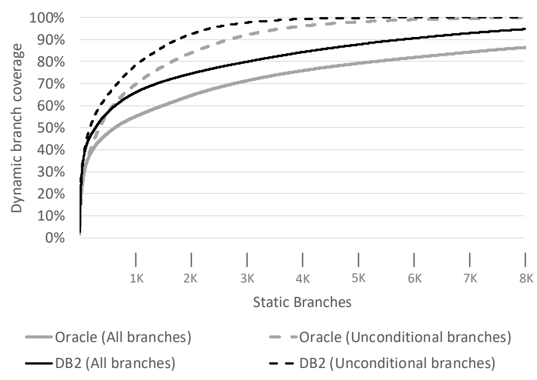

Figure 4. Contribution of static branches towards dynamic branch execution for Oracle and DB2.

- 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 5a 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 [13] and Confluence [10]. Specifically, while prefetching 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, which are shown in Figure 5a.

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 5a, is comprised of the following fields:

Tag: the branch identity.

Size: the size of the basic block containing the branch (like

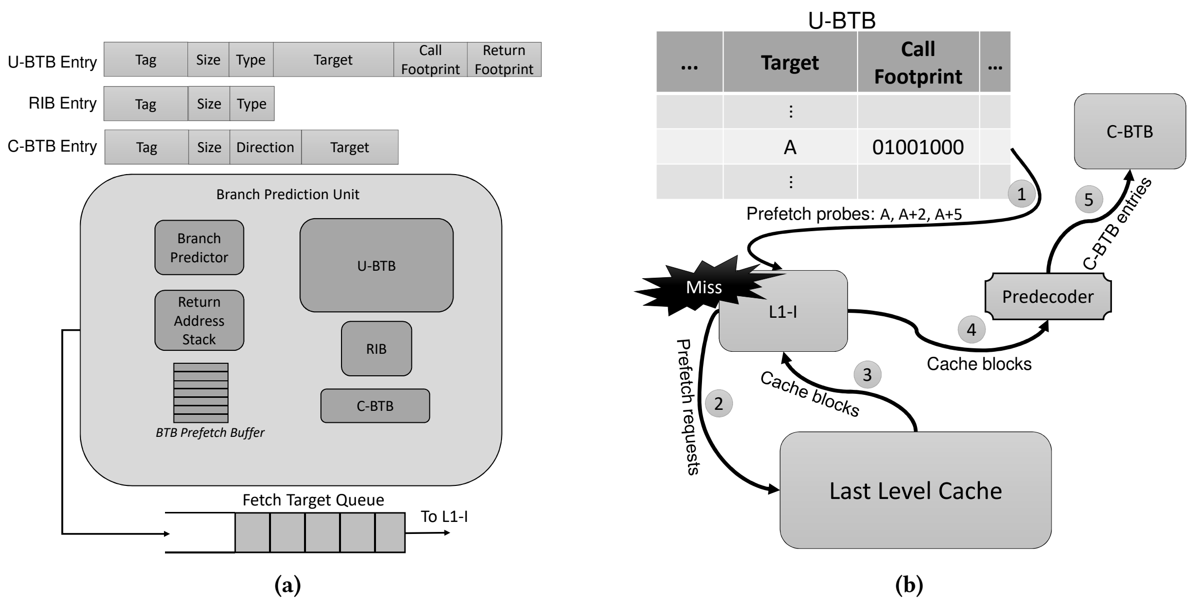

Figure 5. Shotgun: (a) BTB organization and (b) Prefetching using spatial footprints.

Boomerang, Shotgun uses a basic-block-oriented BTB [20])[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.

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 underutilization 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 underutilization 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 5a, 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 5a, 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.

[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

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 [15], Shotgun also employs a Fetch Target Queue(FTQ), as shown in Figure 5a, 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 [10] 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 5b 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 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 [13]. On a hit to the BTB Prefetch Buffer, the accessed branch is moved to the appropriate BTB based on the branch type.

Discussion: Similar to Shotgun, two previously proposed techniques, pTask [7] and (RDIP) [12]), also leverage global

[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.

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 [16] (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

control flow information for prefetching; but unlike Shotgun, they target only L1-I misses. Moreover, 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. Second, RDIP targets only a part of the overall front-end bottleneck as it prefetches only L1-I blocks but does not prefill BTB. Meanwhile, Shotgun offers a cohesive solution to the entire problem. Finally, 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.

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 [10]. To further reduce metadata storage costs, Confluence virtualizes the history metadata into the LLC using SHIFT [9]. We model Confluence as SHIFT augmented with a 16K-entry BTB, which was shown to provide a generous upper bound on Confluence's performance [10]. 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 [13]. 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 start with assessing the impact of spatial footprints in front-end prefetching; we then analyze the impact of using a small C-BTB on Shotgun's performance; finally, we present a sensitivity study to the BTB storage budget.

6.1 Front-end stall cycle coverage

To assess the efficacy of different prefetching mechanisms, we present the number of front-end stall cycles covered by each of them in Figure 6. 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 6, Shotgun covers 68% of the stall cycles experienced by a no prefetch baseline; this is 8% better than each of Boomerang and Confluence. A closer inspection reveals that Shotgun outperforms its direct rival Boomerang on all of the workloads; in particular, Shotgun provides more than 10% coverage improvements on each of DB2 and Streaming, and over 8% on Oracle – these workloads have 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 16%-19% additional coverage. Confluence performs poorly on these applications, as also noted by Kumar et al. [13], 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 2% of Confluence on DB2, but is 10% 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.

6.2 Performance Analysis

Figure 7 shows the performance improvements for different prefetching mechanisms over a baseline without any prefetcher. The performance trends are similar to coverage trends (Figure 6) with Shotgun providing, on average, 32% performance improvement over the baseline and 5% improvement over each of Boomerang and Confluence. The speedup over Boomerang is especially prominent on high BTB MPKI workloads, DB2 and Oracle, where Shotgun achieves 10% and 8% improvement respectively.

Interestingly, Figure 7 shows that Shotgun attains a relatively modest performance gain over Boomerang on Nutch,

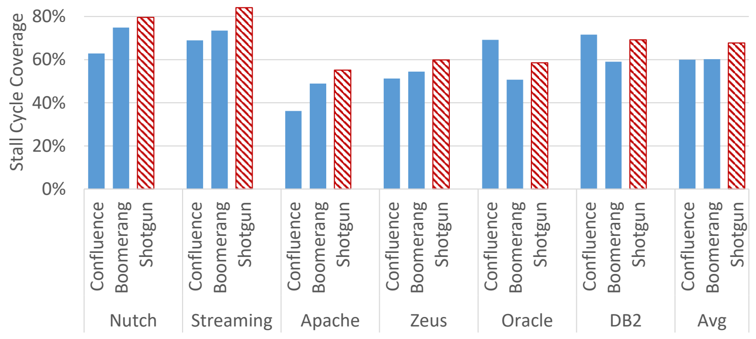

Figure 6. Front-end stall cycles covered by different prefetching schemes over no-prefetch baseline.

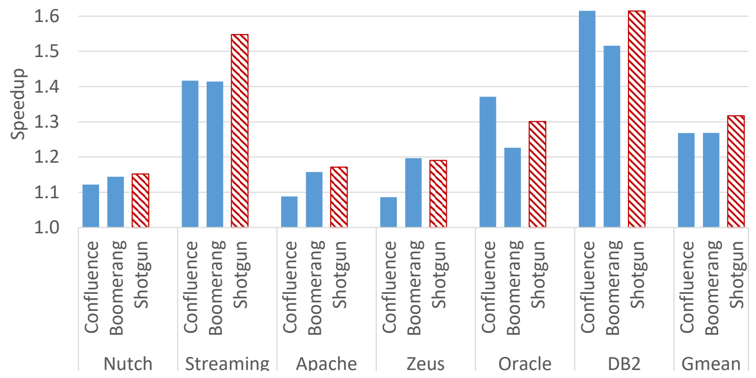

Figure 7. Speedup of different prefetching schemes over no-prefetch baseline.

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 7%.

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: (1) No bit vector: does not perform any region prefetching; (2) 8-bit vector; (3) 32-bit vector; (4) Entire Region: prefetch all the cache blocks between entry and exit points of the target region; and (5) 5-Blocks: prefetch 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.

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 8 and 9 show, an 8-bit vector provides, on average, 6% coverage and 4% performance benefit compared to no spatial region prefetching. In fact, without spatial footprints, Shotgun's coverage is only 2% better than Boomerang. With an 8-bit vector, Shotgun improves the performance of every single workload, with the largest gain of 9% 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 remaining spatial region prefetching mechanisms,

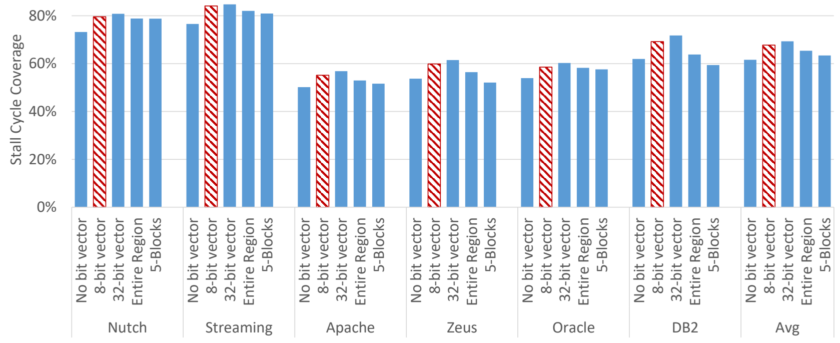

Figure 8. Shotgun front-end stall cycle coverage with different spatial region prefetching mechanisms.

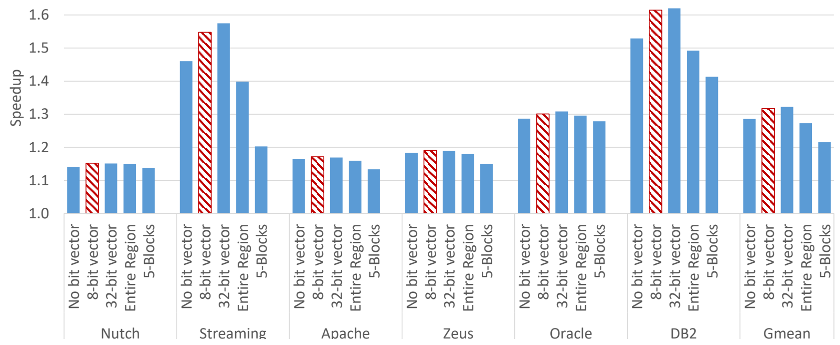

Figure 9. Shotgun performance with different spatial region prefetching mechanisms.

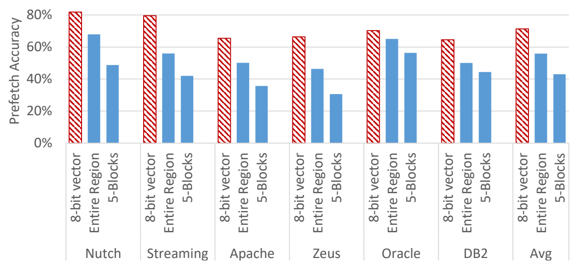

Figure 10. Shotgun prefetch accuracy with different spatial region prefetching mechanisms.

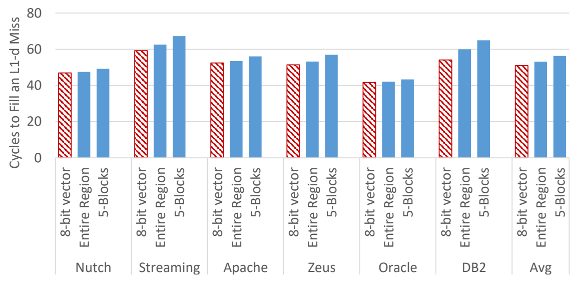

Figure 11. Number of cycles required to fill an L1-D miss with different mechanisms for spatial region prefetching.

Entire Region and 5-Blocks, lead to a performance degradation compared to 8-bit vector as shown in Figure 9. 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 10, 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 11. 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 increased LLC access latency due to over-prefetching makes indiscriminate region prefetching less effective than the 8-bit vector design.

6.4 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

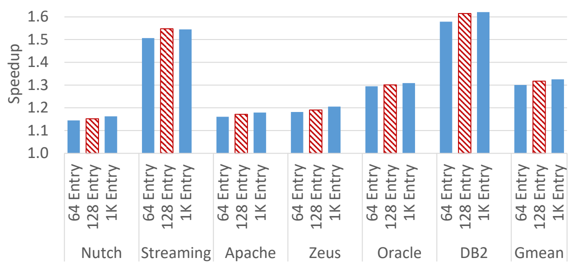

Figure 12. Shotgun speedups with different C-BTB sizes.

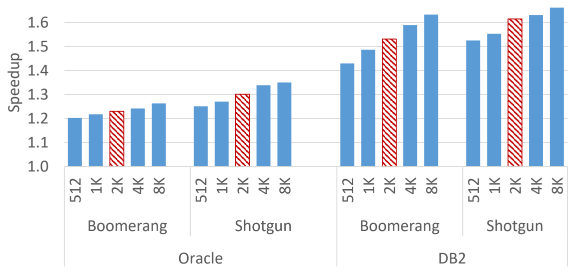

Figure 13. Boomerang and Shotgun speedup for different BTB sizes. The indicated BTB size is for Boomerang; Shotgun uses the equivalent storage budget for its three BTBs.

fill it ahead of time. To measure Shotgun's effectiveness in prefilling the C-BTB, Fig 12 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 0.8% 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 4% lower performance compared to a 128-entry C-BTB. On average, the 128-entry C-BTB outperforms the 64-entry C-BTB by 2% as shown in Figure 12.

6.5 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 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 13 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 (27% vs 26.3% 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 61.5% speedup, whereas Boomerang attains only a 58.9% 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.

7 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.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work is supported by EPSRC grant EP/M001202/1 to the University of Edinburgh.

References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *International Conference on Very Large Data Bases*. 266–277.

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

[3] Ioana Burcea and Andreas Moshovos. 2009. 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*. 313–324. DOI: http://dx.doi.org/10.1145/1508244.1508281

[4] I-Cheng K Chen, Chih-Chieh Lee, and Trevor N Mudge. 1997. Instruction Prefetching Using Branch Prediction Information. In *International Conference on Computer Design*. 593–601.

[5] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive Instruction Fetch. In *International Symposium on Microarchitecture*. 152–162.

[6] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal Instruction Fetch Streaming. In *International Symposium on Microarchitecture*. 1–10.

[7] P. Kallurkar and S. R. Sarangi. 2016. pTask: A smart prefetching scheme for OS intensive applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. DOI: http://dx.doi.org/10.1109/MICRO.2016.7783706

[8] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2015. Profiling a warehouse-scale computer. In *International Symposium on Computer Architecture*. 158–169.

[9] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. SHIFT: Shared History Instruction Fetch for Lean-core Server Processors. In *International Symposium on Microarchitecture*. 272–283.

[10] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: Unified Instruction Supply for Scale-Out Servers. In *International Symposium on Microarchitecture*. 166–177.

[11] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. 1998. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *International Symposium on Computer Architecture*. 15–26.

[12] Aasheesh Kolli, Ali G. Saidi, and Thomas F. Wenisch. 2013. 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*. 260–271.

[13] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A Metadata-Free Architecture for Control Flow Delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. 493–504. DOI: http://dx.doi.org/10.1109/HPCA.2017.53

[14] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. 1998. 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*. 307–318.

[15] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch Directed Instruction Prefetching. In *International Symposium on Microarchitecture*. IEEE, 16–27.

[16] André Seznec and Pierre Michaud. 2006. A case for (partially) TAgged GEometric history length branch prediction. *J. Instruction-Level Parallelism* 8 (2006).

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

[18] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31.

[19] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *International Symposium on Computer Architecture*. 84–95.

[20] Tse-Yu Yeh and Yale N. Patt. 1992. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *International Symposium on Microarchitecture*. 129–139.