Warming Up a Cold Front-End with Ignite

Citation for published version:
Schall, D, Sandberg, A & Grot, B 2023, Warming Up a Cold Front-End with Ignite. in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO).* Association for Computing Machinery (ACM), pp. 254-267, 56th IEEE/ACM International Symposium on Microarchitecture, Toronto, Ontario, Canada, 28/10/23. https://doi.org/10.1145/3613424.3614258

Digital Object Identifier (DOI):
10.1145/3613424.3614258

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)

Warming Up a Cold Front-End with Ignite

David Schall
david.schall@ed.ac.uk
University of Edinburgh
Edinburgh, UK

Andreas Sandberg
andreas.sandberg@arm.com
Arm Ltd.
Cambridge, UK

Boris Grot
boris.grot@ed.ac.uk
University of Edinburgh
Edinburgh, UK

ABSTRACT

Serverless computing is a popular software deployment model for the cloud, in which applications are designed as a collection of stateless tasks. Developers are charged for the CPU time and memory footprint during the execution of each serverless function, which incentivizes them to reduce both runtime and memory usage. As a result, functions tend to be short (often on the order of a few milliseconds) and compact (128–256 MB). Cloud providers can pack thousands of such functions on a server, resulting in frequent context switches and a tremendous degree of interleaving. As a result, when a given memory-resident function is re-invoked, it commonly finds its on-chip microarchitectural state completely cold due to thrashing by other functions — a phenomenon termed *lukewarm* invocation.

Our analysis shows that the cold microarchitectural state due to lukewarm invocations is highly detrimental to performance, which corroborates prior work. The main source of performance degradation is the front-end, composed of instruction delivery, branch identification via the BTB and the conditional branch prediction. State-of-the-art front-end prefetchers show only limited effectiveness on lukewarm invocations, falling considerably short of an ideal front-end. We demonstrate that the reason for this is the cold microarchitectural state of the branch identification and prediction units. In response, we introduce Ignite, a comprehensive restoration mechanism for front-end microarchitectural state targeting instructions, BTB and branch predictor via unified metadata. Ignite records an invocation's control flow graph in compressed format and uses that to restore the front-end structures the next time the function is invoked. Ignite outperforms state-of-the-art front-end prefetchers, improving performance by an average of 43% by significantly reducing instruction, BTB and branch predictor MPKI.

CCS CONCEPTS

• **Computer systems organization → Architectures**; **Cloud computing**.

KEYWORDS

Microarchitecture, instruction delivery, front-end prefetching and serverless

MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614258

ACM Reference Format:
David Schall, Andreas Sandberg, and Boris Grot. 2023. Warming Up a Cold Front-End with Ignite. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614258

1 INTRODUCTION

Serverless computing has emerged as a popular cloud programming paradigm. Serverless applications comprise a task graph of stateless *functions* that run on-demand in response to an invocation (e.g, a mouse click or from another function). In serverless, developers pay for the CPU time and memory footprint only during a function's execution. For developers, this model is attractive since it offers high scalability on a pay-for-what-you-use basis. Meanwhile, cloud providers get high utilization out of their infrastructure by shutting down idle functions — a task made simple by their stateless nature.

The serverless model incentivizes developers to have fine-grained functions that can be scaled independently of each other and that have a low execution time and memory footprint. Indeed, studies show that serverless functions tend to have short execution durations, as low as a millisecond or less [18, 19] and memory footprints often in the range of 128–256 MiB [18]. This allows thousands of functions to be packed onto a modern server, with typical invocation intervals ranging from seconds to minutes [54].

The large number of colocated functions with extremely short running times result in unprecedented frequency of context switching. Moreover, the relatively long inter-invocation intervals mean that thousands of other functions may execute between two invocations of a given function. As a result, a re-invoked function that is live on a server may find its microarchitectural CPU state (including caches and in-core structures) completely cold due to thrashing by interleaved invocations of other functions. Recent work has termed this phenomenon a *lukewarm* invocation [51].

Our characterization of a suite of serverless functions on a contemporary Intel Xeon Ice Lake server CPU reveals that the cold microarchitectural state due to lukewarm invocations causes a performance degradation of 2x or more as compared to back-to-back invocations in which the microarchitecture stays warm. These results corroborate the results of a prior study on an older (Broadwell-class) CPU [51], indicating that the problem is fundamental and is unaddressed by a newer, more aggressive microarchitecture.

A detailed analysis of the sources of performance stalls reveals the front-end (i.e., instruction delivery, branch identification and branch prediction) to be the main culprit behind the poor performance on lukewarm invocations. Collectively, the front-end is responsible for two-thirds of performance degradation as compared to executions with warm microarchitectural state. These results echo earlier work in this space, which identified off-chip instruction misses as the single largest contributor to the performance loss in lukewarm executions [51]. To mitigate the problem, the work proposed a dedicated prefetcher, called Jukebox, to bring the previously-recorded instruction working set into the L2 cache upon a function's invocation [51].

Our evaluation of state-of-the-art front-end prefetchers — Jukebox, Boomerang (which proactively fills the L1-I and the BTB) [41], and the two combined — shows that they fall considerably short of an ideal front-end that delivers 61% average speed-up over an aggressive next-line prefetcher. In comparison, the strongest performer — a combination of Jukebox and Boomerang — provides only a 20% speed-up as it fails to reduce the high miss rate across all front-end structures, namely the L1-I (26 MPKI), BTB (13 MPKI) and the conditional branch predictor (21 MPKI).

We perform a root-cause analysis to understand why the state-of-the-art in front-end prefetching is performing so poorly and find that the cold microarchitectural state of the BTB and the conditional branch predictor (CBP) is compromising prefetching performance. Misses in the BTB and mispredictions of conditional branches constantly drive the front-end (both demand and prefetch) off the correct path, resulting in poor prefetching performance and frequent pipeline flushes. Moreover, the short execution time of serverless functions does not allow the warm-up time of these structures to be amortized.

To overcome the cold front-end challenge of lukewarm invocations, we propose Ignite, a comprehensive *restoration* mechanism for front-end microarchitectural state targeting instructions, BTB and CBP via unified metadata. The underlying insight behind Ignite is that the BTB working set provides an efficient way of approximating a program's (or container's) control flow graph and can be used for instruction, BTB *and* CBP prefetching. Ignite capitalizes on this insight by monitoring BTB insertions to create compressed control flow records that are stored in main memory. When the same function is invoked again, the metadata is streamed from memory and used to generate instruction prefetches and restore the state of the BTB and the bimodal branch predictor. Ignite has low logic complexity, is easy to integrate with existing front-end prefetchers, and seamlessly supports thousands of functions on a server by virtue of having no metadata on-chip. Our evaluation of Ignite shows that it improves performance by 43%, on average, and significantly reduces the miss rate in all front-end structures as compared to prior art.

To summarize our contributions:

- We corroborate prior work, demonstrating a significant performance degradation in the execution of lukewarm serverless functions due to cold microarchitectural state. The main source of the performance degradation is the front end: instruction fetch and the BPU.
- We show that the combination of state-of-the-art front-end prefetchers, Boomerang [41] + Jukebox [51], improves performance by only 20%, on average, as compared to 61% with an ideal front-end. Cold BPU state is to blame.
- We introduce Ignite, a record-and-replay *restoration* mechanism that uses a unified control flow representation recorded during one invocation of the function to prefetch instructions and restore the BPU's state upon the next invocation.
- We demonstrate that Ignite improves performance by 43%, on average, by providing a significant reduction in L1-I, BTB and CBP MPKI.

2 MOTIVATION

2.1 Serverless Basics

In the serverless model, developers structure their applications as a task graph of stateless event-triggered functions[1]. Functions are invoked on-demand, with all resource management decisions (e.g., whether to spawn a new function instance or use an existing one) ceded to the cloud provider. For cloud providers, the serverless model is a way to get a high resource utilization and monetize it — a challenge that's difficult to meet with traditional "rented" VMs that may stay idle for indefinite periods of time while holding expensive hardware resources. Cloud providers pass the efficiency gain of serverless to the developers in the form of pay-per-use billing, whereby developers pay only for the CPU time and memory footprint of each function invocation. This model contrasts starkly with traditional cloud software deployments, where developers "rent" cloud resources to run virtual machines (VMs) and pay for the uptime of their VMs regardless of utilization.

Because cloud providers bill only for the actual CPU usage and memory footprint of a running function, they have an incentive to shut-down inactive function instances to recycle resources — a model that's enabled by the fact that functions are stateless. However, bringing up a new instance is expensive in terms of storage and network bandwidth required to fetch the function image and the CPU time needed to launch the container. Thus, cloud providers tend to keep recently-invoked instances alive (aka *warm*), for some number of minutes in hopes of receiving invocations that can be served by these instances.

For developers, the pay-per-use model incentivizes compact functions that, in many cases, run for merely a few milliseconds or less and consume as little as 128 MiB of memory [18–20, 54]. For cloud operators, the combination of compact functions and keep-alive periods means that thousands (!) of serverless functions can be packed onto a typical cloud server [4, 6]. Given that a typical warm function instance is invoked once every several seconds or minutes [54], hundreds or even thousands of other function instances may run between two consecutive invocations of a given instance, resulting in an unprecedented frequency of context switches and a vast number of interleaving contexts on that server.

2.2 Lukewarm Serverless Invocations

Prior work [51] showed that the massive degree of interleaving causes extensive thrashing of on-chip microarchitectural state, including caches and in-core structures. As a result, when a given warm function is re-invoked, it tends to find the on-chip microarchitectural state completely cold — a phenomenon termed *lukewarm* invocation [51]. Compared to back-to-back invocations of a given

[1]The serverless model easily supports stateful services. Any state that must persist beyond the function call boundaries must be saved to a conventional datastore or propagated to the next function in the task graph as part of the invocation.

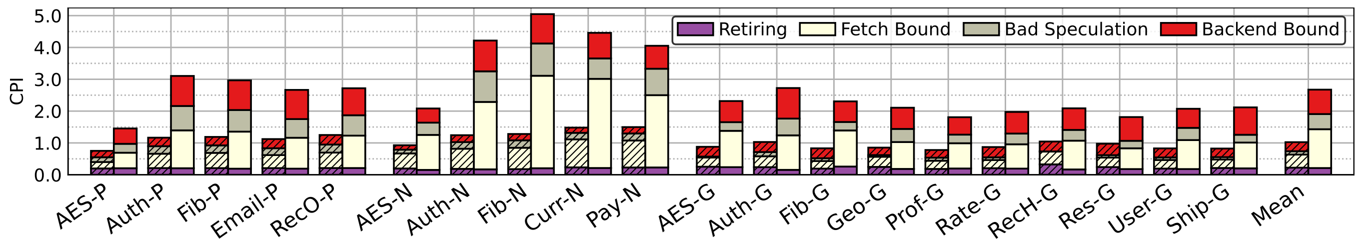


Figure 1: CPI stack for interleaved execution (shaded) vs. back-to-back execution (solid) on an Intel Ice Lake CPU.

function instance, lukewarm invocations incur a significant performance overhead due to cold microarchitectural state that results in frequent cache misses, branch mispredictions, etc. Often, the overhead cannot be amortized over a long execution interval due to the short duration of many serverless functions [51].

We corroborate the prior work by showing that, indeed, cold microarchitectural state degrades execution efficiency for serverless functions. We use a suite of 20 diverse serverless functions, which we run on an Intel Ice Lake CPU. Details of our benchmarking setup can be found in Section 5. To achieve statistically meaningful results in a tractable amount of time and with a high degree of reproducibility, we model the effects of interleaving by using a stressor [16] that runs in-between invocations of a function-under-test (FUT) on the same core as the FUT. As shown in prior work, at the microarchitectural level, this methodology achieves a comparable effect to interleaving numerous invocations of many different functions [51].

Figure 1 plots cycles per instructions (CPI) for back-to-back invocations of the same function instance compared to interleaved invocations. Note that, performance-wise, back-to-back invocations are the best-case scenario since the re-invoked function finds all microarchitectural resources warm. As the figure shows, interleaved executions consistently increase the CPI (i.e., degrade performance) by 100-294% (162% on average) as compared to back-to-back invocations.[2]

To identify the sources of performance degradation due to interleaving, we use performance counters to break down execution cycles into four categories: retiring, instruction fetch stalls (cache and TLB misses for instructions), bad speculation (BTB misses and mispredictions of conditional branches) and back-end stalls (cache and TLB misses for data). Our classification roughly follows the Intel Top-Down methodology [59]. The first category, *retiring*, is the only "good" one, representing cycles where useful work was completed. The three other categories are characterized by pipeline stalls that impede efficient execution [3]. Because the branch predictor unit (BPU), composed of the branch target buffer (BTB) and the conditional branch predictor (CBP), works together with the fetch unit to steer control flow and deliver instructions to the pipeline, we refer to fetch stalls and bad speculation collectively as *front-end* stalls.

[2]While our results are consistent with prior work characterizing lukewarm invocations [51], our numbers cannot be directly compared with those reported in [51]. In addition to the fact that we study a much more recent server featuring an Intel Ice Lake CPU (prior work studied an Intel Broadwell CPU), we have made a number of improvements to the measurement methodology and have used more recent versions of the functions.

[3]The classification is not always precise since stalls can overlap with each other and with retirement of other instructions [59].

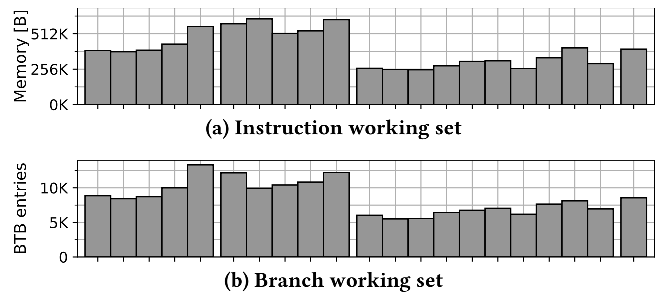


Figure 2: Working sets of serverless functions for CPU front-end structures.

As Figure 1 shows, the performance degradation under interleaved executions is caused by a significant increase in stall cycles across all stall categories. By far, the largest increase is observed in front-end stalls. Collectively, front-end stalls increase by 130-490% (215% on average), which corresponds to two-thirds of the overall performance degradation. These results corroborate prior work [51] and show that even on latest-generation hardware, lukewarm invocations of serverless functions result in poor microarchitectural efficiency largely due to the front-end bottleneck.

2.3 Severless Working Set Characteristics

To get a better understanding of why serverless workloads put such high pressure on the CPU front-end we examine the working sets of serverless function invocations. For this analysis, we use the gem5 simulator [13, 45] to run the same set of serverless functions as in the hardware experiments (Section 2.2). We leverage the vSwarm-μ framework [43], which enables us to run the entire software stack (including Docker, OS, full gRPC stack) as used for our hardware characterization in gem5's full-system mode. Details of our simulation and workload setup can be found in Section 5.3. Starting from a checkpoint, we trace the execution of 25 consecutive invocations and record instruction cache accesses at cache block granularity and allocations in the BTB. We remove repeating addresses to identify the instruction working set and the branch working set. Note that never-taken branches do not consume BTB (and BPU) capacity [2, 3]. Therefore, the BTB working set, which contains branches that are taken at least once, represents the branch working set.

Figure 2 presents the average instruction (a), and branch working set (b) accumulated during one invocation. The graphs show that, despite short execution times, serverless functions execute large amounts of code and a large number of unique branches relative to 32 KiB instruction cache and 5 K entry BTB found in Intel's Ice Lake server [29]. A single function invocation touches 240–620 KiB of code memory and accumulates branch working sets ranging from 5.4 K BTB entries (Auth-G) to almost 14 K BTB entries (RecO-P).

Our findings show that the front-end microarchitectural state may overwhelm existing CPU front-end structures even for a single serverless function. With thousands of functions interleaving their executions on a single server, a CPU is unable to retain the front-end microarchitectural state across invocations, which explains the observed front-end bottleneck under interleaving.

2.4 Prior Art in Front-End Mitigation

The *front-end bottleneck* is a well-established challenge for server applications [33, 35, 40, 41]. The root cause of the bottleneck are deeply-layered software stacks with multi-megabyte instruction working sets and commensurately large control flow state. For an individual server workload, the instruction footprint typically fits into the on-chip last-level cache (LLC) but easily overwhelms per-core private front-end structures, namely the L1-I, BTB and CBP [41, 44]. Serverless amplifies the front-end challenge, since potentially thousands of serverless functions may interleave on a server, meaning that a given invocation is likely to find none of its on-chip microarchitectural state warm.

2.4.1 Front-end Prefetching for Conventional Server Workloads. A significant body of research has studied microarchitectural techniques for overcoming the front-end bottleneck in servers. The state-of-the-art in this space can be classified into two broad categories: temporal streaming and fetch-directed prefetching. We discuss each of these in turn, followed by a discussion of prior art in mitigating the front-end bottleneck for serverless functions.

Temporal streaming [23] leverages the fact that control flow in server applications is recurrent, leading to repeating sequences of instructions and BTB accesses. These sequences can be recorded and subsequently prefetched, with prefetching initiated upon an access (or miss) to a triggering instruction. Temporal streaming has been shown to be highly effective in eliminating the vast majority of instruction misses [22] and BTB misses [14]. The most recent work in this area, called Confluence [33], demonstrated a unified solution for instruction and BTB prefetching whereby the prefetched instruction cache blocks are predecoded on entry into the L1-I, the branch instructions and their targets are extracted and installed into the BTB.

The main downside of temporal streaming is its high storage cost, with hundreds of kilobytes of metadata required for high miss coverage. Prior work studying individual server applications has shown that the overhead can be ameliorated by virtualizing the metadata into the LLC [14, 33]. However, metadata virtualization is hampered by workload colocation, because each colocated workload requires LLC capacity to store the metadata for the instruction prefetcher. Serverless functions exacerbate this problem due to their high colocation density and, thus, prohibitive on-chip metadata costs.

Fetch-Directed Prefetching. The central motivation behind fetch-directed prefetching (FDP) is to leverage the high accuracy of modern branch predictors to identify future control flow and prefetch the predicted instruction cache blocks into the L1-I [50]. FDP decouples the branch predictor from instruction fetch through a Fetch Target Queue (FTQ), which stores predicted targets to be consumed by the prefetcher and allows the branch predictor to run ahead of the fetch stream. Compared to temporal streaming, FDP enjoys very low implementation complexity and requires no metadata, which makes it extremely attractive for industry adoption. Indeed, a number of recent server CPUs have implemented FDP [2, 27, 32, 48].

Alas, the strength of FDP, which is its low cost and complexity, is also its weakness, since its efficacy is limited by BPUs ability to keep the branch working set in its BTB and CBP. Recent work has shown that for traditional server workloads, the BTB is particularly important as it helps identify upcoming branches and detect *discontinuities* in the control flow [40, 41]. By detecting the discontinuities (with the help of the branch predictor for conditional branches), FDP can predict upcoming non-sequential cache blocks and prefetch them into the L1-I. Perhaps not surprisingly, recent server CPUs have considerably beefed up their BTB configurations; for instance, the upcoming Intel Sapphire Rapids CPU features a 12 K entry BTB, more than doubling the capacity over the 5 K entry BTB in the preceding Ice Lake architecture [8, 58].

To further reduce FDP's dependence on BTB capacity, recent research in FDP has focused on BTB prefetching. Thus, Boomerang [41] proposes detecting BTB misses in FDP through the use of a basic-block-oriented BTB, and resolving them by retrieving the missing branches from target cache blocks. BTB prefetching not only improves the efficacy of FDP, but also reduces pipeline flushes stemming from BTB misses. Notably, published results indicate that Boomerang and Confluence achieve similar performance gains on traditional server workloads, but the lower complexity of FDP has made it the preferred choice for front-end mitigation in recent server CPUs [2, 27, 32, 48].

2.4.2 Front-end Prefetching for Serverless. Recent work has identified the front-end bottleneck in lukewarm executions of serverless function [51], noting that interleaving-induced thrashing results in frequent *off-chip* misses for instructions — a phenomenon not previously reported in characterizations of server workloads. In response, the work proposed a specialized instruction prefetcher, Jukebox, which addresses off-chip instruction misses. Jukebox is a temporal streaming prefetcher that records a trace of L2 instruction misses and stores it in a compact format in main memory, thereby supporting thousands of warm functions without the associated on-chip storage overhead. When a function is re-invoked, Jukebox reads the prefetcher metadata from memory and initiates *bulk* prefetching of instructions from memory into the L2 cache of the core executing the function. Evaluation showed Jukebox to be highly effective in eliminating the vast majority of off-chip misses for instructions with just 32 KiB of in-memory metadata per function instance.

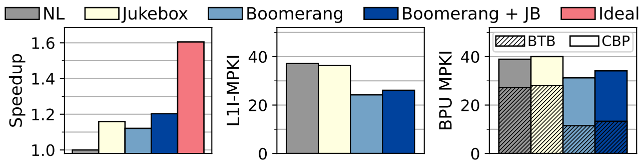


Figure 3: Performance, L1-I MPKI and BPU MPKI for various front-end configurations on lukewarm invocations.

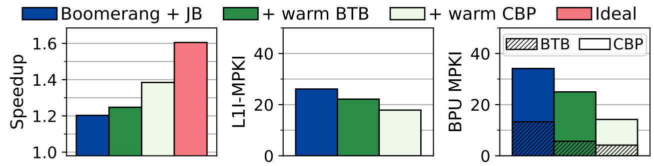


Figure 4: Sensitivity to BPU state

3 FRONT-END PREFETCHING ON LUKEWARM INVOCATIONS

We next study the performance of state-of-the-art microarchitectural front-end prefetchers on lukewarm invocations. We evaluate Jukebox [51], the state-of-the-art for mitigating the off-chip misses for instructions for serverless, and Boomerang [41], a unified FDP instruction and BTB prefetcher. For clarity of exposition, we do not show results for a temporal streaming prefetcher in this section, but note that our findings fully extend to that class of designs, which we demonstrate in Section 6.5. The same gem5 setup as in Section 2.3 is used to evaluate the prefetchers in detailed cycle-accurate simulations.

3.1 Big Picture Results

Figure 3 presents a competitive comparison of the following front-end configurations: *Next-line (NL)* represents our baseline and features an aggressive next-line prefetcher that triggers prefetches on a miss to the L1-I and also on hits to prefetched lines; *Jukebox*; *Boomerang*; and *Boomerang+JB* which combines Boomerang with Jukebox. By combining Jukebox and Boomerang, we relieve Boomerang from hiding the high latency of off-chip misses as Jukebox prefetches these accesses, thus making Boomerang more effective at prefetching into the L1-I and BTB. We also consider an *Ideal* front-end configuration that features a perfect L1-I, perfect BTB, and a pre-trained CBP.

The first graph in Figure 3 shows the speed-up of the various techniques, normalized to NL. Results are averaged across all 20 serverless functions in our benchmark suite. We observe that Boomerang delivers an average speed-up of 12%. It is outperformed by Jukebox (16% average speed-up), despite the fact that Jukebox prefetches only into the L2 while Boomerang prefetches into the L1-I and the BTB. This indicates that FDP struggles to hide the latency of off-chip misses. Combining Boomerang with Jukebox (*Boomerang+JB*) increases speed-up to an average of 20% — a rather modest improvement compared to an ideal front-end that delivers an average performance gain of 61%.

To understand the reasons for the underwhelming performance of existing front-end prefetchers, we first examine their ability to cover L1 misses for instructions. The expectation is that Boomerang, particularly when combined with Jukebox, should be able to cover the majority of L1-I misses. The middle graph in Figure 3 indicates that this is not the case. Compared to the next-line prefetcher, whose L1-I miss rate is 37 MPKI, both Boomerang and Boomerang+JB do reduce the miss rate in the L1-I, but with L1-I MPKI of 24 and 26, respectively, both techniques fail to shield the core front-end from instruction misses.

Next, we examine the BPU by focusing on the BTB miss rate and the CBP misprediction rate. As shown in the last graph of Figure 3, both rates are high, with the average BPU miss rate exceeding 30 MPKI for both Boomerang and Boomerang+JB[4]. We note that while both variants of Boomerang reduce the BTB miss rate as compared to NL, which is expected since Boomerang prefetches into the BTB, the rate of conditional branch mispredictions increases as compared to NL. We examine this phenomenon in the following section.

Take-away: The state-of-the-art front-end prefetching ensemble falls considerably short of an ideal front-end when faced with lukewarm serverless function invocations, exposing the core to high L1-I, BTB, and CBP MPKI.

3.2 Cold uArch State in Focus

We hypothesize that the reason for the poor performance of Boomerang-enabled front-end configurations is the cold BPU state owing to lukewarm invocations. While Boomerang prefetches into the BTB, it does not help with the CBP. Moreover, our results show that when faced with lukewarm invocations, Boomerang's prefetch effectiveness into the BTB is limited, with average BTB MPKI of 13 (Figure 3).

The BPU plays a two-fold role in achieving high front-end performance. The first role is on the prefetching side, where the BPU identifies upcoming branches and their targets via the BTB and, in the case of conditional branches, predicts whether they are taken. Branches that are not present in the BTB or for which the CBP is unable to make an accurate prediction steer the prefetcher onto the wrong path, subsequently resulting in uncovered misses for instructions. The second role of the BPU is in avoiding pipeline resets since every mispredicted or unidentified branch requires a front-end resteer, entailing a pipeline flush and a reset of the fetch PC.

To understand the effect of the cold vs warm microarchitectural state of the BPU, we study the following Boomerang configurations. The baseline is Boomerang+JB, as presented in the previous section. Next, we evaluate the same configuration but with a warm BTB, whereby the BTB state at the end of one invocation is preserved for the next invocation of that function. Finally, we add a configuration that combines a warm BTB and warm CBP (i.e., both the BTB and CBP are preserved across two invocations of a function).

[4]One may wonder why Boomerang+JB has a higher L1-I and BPU miss rate than Boomerang. The reason is that Boomerang+JB is more effective in covering front-end misses (thanks to the Jukebox component), which allows its front-end to go faster than in Boomerang; however, many of the fetched instructions are on the wrong path (due to the high BTB and CBP miss rate). Thus, Boomerang+JB fetches more instructions but also experiences more L1-I and BPU misses/mispredictions as compared to Boomerang.

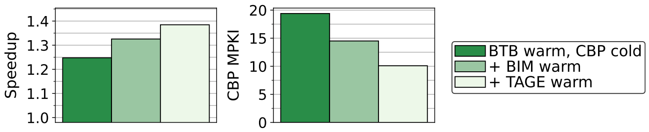


Figure 5: Sensitivity to the CBP state on Boomerang+JB with a warm BTB.

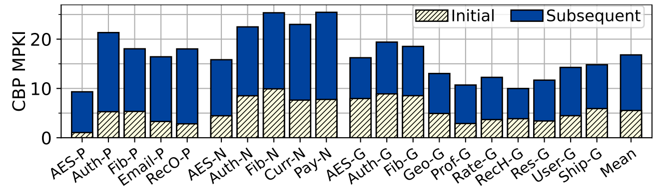


Figure 6: BPU MPKI of Boomerang+JB (warm BTB) split into initial (striped) and subsequent misprediction (solid).

The results of the study are shown in Figure 4, which presents speedup over NL. The figure shows that preserving the microarchitectural state of the BPU across invocations of a function brings a significant performance benefit. A warm BTB helps increase overall performance by 4.2% over a cold front-end. Preserving the CBP across invocations, in addition to the BTB, provides an additional 10% performance gain, bringing performance within 42% of an ideal front-end.

The L1-I and BPU MPKI rates corroborate the performance story. Preserving the BPU state across invocations helps keep the front-end prefetcher on the correct execution path, delivering marked reductions in L1-I and BPU MPKI. With a warm BTB, L1-I misses reduce by 15% while the BPU MPKI reduces by 26%, predominantly stemming from a 50% reduction in BTB misses. When the CBP is also kept warm (together with the BTB), L1-I misses reduce by a further 18% and the BPU MPKI drops by a further 42%.

Take-away: Cold BPU state arising from lukewarm invocations is responsible for poor front-end performance on lukewarm invocations, even in the presence of a state-of-the-art front-end prefetching ensemble.

3.3 Effect of the Cold Branch Predictor

Finally, we focus on the large number of branch mispredictions and the implications of a cold CBP on the front-end machinery. First, we seek to understand the relative importance of CBP's components in the context of cold vs warm microarchitectural state. We model a high-end CBP configuration comprised of 64 KiB L-TAGE and 5 KiB bimodal (BIM) base predictor[5]. Our baseline is Boomerang+JB with a warm BTB and a cold CBP, which corresponds to the second (green) bar from the left in Figure 4. Next, we consider the same configuration but with a warm BIM component; note that the TAGE component is cold. Finally, we consider a configuration with a warm BPU; i.e., when both BIM and TAGE are kept warm across invocations of a given function.

Results of the study are shown in Figure 5. We observe that keeping only the BIM warm decreases the CBP mispredictions from 19.3 to 14.5 MPKI, resulting in a performance improvement of 6.4%, on average. If the TAGE component is also kept warm, CBP accuracy improves further, leading to 10 MPKI and another 4.5% performance gain.

The question arises as to why the BIM has such high relevance for serverless function despite consuming less than 1/10 of the overall CBP size.

We hypothesize that many executed branches are highly biased towards one direction and therefore easy to predict by the BIM. However, as the BIM is cold, those branches are mispredicted during their initial dynamic execution. To validate our hypothesis, we analyze when mispredictions occur during individual function invocations. If a miss happens during the first execution of a branch, we count it as *initial* miss. All other misses are counted as *subsequent* miss. Figure 6 shows the corresponding split of initial and subsequent CBP mispredictions for Boomerang+JB with a warm BTB (cold CBP). We find 12-49% (33% on average) of the mispredictions are caused by branches executed for the first time during an invocation. The results indicate that a significant faction of branches is simple to predict once the CBP is aware of them, corroborating our hypothesis.

The presence of a large number of initial CBP mispredictions reveals a crucial insight to understand Boomerang's poor performance. Two conditions must be met to allow a branch to be speculatively taken: the CBP must predict taken, and the BTB must hold the corresponding target. Otherwise, the branch is not taken. Thus, combining a warm BTB with a cold CBP presents two problematic situations. If the CBP is incorrect and predicts not-taken for a *taken* branch, Boomerang's BTB filling mechanism did not help eliminating the branch misprediction. Conversely, if the branch is *not taken* but the CBP predicts taken, BTB filling was counterproductive since not identifying the branch in the first place (by not placing it into the BTB) would have prevented the misprediction.

Take-away: Keeping only the BIM warm across invocations achieves 51% of the potential, in both MPKI and performance, compared to keeping the entire CBP (which includes the much-larger TAGE component) warm. The BIM's high relevance is due to many initial mispredictions, which compromise the existing BTB filling techniques.

3.4 Summary

Our findings show that *cold microarchitectural state* due to lukewarm invocations results in a *critical front-end bottleneck even in the presence of state-of-the-art front-end prefetchers.* With instructions on-chip, a unified front-end prefetcher filling the L1-I and the BTB fails to achieve a significant MPKI reduction in these structures. Our analysis reveals that the cold BPU state is to blame, with frequent BTB misses and branch mispredictions leading to a high incidence of fetches and prefetches on the wrong path.

Effectively tackling the cold front-end requires having instructions on-chip and the BPU initialized so as to identify branches and predict conditional ones. An important finding is that initializing only the BIM component of the CBP, which is much smaller and

[5]The actual CBP configuration in Ice Lake has not been made public.

simpler than TAGE, achieves 51% of the benefit of initializing the entire CBP (BIM+TAGE).

While prior work (Jukebox [51]) has demonstrated a solution for avoiding off-chip misses for instructions, it does not address the cold microarchitectural state in the BPU, which impedes existing front-end prefetchers from attaining high efficacy. What is needed is a light-weight mechanism to not only deliver instructions on-chip, but to also restore the branch working set into the BTB and the CBP upon function invocation.

4 IGNITE

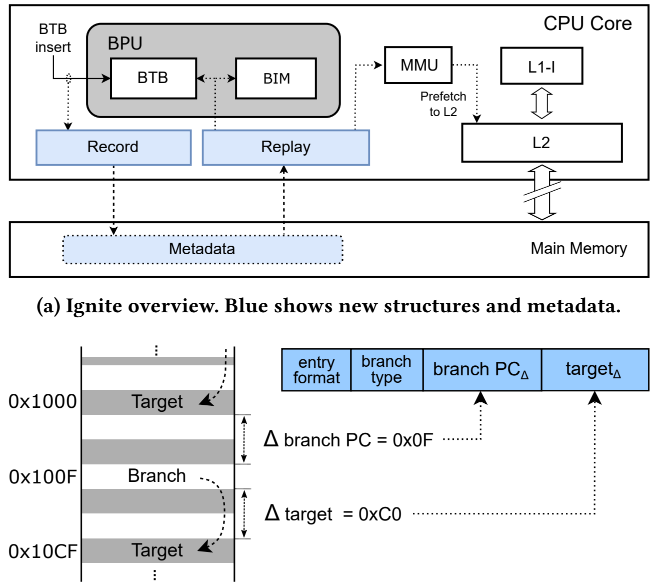
We introduce Ignite, a comprehensive solution for *restoring* the front-end microarchitectural state. At the heart of Ignite is a compact and unified representation of the front-end microarchitectural working set spanning instructions, BTB and CBP. Ignite operates by recording the observed working set during the execution of a given serverless function, then restoring it upon re-invocating that function again. We use the term *restoration* to differentiate Ignite from traditional prefetchers that continuously monitor the current process and reactively prefetch at fine granularity (e.g., a cache block or a page) triggered by a particular address, stride, or PC. In contrast, Ignite unconditionally restores the entire recorded instruction, BTB, and partial CBP working set at the start of an invocation. Such bulk restoration is essential for enabling a rapid warm-up of the core front-end.

In simplest terms, Ignite records control flow discontinuities as a single stream of metadata. Control flow discontinuities arise when the sequential flow of instructions is interrupted by a taken branch (conditional, unconditional, function call/return). Each record in Ignite's stream represents a discontinuity in otherwise sequential code, and is comprised of a branch PC, branch type, and a target. The records form a chain of control flow, where the target of one branch is the start of a contiguous block of code ended by the next taken branch, identified by the next record in the stream.

The stream described above is comprehensive, recording every observed taken branch, which allows the address of every executed instruction to be trivially determined. However, such a trace is highly redundant due to recurrent control flow (e.g., loops and functions with multiple callers), thereby incurring significant metadata storage costs. We exploit two insights to make unified front-end metadata practical.

Our first insight is that the BTB working set (which may exceed the actual capacity of the BTB) provides a *complete* and *non-redundant* representation of the control flow graph of the program. Ignite leverages this insight to minimize metadata redundancy and storage costs by only recording BTB *insertions*. Given a recorded BTB working set, it is trivial to reconstruct the working set of instruction cache blocks by chaining branch PCs and their target addresses. But what about the CBP?

Our second insight is that modern CPUs create new BTB entries (i.e., insert branches into the BTB) only when a *taken* branch is committed [2, 32]. Thus, the mere fact that a BTB entry is created for a conditional branch implies that the branch was taken. Ignite uses this insight at replay time to initialize the BIM to 'taken' for each conditional branch encountered in its metadata. Note that Ignite does not restore TAGE, whose size and complexity would



(a) Ignite overview. Blue shows new structures and metadata.

(b) Ignite metadata record.

Figure 7: Ignite design overview and metadata record layout.

considerably encumber Ignite's design. Thus, Ignite opts for simplicity and low metadata cost in exchange for a modest loss in branch prediction accuracy (Section 3.3).

Figure 7a provides an overview of Ignite. At record time, Ignite simply monitors BTB insertions and writes the entries to a dedicated region of memory. At replay time, Ignite reads the stream from the beginning and uses it to restore instructions, BTB and BIM as follows. The branch PC is used to prefetch the corresponding instruction block into the L2 cache. Each stream entry directly corresponds to a BTB entry and can be inserted into the BTB as such. For conditional branches (identified via the branch type field), the BIM entry corresponding to that PC is initialized as taken.

Ignite naturally integrates with FDP (e.g., FDIP [50] or Boomerang [41]), thereby allowing effective instruction prefetch from the lower levels of the cache hierarchy into the L1-I. Ignite's metadata is stored in main memory, thus naturally scaling with the number of active serverless functions. Ignite has low microarchitectural complexity: its record logic needs to monitor only BTB insertions as it uses the same information for its metadata as the BTB entry being created, while the replay logic reads the recorded stream in sequential order and, for each entry, issues an instruction prefetch and inserts BTB and BIM entries. Thus, Ignite enables high front-end miss coverage, high scalability and low integration complexity.

Ignite was designed in the context of serverless functions. However, the approach is applicable in other contexts where frequent switching between threads hurts performance [57, 61] due to cold microarchitectural state. For example, Ignite could be beneficial in modern mobile applications that are characterized by frequent context switches or cases where microarchitectural state needs to be flushed at context switches for security reasons [57].

4.1 Record

The record logic of Ignite is responsible for recording the front-end working set by capturing BTB entries at the point of their creation and storing them into a dedicated, per-container, memory region. The recorded working set needs to satisfy three requirements to be useful at the replay stage. First, it needs to accurately capture the branch working set. Second, it must be recorded in the order of expected reuse to ensure timely instruction prefetching. Third, it needs have low redundancy to minimize memory bandwidth and storage requirements.

As noted in Section 4, the BTB in modern processors only inserts taken branches. Furthermore, as a cold BTB can be expected when recording starts (see Section 2.3), every *new* taken branch will result in a BTB entry being allocated. This means that we can use BTB allocation events to record new branches as they are encountered by the front-end. With an unbounded BTB, the resulting trace would contain a complete record of unique branches and their targets in the order that they were first executed (i.e., in the order we expect them to be executed in the future). In practice — with a finite BTB — a branch may be evicted and, later, re-inserted, resulting in a small degree of redundancy in the recorded trace.

Metadata compression: A naive way of storing branches and their targets would be to store the branch PC and the target PC. Assuming 48 bit virtual addresses, such a format would use at least 96 bit of storage per entry. This is clearly wasteful. We can use two important observations to compress records. First, most branches tend to be local, for example, inside a method call. This implies that the target can be encoded as a small delta from the PC of the branch instruction [11, 55]. Second, the distance to the next branch from the target of the previous branch tends to be small, indicating that a delta (from the previous target) can be used instead of the full branch PC.

To compute the deltas, Ignite stores the last-inserted BTB entry in a dedicated register. When a new entry is BTB created, simple logic computes the delta from the target of the previous BTB entry to the branch PC of the newly created entry. Similarly, a delta is computed from the new branch PC to its target. Once an Ignite metadata entry is formed, the register is updated with the content of the newly-created BTB entry.

Ignite uses a fixed-size delta for the branch PC and another delta for the target to simplify record and replay logic[6]. When computed deltas exceed the pre-determined size, the full PC is used. A single bit in each metadata entry specifies the *format* of the entry with respect to whether deltas or full addresses are used. Figure 7b visualizes the creation of metadata entry and its format.

4.2 Replay

The purpose of the replay phase is to deliver instructions into the L2 cache and to prime the BTB and CBP to enable efficient speculation. Priming the BTB and CBP has two complementary benefits: it reduces front-end stalls for demand accesses due to more accurate prefetches by FDP, and it reduces pipeline flushes due to BTB misses and branch mispredictions. Meanwhile, prefetching of instruction into the L2 reduces the risk of long-latency instruction misses that cannot be hidden by FDP alone.

Ignite sequentially reads the metadata trace created in the record phase and, for each metadata record, performs the following actions. First, if the record uses delta-encoded branch and target fields, it expands them. Using the full-length fields, it creates a BTB entry and inserts it into the BTB. If the entry corresponds to a conditional branch, it sets the appropriate BIM entry to 'weakly taken'. In parallel with the BPU insertion, the replay logic uses the MMU to translate the address of the branch PC comprised in the entry and issues a prefetch to the L2 cache for the corresponding cache block. Note that the act of address translation populates the I-TLB, hence effectively serving as an I-TLB prefetcher.

Prefetch throttling: To avoid thrashing the BTB, Ignite throttles the replay rate. For workloads with large branch working sets, this effectively increases the reach of the BTB beyond its natural size. We implement throttling by tracking the number of restored BTB entries that have not been accessed by the core front-end either for demand fetch or for prefetching. The tracking itself is implemented using a dedicated per-entry bit in the BTB that gets set when a BTB entry is inserted by Ignite and cleared when the entry is accessed or evicted. A counter keeps track of the total number of restored BTB entries that have not been touched; the counter is incremented when an entry is restored and decremented whenever a restored entry is first accessed or evicted without having ever been used. Prefetching is throttling whenever the number of unaccessed restored entries exceeds a predetermined threshold.

Divergence at replay time: In the unlikely event that a function's behavior changes substantially between two invocations (i.e., from record to replay), Ignite may fail to accurately capture the branch working set. In such cases, Ignite behaves similar to a system without Ignite since BTB and CBP lookups would fail to capture the new behavior in both cases. While we have not observed such cases in our studies, they could be mitigated by running record and replay simultaneously (see Section 4.3) to capture a branch working set that evolves between invocations.

4.3 Operating System Interface

Ignite integrates with the operating system to manage memory and to trigger record and replay when a function invoked. These two components of Ignite have an independent set of control registers to set the base address and size of the metadata region and to activate recording or replay. This interface is, in fact, identical to Jukebox[51]; we refer an interested reader to that work for a more detailed description.

When a new function starts, the operating system allocates a contiguous region of memory for metadata. It then points Ignite's record component to the metadata region using its base and size registers. Once the metadata region has been configured, the operating system enables recording by setting a control bit and launches the function. On subsequent invocations of the function, the operating system configures the replay mechanism with a pointer to the recorded metadata and its size. It then sets a control bit to activate replay as soon as a function has been scheduled on a core. Note that by starting replay together with the function, Ignite loses the opportunity to cover misses at the very start of a function's

[6] We empirically found 7 bits for branch PC delta and 21 bits for target delta to achieve the highest compression.

execution. However, Ignite rapidly establishes a sufficient prefetch distance because the CPU stalls every time an instruction cache miss is encountered, while Ignite's prefetching does not.

Since replay and record are independent components, an operating system may choose to double-buffer metadata and activate both replay and record at the same time. Doing so increases metadata bandwidth and storage requirements but lets Ignite react to changes in the branch working set.

4.4 Security Aspects

Ignite records microarchitectural state as metadata into main memory, raising the question of whether this opens up security vulnerabilities. Ignite and its metadata are managed by the host OS, which already has visibility into application state, including microarchitectural state. For instance, most recent CPUs offer features like Intel's last branch record register (LBR), Intel's processor trace (Intel-PT) [30] or Arm's branch record register (BRB) [10] that allow collecting application traces.

As Ignite injects branch targets into the BTB, a malicious VM can use Ignite to create a speculative side channel and extract information from other VMs. However, as it is already possible to inject arbitrary branch targets into the BTB [9] Ignite does not increase the attack surface. Additionally, Ignite is compatible with side channel mitigations like Arm's BTB tagging feature (FEAT_CSV2) [9]. In a CPU featuring BTB tagging, Ignite would use the currently running VM ID to tag restored BTB entries. In that way, replayed entries from a malicious VM are not executable by other VMs.

5 METHODOLOGY

5.1 Workloads

We use 20 distinct serverless functions from the vSwarm benchmark suite [42] listed in Table 1. The functions feature three different languages/runtimes: Python, NodeJS and Go. In both our hardware experiments and simulation we use the same software stack and the same version of function images (Ubuntu 20.04 with Linux kernel version v5.4 and Docker version 20.10 as container host). The function container instance is pinned to a core isolated from the OS scheduler. A client for driving the invocations is pinned to other cores. Before measuring, the function is invoked 20 000 times to warm up the runtimes of function containers[7].

5.2 Hardware Infrastructure

For the hardware studies, we rent a *r650* server node in the CloudLab cluster at Clemson University, South Carolina [21]. The *r650* instances implement a 3rd Gen. Intel Xeon Ice Lake (dual socket 36-core Intel Xeon Platinum 8360Y) running at 2.4 GHz [29]. Each core features a private 32 KiB L1-I cache, a 48 KiB L1-D and 1.25 MiB L2 cache. All cores share a 54 MiB L3 cache per NUMA node and can access 256 GiB DDR4 DRAM. SMT is disabled as done in production [4, 56].

[7]We empirically found that 20 000 invocations is sufficient for NodeJS's JIT engine to perform code optimizations for all of our workloads.

Function	Abbr.	Function	Abbr.
Hotel Reservation [24]		**Online Boutique** [26]	
Geo	Geo-G	Currency	Curr-N
Profile	Prof-G	Email	Email-P
Rate	Rate-G	Payment	Pay-N
Recommend.	RecH-G	ProductCatalog	ProdL-G
User	User-G	Shipping	Ship-G
Other [7, 38, 39]		Recommend.	RecO-P
Authentication	Auth-P/N/G		
Fibonacci	Fib-P/N/G		
AES	AES-P/N/G		

Table 1: Serverless functions and their language runtimes (Abbreviation legend – P: Python, N: NodeJS, G: Go).

Core	
Architecture:	Ice lake-like, ISA: x86-64, Freq.: 2.6 GHz
Fetch BW	16 bytes / cycle
BP Unit	L-TAGE [52]: 64 KiB Bimodal: 5 KiB BTB: 12 K entries, 6-way, 12 b tag, 48 b target
Back-end	ROB: 353 entries LSQ: 128 load + 72 store Scheduler: 160 entries RF: 280 Int+ 224 FP
Memory Hierarchy	
L1-I Cache	32 KiB, 64 B line, 8-way, 1 cycle[8], private, LRU, 10 MSHRs
L1-D Cache	48 KiB, 64 B line, 12-way, 4 cycles, private, 10 MSHRs, LRU
L2 Cache	1280 KiB, 20-way, 13 cycles, private, LRU, 32 MSHRs
LLC	8 MiB, 16-way, 50 cycles, shared, non-inclusive, 32 MSHRs, 64 store buffers
Memory	DDR4 2400 MHz, 14 ns RCD, 14 ns RP, 14 ns CL

Table 2: Parameters of the simulated processor.

After warming the function container instance we collect PMU performance counters using *Linux perf* [34] for both user and kernel space from 500 consecutive invocations. The effect of interleaving with other functions is modeled by using `stress-ng` [16] as a stressor to thrash microarchitectural state of the core running the function container.

5.3 Simulator Infrastructure

We use gem5 v22.0.0.1 [13, 25, 45], a cycle-approximate full-system simulator configured to model the Intel Xeon Ice Lake CPU used in the hardware studies [1, 29]. In light of the fact that industry trends are toward much larger BTBs than in recent past, we enlarge the BTB from 5 K entries in Ice Lake to 12 K entries as found in the latest

[8]Since gem5 does not support a micro-op cache, the L1-I cache is configured with the micro-op cache latency, instead of the L1-I cache latency as described in [28].

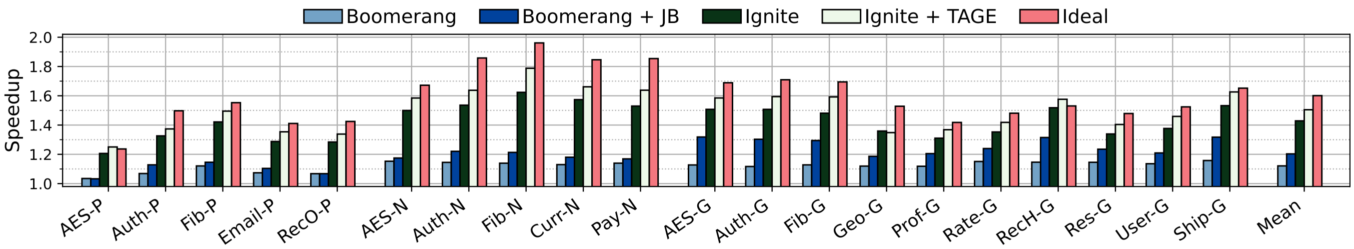


Figure 8: Performance results over next-line prefetcher.

Intel Xeon Sapphire Rapids CPU [8]. We find that overall trends and conclusions are not affected by this choice. Table 2 summarizes the modeled parameters.

We create a two-machine simulation setup using the vSwarm-μ framework [43]. The first machine runs a test client that sends requests via gem5's Ethernet model to the second machine that models the detailed Ice Lake CPU. To simulate the effect of interleaving we flush the microarchitectural structures of the simulated Ice Lake CPU between two invocations and overwrite the bimodal predictor with a random state.

We evaluate the following prefetchers:

Baseline (NL): Next-line prefetching for instructions and stride prefetching for data. Used in all configurations below.

FDP: We implement the decoupled front-end (FDP) in gem5's out-of-order CPU following industry reports [31][9]. FTQ: 32 entries; branch predictor bandwidth is double the fetch width [48]; branch predictor uses taken-only history [2, 3].

Boomerang: FDP augmented with the BTB filling mechanism as described in [41]. 6-cycle pre-decode latency, 16-entry BTB prefetch buffer.

Confluence: 8 K entry index and a 32 K entry history buffer [33]. Instead of modeling virtualized metadata in the LLC, we use dedicated structures for index and history buffers with an LLC-like look-up latency of 50 cycles [1].

Jukebox: 16-entry CRRB and a region size of 1 KiB. For both record and replay, metadata is limited to 16 KiB each (32 KiB in total). Prefetched instruction blocks land in L2.

Ignite: 21 bits to encode branch PC (i.e., source) delta, 7 bits to encode target delta. Replay throttled when >1 K restored BTB entries have not been accessed. Maximum metadata size: 120 KiB. Our implementation is on top of FDP, but could equally be used with Boomerang.

6 EVALUATION

6.1 Performance Analysis

We first study the performance of the various front-end prefetchers under lukewarm invocations. We evaluate Boomerang, Boomerang augmented with Jukebox (Boomerang+JB), and Ignite. Because Ignite restores only the BIM component of the CBP, we also consider a variant of Ignite that restores the TAGE component as well (Ignite+TAGE). Note that the latter configuration may not be feasible, as there is no known mechanism to efficiently save and restore TAGE context [57], but it is useful for understanding the opportunity in restoring TAGE.

Figure 8 presents the results of the evaluation, normalized to our Baseline (NL). As reported in Section 3.1, Boomerang improves performance over NL by 3-16% (12% on average). For Boomerang+JB, the improvement increases to 20%, on average, over NL.

Ignite achieves a 21-62% (43% on average) speed-up over NL, an improvement of 3.6x over Boomerang and 2.2x over Boomerang+JB. The highest speed-ups are observed on functions written in NodeJS, which tend to be branch-heavy (refer to Figure 2b) and thus have a high dependence on the BPU. Ignite improves the performance of these applications by 50-62%. Ignite+TAGE improves performance by 50%, on average, covering roughly half of the performance difference between Ignite and the Ideal front-end. We observe Ignite outperform Jukebox by 2.4x despite both addressing lukewarm function invocation. Jukebox prefetches only the instruction working set into the L2 to cover off-chip misses for instructions but leaves the remaining front-end completely cold. Ignite prefetches into the L1-I, BTB and BIM. Thus, Ignites covers misses in multiple front-end structures that are ignored by Jukebox.

6.2 Miss Coverage and Accuracy

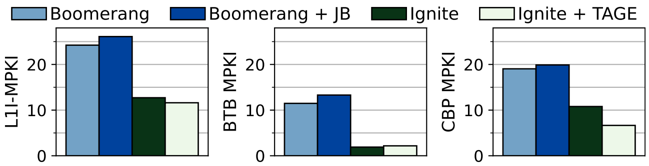
We next study Ignite's ability in covering front-end misses for instructions, branch targets and branch direction predictions. As before, we consider Boomerang, Boomerang+JB, Ignite, and Ignite+TAGE.

L1-I miss coverage: Figure 9a, left, shows MPKI for the various front-end prefetchers. Ignite reduces L1-I misses by about 2x as compared to Boomerang and Boomerang+JB. There are two reasons for Ignite's strong performance. The primary reason is a much lower BPU MPKI (discussed below) owing to Ignite's effective BPU restoration. This allows the front-end prefetcher (FDP) to stay on the correct path, thus achieving higher coverage than Boomerang and Boomerang+JB. The second reason is that Ignite covers more off-chip misses for instructions than Boomerang.

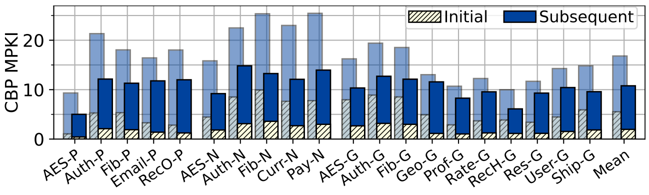
BTB miss coverage: The center graph in Figure 9a shows Ignite's efficacy in restoring branches into the BTB. We observe Ignite is highly effective at eradicating BTB misses. Boomerang achieves a BTB miss rate of 11 MPKI (13 MPKI for Boomerang+JB). Meanwhile, Ignite achieves a BTB miss rate of 1.9 MPKI — an improvement of over 5x versus prior front-end prefetchers.

Branch miss coverage: As shown in Figure 9a, right, Ignite reduces the incidence of branch mispredictions by nearly half versus other front-end prefetchers — from 19 MPKI or more to just over

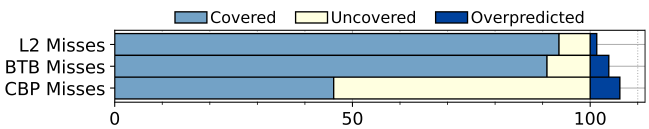
[9]The gem5 implementation of FDP has been released and made available for the research community at https://github.com/dhschall/gem5-fdp/



(a) Miss coverage



(b) Initial miss coverage



(c) Restore accuracy

Figure 9: Ignites miss coverage (a), detailed *initial* miss coverage (b), and restore accuracy (c). Initial miss coverage shows Ignite in the front and Boomerang+JB (warm BTB) in the back.

10 MPKI. Preserving in addition to Ignite the full TAGE prediction tables (Ignite+TAGE bar) reduce CBP mispredictions to 6.6 MPKI.

As Ignite's primary objective is to eliminate *initial* branch mispredictions, we analyze its efficacy in more detail. As done in Section 3.3, we split initial and subsequent mispredictions for Ignite and show the results Figure 9b. For ease of comparison, the previous results (Figure 6) are plotted in the background. We observe Ignite is effective in covering 67% of the initial mispredictions.

Restore accuracy: Ignite places microarchitectural state into the L2 cache, BTB and CBP possibly evicting valuable state. Therefore, we evaluate Ignite's accuracy in Figure 9c where we show for each of the three restored structures the fraction of misses covered, not covered and overpredicted. For L2 and BTB, overpredicted means entries that were installed by Ignite but never used, and for the CBP, mispredictions which the BIM causes because Ignite initialized an entry incorrectly.

We find Ignite has high accuracy in restoring state. On average, only 1.4% of Ignite's L2 instruction prefetches and 3.9% of BTB entries that Ignite restores are *not* useful. Furthermore, Ignite induces only 6.2% additional mispredictions (while eliminating 46% of the mispredictions of Boomerang+JB). The high accuracy stems from the high commonality in the execution of serverless functions.

6.3 Memory Bandwidth

We analyze the amount of memory bandwidth consumed by the various front-end prefetchers. We consider four sources of memory bandwidth usage: useful instructions, useless instructions, record metadata (i.e., metadata streamed *to* memory), and replay metadata

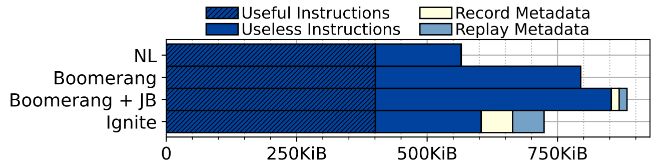


Figure 10: Ignite's impact on memory bandwidth.

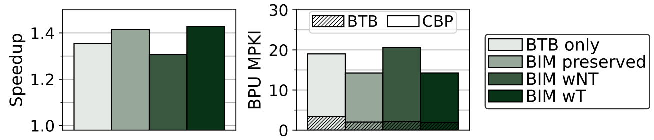


Figure 11: Comparison of different BIM initialization policies. *wNT*: weakly not-taken, (*wT*): weakly taken.

(metadata streamed *from* memory). We study the worst-case memory usage, where record and replay happen simultaneously. Note that instruction cache blocks include both demand requests and prefetches both on correct and misspeculated paths.

Figure 10 compares memory bandwidth for NL prefetcher, Boomerang, Boomerang+JB and Ignite. We observe that 25% of the overall instruction traffic with the next-line prefetcher is useless and is fetched while the front-end is on the wrong path, which happens due to the cold state of the BPU. Boomerang employs fetch-directed prefetching; however, owing to the cold BPU, Boomerang (and the underlying FDP mechanism) exacerbates wrong-path instruction fetches. As shown in the second bar of Figure 10, Boomerang more than doubles useless instruction fetches, which translates to an overall increase in traffic for instructions by 41% over the next-line prefetcher. Boomerang+JB further increases the memory traffic by an additional 10% over Boomerang. The reason for the increase is that Jukebox helps cover more off-chip misses for instructions, allowing the front-end to run faster than without Jukebox, thus fetching more instructions per unit time. However, due to the cold BPU, most of these are on the wrong execution path. Thus, Boomerang+JB generates even more useless fetch and prefetch requests to memory than Boomerang.

Finally, by restoring the content of the BPU, Ignite dramatically reduces wrong-path instruction accesses. As a result, Ignite uses 24% less memory bandwidth for instructions than Boomerang (29% less than Boomerang+JB). However, the reduction in useless memory bandwidth is partially negated by Ignite's metadata traffic. Nonetheless, even with both record and replay metadata traffic accounted for, Ignite requires 8.6% less memory bandwidth than Boomerang and 17% less bandwidth than Boomerang+JB.

6.4 Sensitivity to Bimodal Initialization

We evaluate different BIM initialization policies for Ignite. As our baseline, we use Ignite to restore only L2 and BTB state but not to initialize the BIM. Next, we compare our baseline against an upper bound that fully preserves the BIM state from the previous invocation. Finally, we compare two configurations in which we initialize BIM entries together with inserting branches into the BTB to a *weakly not-taken* state (wNT) and a *weakly taken* (wT) state[10]. The latter policy — initializing inserted entries to wT — is used by Ignite.

In Figure 11, we show speedup (over NL) and the BPU MKPI. We observe that using Ignite to restore only L2 and BTB state results in a speedup of 35%. Preserving the entire BIM state across invocations gains a further 5.5% speedup and a 25% MPKI reduction, underscoring the importance of a warm BIM state.

The evaluation of different initialization policies reveals that resetting BIM entries to *weakly not-taken* degrades the performance by 3% as compared to a baseline that does not initialize the BIM at all. In contrast, resetting BIM entries to *weakly taken* results in a 6% performance boost. The results correlate with our observation from Section 3.3 that restoring BTB entries is only effective if the CBP predicts taken. As Ignite fills only branches taken in the last invocation, it must initialize BIM entries as *weakly taken*.

Finally, we notice that using a *weakly taken* policy for Ignite achieves similar performance as preserving the BIM state. In fact, in some cases, the *weakly taken* initialization policy slightly outperforms preserving the BIM. The reason why Ignite's BIM initialization policy may, at times, outperform preserving the BIM across invocations is that the BIM's state at the end of an invocation reflects the effect of the *last* execution(s) of a given branch. In contrast, Ignite records the *first* execution of a branch. Ignite's strategy favors branches whose behavior differs between first and last execution (e.g., branches associated with predicates guarding a loop). Overall, the study validates Ignite's design by showing the importance of initializing the BIM state and demonstrating that *weakly taken* is the right initialization policy.

6.5 Temporal Streaming Prefetchers

So far, we have only considered FDP-based front-end prefetchers. We now examine temporal streaming prefetching (Section 2.4.2) and demonstrate that the observations made throughout the paper, including the effect of cold microarchitectural state on front-end performance, apply to this class of prefetchers as well. We further show that Ignite is compatible with this class of prefetchers, making the observations behind Ignite general.

We consider Confluence [33], a state-of-the-art unified temporal streaming prefetcher discussed in Section 2.4.2. Confluence uses dedicated metadata to drive instruction prefetching into the L1-I, where it relies on instruction pre-decoders to extract branches and insert them into the BTB. See Section 5.3 for configuration parameters of Confluence.

We evaluate Confluence, Confluence together with Ignite (Confluence+Ignite), and FDP with Ignite (FDP+Ignite); the latter is the configuration evaluated elsewhere in this section under the name 'Ignite'. Results are presented in Figure 12.

As the figure shows, the general trends presented for Boomerang (an FDP-style prefetcher) in Figure 3 hold for Confluence. Specifically, Confluence delivers only a small performance improvement over NL due to high L1-I and BPU MPKI. Although Confluence does not rely on fetch-directed prefetching, it is nonetheless highly sensitive to branch mispredictions, since they require Confluence

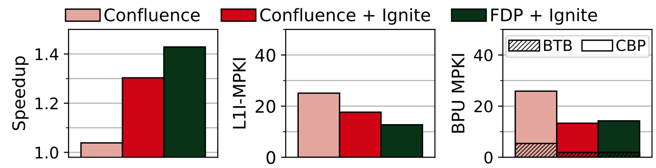


Figure 12: Evaluation of Temporal Streaming Prefetchers

to re-index and re-initiate prefetching from a different stream than the one that was being followed. We thus conclude that the same limitation found in Section 3.3 for Boomerang applies to Confluence. While Confluence delivers branch targets to the BPU, the cold CBP hinders the BTB filling mechanism to become effective.

The figure further demonstrates that Confluences pairs well with Ignite, which helps avoid off-chip misses for instructions and restores the state of the BTB and the BIM. As a result, Confluence+Ignite enjoys a 28% reduction in L1-I misses and 50% reduction in BPU misses as compared to Confluence.

Finally, we note that FDP+Ignite achieves somewhat better performance than Confluence+Ignite, which we attribute to the fact that Confluence requires more training time to form sufficiently-long streams, whereas FDP trains faster, especially with Ignite restoring the BPU.

7 RELATED WORK

Serverless: Little work has been done on understanding microarchitectural implications of serverless computing. In one previous study, Shahrad et al. [53] analyzed the performance of five serverless functions and identified problems including high cold-start latency and high variability in execution time. The work noted a high incidence of branch mispredictions upon a function cold start but did not propose a mitigation. Schall et al. [51] characterise microarchitectural implications of 20 diverse serverless functions and identified off-chip instruction misses due to a high degree of function interleave as a key performance bottleneck.

Mitigating Context switches: Prior research has tackled the issue of context switches in virtualised systems and proposed techniques to preserve LLC state across context switches. Ahn et al.[5] control LLC capacities available for individual virtual machines while others leverage record-and-replay to prefetch the entire LLC state upon context switch [17, 60]. The focus of those works is only on preserving LLC state. Vougioukas et al. address cold branch predictor states due to flushes upon context switches in order to avoid side-channel attacks [57]. The work proposes a small specialized predictor that can be quickly restored on a context switch, along with a buffer that allows retaining the state of that predictor for a small number of concurrently active applications. In contrast, Ignite proposes a unified restoration mechanism for the entire core front-end including the CBP. Notably, Ignite requires no modifications to the branch predictor organization and does not need to store any metadata on-chip.

Software techniques: Recent work has proposed specialized instructions to prefetch code [12, 36, 46], BTB entries [35] or branch prediction hints [37]. Each of these techniques requires architectural support and increases code size. Furthermore, prior techniques

[10]We also evaluated strongly taken/not-taken policies but found no significant differences compared to weakly taken/not-taken

address instruction misses, BTB misses or branch mispredictions individually. Our work holistically addresses all sources of front-end misses with no code or ISA modification. Other techniques leverage profile information to optimize code layout [15, 47, 49]. But doing so does not help with BTB misses or branch mispredictions.

8 CONCLUSION

Lukewarm invocations compromise performance of serverless functions due to cold microarchitectural state, particularly in the core front-end. Meanwhile, existing front-end prefetchers show limited effectiveness on lukewarm invocations because the cold BPU throws both fetch and prefetch streams off the correct path. In response, this work introduces Ignite, a comprehensive mechanism for restoring front-end microarchitectural state recorded during a previous invocation of a given function. To the best of our knowledge, Ignite is the first approach to restore instructions, BTB and CBP state using unified metadata. Ignite is enabled by the insight that the BTB working set provides a good approximation of a program’s control flow graph. Ignite records this working set and uses it to restore the front-end metadata. A detailed evaluation of Ignite shows that it outperforms state-of-the-art prefetchers and delivers significant performance gains on lukewarm invocations by reducing the front-end MPKI.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers as well as the members of the EASE Lab at the University of Edinburgh for their valuable feedback on this work. This research was generously supported by the University of Edinburgh, Arm and by EASE Lab’s industry partners and sponsors including Huawei, Intel and Microsoft.

REFERENCES

[1] 7-Zip LZMA Benchmark. 2019. Intel Ice Lake. Retrieved April 28, 2023 from https://www.7-cpu.com/cpu/Ice_Lake.html

[2] Narasimha Adiga, James Bonanno, Adam Collura, Matthias Heizmann, Brian R. Prasky, and Anthony Saporito. 2020. The IBM z15 High Frequency Mainframe Branch Predictor Industrial Product.. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. IEEE, 27–39. https://doi.org/10.1109/ISCA45697.2020.00014

[3] Inc. Advanced Micro Devices. 2023. *Software Optimization Guide for the AMD Zen4 Microarchitecture*. Technical Report. Advanced Micro Devices, Inc., Cambridge, MA, USA.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications.. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 419–434.

[5] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh. 2014. Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems.. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 394–405. https://doi.org/10.1109/MICRO.2014.49

[6] Amazon. 2022. A Demo Running 4000 Firecracker MicroVMs. Retrieved April 12, 2022 from https://github.com/firecracker-microvm/firecracker-demo

[7] Amazon Web Services. 2022. Use API Gateway Lambda Authorizers. Retrieved April 12, 2022 from https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-use-lambda-authorizer.html

[8] Andrei Frumusanu. 2021. Golden Cove Microarchitecture (P-Core) Examined. Retrieved January 27, 2023 from https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3

[9] Arm. 2022. *Spectre-BHB: Speculative Target Reuse Attacks, Version 1.7*. Technical Report. Arm Limited.

[10] Arm. 2023. *Feature names in A-profile architecture*. Retrieved July 01, 2023 from https://developer.arm.com/downloads/-/exploration-tools/feature-names-for-a-profile

[11] Truls Asheim, Boris Grot, and Rakesh Kumar. 2023. A Storage-Effective BTB Organization for Servers.. In *Proceedings of the 29th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1153–1167. https://doi.org/10.1109/HPCA56546.2023.10070938

[12] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers.. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. ACM, 462–473. https://doi.org/10.1145/3307650.3322234

[13] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718

[14] 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-XIV)*. ACM, 313–324. https://doi.org/10.1145/1508244.1508281

[15] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications.. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, (CGO)*. ACM, 12–23. https://doi.org/10.1145/2854038.2854044

[16] Colin Ian King. 2022. Stress-ng. Retrieved April 12, 2023 from https://github.com/ColinIanKing/stress-ng

[17] David Daly and Harold W. Cain. 2012. Cache restoration for highly partitioned virtualized systems.. In *Proceedings of the 18th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 225–234. https://doi.org/10.1109/HPCA.2012.6169029

[18] Datadog. 2020. The State of Serverless 2020. Retrieved October 27, 2022 from https://www.datadoghq.com/state-of-serverless-2020

[19] Datadog. 2021. The State of Serverless 2021. Retrieved October 27, 2022 from https://www.datadoghq.com/state-of-serverless-2021/

[20] Datadog. 2022. The State of Serverless 2022. Retrieved October 27, 2022 from https://www.datadoghq.com/state-of-serverless/

[21] Datadog. 2023. The University of Utah. Retrieved April 28, 2023 from https://www.cloudlab.us/hardware.php

[22] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch.. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 152–162. https://doi.org/10.1145/2155620.2155638

[23] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming.. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 1–10. https://doi.org/10.1109/MICRO.2008.4771774

[24] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems.. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*. ACM, 3–18. https://doi.org/10.1145/3297858.3304013

[25] gem5 developers. 2022. *gem5*. Retrieved August 21, 2023 from https://github.com/gem5/gem5/releases/tag/v22.0.0.1

[26] GoogleCloudPlatform. 2022. Online Boutique. Retrieved April 12, 2022 from https://github.com/GoogleCloudPlatform/microservices-demo

[27] Brian Grayson, Jeff Rupley, Gerald D. Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. 2020. Evolution of the Samsung Exynos CPU Microarchitecture.. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. IEEE, 40–51. https://doi.org/10.1109/ISCA45697.2020.00015

[28] UC Davis Computer Architecture Research Group. 2020. *gem5 skylake config*. Retrieved April 12, 2023 from https://github.com/darchr/gem5-skylake-config/blob/master/configuration-details.md

[29] Intel. 2023. Ice Lake SP. Retrieved April 28, 2023 from https://www.intel.com/content/www/us/en/products/platforms/details/ice-lake-sp.html

[30] Intel. 2023. *Intel 64 and IA-32 Architectures Software Developer Manuals*. Retrieved July 01, 2023 from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[31] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective.. In *Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 172–182. https://doi.org/10.1109/

ISPASS51385.2021.00034
[32] Christian Jacobi, Anthony Saporito, Martin Recktenwald, Aaron Tsai, Ulrich Mayer, Markus M. Helms, Adam Collura, Pak kin Mak, Robert J. Sonnelitter, Michael A. Blake, Tim Bronson, Arthur O'neill, and Vesselina K. Papazova. 2018. Design of the IBM z14 microprocessor. *IBM J. Res. Dev.* 62, 2/3 (2018), 8:1–8:11. https://doi.org/10.1147/JRD.2018.2798718
[33] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers.. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 166–177. https://doi.org/10.1145/2830772.2830785
[34] kernel.org. 2020. perf: Linux profiling with performance counters. Retrieved April 12, 2023 from https://perf.wiki.kernel.org/index.php/Main_Page
[35] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K. Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A. Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications.. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 816–829. https://doi.org/10.1145/3466752.3480124
[36] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing.. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 146–159. https://doi.org/10.1109/MICRO50266.2020.00024
[37] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A. Jiménez, and Baris Kasikci. 2022. Whisper: Profile-Guided Branch Misprediction Elimination for Data Center Applications.. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 19–34. https://doi.org/10.1109/MICRO56248.2022.00017
[38] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service.. In *Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504. https://doi.org/10.1109/CLOUD.2019.00091
[39] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS.. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*. ACM, 477. https://doi.org/10.1145/3357223.3365439
[40] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the Front-End Bottleneck with Shotgun.. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*. ACM, 30–42. https://doi.org/10.1145/3173162.3173178
[41] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A Metadata-Free Architecture for Control Flow Delivery.. In *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 493–504. https://doi.org/10.1109/HPCA.2017.53
[42] EASE Lab. 2022. *vSwarm: A suite of representative serverless cloud-agnostic (i.e., dockerized) benchmarks*. Retrieved April 12, 2022 from https://github.com/ease-lab/vSwarm
[43] EASE Lab. 2022. *vSwarm-u: Microarchitecture for serverless workloads*. Retrieved April 12, 2022 from https://github.com/ease-lab/vSwarm-u
[44] Chit-Kwan Lin and Stephen J. Tarsa. 2019. Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions.. In *Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 228–238. https://doi.org/10.1109/IISWC47752.2019.9042108
[45] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *CoRR* abs/2007.03152 (2020).
[46] Chi-Keung Luk and Todd C. Mowry. 1998. Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors.. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM/IEEE Computer Society, 182–194. https://doi.org/10.1109/MICRO.1998.742780
[47] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond.. In *Proceedings of the 2019 International Symposium on Code Generation and Optimization, (CGO)*. IEEE, 2–14. https://doi.org/10.1109/CGO.2019.8661201
[48] Andrea Pellegrini, Ashok Kumar Tummala, Jamshed Jalal, Mark Werkheiser, Anitha Kona, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, and Tushar Ringe. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* 40, 2 (2020), 53–62. https://doi.org/10.1109/MM.2020.2972222
[49] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning.. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 16–27. https://doi.org/10.1145/93542.93550
[50] Glenn Reinman, Brad Calder, and Todd M. Austin. 1999. Fetch Directed Instruction Prefetching.. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM/IEEE Computer Society, 16–27. https://doi.org/10.1109/MICRO.1999.809439
[51] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. 2022. Lukewarm serverless functions: characterization and optimization.. In *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*. ACM, 757–770. https://doi.org/10.1145/3470496.3527390
[52] André Seznec. 2007. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* 9 (2007), 1–6.
[53] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing.. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 1063–1075. https://doi.org/10.1145/3352460.3358296
[54] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider.. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. USENIX Association, 205–218.
[55] Niranjan K. Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. PDede: Partitioned, Deduplicated, Delta Branch Target Buffer.. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 779–791. https://doi.org/10.1145/3466752.3480046
[56] The Firecracker Authors. 2022. Production Host Setup Recommendations. Retrieved April 12, 2022 from https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md
[57] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2019. BRB: Mitigating Branch Predictor Side-Channels.. In *Proceedings of the 25th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 466–477. https://doi.org/10.1109/HPCA.2019.00058
[58] WikiChip. 2023. *Sunny Cove - Microarchitectures - Intel*. Retrieved April 12, 2023 from https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove
[59] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture.. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 35–44. https://doi.org/10.1109/ISPASS.2014.6844459
[60] Jason Zebchuk, Harold W. Cain, Xin Tong, Vijayalakshmi Srinivasan, and Andreas Moshovos. 2013. RECAP: A region-based cure for the common cold (cache).. In *Proceedings of the 19th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 83–94. https://doi.org/10.1109/HPCA.2013.6522309
[61] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. 2015. Microarchitectural implications of event-driven server-side web applications.. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 762–774. https://doi.org/10.1145/2830772.2830792