Boomerang: a Metadata-Free Architecture for Control Flow Delivery

Rakesh Kumar Cheng-Chieh Huang Boris Grot Vijay Nagarajan
Institute of Computing Systems Architecture
University of Edinburgh
{rakesh.kumar, cheng-chieh.huang, boris.grot, vijay.nagarajan}@ed.ac.uk

***Abstract*—Contemporary server workloads feature massive instruction footprints stemming from deep, layered software stacks. The active instruction working set of the entire stack can easily reach into megabytes, resulting in frequent front-end stalls due to instruction cache misses and pipeline flushes due to branch target buffer (BTB) misses. While a number of techniques have been proposed to address these problems, every one of them requires dedicated metadata structures, translating into significant storage and complexity costs.**

In this paper, we ask the question whether it is possible to achieve high-performance control flow delivery without the metadata costs of prior techniques. We revisit a previously proposed approach of branch-predictor-directed prefetching, which leverages just the branch predictor and BTB to discover and prefetch the missing instruction cache blocks by exploring the program control flow ahead of the core front-end. Contrary to conventional wisdom, we find that this approach can be effective in covering instruction cache misses in modern CMPs with long LLC access latencies and multi-MB server binaries. Our first contribution lies in explaining the reasons for the efficacy of branch-predictor-directed prefetching. Our second contribution is in Boomerang, a metadata-free architecture for control flow delivery. Boomerang leverages a branch-predictor-directed prefetcher to discover and prefill not only the instruction cache blocks, but also the missing BTB entries. Crucially, we demonstrate that the additional hardware cost required to identify and fill BTB misses is negligible. Our experimental evaluation shows that Boomerang matches the performance of the state-of-the-art control flow delivery scheme without the latter's high metadata and complexity overheads.

I. Introduction

Modern server software stacks are organized as layered services, each implementing complex functionality. Because of this, server workloads tend to have multi-MB instruction footprints that defy private instruction caches, causing frequent long-latency accesses to lower cache levels. Similarly, the large number of branches associated with such massive instruction working sets exceed the capacity of practical single-level BTBs, resulting in either frequent pipeline flushes or necessitating dedicated high-capacity second level BTBs. In the case of both instruction caches and BTBs, frequent misses considerably degrade core performance by exposing the fill or flush (in the case of a BTB miss) latency.

The front-end bottleneck in servers is not a new phenomenon and has been highlighted by a number of studies over the years [1], [2], [3], [4], [5], [6], [7]. Mitigation techniques for both instruction cache (L1-I) and BTB misses generally rely on sophisticated hardware prefetchers, as software solutions such as code layout optimizations [8], provide only partial improvements due to complex control flow and massive code footprints in server workloads.

On the instruction cache side, state-of-the-art prefetchers for servers rely on temporal streaming to record and replay long sequences of instructions [9], [10], [7], [11]. While highly effective at eliminating L1-I misses, for maximum coverage, these techniques require 100s of KBs of metadata to store and index the temporal streams. On the BTB side, both spatial and temporal correlating prefetchers have been proposed to move BTB entries from a large second level BTB to a small first level [12], [13]. In order to capture the tens of thousands of branch targets that are typical of server workloads, the second-level BTBs also necessitate well over 100KB of storage.

Whereas the above works have looked at the instruction cache problem and the BTB problem separately, recent research has looked at addressing both together [14]. Specifically, it makes the critical observation that control flow is common across the different structures, and dedicated history-based instruction and BTB prefetchers implicitly replicate it in their respective histories. Because instruction cache blocks carry the branch instructions with their targets or offsets, the authors suggest using a temporal-stream-based instruction prefetcher to fill the BTB by extracting branch information from prefetched cache blocks. The resulting scheme, called Confluence, dispenses with dedicated BTB metadata (i.e, a second BTB level) but still requires expensive metadata for the instruction prefetcher.

In this paper, we ask the question whether it is possible to achieve high-performance control flow delivery without the metadata costs of prior techniques. In other words, is it possible to solve both the instruction cache problem and the BTB problem, with no additional metadata beyond what is already contained in a modest-complexity core? An affirmative answer to this question would enable high-performance control flow delivery in emerging many-core [15] and many-thread [16] RISC processors that optimize for silicon efficiency and avoid the use of area-hungry structures in favor of additional cores or threads [17].

Our key contribution is in demonstrating that the previously-proposed idea of branch-predictor-directed prefetching [18], [19] can be successfully applied in the server context and used to fill both the instruction cache and the BTB using

To Appear in HPCA, Austin, USA; February, 2017

only the existing structures inside the core, namely the BTB and the branch predictor, thus eliminating the need for additional metadata. The result is not only powerful, but is also contrary to conventional wisdom, since prior work in the server instruction prefetching space has held that branch-predictor-directed prefetching suffers from two fundamental drawbacks that limit its usefulness in the server space. The two drawbacks are (i) the limited accuracy of the branch predictor that limits its ability to cover long LLC access delays and (ii) the need for massive BTB capacity to capture the branch target working set.

We present detailed evidence demonstrating that neither of these issues is significant and that, indeed, branch-predictor-directed prefetching can effectively fill the instruction cache in advance of the core front-end even with large LLC delays. We then present Boomerang, a metadata-free control delivery architecture that uses a state-of-the-art branch-predictor-directed prefetcher to fill both the instruction cache and the BTB. We discuss practical issues and optimizations in the design of Boomerang, showing that its cost and complexity is negligible.

An evaluation of Boomerang on a set of traditional and scale-out server workloads in the context of a 16-core RISC processor reveals that Boomerang eliminates nearly all BTB-related pipeline flushes, and reduces front-end stall cycles by 50-75%. In doing so, Boomerang improves performance by 27.5%, on average, over the baseline. More importantly, Boomerang averages similar performance to the state-of-the-art (Confluence) without the latter's metadata cost and higher overall complexity.

II. Motivation

A. Importance of Control Flow Delivery in Servers

Contemporary server workloads are characterized by massive instruction footprints stemming from deep, layered software stacks. As an example, consider a typical web server deployment, consisting of the web server itself, a caching layer, CGI, a database, and an OS kernel responsible for network I/O and scheduling. The active instruction working set of the entire stack can easily reach into megabytes, resulting in frequent front-end stalls due to instruction cache misses. Similarly, the large code footprint can contain tens of thousands of active branches that can cause pipeline flushes if their targets are not found in a BTB.

The performance degradation caused by massive instruction working sets of commercial and open-source server software stacks has been highlighted by a number of studies over the years [1], [5], [6], [3]. Moreover, a recent characterization study at Google suggests that the problem is getting worse [4]. The authors highlight a search workload with a multi-megabyte instruction working set that has expanded at a rate of 27% per year for several years running, doubling over the course of their study [4].

To quantify the opportunity in eliminating front-end stalls and pipeline flushes stemming from instruction cache and BTB misses, we study a set of enterprise and open-source scale-out applications using a full-system microarchitectural simulator.

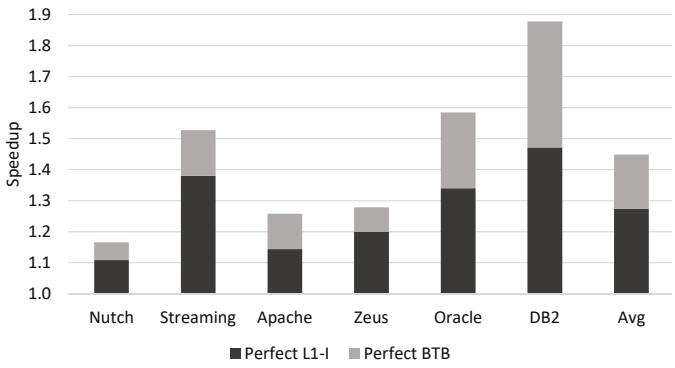

Figure 1. Opportunity in improving control flow delivery on server workloads.

The baseline core is configured with a 2K-entry BTB and a 32KB L1-I. Complete workload and simulation parameters can be found in Section V. As Figure 1 shows, eliminating all L1-I misses improves performance by 11-47%, with another 6-40% performance improvement attained by eliminating all BTB misses.

In the quest for higher core performance, we next examine techniques for mitigating instruction cache and BTB misses.

B. Mitigating Instruction Misses

Spracklen et al. [20] were the first to provide a detailed microarchitectural analysis of the sources of instruction cache stalls in commercial server workloads. A particular problem identified in the work is that of *discontinuities* resulting from non-sequential control flow. Such discontinuities challenge next-N-line prefetchers, necessitating control-flow-aware prefetch techniques.

Prior work has proposed using the branch predictor to anticipate future control flow and prefetch cache blocks into the L1-I ahead of the fetch stream [18], [19]. A particular strength of such an approach is its low cost and complexity, since it exploits existing BTB and branch predictor structures. However, branch-predictor-directed prefetch was proposed in the context of SPEC workloads with modest instruction working sets. On the server side, researchers have argued that the vast code footprints of server workloads defy capacities of practical BTBs, rendering branch-predictor-based approaches ineffective due to their inability to discover discontinuities [20]. Another challenge for branch-predictor-driven approaches is presented by the limited accuracy of branch predictors, which must predict many branches ahead of the fetch stream to cover large LLC access latencies [9].

Because of the aforementioned challenges, instruction prefetchers for servers have introduced dedicated prefetcher metadata that is entirely decoupled from branch prediction structures [20]. State-of-the-art server instruction prefetchers are based on the principle of temporal streaming, whereby entire sequences of instructions are recorded and, subsequently, replayed by the prefetcher to fill the cache ahead of the fetch stream [9], [10], [7], [11]. While extremely effective at eliminating instruction cache misses, temporal-stream-based prefetchers

incur large metadata storage costs due to massive instruction working sets of server workloads and high redundancy across streams. For instance, Proactive Instruction Fetch requires over 200KB of prefetcher metadata per core [10] .

Recent work has attempted to reduce the storage requirements of temporal streaming. RDIP [7] correlates a set of targets with an execution context, effectively reducing some of the metadata redundancy. Nonetheless, RDIP still requires over 60KB of dedicated metadata storage per core. Another approach, SHIFT, proposes to virtualize the metadata in the LLC and share it across cores executing a common workload [11]. With an 8MB LLC, SHIFT requires over 400KB of metadata storage, which is amortized among the cores executing a common workload. If multiple workloads share a CMP, they each necessitate their own prefetch metadata, placing additional pressure on the LLC.

To summarize, state-of-the-art instruction prefetchers for servers are effective but, despite recent efforts to make them practical, incur significant costs associated with storing and managing the metadata.

C. Mitigating BTB Misses

Providing the instruction cache with correct blocks is only part of the challenge; the other part is feeding the core with the right sequence of instructions. To do so, modern processors employ conditional branch predictors and branch target buffers to predict discontinuities and redirect the fetch stream to the target address.

The large instruction footprints in server workloads place significant pressure on the BTBs, which requires multiple bytes per entry to precisely identify branch PCs and their targets. In contrast, branch direction predictors mandate only a small amount of state per branch and can often deal with aliasing. As a result, recent work has shown that minimizing mis-speculation-induced flushes requires maintaining 16-32K BTB entries, costing up to 280KB of state per core [14].

Several approaches have suggested augmenting a low-latency first-level BTB with a large-capacity second level BTB and a dedicated BTB transfer engine for moving entries between BTB levels. One such approach, called Bulk Preload and implemented in an IBM z-series processor, relies on a 24K-entry second-level BTB and uses spatial correlation to preload a set of spatially-proximate entries into the first level upon a miss there [12]. Another approach, PhantomBTB, forms temporal streams of BTB entries and virtualizes them into the LLC [13]. Both designs incur high storage overhead (in excess of 200KB per core) and rely on misses in the first-level BTB to trigger fills, thus exposing the core to the high access latency of the second BTB level.

Recent work has suggested an effective way to mitigate the cost and performance overheads of hierarchical BTBs. Noting that instruction cache blocks effectively embed the BTB metadata for the branches they contain, Confluence proposes using a temporal-stream-based instruction prefetcher to fill both the instruction cache and the BTB, the latter by predecoding incoming cache blocks and extracting branch targets from branch instructions inside the cache block [14]. By avoiding the need for a dedicated second BTB level and a dedicated BTB prefetcher, Confluence greatly reduces the cost of a high-performance front-end. However, it still relies on a temporal-stream-based instruction prefetcher that itself incurs high metadata costs.

III. Toward Metadata-Free Control Flow Delivery

In this paper, we ask the question whether it is possible to achieve high-performance control flow delivery without the staggering metadata costs of prior techniques. Reducing metadata costs is particularly important for emerging many-core and many-thread server processors, such as Cavium ThunderX [15] and Oracle T-Series [16], that seek to maximize the number of hardware contexts on chip, thus delivering better performance per unit area and per watt over conventional server CPUs [17]. To maximize these metrics, many-core and many-thread server processors eschew high microarchitectural complexity, including massive BTBs and vast metadata stores, while still relying on out-of-order cores to meet stringent per-thread performance requirements of online services.

To provide effective control flow delivery in such designs, we revisit the previously proposed idea of branch predictor-directed prefetching [19], as it does not require any metadata beyond what is already present in a core – a single-level BTB and a branch predictor. However, as noted in the previous section, prior work on server instruction prefetching has dismissed branch predictor-directed prefetching on the basis of two concerns:

1: The branch predictor must predict a large number of branches correctly in order to run far enough ahead of the core front-end so as to cover the large LLC delays in many-core NUCA processors [9]. Because branch predictor accuracy decreases geometrically with the number of branches predicted, covering large LLC delays while staying on the correct path is infeasible.

2: The BTB must capture a large branch target footprint to discover discontinuities [20]. With a small BTB, frequent BTB misses will lower prefetch coverage and cause frequent pipeline flushes, preventing the branch predictor from running ahead of the core front-end.

A. Does Branch Prediction Accuracy Limit Coverage?

In order to understand to what extent the branch predictor affects prefetch coverage, we assess the benchmarks from Figure 1 with a state-of-the-art TAGE branch predictor [21] and FDIP [19] as branch-predictor-directed prefetcher. FDIP decouples the L1-I from the front-end via a deep fetch target queue (FTQ), and uses the BTB and branch predictor ensemble to fill it. To isolate the effect of the branch predictor, we use a near-ideal 32K-entry BTB. Detailed microarchitectural parameters can be found in Section V.

Figure 2 compares a TAGE-based FDIP prefetcher to PIF [10], a state-of-the-art temporal streaming instruction prefetcher with private metadata. We study a range of LLC access latencies and use percentage of front-end *stall cycles*

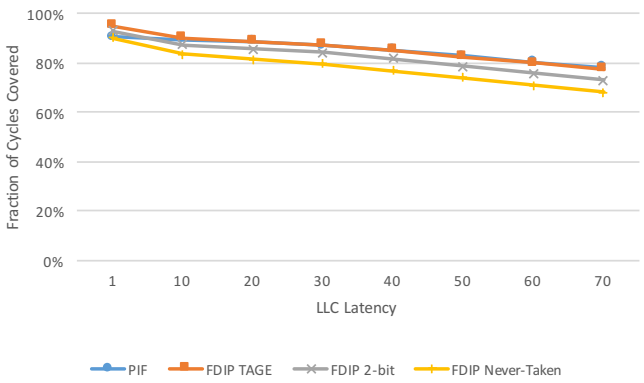

Figure 2. Percentage of front-end stall cycles covered by FDIP for different branch predictors and LLC access latencies.

covered as a metric of choice. We use the *stall cycles covered* metric over the more common *misses covered* metric to precisely capture impact of in-flight prefetches – the ones which have been issued but the requested block has not arrived to L1-I when needed by the fetch unit. Our metric captures stall cycles only on the correct execution path, since wrong-path stalls do not affect performance.

As shown in the figure, FDIP with TAGE performs nearly identically to PIF over a range of LLC access latencies. For very small LLC access latencies, PIF's coverage actually lags behind that of FDIP because PIF monitors the retire stream to advance the prefetcher. As a result, PIF is exposed to the pipeline latency upon a branch mispredict; in contrast, FDIP immediately redirects the prefetcher to the correct path.

To better understand the result, we plot the breakdown of sources of miss cycles for various prefetchers normalized to a no-prefetch baseline in Figure 3. We model a 16-core NUCA processor with a mesh-based interconnect, yielding an average LLC access latency of 30 cycles. In the figure, we focus on three prefetcher configurations: next line (NL), FDIP 32K, and PIF 32K. We breakdown the sources of misses into three categories: (i) sequential, (ii) discontinuities due to conditional branches, and (iii) discontinuities due to unconditional branches, which include function calls and returns.

As observed in prior work, sequential misses dominate, accounting for 40-54% of all miss cycles in the no-prefetch baseline. This explains why simple next-N-line prefetchers are extremely effective, as also observed in prior work on server instruction cache prefetching [20]. FDIP 32K covers the vast majority of cache misses across all three categories, delivering essentially identical coverage as PIF within each category.

While the TAGE branch predictor is quite accurate, it is not perfect. So how does FDIP attain such high coverage across a large LLC latency range despite the mispredicts? The answer lies in the fact that most Taken conditional branches have targets within a few cache lines of the branch instruction. As Figure 4 shows, nearly 92% of all taken branches have targets within four blocks of the current one. Because of the short offset, correctly predicting these branches is not essential for high coverage of conditional discontinuities. For such short backward branches, the targets are typically already found in the cache, while for forward branches, a prefetcher running far enough ahead will reach the cache block containing the target of the branch by simply following the fall-through path.

With sequential and conditional branches largely covered, the unconditional branches are the remaining source of discontinuities. The targets of these branches tend to be far away from the branch instruction itself, which is why next-N-line prefetchers generally fail to provide any coverage in this category. However, because these branches are unconditional, following them in FDIP does not depend on branch predictor's accuracy, thus guaranteeing high coverage for these discontinuities regardless of the branch predictor.

To confirm this intuition, we pair FDIP with a naive "never-taken" predictor that, for each conditional branch, simply follows the fall through path. We also evaluate FDIP with a simple bimodal predictor. To focus the study on the effects of branch predictor on FDIP, we use these two predictors only to drive FDIP; the back-end is still using TAGE to guarantee that pipeline resets due to branch mispredicts are identical to the baseline FDIP+TAGE configuration.

Figure 2 shows the results of the study. As expected, FDIP with the "never taken" predictor attains much of the coverage benefit of FDIP with TAGE. In other words, while a good branch predictor is essential to avoid pipeline squashes, even a naive branch predictor coupled with FDIP can be highly effective in covering discontinuities.

B. Does BTB Size Limit Coverage?

We next consider the BTB as a potential bottleneck. A small BTB may limit coverage by failing to discover discontinuities and by causing pipeline flushes due to branch mispredicts, thus preventing the branch predictor from running sufficiently far ahead of the core front-end.

Figure 5 shows FDIP's stall cycle coverage as a function of the BTB size and the LLC access latency. We use the same set of workloads as before and pair FDIP with the TAGE branch predictor. As the figure shows, going from a 32K to 2K BTB results in a 12% drop in stall cycle coverage. The reduction is relatively modest, and can be explained by the insight in Section III-A that most misses are due to a combination of sequential and conditional branches, and these can be covered by following the straight-line path. Thus, the difference in coverage between a large and small BTB must be attributed to unconditional branches. Because the targets of unconditional branches tend to reside far from their branch instructions, a BTB is essential to uncover these discontinuities.

To validate the intuition, we revisit Figure 3, this time focusing on the three FDIP configurations featuring 2K-, 8K-, and 32K-entry BTB. As expected, the largest difference in stall cycle coverage between a 2K- and 32K-entry BTB is due to unconditional branches. For instance, on Nutch, the 32K-entry BTB FDIP configuration improves coverage over the 2K-entry BTB by 3.4%, 2% and 7% for sequential, conditional and unconditional branches, respectively.

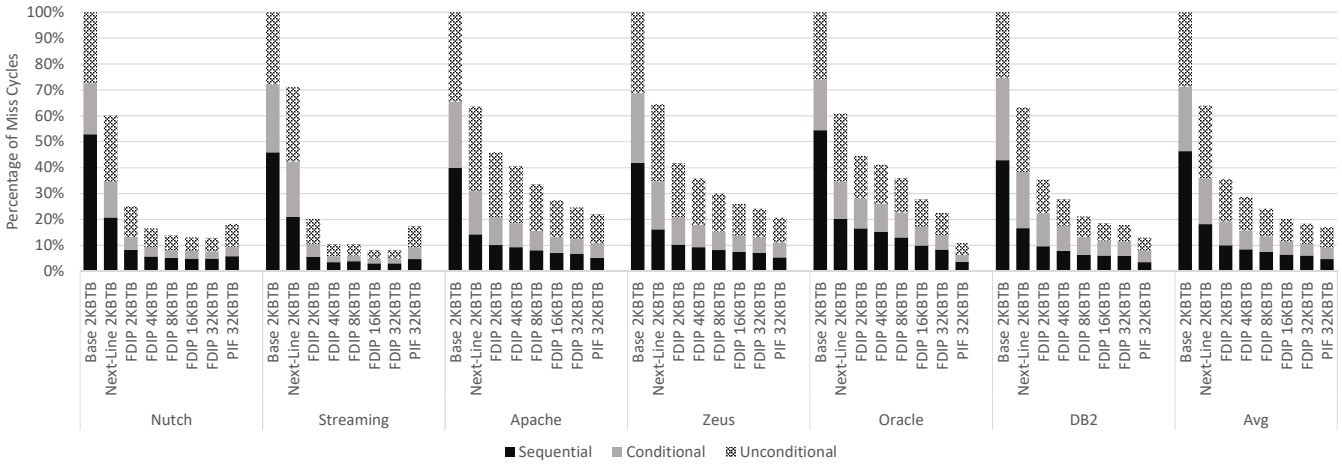

Figure 3. Source of miss cycles.

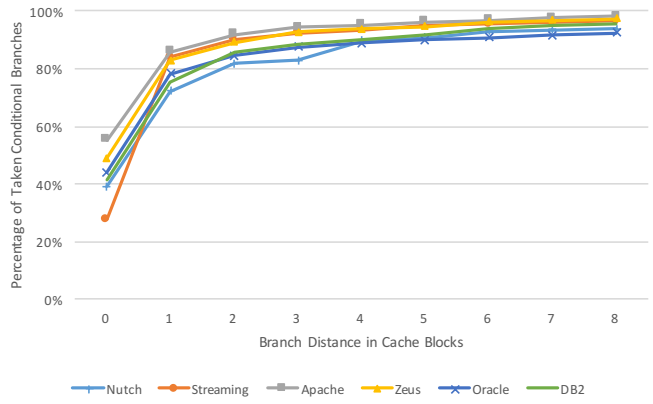

Figure 4. Taken Conditional Branch jump distance in number of cache blocks.

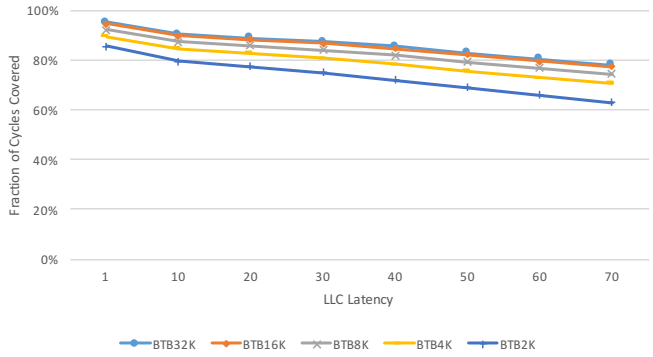

Figure 5. Percentage of front-end stall cycles covered by FDIP for different BTB sizes as a function of LLC latency

C. The Big Picture

As discussed in Section II, there are two major bottlenecks in the control flow delivery for server workloads that led to the rise of storage intensive prefetchers: L1-I misses and BTB misses. It is important to not only mitigate both of these bottlenecks, but to do so while avoiding the high metadata costs and complexity overheads incurred by prior techniques.

So far, we have demonstrated that a branch-predictor-directed prefetcher can replace the storage intensive state-of-the-art instruction prefetechers without compromising on L1-I miss coverage. The branch-predictor-directed prefetching is effective because the branch prediction accuracy is not a concern, since only conditional branches require a branch predictor and those tend to have short target offsets. Thus, branch mispredictions have little or no effect on L1-I miss coverage. Large BTBs are useful in discovering unconditional discontinuities, which account for only 16%, on average, of front-end stall cycles in a FDIP-enabled system with a 2K-entry BTB.

While a branch-predictor-directed prefetcher is effective in mitigating L1-I miss stalls, it does not tackle the BTB miss problem. As our opportunity study in Figure 1 shows, eliminating these misses leads to a performance improvement of up to 40%. The state-of-the-art approaches to capture this performance opportunity incur 100s of KB of storage overhead.

To avoid these massive storage overheads, we propose Boomerang - a metadata-free control flow delivery architecture that augments a conventional branch-predictor-directed front end to identify and resolve BTB misses, in addition to instruction cache misses. In doing so, Boomerang eliminates the front-end stalls associated with L1-I misses *and* the pipeline flushes caused by the BTB misses.

IV. Boomerang

Boomerang provides a unified solution to the L1-I and BTB miss problems while relying exclusively on the existing in-core metadata. For mitigating instruction cache misses, Boomerang leverages an existing branch-predictor-directed prefetcher, FDIP [19]. For resolving BTB misses, Boomerang exploits an insight made in prior work that the BTB can be populated by extracting branches and their targets from incoming cache blocks [14]. Unlike the prior work, however, Boomerang discovers and fills BTB misses using existing in-core structures and small augmentations to the FDIP prefetcher.

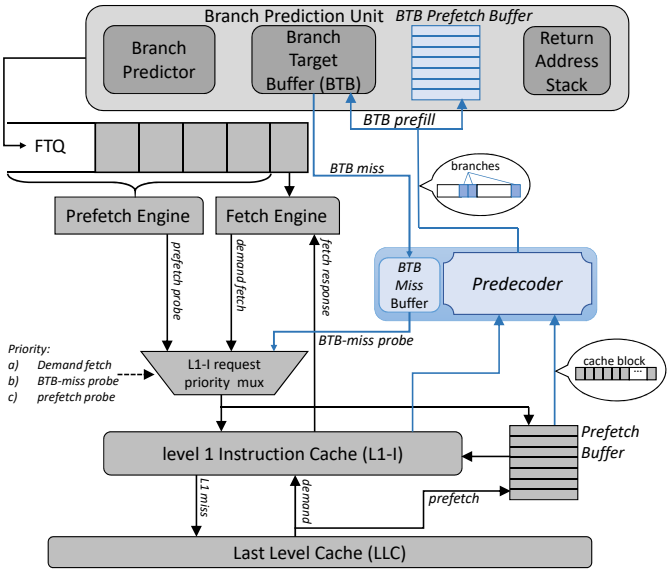

Figure 6. Boomerang microarchitecture. New components in light blue. (FTQ: Fetch Target Queue.)

Figure 6 shows the microarchitecture of Boomerang. The baseline components, including FDIP, are shown in dark grey, while Boomerang additions are in light blue. As Boomerang builds upon FDIP, we first describe the baseline FDIP microarchitecture. Next, we detail the additional components necessary to detect and prefill the BTB misses. Finally, we discuss the trade-offs and optimizations in the Boomerang microarchitecture.

A. Baseline: FDIP

FDIP employs a decoupled front-end where the fetch engine is decoupled from the branch prediction unit (consisting of the branch predictor, BTB and return address stack) by means of a deep fetch target queue (FTQ). Each FTQ entry holds fetch address information for a basic block which is defined as a sequence of straight-line instructions ending with a branch instruction[1]. The *branch prediction unit* makes a basic block prediction every cycle and pushes it to the FTQ. The *fetch engine* reads the head of FTQ and issues N *demand-fetch* requests to L1-I, where N is the fetch width. A typical in-order fetch engine waits for a response from L1-I, before issuing subsequent requests. Therefore, in case of an L1-I miss, the fetch engine needs to stall until the instructions are returned from the lower cache levels.

The *prefetch engine* is a critical component of FDIP that is responsible for prefetch address generation. As new basic block fetch addresses are inserted into the FTQ, the prefetch engine scans them to discover the L1-I blocks that will be accessed by the fetch engine in the near future. For every new L1-I block discovered, the prefetch engine sends a *prefetch probe* to the L1-I. The prefetch probe simply checks if the block is present in L1-I or in the *prefetch buffer*. If the block is already present, no further action is taken. However, if the probed block is missing, it is fetched from the lower levels of caches and is inserted into the L1-I's prefetch buffer. A subsequent demand-fetch hit in the prefetch buffer moves the block to L1-I.

Unlike the fetch engine, the prefetch engine does not require a response from the L1-I; therefore, it can sustain a high issue rate for prefetch probes, even if the probes miss in the L1-I. This attribute allows the prefetch engine to run far ahead of the fetch engine. Moreover, as the prefetch engine operates at a cache block granularity, it issues only a single prefetch probe for all the basic blocks residing in that cache block. This allows the prefetch engine to further race ahead of the fetch stream. As long as the prefetch engine is sufficiently ahead of the fetch engine, it can hide all of the L1-I misses through timely prefetching.

B. Boomerang Overview

While FDIP is effective in solving the front-end (frequent L1-I misses) problem, the overall performance still suffers because of pipeline flushes due to frequent BTB misses for eventually taken branches. Boomerang goes a step ahead of FDIP and prefills not only the missing L1-I blocks but also the missing BTB entries. By prefilling the BTB, it reduces the number of pipeline flushes and hence unblocks both front-end and back-end bottlenecks.

In order to fill the BTB, Boomerang first needs to detect that a BTB miss has occurred. Unfortunately, a conventional instruction-based BTB interprets all BTB lookups that miss in the BTB as a non-branch instruction. In other words, such a BTB design cannot distinguish between a non-branch instruction and a genuine BTB miss. Therefore, Boomerang leverages a different BTB organization – a basic-block-based BTB [22], which stores BTB entries at basic block granularity. This design guarantees that each BTB entry contains exactly one branch, whose target is another BTB entry. Therefore, if a BTB lookup fails to return a valid entry, it is guaranteed to be a genuine BTB miss.

Upon detecting a BTB miss, because the target and the basic block size of the missing entry are not known, the branch prediction unit stops feeding the FTQ with new entries until the BTB miss is resolved. The following actions are then executed to resolve the BTB miss:

1) A BTB miss probe for the cache block containing the starting address of the missing BTB entry is issued to the L1-I.
2) The corresponding cache block is fetched from L1-I or from the lower cache levels if not present in L1-I.
3) The cache block is sent to a *predecoder* that extracts all the branches and their targets.
   a) If branches are found after the starting address of missing BTB entry: the first such branch is the terminating branch of the missing BTB entry. A new BTB entry is created and stored in BTB.
   b) If no branch is found after the starting address of missing BTB entry: a BTB miss probe for the next

[1]This definition of basic block is slightly different from a conventional single-entry single-exit straight-line code

sequential cache block is issued and the process above repeats starting from step 2.

Furthermore, the BTB entries corresponding to the branches inside the predecoded cache block(s), except for the branch terminating the missing BTB entry, are stored in the *BTB prefetch buffer*. Whenever the BTB is accessed, the BTB prefetch buffer is accessed in parallel. On a hit to the BTB prefetch buffer, the corresponding entry is moved to the BTB. The remaining entries are replaced in a first-in-first-out manner.

Once the BTB miss is resolved, the branch prediction unit resumes its normal operation of feeding the FTQ.

C. Boomerang: Details

1) Prefetching Under a BTB Miss: As described above, Boomerang stops filling the FTQ on discovering a BTB miss, thereby potentially losing prefetch opportunities if the branch turns out to be not taken. In this section, we discuss the alternative design choices that can be opted for on a BTB miss.

No prefetch. As described in the previous section, the simplest design choice is to stop feeding the FTQ once the branch prediction unit detects a BTB miss. However, this approach results in missed prefetching opportunities and a loss of coverage if the branch is not-taken after the BTB miss resolution.

Unthrottled prefetching. In this design point, the branch prediction unit speculatively assumes that the branch corresponding to the missing BTB entry is not going to be taken and continues to feed the FTQ sequentially until the next BTB hit. However, such unthrottled prefetching can potentially pollute the L1-I prefetch buffer by over-prefetching on the wrong path. Moreover, wrong-path prefetching wastes bandwidth at the LLC and in the on-chip interconnect, which can cause a degradation in processor performance.

Throttled prefetch. This design point provides a balance between the lost opportunities in not prefetching on a BTB miss and potentially over-prefetching in Unthrottled prefetch. To capture the opportunity for sequential prefetching under a BTB miss, Throttled prefetch issues a prefetching request for next N sequential cache blocks if the BTB miss cannot be filled from the L1-I. Therefore, if the branch is not-taken following BTB miss resolution, prefetching opportunity is not lost due to next-N-block prefetching. On the other hand, if the branch is taken, the number of uselessly prefetched cache blocks is limited to just the next-N.

In our study, we found that Throttled Prefetch using the next-2-blocks policy outperforms other policies. A study showing the trade-offs of the design space is presented in Section (§VI-E1).

2) BTB miss probe prioritization: Because a BTB miss causes the branch prediction unit to stop feeding the FTQ, it also stops L1-I prefetching once the prefetch probes for the pending FTQ entries have all been issued. However, if the BTB miss can be resolved before all the prefetch probes have been sent, the branch prediction unit can again start feeding the FTQ and prefetching can continue uninterrupted. Thus, it is essential to resolve the BTB misses as early as possible.

To ensure swift resolution of BTB misses, Boomerang prioritizes BTB miss probes over pending prefetch requests. As shown in Figure 6, the *L1-I request priority mux* steers a BTB miss probe to L1-I before any prefetch probe generated by the prefetch engine. This prioritization facilitates a fast resolution of BTB misses and reduces the likelihood of stalling L1-I prefetching.

Processor	16-core, 2GHz, 3-way OoO 128 ROB, 32 LSQ
Branch Predictor	TAGE [21] (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 I
MICROARCHITECTURAL PARAMETERS

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 II
WORKLOADS

V. METHODOLOGY

We evaluate Boomerang on a set of enterprise and open-source scale-out applications listed in Table II using Flexus [23], a full system multiprocessor simulator. 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 [24] 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 collects statistics over the next 50K cycles. We use ratio of the 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 [23]. The performance is measured with an average error of less than 2% at a 95% confidence level.

Our modeled processor is a 16-core tiled CMP. Each core is 3-way OoO resembling an ARM Cortex-A57 core. The

microarchitectural parameters are listed in Table I. We model a 2K-entry BTB, which is a practical size for a single-cycle access latency.

A. Control Flow Delivery Mechanisms

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

Discontinuity Prefetcher (DIP): DIP records the control flow discontinuities that result in L1-I misses in a discontinuity prediction table. For maximum L1-I miss coverage, the table needs to store upto 8K entries. Spracklen et al. [20] proposed to complement DIP with a Next-4-Line prefetcher to cover the sequential misses. We found that Next-2-Line prefetcher works better than Next-4-Line due to higher prefetch accuracy in our settings. Therefore, we use a Next-2-Line prefetcher along with an 8K entry discontinuity prediction table.

Fetch Directed Instruction Prefetch (FDIP): As described in Section IV-A, FDIP decouples the branch prediction unit from the fetch engine by means of a fetch target queue (FTQ). The instruction prefetches are issued from the FTQ entries. We model a 32-entry FTQ with each entry holding the start address of a basic block and its size. We use a basic block-oriented BTB to drive FDIP. On a BTB miss, FDIP enqueues a single sequential instruction address into the FTQ per cycle and access the BTB with the this sequential address until the next BTB hit.

Shared History Instruction Prefetch (SHIFT): SHIFT is a temporal-stream-based instruction prefetcher that records the correct-path instruction history and replays it to predict future instruction accesses [11]. SHIFT virtualizes the instruction history metadata into the LLC and shares it among all cores executing a common workload. For high L1-I miss coverage, SHIFT requires at least a 32K-entry instruction history and an 8K-entry index table.

Confluence: Confluence, the only other technique that tackles both L1-I and BTB misses, relies on SHIFT for instruction prefetching. Confluence predecodes the prefetched L1-I blocks, identifies branch instructions, and inserts them into the BTB. We model Confluence as SHIFT augmented with a 16K-entry BTB, which provides a generous upper bound on Confluence's performance [14]. Our storage calculation assumes a 1K-entry block-oriented BTB per the original Confluence design.

Boomerang: As described in Section IV, Boomerang employs FDIP for L1-I prefetching and augments it with BTB prefilling. Like FDIP, Boomerang employs a 32-entry FTQ. Furthermore, Boomerang uses a throttled prefetch approach that prefetches the next-2 sequential cache blocks on a BTB miss that is not filled from the L1-I. Also, our evaluated Boomerang design employs a 32-entry BTB prefetch buffer.

VI. Evaluation

In this section, we first evaluate how effective Boomerang is in delivering control flow, i.e. reducing pipeline squashes and front-end stall cycles, compared to other alternatives. Second, we evaluate the performance benefits attained owing to Boomerang's control flow delivery. Third, we compare the storage cost of Boomerang with other control flow delivery mechanisms. Then, we assess the efficacy of throttled (next-N-block) prefetching and finally, evaluate Boomerang's sensitivity to LLC latency.

A. Branch Misprediction Squashes

The BTB misses and branch direction/target mispredictions are the two major sources of pipeline squashes. Figure 7 shows the number of pipeline squashes per 1K instructions coming from these two sources for different prefetching schemes. On average, both BTB misses and branch mispredictions are equally responsible for pipeline squashes as can be seen for prefetching schemes that don't target reducing BTB misses, i.e. Next-line, DIP, FDIP and SHIFT. Moreover, the contribution of BTB misses in overall squashes is especially evident in DB2, where about 75% of pipeline squashes are caused by BTB misses.

Only Boomerang and Confluence target BTB misses and their associated pipeline squashes. Both techniques are able to eliminate more than 85% of BTB miss-induced squashes. Compared to Confluence, Boomerang is generally more effective, exceeding Confluence's squash reduction by over 10%, on average. The reason Boomerang is more effective is because it detects every BTB miss and fills it, thus ensuring that the execution stays on the correct path. In contrast, Confluence does not detect BTB misses; rather, it relies on a prefetcher to avoid them altogether. The downside of Confluence's approach is that if an L1-I prefetch is incorrect or not timely (i.e., has not reached the L1-I before the front end), the branches corresponding to the block are absent from the BTB. In these circumstances, Confluence's front end follows a sequential instruction stream, as if there were no branch instructions present.

By eliminating BTB misses, Boomerang and Confluence achieve almost 2x reduction in total squashes compared to all other configurations. It is also important to note that some of the eliminated BTB misses can still cause pipeline squashes due to direction/target misprediction. For example, as shown in Figure 7, on average SHIFT sees 10.22 squashes per 1K instructions due to branch direction/target mispredictions. This number rises to 11 squashes per kilo-instruction for Confluence due to additional direction and target mispredictions incurred by the prefilled BTB entires. However, as evident from the figure, the incidence of these additional squashes is negligible.

B. Front-end Stall Cycles Covered

To show the effectiveness of different L1-I prefetching techniques, we present the number of front-end stall cycles covered by them in Figure 8. The average coverage is similar for all control-flow-aware prefetchers; however, there are important differences across the individual benchmarks. On average, Boomerang eliminates 61% of the stall cycles performing similarly to Confluence, which covers 60% of stall cycles. Upon closer inspection, we find that Boomerang performs better than Confluence on four out of six applications: Apache, Nutch, Streaming and Zeus. On these, Boomerang benefits from

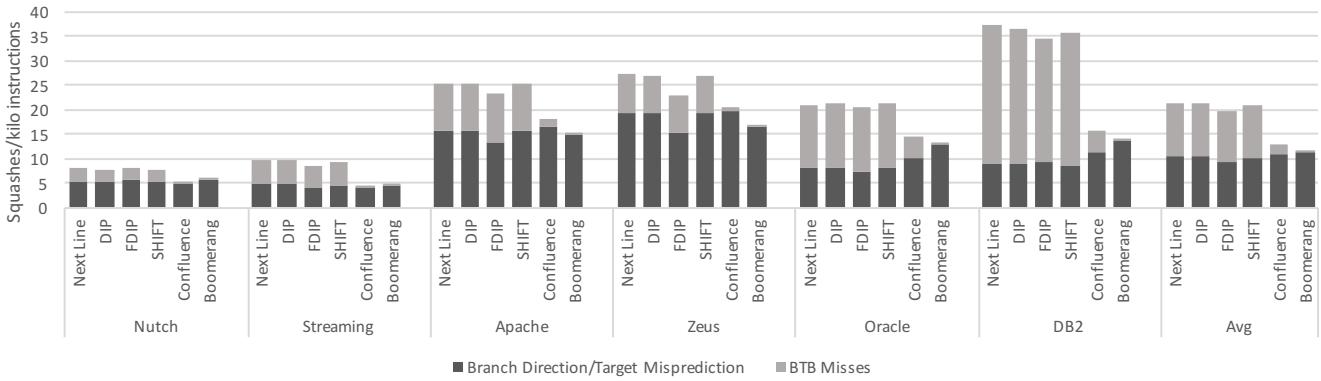

Figure 7. Number of pipeline squashes per kilo instructions with a 2K-entry BTB.

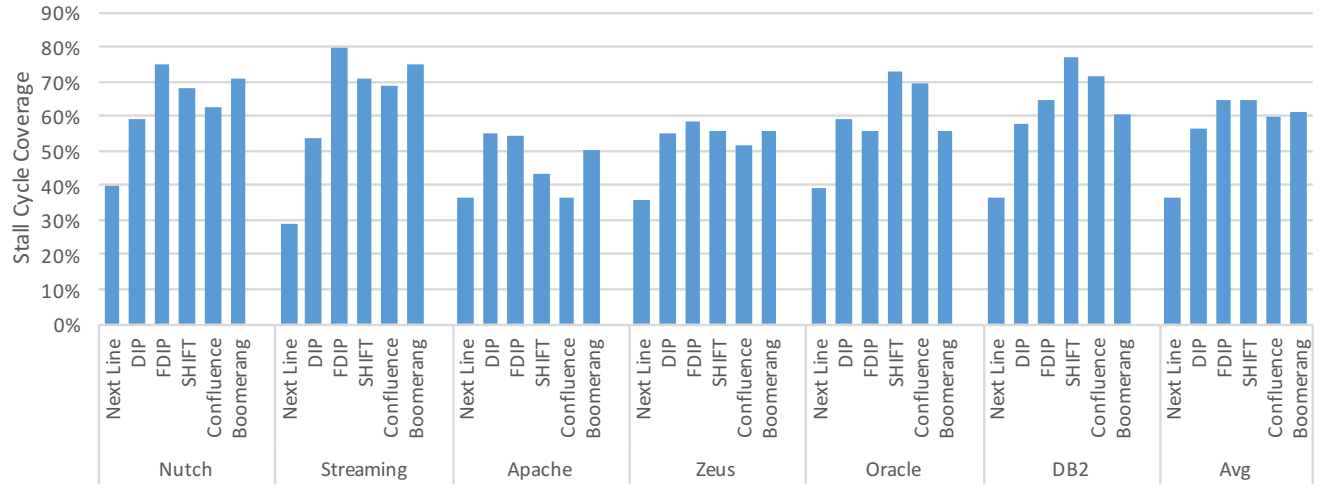

Figure 8. Front-end stall cycles covered with a 2K-Entry BTB over no-prefetch baseline.

fast accesses to local state (i.e., its branch prediction structures). In contrast, the SHIFT prefetcher that Confluence relies on must access LLC-embedded history metadata. Therefore, every time SHIFT mispredicts an instruction cache block access sequence, it first needs to load the correct sequence from the LLC before starting issuing prefetches on the correct path. In contrast, Boomerang can start issuing prefetches on the correct path as soon as a misprediction is detected.

On the two other applications, Oracle and DB2, Boomerang is surpassed by Confluence. The reason for Boomerang's inferior coverage is a high BTB miss rate, which forces Boomerang to frequently stall for prefilling each BTB miss. Because no BTB-directed instruction prefetches are generated while a BTB miss is pending, instruction stall cycle coverage suffers.

It is also interesting to note that FDIP and SHIFT provide slightly better coverage than Boomerang and Confluence, even though the latter rely on the respective former mechanisms for instruction prefetching. The reason for this seeming paradox lies in the fact that the data in Figure 8 shows only the correct-path stall cycles covered. Meanwhile, wrong-path accesses may prefetch instruction blocks on the eventually-correct path, thus effectively reducing stall cycles. As FDIP and SHIFT go on the wrong path more frequently than Boomerang and Confluence due to more frequent BTB misses (Figure 7), their wrong-path prefetches lower the stall cycles on the correct path.

C. Performance Analysis

Figure 9 shows the performance improvements for different instruction supply mechanisms over a baseline without any instruction/BTB prefetching. The results follow those of Figure 7 and Figure 8. Boomerang, on average, provides 28% speedup over the baseline, outperforming Confluence by 1%. Similar to the stall cycle coverage results, Boomerang lags behind Confluence on Oracle and DB2 due to lower stall cycle coverage. For Zeus and Apache, Boomerang significantly outperforms Confluence due to the combination of higher stall cycle coverage and fewer pipeline squashes.

It is worth noting that the complete control flow delivery mechanisms, Boomerang and Confluence, outperform the

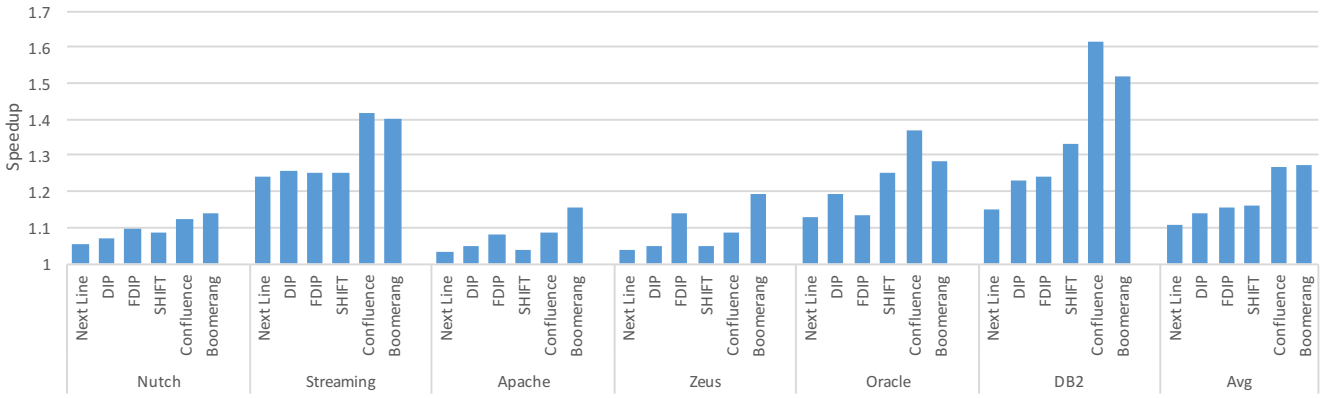

Figure 9. Speedup with a 2K-Entry BTB over no-prefetch baseline.

instruction prefetchers, including state-of-the-art SHIFT and DIP, by a large margin, averaging 11%, by eliminating pipeline squashes on top of the instruction cache stalls. This result underscores the advantage of complete control flow delivery as opposed to just L1-I prefetching.

D. Boomerang vs Confluence: Storage, Complexity and Energy

We first compare the storage requirements of Boomerang and Confluence. The baseline architecture, without any prefetching, maintains a BTB and branch predictor to guide the instruction fetch engine. An FTQ of a few entries is employed to buffer the fetch addresses before they can be used to access the L1-I. A prefetch buffer is usually employed by L1-I prefetchers to limit L1-I pollution.

Given all the components in baseline, Boomerang requires minimal additional hardware to enable both L1-I and BTB prefetching. First, it needs a deeper FTQ to detect and prefetch the missing L1-I and BTB entries ahead of the core front-end. Each FTQ entry contains the start address of the basic block (46-bits) and its size (5-bits). Boomerang uses a 32 entry FTQ therefore requiring 204 bytes of storage. Second, Boomerang employs a 32 entry BTB prefetch buffer to avoid BTB pollution. Each buffer entry contains a tag (46-bits), target address (30-bits, maximum offset in SPARC), branch type (3-bits) and basic block size (5-bits). Therefore, the 32 entry BTB prefetch buffer requires 336 bytes of storage. Thus, the total storage requirement of Boomerang is 540 bytes.

Confluence, on the other hand, employs a 32K-entry instruction history table and an 8K-entry index table for L1-I prefetching. To store the index table, Confluence extends the LLC tag array, requiring 240KB of dedicated storage. The instruction history table is virtualized into the LLC. As such, it does not require dedicated storage but does result in a lower effective LLC capacity.

On the complexity side, we argue that Boomerang is considerably simpler than Confluence. The complexity of Confluence stems from the following factors:

System Level Support: Confluence reserves a portion of physical address space to store instruction history in LLC. Furthermore, the cache lines holding the history metadata must be pinned. To fulfill these requirements Confluence requires system-level support. Boomerang, on the other hand, is transparent to the software stack.

LLC Tag Array Extension: Confluence extends LLC tag array to store the index table. Therefore, the storage cost becomes a factor of LLC size in addition to instruction history size. For an 8MB LLC and 32K entry instruction history, the LLC tag array extension results in 240KB of storage overhead. On the contrary, Boomerang does not require any changes to LLC.

Workload Consolidation: Confluence virtualizes instruction history in LLC and shares it among all the cores to reduce per core storage overhead. However, this technique is effective only when all the cores are running the same application. As the number of applications running on the CMP increases, Confluence needs to store one instruction history table per application in LLC, reducing the effective LLC capacity by over 200KB with each additional application. Boomerang does not carve LLC capacity in any way.

Increased On-chip Interconnect Traffic: As the instruction history and index tables are stored in LLC, Confluence generates additional network traffic to retrieve the prefetching metadata from LLC. Boomerang, on the other hand, uses only core-private state from its local BTB and branch direction predictor.

History Generation: Confluence relies on one of the cores to generate instruction history which is then shared among all the cores to issue prefetches. If the history generator core switches to a housekeeping task, such as garbage collection, the history generation will suffer, which might adversely affect the prefetch accuracy in other cores. Prefetch generation in Boomerang, on the other hand, is private to each core and hence, is not affected by the activities of other cores.

All the above factors make Confluence significantly more complex than Boomerang, whose only control logic requirements are for:

- Halting fetch address generation on a BTB miss.

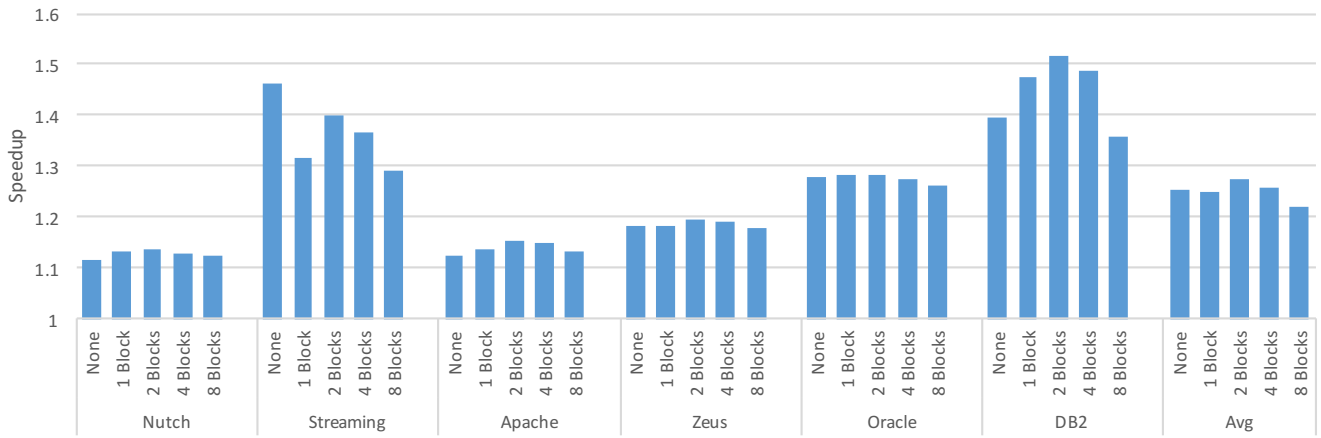

Figure 10. Boomerang's performance sensitivity to next-N-block prefetching on BTB misses.

- Prioritizing BTB misses over other prefetch requests.
- Looking up an entry in the BTB prefetch buffer in parallel with the BTB.
- Issuing prefetches for next-2-lines on a BTB miss.

Even though complexity is not straightforward to quantify, in practice it has a large impact on design decision.

In terms of energy-efficiency, Boomerang has an advantage over prior techniques, including Confluence, because it does not introduce any dedicated storage-intensive hardware structures or cause additional metadata movement. In general, however, prior work has shown that even for storage-intensive prefetchers, the energy costs comprise a small fraction of the processor's total power budget [25].

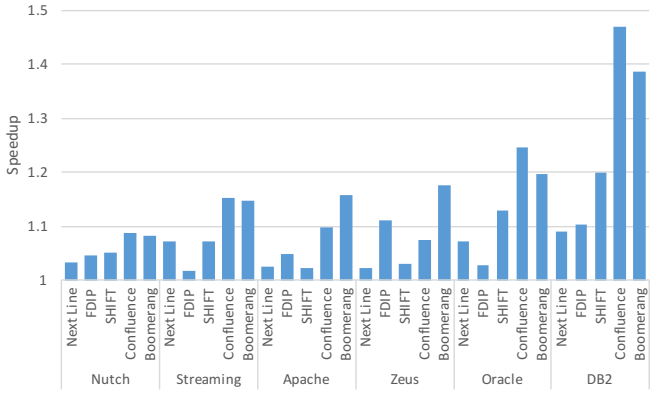

Figure 11. Performance at a lower LLC round-trip latency.

E. Sensitivity Analysis

1) Next-N-line prefetches on a BTB Miss: As discussed in Section IV-C1, on a BTB miss that cannot be prefilled from L1-I, Boomerang issues prefetch for next two sequential cache blocks in addition to the block that contains the missing BTB entry. Figure 10 shows the sensitivity of performance to the number of next-N-blocks prefetched. As the figure shows, prefetching next-2-blocks provides optimal performance. The effect of prefetching next-N-blocks is notable especially in DB2, where prefetching next-2-blocks provide 12% performance improvement over not prefetching at all. It is also important to note that prefetching more than two blocks generally results in performance degradation compared to next-2-blocks as erroneous prefetches delay the useful blocks.

Streaming is an exception where not prefetching any block provides the maximum performance. Prefetching next-N-blocks degrades performance because the majority of these blocks end up being discarded, and thus polluting network and LLC bandwidth and L1-I prefetch buffer. Next-1-block prefetching performs worse than next-2 and next-4-block prefetching due to the taken branches. These branches skip the next sequential block and jump to the blocks following it. Therefore, the next-1-block prefetching suffers from particularly poor accuracy as it fails to prefetch useful blocks, whereas next-2 and next-4-block prefetching does bring in some useful blocks even for taken branches.

2) Effect of LLC Round-trip Latency: Figure 11 shows the speed up of the different techniques under a lower LLC round-trip latency. In particular, we model a wide crossbar interconnect that lowers the average LLC round-trip latency from 30 cycles in the mesh down to 18 cycles.

As the figures shows, the general trends remain the same as in a mesh-based NOC. Boomerang maintains its slight performance advantage over Confluence even at the lower LLC latency. The absolute benefits of all schemes reduce, however, because the L1-I misses are now less costly due to the lower LLC latency.

VII. Conclusion

Effective control flow delivery is crucially important for server workloads with their massive instruction footprints. Indeed, instruction cache and BTB misses can cause a significant performance degradation. Although there have been a number of techniques proposed to address the control flow delivery bottleneck, every one of them requires separate metadata

structures, translating into significant storage and complexity costs.

This work introduced Boomerang, a metadata-free architecture for control flow delivery. Boomerang leverages a branch-predictor-directed prefetcher that uses existing in-core metadata for solving the instruction cache problem. Contrary to conventional wisdom, we have shown that a branch-predictor-directed prefetcher can be effective in discovering the future instruction stream despite limited branch predictor accuracy and a modest BTB storage budget. Our second contribution is in demonstrating that BTB misses can be identified and filled by the branch-predictor-directed instruction prefetcher at minimal additional cost and complexity. By eliminating BTB misses, Boomerang is able to avoid a large fraction of performance-degrading pipeline flushes. Our results show that Boomerang is able to match the performance of Confluence, the state-of-the-art control-flow delivery scheme, without its associated storage and complexity costs.

VIII. ACKNOWLEDGEMENTS

We 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] 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*, pp. 266–277, 1999.

[2] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS*, pp. 37–48, 2012.

[3] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *International Symposium on Computer Architecture*, pp. 184–195, 2009.

[4] 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*, pp. 158–169, 2015.

[5] 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*, pp. 15–26, 1998.

[6] 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*, pp. 307–318, 1998.

[7] 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*, pp. 260–271, 2013.

[8] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero, "Code Layout Optimizations for Transaction Processing Workloads," in *International Symposium on Computer Architecture*, pp. 155–164, 2001.

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

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

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

[12] 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*, pp. 71–82, 2013.

[13] 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*, pp. 313–324, 2009.

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

[15] *ThunderX ARM Processors: Workload Optimized Processors for Next Generation Data Center and Cloud Applications*. www.cavium.com/ThunderX_ARM_Processors.html.

[16] *Oracle SPARC T5-2 Server*. www.oracle.com/servers/sparc/t5-2/index.html.

[17] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out Processors," in *International Symposium on Computer Architecture*, 2012.

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

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

[20] 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*, pp. 225–236, 2005.

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

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

[23] 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.

[24] 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*, pp. 84–95, 2003.

[25] J. N. Amaral and J. Torrellas, eds., *International Conference on Parallel Architectures and Compilation*, ACM, 2014.