Addressing Microarchitectural Implications of Serverless Functions

David Helmut Schall

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2024

Abstract

Serverless computing has emerged as a widely-used paradigm for running services in the cloud. In this model, developers organize applications as a set of functions invoked on-demand in response to events, such as HTTP requests. Developers are charged for CPU time and memory footprint during function execution, incentivising them to reduce runtime and memory consumption. Furthermore, to avoid long start-up delays, cloud providers keep recently-triggered instances idle (or *warm*) for some time, anticipating future invocations. Consequently, a server may host thousands of warm instances of various functions, their executions interleaved based on incoming invocations.

This thesis investigates the workload characteristics of serverless and observes that: (1) there is high interleaving among warm instances on a given server; (2) individual warm functions are invoked relatively infrequently, often at intervals of seconds or minutes; and (3) many function invocations complete within milliseconds. This interleaved execution of rarely invoked functions leads to thrashing of each function's microarchitectural state between invocations. Meanwhile, the short execution time of functions impedes the amortization of warming up on-chip microarchitectural state. 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 we term *lukewarm* execution. Our analysis reveals that the cold microarchitectural state severely affects CPU performance, with the main source of degradation being the core front-end, comprising instruction delivery, branch identification via the BTB, and conditional branch prediction.

Based on our analysis, we propose two mechanisms to address performance degradation due to lukewarm invocations. The first technique is Jukebox, a record-and-replay instruction prefetcher specifically designed to mitigate the high cost of off-chip instruction misses. We demonstrate that Jukebox's simplistic design effectively eliminates more than 95% of long-latency off-chip instruction misses.

The second technique is Ignite, which builds on Jukebox to offer a comprehensive solution for restoring front-end microarchitectural state, including instructions, BTB, and branch predictor state, via unified metadata. Ignite records an invocation's control flow graph in compressed format and uses that to restore the state of the front-end structures the next time the function is invoked. Ignite significantly reduces instruction misses, BTB misses, and branch mispredictions, resulting in an average performance improvement of 43%.

In summary, this thesis demonstrates that serverless systems present distinct workload characteristics that fail to match traditional CPU designs, severely impacting performance. Two simple techniques can overcome these bottlenecks by preserving the microarchitectural state across function invocations.

Lay summary

In our increasingly digital world, the demand for data processing and computing power is growing exponentially. Data centres, which provide massive computing facilities to meet this demand, consume vast amounts of energy and contribute significantly to global carbon emissions. Maximizing the utilization of existing hardware resources is paramount for sustainability.

Serverless computing has emerged as a promising approach to enhance resource management in data centres. By abstracting infrastructure management away from developers and fully ceding it to cloud providers, serverless computing improves developer productivity and allows providers to optimize their systems for better utilization. However, this model introduces a radical shift in workload characteristics that misaligns with traditional server CPU designs.

This thesis investigates the performance implications of serverless workloads on modern CPUs. It reveals that the transient nature of serverless functions, characterized by short execution times and frequent switching between different functions, conflicts with CPU designs optimized for long-running applications. Typically, CPUs undergo a *warm-up* phase where they learn program behaviour to optimize execution. However, serverless functions are often too short-lived to benefit from this warm-up process. As a result, they frequently operate with a *cold* microarchitectural state, leading to significant performance degradation and inefficiencies.

Through comprehensive analysis, this research identifies the key bottlenecks in instruction delivery — the machinery that feeds the CPU with work. To overcome these bottlenecks, two novel, lightweight techniques are proposed that effectively warm up key components of the CPU. Both techniques demonstrate substantial performance improvements while requiring minimal hardware complexity and integration effort, ensuring that serverless computing can remain an efficient and sustainable model for future data centres.

Acknowledgements

My Ph.D. journey was one of the most remarkable times in my life, filled with fun and challenging moments. When asked, *"Would you do it again?"* I have no hesitation in saying, *"Yes"* (though one Ph.D. is enough for me). Beyond all the skills I gained, Edinburgh as a city, and Scotland as a country (including the weather), the most important part was and will always be the people who inspired, supported, and enriched my journey.

First and foremost, I want to express my deepest gratitude to my advisor, Boris Grot. By always being open to ideas and discussion, providing honest feedback, and offering unconditional support, Boris helped me develop skills far beyond those essential for research. His remarkable energy and never-give-up attitude always gave me a boost of motivation, pulling me back on track and fueling me with new ideas, even when everything seemed broken. Most importantly, I could always rely on him, giving me the freedom to explore. Boris will always remain a role model and inspiration in my future. I am happy and grateful to have been his student.

I am also grateful to Andreas Sandberg, first, for convincing me to pursue a Ph.D., suggesting Edinburgh and Boris as supervisors, and for his constant support. Without his technical knowledge, especially with gem5, I could not have implemented everything needed for this work. Meetings with him were always fun, a great opportunity to express my thoughts, and a source of valuable ideas and a different perspective from inside the industry. Together with Andreas, I want to thank Arm Ltd for funding my research, along with the School of Informatics.

The importance of good friends and a great group became clear to me during COVID, and I am thankful to all the students from Boris's group. First, I want to thank Artemiy for supporting me throughout my Ph.D. and motivating me with his perseverance. I'll always remember our wild discussions on branch prediction and life, Jukebox, cycling, running, and numerous other things I cannot list. Similar things can be said about Shyam for his open-mindedness, energy, and motivation to explore new things. I will never forget our discussions until late in the night, Frisbee sessions, cycling trips, and shared meals. Thank you, Dimitrii, for your prompt feedback on presentations and writing, and for your help with serverless; Siavash, for your guidance on writing, particularly this thesis; Esra, for our cultural food explorations and the salad that kept me going during all-nighters before deadlines; Dilina, for your help drawing graphs and broadening my perspective on elephants being ordinary pets; Allan, for your unstoppable energy and motivation, and for showing how much hair can

grow in a year; Shengda, for your insightful questions and demonstrating the power of coffee; and Maria, for the numerous cakes, memorable musical evenings, and your fun, energetic and open nature.

Thanks also to my office mates, Jackson and Dimitra, for our fun discussions in the office and pubs. Jackson, in particular, thank you for exploring and trail-running the Scottish mountains around Glen Coe with me and for helping to make our office the most “interesting“ one in the building.

I want to thank Roman for our weekly discussions and for the great times at ISCA, MICRO, and ACACES. Talking to him always sparked wild and crazy ideas.

Thanks also to Albrecht for being a motivation from a young age and, ultimately, the reason I pursued a Ph.D.

I would also like to thank Nora and my grandma Regina for shaping two remarkable periods during my journey. Nora made the first 15 months of my Ph.D., despite being during COVID, a great experience in Sweden, which I will never forget. I am deeply grateful for those memories with her. My grandma played an important part in the spring of 2022, showing unconditional love and gratitude to everyone around without a single word of her suffering, bridging generations, and making even the hardest times worth remembering. The time spent with her, combining moments of joy and sorrow, has become a cherished treasure and a source of motivation to follow my dreams and enjoy life.

Finally, and most importantly, I want to express my deep gratitude to my family, who always supported and encouraged me to pursue this path. My father, Claus; my mother, Dorothee; and my siblings, Johannes, Lukas, Jonathan, and Charlotte, were always there with motivation, encouragement, distraction, or whatever I needed most. Despite the distance, you were my foundation and safety net, giving me the strength and confidence to pursue my dreams without fear. I know I can always rely on you, and I cannot express my thankfulness enough.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

1. **David Schall**, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. "Lukewarm Serverless Functions: Characterization and Optimization". In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA'22)*. 2022. **IEEE MICRO Top Picks Honorable Mention 2022** [135]

2. **David Schall**, Andreas Sandberg, and Boris Grot. "Warming Up a Cold Front-End with Ignite". In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*. 2023.[136]

In addition to the two above-listed works forming the foundation of this thesis, I also contributed to the following other publications during my PhD program:

1. **David Schall**, Andreas Sandberg, and Boris Grot. "The Last-Level Branch Predictor". 2024. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'24)*.

2. Jackson Woodruff, **David Schall**, Michael F.P. O'Boyle, and Christopher Woodruff. "When Does Saving Power Save the Planet?" In *Proceedings of the 2nd Workshop on Sustainable Computer Systems (HotCarbon'23)*. 2023. [169]

(*David Schall*)

To my parents Claus and Dorothee,
and my siblings, Johannes, Lukas, Jonathan and Charlotte.

Table of Contents

1 Introduction **1**

1.1 The Problem: Serverless Doesn't Fit Modern CPUs 2

1.2 Our Approach 4

1.2.1 Understanding Serverless Workloads 5

1.2.2 Addressing Lukewarm Execution 6

1.3 Thesis Contributions 8

1.4 Thesis Organization 9

2 Background and Related Work **11**

2.1 Cloud Applications: from Monoliths to Serverless 11

2.1.1 Cloud Deployment Models 11

2.1.2 Microservices 12

2.1.3 Serverless Computing 13

2.1.4 Serverless Workload Characteristics 15

2.1.5 Prior Work on Serverless Characteristics 16

2.2 The Server in Serverless 17

2.2.1 The Importance of Microarchitectural State 20

2.3 Front-End Bottleneck in Servers 21

2.4 Conclusion 23

3 Characterizing Serverless Workloads **25**

3.1 Methodology 26

3.2 Serverless Functions on a Cloud Server 27

3.3 Top-Down Analysis of Lukewarm Executions 29

3.4 The Story of Cache Misses 32

3.5 Severless Working Set Characteristics 33

3.5.1 Working Set Size 34

3.5.2 Working Set Commonality 35
3.6 Conclusion 36

4 Jukebox: Preserving Microarchitectural State 39
4.1 Jukebox Overview 40
4.1.1 Record 41
4.1.2 Replay 43
4.1.3 Discussion 44
4.2 Methodology 46
4.2.1 Simulation Infrastructure 46
4.2.2 Workloads 47
4.3 Evaluation 47
4.3.1 Parameterizing Jukebox 47
4.3.2 Performance 49
4.3.3 Miss Coverage 50
4.3.4 Memory Bandwidth 50
4.3.5 Comparison to a State-of-the-Art Instruction Prefetcher 51
4.3.6 Jukebox with a Smaller L2 Cache 53
4.4 Related Work 54
4.5 Conclusion 56

5 Warming Up a Cold Front-End with Ignite 57
5.1 Front-End Prefetching on Lukewarm Invocations 58
5.1.1 Big Picture Results 59
5.1.2 Cold uArch State in Focus 60
5.1.3 Effect of the Cold Branch Predictor 62
5.1.4 Putting it all Together 63
5.2 IGNITE 64
5.2.1 Record 66
5.2.2 Replay 67
5.2.3 Operating System Interface 68
5.2.4 Security Aspects 70
5.3 Methodology 71
5.4 Evaluation 72
5.4.1 Performance Analysis 72
5.4.2 Miss Coverage and Accuracy 73

5.4.3 Memory Bandwidth 74
5.4.4 Sensitivity to Bimodal Initialization 75
5.4.5 Temporal Streaming Prefetchers 77
5.5 Related Work 78
5.6 Conclusion 79

6 Conclusions and Future Work **81**
6.1 Summary of Contributions 82
6.1.1 Understanding Serverless Workloads 82
6.1.2 Addressing Lukewarm Execution 82
6.2 Future Work 83
6.2.1 Addressing Back-End Inefficiencies 83
6.2.2 Software Approach 84
6.2.3 Accelerate Context Switches 85
6.3 Impact Beyond Severless 85

Bibliography **87**

Chapter 1

Introduction

Our world is increasingly digitized, with data becoming the new oil [42]. From healthcare to supply chains to transportation, all sectors rely heavily on data and computing, with demand experiencing exponential growth [71]. Data centres play a crucial role by providing massive computing and storage resources, and their demand is expected to grow at a compound annual growth rate (CAGR) of 10.9% from 2023 to 2030 [59].

However, building and powering data centres has a severe environmental impact and a large carbon footprint. Data centres consume enormous amounts of energy — hundreds of terawatt-hours — exceeding the energy usage of entire countries like Iran [82]. At their current growth rate, data centres are expected to consume more than 2.5 times the energy in 2030 compared to 2024 [13]. Additionally, as technology scaling slows down and producing chips in smaller technology nodes consumes more energy and rare elements [52], the embodied carbon emission of servers and, correspondingly, data centres increase and become the dominant factor [64, 65]. Most concerning is that most of the energy and carbon spent on data centres goes wasted due to improper resource planning and, as a result, low server utilization. A study by Google on over 5,000 servers found utilization rates between 10-50% over six months [25]. Other studies confirm similar utilization rates, with an average utilization far below 50% [6, 60, 106]. Given that the information and communications technology (ICT) ecosystem accounts for 2% of the global carbon footprint — on par with the aviation industry [82] — and data centres are the second-largest growing factor within this sector, designing systems that maximize hardware resource utilization is of paramount importance.

Problematically, efficient resource management is challenging and further exacerbated by the increasing complexity of modern online services. A typical online service

comprises hundreds of components [54, 113], requiring a deep understanding of the entire software and system stack to allocate compute resources optimally. This need for expertise and time conflicts with developer productivity, which is critical for business success [70]. A survey by Granulate shows that even with cloud infrastructures and auto-scaling techniques, only 30-50% CPU utilization is achieved, and the lack of manpower, budget constraints, and time-to-value being major obstacles to further improving utilization [60].

Serverless computing emerged as a promising approach to simplifying the resource management of online applications running in data centers while improving utilization. In serverless, the underlying infrastructure required to run a large-scale service is abstracted away from the developer and fully ceded to the cloud provider. Developers, freed from the burden of configuring, maintaining, and planning their cloud resource consumption, can focus on the core business logic and innovate their offerings. At the same time, the cloud provider, equipped with knowledge and expertise about its infrastructure, can optimize its system stack and concentrate resources consumed by multiple customers, maximizing its data centre computing capacity. This way, serverless bridges the gap between the need for efficient resource management and productivity, providing a cost-effective solution for both the customer and the cloud provider. Not surprisingly, serverless computing experiences one of the highest rates of adoption of public cloud offerings — 9% increase in one year to 48% of organizations using serverless in 2024 [48] — and is expected to grow at a CAGR of 22.2% from 2024 to 2032 [129].

1.1 The Problem: Serverless Doesn't Fit Modern CPUs

Although highly attractive for developers and cloud providers, serverless computing presents a radical shift in the cloud execution model accompanied by a likewise drastic change in workload characteristics. Traditionally, monolithic or microservice-based architectures are used to implement cloud applications in one or more virtual machines (VMs) that constantly occupy one of the cloud provider's machines. On the contrary, serverless applications are composed of many small, typically short-running, stateless functions that are invoked sporadically on demand [36, 37, 80, 150]. In fact, developers are incentivized to break down their applications into a collection of fine-grained functions, allowing the provider to scale different parts of the business logic independently, thus maximizing elasticity. Moreover, serverless providers, such as

AWS Lambda and Google Cloud Functions, charge users for the amount of memory a function consumes times the function's execution time [11, 55], which further pushes the developers toward leaner, short-running functions. For instance, studies show that serverless functions tend to have short execution durations, as low as a millisecond or less [36, 37], and memory footprints often in the range of 128–256MiB [36]. Likewise, in the case of interactive services, for which serverless is a common implementation model [44], the end-to-end latency must often meet an SLO target in the order of a few tens of milliseconds [57]. To achieve this, the individual functions may be expected to be complete in a millisecond or less [80, 154, 172].

Furthermore, to avoid the long delays of booting a new function instance, cloud providers tend to keep recently-invoked instances alive (or *warm*, in serverless parlance) instead of shutting them down in anticipation of additional invocations to that instance. Recent work has shown that Amazon, Google, and Microsoft all keep recently invoked function instances warm for at least several minutes and up to an hour [163]. The combination of small memory footprints for many functions, long keep-alive intervals enforced by cloud providers [163], and hundreds of gigabytes of memory in a representative server results in *thousands* of warm function instances residing on a typical cloud server [5]. The execution of these functions is interleaved in time based on invocation traffic, with many warm functions experiencing invocation inter-arrival rates measured in seconds or minutes [150] — an invocation rate that is relatively infrequent compared to their run time.

Problematically, the microarchitecture of a CPU is designed and optimized for long-running workloads. By taking advantage of spatial and temporal locality during program execution, CPUs cache data and instructions close to the core to overcome the high latency of memory accesses [69]. Furthermore, long execution times allow structures like branch predictors or prefetchers to learn the program's control flow behaviour and data access patterns, enabling highly accurate branch and prefetch predictions. Today, server-class CPUs are equipped with multi-megabyte caches, consuming most of the chip real estate [3, 112, 131], as well as sophisticated branch predictors [3, 148] and data prefetchers [61, 122, 125] all with the sole reason to collect data, instructions or information about the program behaviour the processor can use to optimize instruction execution. However, when launching a new application, the CPU microarchitecture is *cold*, and it may take millions of instructions for the microarchitectural structures to cache more data and instructions and learn the behaviour of the new application until it finally reaches its peak performance. This phase is called *warm-up*

and has been well-studied in the context of simulating workloads [33, 67, 93, 115]. For example, Yue et al. [105] found it takes 3-23M instructions to warm up a 1MiB cache, while [115] showed the warm-up increases to more than 100M instructions as the cache size grows. A similar study on branch predictors shows that it can take up to 10M branch instructions to train a state-of-the-art branch predictor until it reaches its peak performance [166]. In the cold state, the branch predictor has a misprediction rate 10x higher compared to after training, which can result in performance degradation of up to 45% [166]. On top of hurting performance, inaccurate predictions — as a result of the cold state — waste energy to correct predictions or trigger unnecessary memory transactions [77]. Considering a typical IPC of around 1 instruction per cycle found for modern server workloads [45, 79, 124], it can take up to 10-100M cycles to warm up a cache and $\sim$50M cycles[1] to warm up a branch predictor. With a typical CPU frequency of 2-3GHz [4, 14, 125], it can easily take tens of milliseconds to warm up the microarchitecture. While the warm-up time is negligible for conventional always-on cloud applications, where VMs stay alive for hours or days, it is significant for serverless functions that run for milliseconds or even less.

The mismatch between CPU microarchitecture design goals and the workload characteristics of serverless functions is not yet fully understood, and it can lead to high inefficiencies in the execution of serverless functions. Considering the increasing scale of serverless, already serving ”trillions of requests per month” [5], these inefficiencies may result in performance degradation worth millions of dollars and a huge amount of wasted energy. Therefore, this mismatch must be thoroughly understood and addressed to ensure serverless computing is a sustainable solution for resource management challenges.

1.2 Our Approach

To ensure the sustainability of serverless computing as a solution for data centre resource management challenges, this thesis sets the following objectives:

1. **Examine Performance Implications:** Develop a comprehensive understanding of the performance implications of serverless workload characteristics on modern CPUs.

[1] Assuming that every fifth instruction is a branch [22]

2. **Understand Bottlenecks:** Identify bottlenecks and offer insights into their existence and potential solutions.
3. **Overcome Bottlenecks:** Design and evaluate solutions to address bottlenecks resulting from the mismatch in CPU designs and the serverless workload characteristics.
4. **Ensure Lightweightness:** The proposed solutions must be lightweight, non-disruptive, and easy to integrate with existing hardware and software to ensure sustainability.

1.2.1 Understanding Serverless Workloads

To achieve objective **1.**, we examine a typical serverless scenario where a contemporary Intel Xeon Ice Lake server is loaded with many interleaved executing functions. We find that the large number of co-located functions with extremely short running times (milliseconds or less) results in a high degree of interleaving and an unprecedented frequency of context switching. Moreover, the relatively long inter-invocation intervals (seconds to minutes) mean that hundreds to 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. We term this phenomenon *lukewarm execution*.

Our analysis reveals that lukewarm executions of functions cause a performance degradation of 2x or more compared to executions with a fully warmed-up microarchitectural state (i.e., back-to-back executions of the same function on the same core). The reason for such high performance degradation is that the short running time of the functions (typically on the order of a few milliseconds or less) means functions will either never complete the warm-up phase or the execution time is insufficient to amortize and benefit from warming up the microarchitectural structures.

Objective **2.** In a detailed Top-Down performance analysis [170], we identify the front-end (i.e., instruction delivery, branch identification, and branch prediction) as the main culprit behind the poor performance on lukewarm invocations. Collectively, the front-end is responsible for two-thirds of the performance degradation compared to executions with a warm microarchitectural state. Specifically, we find that off-chip misses for instructions are the major pain point, increasing the fetch latency by 45%

Based on this finding, we examine the instruction and branch working sets across multiple invocations of the same function and find a significant commonality: ??% of all instruction cache blocks accessed by one invocation are also accessed by a subsequent invocation. We further find that the instruction footprints of individual invocations for the studied functions range from 300KiB to over 800KiB. With hundreds or thousands of co-running warm function instances on a serverless host, keeping the combined instruction footprints of all functions in processor caches becomes clearly infeasible. The same observations are made for the branch working sets; the branch predictor units (BPU) capacity can comfortably hold the working sets of a single function but is completely overwhelmed by the vast amount of interleaved functions.

1.2.2 Addressing Lukewarm Execution

Objective **3.** Spurred by our observations, we strive for solutions to overcome the performance degradation caused by lukewarm execution. However, we note that data centre servers execute a variety of workloads from different clients and applications, which might not be equally well-suited for the serverless model. Furthermore, serverless is a relatively new paradigm in a rapidly changing software ecosystem. Thus, while a highly tailored solution might achieve optimal results for serverless, it would inevitably be suboptimal for other existing or future workloads. Since our primary goal is to use hardware resources as *efficiently and sustainably* as possible while maximizing their utilization, we make it a key requirement for our solutions to be non-disruptive, easy to integrate with existing hardware and software, and not inhibit the generality of the CPU (Objective **4.**).

In response, we present the following two highly effective lightweight techniques that address lukewarm execution, ensure easy integration, and come at a small hardware cost.

Jukebox The first technique is *Jukebox*, a record-and-replay instruction prefetcher that targets the largest source of performance loss in lukewarm serverless function executions: off-chip instruction misses. It is based on the insight of high commonality in the instruction working sets of individual invocations of the same function. Jukebox records the instruction working set of a function upon its first invocation and replays it upon subsequent invocations. The idea of a record-and-replay instruction prefetcher is not new; indeed, prior works have proposed them for long-running server workloads by

recording entire streams of instruction cache accesses or misses [46, 85, 86], requiring over 200KB of on-chip storage for metadata. In contrast, Jukebox solves a different problem: how to accelerate short-running tasks that have no microarchitectural state or metadata on-chip. To accommodate thousands of warm functions, Jukebox stores its metadata in main memory using simple spatiotemporal compression designed for high coverage and low metadata redundancy. Our evaluation shows that 48KiB of metadata (i.e., twelve OS pages) is sufficient for high efficacy; for a thousand warm function instances, the required metadata cost is a mere 48MiB. Jukebox requires simple hardware support and a negligible amount of on-chip state without modifications to the processor caches. Our full-system simulation of Jukebox reveals that it speeds up the execution of lukewarm functions by 17.8%, on average.

Ignite While Jukebox is highly effective in eliminating off-chip instruction misses by restoring working sets into the L2, it leaves other microarchitectural structures in a cold state. However, lukewarm execution creates various bottlenecks, predominantly in the CPU front-end, including a high number of L1 instruction misses and branch mispredictions. We find that even the state-of-the-art front-end prefetcher Boomerang (which proactively fills the L1-I and the BTB) [96] cannot capitalize on Jukebox and 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). As a result, combining Jukebox with Boomerang provides only 20% speedup on average — a disappointing 2.2% improvement than Jukebox alone (17.8%) — falling considerably short of an ideal front-end that delivers a 61% average speed-up over an aggressive next-line prefetcher.

We perform a root-cause analysis to understand why the state-of-the-art 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.

Therefore, based on our findings and motivated by the results of Jukebox, we aim even higher and propose 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 compared to prior art.

1.3 Thesis Contributions

In summary, this dissertation demonstrates that:

Thesis Statement

Serverless presents distinct workload characteristics that fail to match the designs of traditional CPUs. This mismatch creates severe bottlenecks, which can be overcome by preserving the microarchitectural state across function invocations.

We support this statement by making the following contributions:

1. We **evaluate serverless workload characteristics** and reveal a severe performance degradation by those workloads. The reason is the combination of extremely short execution time and, in contrast, long inter-invocation times of a typical serverless function, which causes the CPU to obliterate the on-chip microarchitectural state between two invocations. The result is a 100-294% (162% on average) performance loss compared to an optimal scenario with completely warm microarchitectural structures.

2. We conduct a systematic **Top-Down analysis** to identify the sources of the performance loss. While we found that a cold microarchitectural state affects efficient instruction execution in multiple ways, the main bottlenecks are caused by the processor front-end, which cannot fill the pipeline with instructions. Namely, the main bottlenecks are off-chip instruction misses, BTB misses, and branch mispredictions.

3. We provide insights on the sources of performance degradation by **analyzing** the instruction and branch **working sets** find individual serverless functions have an extremely high commonality in their working sets which, in general, comfortably fit into the on-chip structures. However, the combined working set of thousands of collocated, interleaved executing functions is orders of magnitude larger than the on-chip structures can possibly hold.

4. We propose **Jukebox, a record-and-replay instruction prefetcher** accelerates lukewarm function executions eliminating off-chip instruction misses. Jukebox requires a small amount (32KiB) of metadata in main memory per function instances (32MiB for a thousand functions) and provides 17.8% performance improvement on lukewarm invocations, on average.

5. We aim higher with **Ignite, a comprehensive restoration technique** that expands upon Jukebox and uses a unified set of metadata to warm up the entire CPU front-end, including instruction cache, BTB, and branch predictor. Hereby, Ignite improves performance by 43%, on average. Most importantly, while being highly effective, Ignite, as well as Jukebox, come at a small hardware cost and are easy to integrate with existing hardware and software, ensuring the sustainability of the solutions.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides the background related to the work presented in this thesis. Chapter 3 presents a comprehensive characterization of serverless functions running on a commodity cloud server and reveals severe performance bottlenecks stemming from serverless functions' drastically different workload characteristics compared to traditional cloud applications. We address these performance degradations in chapters 4 and 5 by proposing two simple but highly effective mechanisms that are based on the insight of high commonality in the execution of serverless functions. Chapter 4 presents Jukebox, a lightweight technique designed to overcome the largest source of performance degradation in serverless functions, cold on-chip instruction state. In chapter 5 we expand upon Jukebox and present a comprehensive solution for the cold CPU front-end. Finally, chapter 6 summarizes our key findings and provides potential directions for future research.

Chapter 2

Background and Related Work

This chapter provides the background relevant to the dissertation. The first half briefly overviews how complex online services are deployed in the cloud. We present the challenges of conventional cloud execution models and how serverless computing addresses them. We then review the workload characteristics of serverless and what distinguishes them from conventional workloads.

In the second half, we will provide an overview of the microarchitecture of modern server CPUs and their key components. We will then discuss the main performance bottlenecks in modern server CPUs and how they are addressed in conventional cloud workloads.

2.1 Cloud Applications: from Monoliths to Serverless

2.1.1 Cloud Deployment Models

Modern online services, such as online shops, streaming platforms, or corporate websites, have become highly complex and comprise an extensive number of features and functionality. For example, Netflix, a popular streaming platform, reported in 2017 that its streaming service comprises more than 700 individual components [66, 113]. While traditionally, these services are implemented as monolithic applications running on private servers, at the increasing complexity and scale, the cost and knowledge needed to maintain the own hardware infrastructure becomes a significant concern [54].

The advent of cloud computing opened up an alternative deployment model where services are deployed as virtual machines (VMs) on the infrastructure of a cloud provider. Moving to the cloud relieves the customer from all tasks related to managing

its physical hardware. The cloud provider takes care of buying, maintaining and upgrading the hardware and ensuring high availability and reliability of their cloud infrastructure. The cloud customer can focus on innovating the service and flexibly scale it by adding or removing VMs as needed. The cloud provider can also allocate resources more efficiently by consolidating services from different users on the same hardware. This increases the overall hardware utilization, translating into lower carbon emissions and lower costs for both the cloud provider and customer [138].

2.1.2 Microservices

While cloud computing relieves the cloud customer from hardware management tasks, it does not solve the problem of steadily increasing functionality and complexity. In a monolithic application, all features are tightly coupled, making it difficult to develop and add new features [66, 137]. Furthermore, scaling up an entire resource-intensive monolithic VM is expensive and inefficient, as the service may not be fully utilized at all times [138].

To address this issue, the industry, led by major players such as Amazon, Twitter, Netflix, Apple, and eBay, has moved towards a microservices architecture [31, 51, 102, 162]. In this architecture, services are decomposed into smaller, independent units called microservices. Each microservice is responsible for a specific functionality and communicates with other microservices over the network [51]. This allows teams of developers to work independently on different functionality, making it easier to maintain, develop, and add new features [20]. It further improves the service's resilience to failures, as a failure in one microservice does not necessarily affect the entire application [151]. Not surprisingly, microservices have become the prevailing architecture for building large-scale online applications like Amazon, Uber, or Netflix [66, 133].

In a microservice architecture, each microservice is deployed in a separate VM and can be scaled independently, providing more elasticity than monoliths. However, scaling microservice on a per VM basis is still inefficient as the VMs may not always be fully utilised, and the cloud customer must pay for all the enabled VMs even when the service is idle [139]. Furthermore, due to the increased complexity, deploying and scaling a large number of microservices and VMs is a complex and time-consuming task, requiring expertise in the entire software and system stack, which is often not affordable under budget constraints and time-to-market pressure [60]. As a result, a significant fraction of cloud resources are wasted by idle VMs consuming energy and

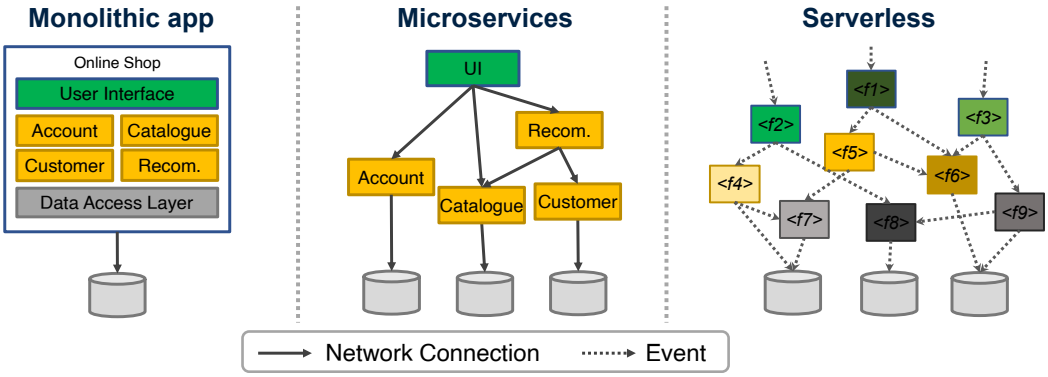


Figure 2.1: Cloud deployment models: monolithic, microservices, and serverless.

increasing the cost and carbon footprint of cloud providers [6].

2.1.3 Serverless Computing

To address the challenges of conventional cloud deployments, serverless computing, also known as Function-as-a-Service (FaaS), has emerged as a popular paradigm [138]. In serverless computing, developers structure their applications as a set of *stateless* functions, fully delegating the management of resources to the cloud provider [98]. Similar to lambda expressions, serverless functions are small pieces of code executed in response to events such as HTTP requests, database updates, or timers. These functions communicate via events and can be composed into workflows [21]. Functions can be written in various programming languages, including Python, JavaScript, Java, or Go, depending on the required functionality or the programmer's preference [39].

Figure 2.1 illustrates the differences between three deployment models by a simple online shop example: monolithic, microservices, and serverless. In the monolithic model, the entire application runs on a single VM. The microservices model decomposes the application into several independent services communicating over fixed network connections. In the serverless model, the application is constructed of individual functions, and communication happens via events. The outstanding features of serverless are simplicity for the developer, extreme elasticity, and a unique cost model.

Simplicity The abstraction of *event-based* serverless functions allows developers to focus on their primary expertise: writing code to innovate their services without worrying about the underlying hardware or software infrastructure. The cloud provider handles provisioning, scaling, and managing all resources needed to execute the functions. To be more specific, in modern serverless runtimes such as AWS Lambda [143],

Google Cloud Functions [30], or Azure Functions [23], developers only need to upload their function code (packaged as a zip file or Linux container [144]), specify the trigger event, and configure the function's permissions. When an event occurs, the cloud provider launches a function instance, schedules it on an available server in its fleet, executes the function code, and shuts down the instance upon completion.

To ensure isolation, each function runs in its own container or VM. However, starting a new function instance requires booting up a new container or VM, which can take up to several seconds [164]. To mitigate this delay, cloud providers keep function instances alive (*warm* in the serverless parlance) for a certain period after completion, anticipating to reuse the instance for subsequent invocations [56, 114, 116]. This practice reduces cold-start latency and improves the responsiveness of functions.

The key point is that the cloud provider manages the entire function lifecycle, completely hiding it from the developer. The developer only pays for the time the function is *actively* processing requests.

Elasticity The *stateless* nature of serverless functions is a crucial characteristic of the serverless model. This means that functions do not carry any state between invocations, and each invocation is independent of the previous one. All the information needed to process an invocation is contained within the function image and the event input. This statelessness enables rapid and independent scaling of functions, allowing the cloud provider to create and shut down function instances at any time without affecting the running application. Based on incoming request traffic, a function can be scaled from zero to thousands of instances within seconds [10], making serverless highly dynamic and suitable for spiky and unpredictable workloads [142]. Combining the simplicity of severless with the stateless, event-driven model enables developers to build highly scalable and responsive applications that can handle a large number of concurrent requests within days [40, 43, 140].

Cost model Being event-driven presents a radical shift in the cloud execution model. In traditional cloud deployments, the application is constantly running on a VM instance, incurring a cost to the developer regardless of whether it is processing requests — doing actual work — or is idle. In contrast, serverless functions consume cloud resources only when needed. If a function is not processing requests, it is not running, and the developer does pay. This makes serverless computing truly elastic and cost-effective, as developers pay only for the time their function executes (i.e., per invoca-

tion) and the memory consumed [94]. Similarly, cloud providers embrace the serverless model as it enables more effective resource allocation without spending expensive hardware resources on idle VMs. The cloud provider can consolidate functions from many different users on the same hardware, increasing the hardware utilization and reducing the cost for the cloud provider. Furthermore, managing the entire software and hardware stack allows cloud providers to optimize the infrastructure for serverless workloads, improving the performance and efficiency of overall data centres. AWS, for example, introduced a light-way virtualization technology called Firecracker, specifically designed for serverless workloads to reduce the overhead of starting and stopping VMs [5]. In addition, AWS advertises that lambda functions execute more efficiently on their customized Graviton2 Arm-based processors than on x86 processors, translating into lower carbon emissions and lower cost for the cloud customer [141].

2.1.4 Serverless Workload Characteristics

Serverless functions are designed to execute fine-grained, short-running tasks [80, 150]. In fact, to take full advantage of the serverless model, developers are incentivized to increase granularity by breaking their applications into a collection of fine-grained functions, thus maximizing elasticity and allowing different parts of the application's business logic to scale independently. As a result, most functions in serverless production deployments are short-running with small memory footprints [36, 37, 38]. For instance, 67% of Lambda@Edge functions complete within 20ms [37]. The demand for short functions continues to increase; for example, one AWS Lambda study shows that the median function duration became 2× shorter in 2020 compared to the median duration in 2019 [37]. In response to this trend, AWS Lambda decreased the billing granularity from 100ms to 1ms [11]. Moreover, serverless providers, such as AWS Lambda and Google Cloud Functions, charge users for the maximum amount of memory each of their functions consumes [11, 55], further pushing developers toward leaner, finer-grained functions. Both AWS Lambda and Azure Functions demonstrate that over 70% of their functions have small memory footprints, allocating less than 300MB of memory [36, 150].

Despite the short running time of many function instances and their small memory footprints, cold-booting a function is a long-latency operation that can take hundreds of milliseconds in today's clouds [163, 164]. To avoid this latency in the critical path of a function invocation, cloud providers tend to keep idle function instances alive (or

warm, in serverless parlance) for 5-60 minutes in anticipation of additional invocations to that instance [107, 108, 109, 163]. Although keeping function instances warm comes at a cost for the providers because users are billed only for the actual processing time of individual invocations, all major providers deploy this performance optimization as it enables users to meet SLO targets, thus boosting the attractiveness of their serverless offerings [57, 80, 154, 172].

With providers keeping function instances warm for 5-60 minutes, approximately 20-40% of all deployed functions have a warm instance when a request arrives, according to the study of Azure Functions [150]. The same study shows that fewer than 5% of all invocations have an inter-arrival time (IAT) of under a second. Thus, the IAT of a vast majority of invocations to warm instances lies in the range of 1 second to a few minutes – an invocation rate that is relatively infrequent compared to their run time.

The combination of small memory footprints for many functions, long keep-alive intervals enforced by cloud providers [163], and hundreds of gigabytes of memory in a representative server allow cloud providers to collocate *thousands* of warm function instances on a typical cloud server [5] The execution of these functions is interleaved in time based on invocation traffic, achieving high density and resource utilization at a server level.

2.1.5 Prior Work on Serverless Characteristics

Modern processors are sophisticated designs highly optimized for the workload characteristics of applications they typically execute to extract more performance. A shift in workload characteristics inevitably has as consequence that current designs might not be equally well suited for future workloads. Serverless presents such a shift in workload characteristics, and its implications must be understood.

While a lot of work was done to understand the characteristics of conventional cloud workloads [45, 51, 119], to date, little work has tried to understand the microarchitectural implications of serverless programming. Shahrad et al. [149] examined the execution of five serverless functions, identifying issues such as high cold-start latency, high variability in execution time, and performance overheads due to containerization. The work observed that short-running functions experienced much higher variability in execution time than long-running tasks but did not attempt to understand the sources of variability other than high branch mispredictions.

The goal of this thesis is to bridge the missing gap and provide a deep understand-

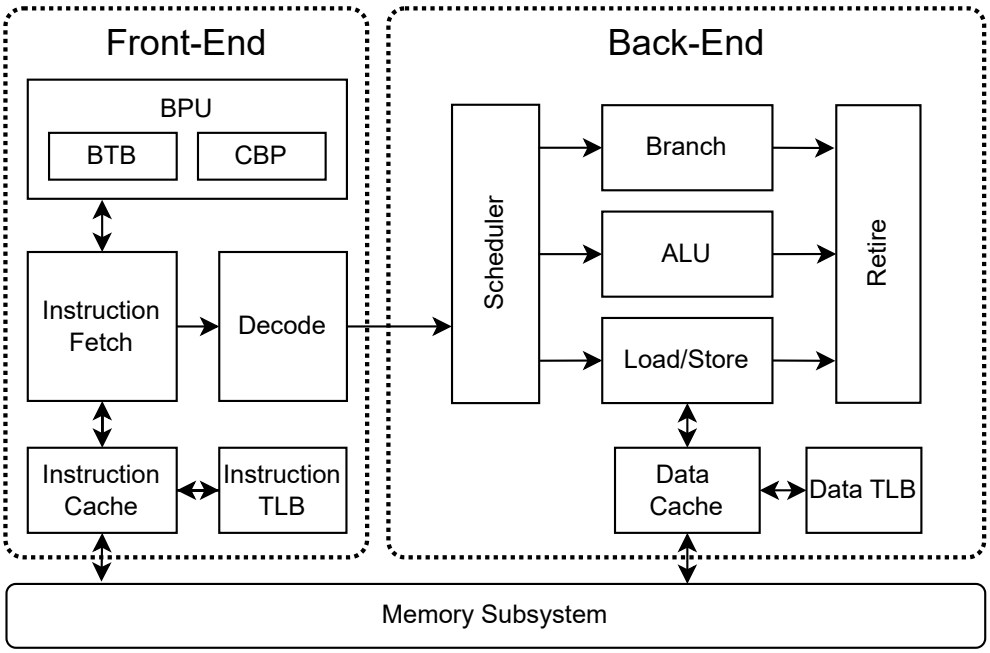


Figure 2.2: CPU pipeline

ing of how serverless functions perform on modern hardware.

2.2 The Server in Serverless

The term *serverless* stems from the illusion provided to developers freed up from hardware management tasks and alludes that no server hardware is needed. However, in fact, serverless functions are executed in large data centres on powerful servers. At its heart, each server features hundreds of gigabytes of memory and a highly sophisticated CPU to store and process data. A simplified view of the architecture of a modern high-performance CPU is depicted in Figure 2.2, which can conceptually be divided into two halves: the front-end and the back-end.

CPU Pipeline The CPU *front-end* is responsible for fetching program instructions from memory and decoding them into hardware operations (micro-ops). After decoding, the micro-ops are passed to the *back-end*, which comprises various functional units for different types of operations and a register file to store intermediate results. Since different operations require different functional units and may take varying amounts of time to execute, the back-end of a modern CPU is designed to execute multiple instructions in parallel and out of order to maximize performance and resource utilization. To achieve this, the scheduler monitors micro-ops dependencies and schedules

them for execution as soon as all dependencies are resolved and the required functional unit is available. Once the micro-ops are executed, the results are written back into the register file, and the next dependent micro-ops are scheduled for execution.

In an optimal scenario, the CPU executes a constant flow of instructions, and the pipeline never comes at a hold. However, due to limited resources and the increasing gap between processor and memory speed, the CPU has various bottlenecks, manifesting as *pipeline stalls*. Specifically, whenever the back-end runs out of functional units or must wait for the required data to arrive from memory, it stalls the pipeline as no more instruction can be consumed. Correspondingly, the front-end stalls the pipeline if it cannot fetch and decode instructions fast enough from memory.

To mitigate these stalls, the pipeline is augmented with a wide range of microarchitectural structures based on three key concepts: caching, prefetching, and speculative execution.

Caching Instruction and data *caches* exploit spatial and temporal locality in memory access patterns by storing larger blocks (known as cache lines) of recently used memory in fast-access on-chip memory (SRAM) close to the pipeline. Future access to the same block can then be served from the fast SRAM, avoiding going again to the slow off-chip memory, reducing memory access latency from hundreds of cycles to a few [69]. Similarly, the translation look-aside buffer (TLB) keeps recent virtual-to-physical address translations at the core, avoiding the need for translation for every request. Since larger structures always come at the cost of increased access latency, the first-level caches and TLBs are backed up by multiple levels of progressively larger and slower structures that balance size and latency.

Prefetching Caches can mitigate the memory latency for subsequent accesses to a cache line once it's cached, but not for the very first access. To avoid paying the cost for the first access, prefetchers observe memory requests to train an internal model of the currently running process's access patterns. This model anticipates future accesses and brings the data into upper-level caches before it is needed. Accurate prefetching can eliminate memory latency for the first access, extend the perceived capacity of caches, and provide the illusion of large *and* fast memory. Prefetching remains an active field of research, with many techniques and optimizations proposed for instruction [128, 132], data [118], and translation [165]

Speculative Execution While caching and prefetching address long memory latencies, speculation addresses pipeline hazards by performing work that might not be needed beforehand [134]. The most widely adopted speculative technique is *branch prediction*, which addresses control flow hazards by predicting branch instructions' direction and target address. If the prediction is wrong, the result will be rolled back by flushing the pipeline and resetting the frond-end on the correct path before execution can continue [8]. Branch prediction is a fundamental technique to unlock performance in modern pipelined processors. It is performed in a dedicated structure called the branch predictor unit (BPU), which includes various predictors specialized for different types of branches, such as conditional branches, indirect branches, or returns. Since over 80% of branches in typical workloads are conditional [22], the most relevant predictor is the conditional branch predictor (CBP). The CBP learns control flow dynamics by correlating branch directions with the history of previously executed branches. Modern CBPs can detect correlations across hundreds of branches using partial pattern-matching algorithms, achieving accuracies exceeding 99% [3, 125, 145, 146, 147, 148].

In a modern CPU, the front-end comprises several stages, taking multiple cycles until instructions are decoded and identified as branches. For example, in Arm's Neoverse N1 server chip, this process takes about six cycles [49]. This causes two challenges. First, if the branch predictor predicts a branch, it takes multiple cycles until the branch target is decoded from the instruction, and the front-end doesn't know where to continue. Second, it is not even known if the current instruction address (PC) is a branch, and a prediction should be made. To avoid wasting these fetch-to-decode cycles and overcome the two challenges, the front end maintains the targets of recently taken branches in a cache-like structure called the branch target buffer (BTB). Furthermore, by indexing the BTB with the current fetch PC, a hit will tell the front-end that the current address contains a branch, and a prediction should be made. Note that the BTB contains only branches that have been taken at least once [74]. Therefore, a BTB miss could also mean a branch that has never been taken. In most cases, this is not problematic because the front end only needs to know when to stop fetching sequentially and redirect to another location.

Essentially, the BTB comprises a mapping of control flow discontinuities, which the front-end uses to determine when to redirect fetching. This insight, combined with the high accuracy of the branch predictor, led researchers to decouple the BPU from the fetch unit, let it run ahead, and use it as a highly efficient instruction prefetcher [128]

making the BTB a key component for modern processors [3].

2.2.1 The Importance of Microarchitectural State

Caching, prefetching, and speculation are not directly involved in executing instructions, but they are crucial for achieving high performance and efficiency. In essence, microarchitectural structures implementing these mechanisms consist of fast-access memory that maintains various types of *microarchitectural state*, comprising information about the currently executing program and logic to access it. For caches, this state consists of raw data and instructions, while for the BPU and prefetchers, it includes metadata (i.e., profile information) on control flow and memory access behaviour. More on-chip state storage equates to higher performance, hence modern microprocessors devote most chip area to storing microarchitectural state [3, 111, 112, 130, 155]. For example, Intel reports that 70% of the transistors in their Itanium processor are spent on the L3 cache [155]. Similarly, IBM revealed that over 30% of a single z15 core area is dedicated to the L2 cache, with another 12% allocated to the BPU [3].

Devoting a significant portion of chip area to storing microarchitectural state is justified by its immense effect on performance, particularly evident when execution transitions to a new application. In such cases, the microarchitecture starts with no stored information about the new application or is *"cold"* in microarchitecture terms. The CPU must first learn the new application's behaviour and cache sufficient information before achieving high performance. With multi-megabyte caches [14] and hundreds of kilobytes of BPU states [3, 61], a modern server processor can acquire runtime information over millions of instructions, significantly boosting performance. For example, a study on the effect of a cold branch predictor state shows that over the course of 10 million branch instructions (approximately 50 million instructions, assuming every fifth instruction is a branch [22]) following a context switch, a state-of-the-art branch predictor increases its accuracy by more than 10x, delivering 45% higher performance than when the branch predictor is cold [166].

This design is suited for traditional workloads, such as VMs running always-on online applications [51], where the microarchitecture remains in a *warm* state. The application benefits from the performance boost delivered by the warmed microarchitecture, making the warm-up time negligible. However, in a serverless environment where function execution times can be as low as a millisecond, the time required to warm up the microarchitecture is substantial and may even exceed the function's ex-

ecution time. Considering the previous branch predictor example and a typical CPU running at 2-3 GHz [4, 14, 125], it can take tens of milliseconds to fully warm up the microarchitecture. Consequently, for serverless functions, the CPU might not have sufficient time to reach peak performance. In chapter 3, we delve deeper into the implications of a cold microarchitecture on the performance of serverless functions.

2.3 Front-End Bottleneck in Servers

The pipelines of modern server CPU's are highly complex, keep hundreds of instructions in-flight and feature massive caches and branch predictors to maximize performance and efficiency [14, 168]. A particular and well-known challenge for server workloads is the so-called *front-end bottleneck* [86, 88, 95, 96]. The root cause of the bottleneck lies in the deeply-layered software stacks with multi-megabyte instruction working sets and commensurately large control flow state [83]. 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 instruction cache (L1-I), BTB and CBP [96, 100]. As a result, the front-end is unable to sustain a constantly high supply of new instructions to feed the back-end efficiently.

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.

Temporal streaming [47] 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 first been shown to be highly effective in mitigating instruction misses, with the most recent design, PIF, achieving almost perfect coverage [46]. Demonstrating its effectiveness for instructions, temporal streaming has been extended to BTB prefetching [27]. The most recent work in this area, called Confluence [86], proposes to unify instruction and BTB prefetching by exploiting the insight that the branch target can be predecoded from the instruction block that is prefetched into the L1-I. This allows Confluence to prefetch instruction cache blocks, predecode the prefetched blocks on their entry to the L1-I to extract branches and install the extracted branches and their targets into the BTB. In

effect, Confluence enables a BTB prefetcher with no additional metadata beyond that necessary for temporal instruction streaming.

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 [27, 86]. 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 [128]. 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. Essentially, all recent server CPUs have implemented FDP [3, 50, 61, 74, 76, 122].

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 [95, 96]. 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 [14, 168].

To further reduce FDP's dependence on BTB capacity, recent research in FDP has focused on BTB prefetching. Thus, Boomerang [96] 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 [3, 61, 76, 122].

2.4 Conclusion

Cloud deployment models are highly attractive to online businesses because they shift the responsibility of managing computing resources to the cloud provider. Serverless computing, a rapidly emerging model, exemplifies this shift. In serverless computing, the cloud provider handles the full lifecycle of running code in the cloud. The simplicity, high elasticity, and the pay-as-you-go cost model of serverless make it particularly appealing to customers.

Despite its appeal, serverless represents a radical change in the execution model, resulting in unique workload characteristics: extremely short execution times, small memory footprints, and long inter-arrival times. These characteristics contrast sharply with those of constantly running VMs in traditional monolithic or microservice-based applications. However, server CPUs are optimized for traditional workloads and perform best with long-running applications where the microarchitecture remains warm, significantly boosting performance. Understanding the mismatch between serverless workload characteristics and CPU design is critical for serverless computing to ensure it is an efficient and sustainable cloud execution model. The next chapter aims to fill this gap by comprehensively studying the performance of serverless functions on a modern cloud server.

Chapter 3

Characterizing Serverless Workloads

Serverless has emerged as a new and popular cloud execution model that promises to relieve developers of infrastructure maintenance costs. In the serverless model, applications are implemented as a graph of fine-grained, stateless functions that run on-demand in response to invocations. The stateless and fine-grained nature of functions enables extremely high elasticity for applications. Functions run only when there is demand, which reduces costs for developers and enables cloud providers to achieve high resource utilization of their expensive hardware.

While serverless is highly attractive to both developers and cloud providers, its workload characteristics, with short execution times, small memory footprints, and relatively infrequent invocations, fundamentally diverge from traditional workload characteristics. Since modern hardware is designed and optimized for traditional workloads, this chapter explores the following question:

What are the effects of serverless workload characteristics on the CPU microarchitecture?

We answer this question by performing an extensive characterization of a rich set of serverless functions on a modern commodity server. We find that the distinct workload characteristics of serverless functions have a severe impact on CPU performance compared to traditional workloads. Through a root cause analysis, we discover that the high degree of function interleaving results in the microarchitectural state being obliterated between consecutive invocations of the same function. This cold microarchitecture state has detrimental effects on performance. The CPU front-end is especially affected by an increased number of branch mispredictions and long off-chip instruction

misses, preventing it from efficiently feeding the processor pipeline with instructions. We further evaluate the working sets of serverless functions and show that while on-chip microarchitectural structures can hold the working set of a single function, they are overwhelmed by maintaining the working sets from hundreds of constantly interleaved executing functions. Finally, we present that executing multiple invocations of the same function is highly common, with most accessed instruction blocks and branches being the same across all invocations.

3.1 Methodology

Workloads In our experiments, we use a large set of short-running serverless functions, developed to work with a number of runtimes (namely, Python, NodeJS and Go), as listed in Table 3.1. The functions were adopted from the Hotel Reservation application from the DeathStarBench suite of microservices [51], Google's Online Boutique application [58], AWS' authentication serverless functions [12], and AES encryption application from FunctionBench [91, 92]. Similarly to vHive [164], the state-of-the-art serverless experimentation framework, each function is deployed as a handle of a gRPC [63] server, which represents a function instance. Each function is deployed in a separate container.[1].

Several functions in our workload are written in NodeJS, a language that utilizes just-in-time (JIT) compilation for code optimization. In order to avoid performance noise induced by the JIT engine to ensure stable and reproducible results, we invoke each JIT'ed function 20 000 times before starting measurements [167]. We empirically found that for our functions more invocations do not trigger further code optimization.

Setup: The measurements are performed on a *r650* server node in the CloudLab cluster at Clemson University, South Carolina [161]. The *r650* instances implement a 3rd Gen. Intel Xeon Ice Lake (dual socket 36-core Intel Xeon Platinum 8360Y) running at 2.4 GHz [72]. 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 [5, 160].

The machine runs 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.

[1] All function codes have been released and made available for the research community at `https://github.com/ease-lab/vSwarm`

Function	Abbreviation	Function	Abbreviation
Hotel Reservation [51]		**Online Boutique** [58]	
Geo	Geo-G	Currency	Curr-N
Profile	Prof-G	Email	Email-P
Rate	Rate-G	Payment	Pay-N
Recommendation	RecH-G	ProductCatalog	ProdL-G
User	User-G	Shipping	Ship-G
Other [12, 91, 92]		Recommendation	RecO-P
Authentication	Auth-P/N/G		
Fibonacci	Fib-P/N/G		
AES encryption	AES-P/N/G		

Table 3.1: Serverless functions and their language runtimes (legend – P: Python, N: NodeJS, G: Go).

Before measuring, the function is invoked 20 000 times to warm up the runtimes of function containers.

After warming the function container instance, we collect PMU performance counters using *Linux perf* [87] for both user and kernel space from 500 consecutive invocations. The effect of interleaving with other functions is modeled by using `stress-ng` [32] as a stressor to thrash microarchitectural state of the core running the function container.

3.2 Serverless Functions on a Cloud Server

As discussed in Section 2.1.4, many serverless functions have modest memory footprints and are kept warm for a number of minutes by the cloud provider to reduce the incidence of cold boots. With typical cloud servers today configured with hundreds of gigabytes of main memory [159], a thousand or more instances of warm functions may be resident on a server [9].The warm instances tend to stay memory-resident as providers disable swapping to avoid the associated performance and security issues [160]. Meanwhile, many functions have short execution times of a few milliseconds or less, with invocations that are rare compared to their processing time.

The combination of these trends results in a high degree of function interleaving. Simplistically assuming a server with instances of functions whose invocation process-

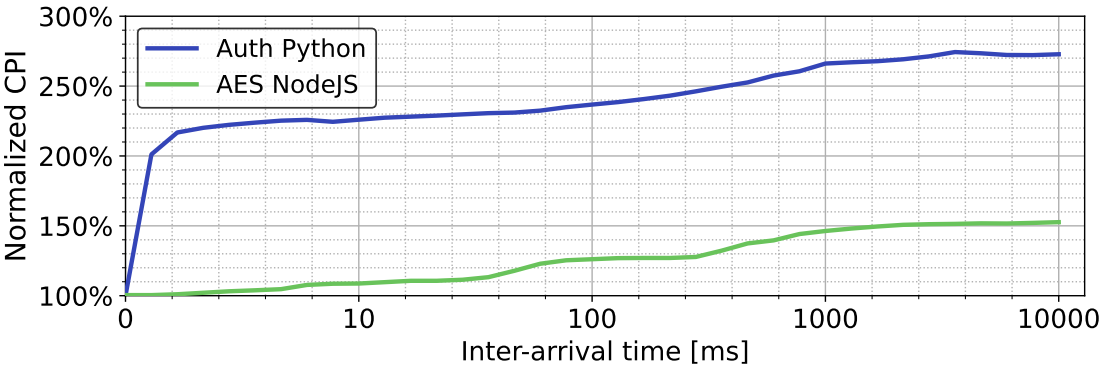


Figure 3.1: Effect of request inter-arrival time on the CPI of a given function on a high-occupancy server. CPI is normalized to back-to-back invocations.

ing time is 1ms with a 1s inter-arrival time (IAT) for each instance, a thousand unrelated invocations will be interleaved between two invocations of the same function. In fact, function execution time may vary and their inter-arrival time distribution is not uniform, but with thousands of function instances kept warm on a host and typical inter-arrival times of seconds to minutes, a huge degree of interleaving is likely.

The problem that stems from such extensive interleaving is that a new invocation to a warm function instance is likely to find its on-chip microarchitectural state largely obliterated. Thus, a function instance that is warm from a runtime's perspective (i.e., has its state fully loaded in memory) faces a cold CPU, requiring the instance to fill all of the microarchitectural structures both in the core and throughout the cache hierarchy – a phenomenon we refer to as a *lukewarm* invocation. Lukewarm invocations pose a particularly acute problem for serverless functions with invocations times in the range of milliseconds, since the short execution times do not offer the opportunity to amortize the latency needed to warm up microarchitectural structures over a long execution period.

To illustrate the problem of lukewarm execution, we start our characterization by studying the performance of our set of serverless functions, running on the evaluated Ice-lake server. Multiple instances of many functions are kept warm on the server. Clients send invocation requests to the various instances maintaining a stable load on the server (around 50% of peak CPU load). In each experiment, one function instance is selected as a function-under-test (FUT). The invocation IAT for the FUT is fixed for the duration of the experiment. For each invocation, we sample a set of performance counters using *perf*. We then repeat the experiment with a different IAT for the FUT. Each experiment with IATs lower than 100ms was run for 3 minutes while the

experiments with 100ms or longer IATs – for 10 minutes.

Figure 3.1 represents the cycles per instruction (CPI) for two representative FUTs: an authentication function written in Python and an AES encryption function written in NodeJS. For this experiment, we choose functions written in different languages to highlight the language-independent nature of the behavior. The figure clearly shows that increasing the invocation IAT tends to increase the CPI. The number of cycles spent per invocation of an authentication function increases by more than 2x and stabilizes at a 270% higher CPI with IAT of over 1 second. With the same IAT, the AES encryption function requires 150% more cycles per instruction compared to back-to-back execution. The reason why the CPI grows as the invocation IAT is increased is that the execution of numerous other instances between two invocations of the FUT thrashes all of the microarchitectural state on the CPU core where the FUT executes and throughout the cache hierarchy. Thus, when a new invocation to a FUT arrives after a long IAT, the FUT experiences a lukewarm execution, with poor performance.

3.3 Top-Down Analysis of Lukewarm Executions

To gain a deeper understanding of the sources of performance loss in lukewarm executions, we study each of the 20 functions in our suite (Table 3.1) in two configurations. In the first configuration, the FUT is invoked repeatedly on the same core on an otherwise idle server – this yields the lowest possible execution time for the studied function as each invocation after the first one enjoys a fully warmed-up microarchitectural state and caches. We refer to this as the *back-to-back* execution. In the second configuration, we model the effect of interleaving with other functions, using `stress-ng` [32] as a stressor that runs between FUT invocations on the same core. The stressor thrashes caches and the core's microarchitectural state and achieves a similar performance degradation to that of combining a high degree of interleaving with high IAT (Section 3.2) in a tractable amount of time and a high degree of reproducibility.

Figure 3.2 plots cycles per instructions (CPI) for back-to-back invocations of the same function instance compared to interleaved invocations. As the figure shows, aggressive interleaving (modeled by the stressor in this experiment) has a detrimental effect on all functions, increasing the CPI (i.e., degrade performance) by 100-294% (162% on average) as compared to back-to-back invocations.

To identify the sources of performance degradation due to interleaving, we use performance counters to break down execution cycles into four categories: retiring,

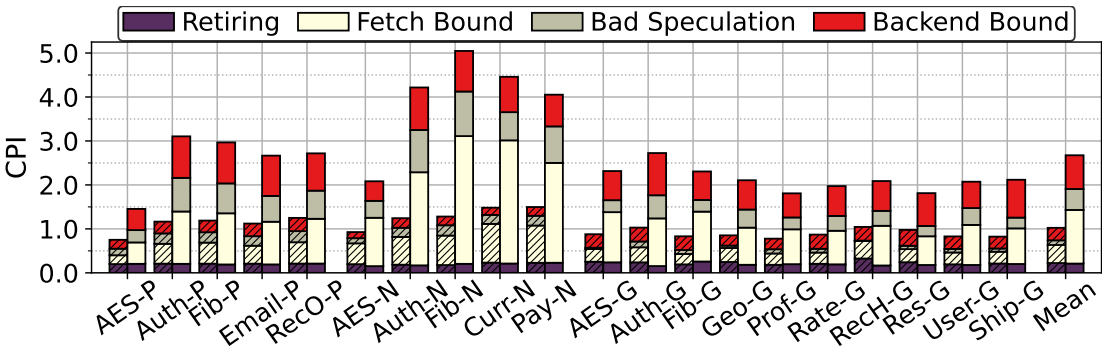


Figure 3.2: CPI analysis of serverless functions for interleaved execution (shaded) vs. back-to-back execution (solid) on an Intel Ice Lake CPU.

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 [170]. 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 [2]. 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.

As Figure 3.2 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 reveal lukewarm invocations of serverless functions result in poor microarchitectural efficiency largely due to the front-end bottleneck.

Next, we focus on the front-end stall cycles to understand the source(s) of the bottleneck. We break up fetch-bound stall cycles into two categories: fetch latency and fetch bandwidth. Note that our categorization differs slightly from Top-Down in that front-end stalls include fetch bound and bad speculation. Figure 3.3 isolates the portion of the CPI due to front-end stalls from Figure 3.2 and breaks down the stall cycles into bad speculation, fetch latency, and fetch bandwidth, putting the front-end performance

[2] In Top-Down, a stall cycle is defined as a CPU cycle in which the pipeline cannot make progress because at least one architectural component is fully utilized and cannot take additional work. However, we note that in an out-of-order architecture, other components can often make progress in the shadow of a pipeline stall. Furthermore, stalls can overlap with each other as well as with retiring [170].

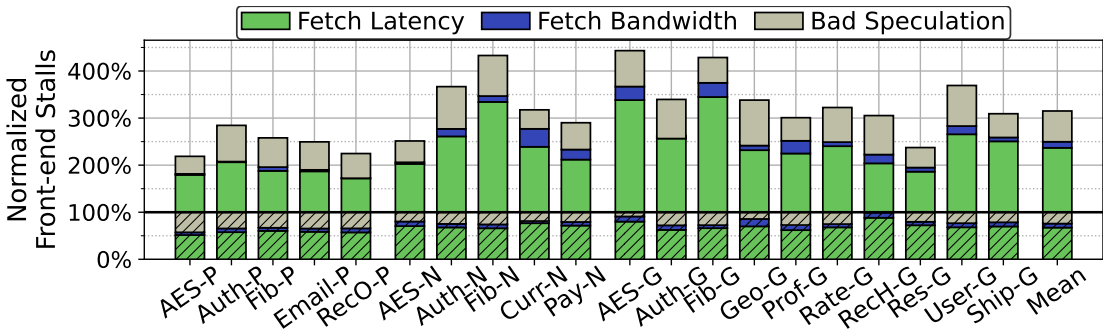


Figure 3.3: CPI analysis of the front-end stall cycles. The striped portions show the back-to-back execution, solid portions show the additional cycles due to interleaving. Normalized to the front-end portion of the CPI for the back-to-back execution in Figure 3.2.

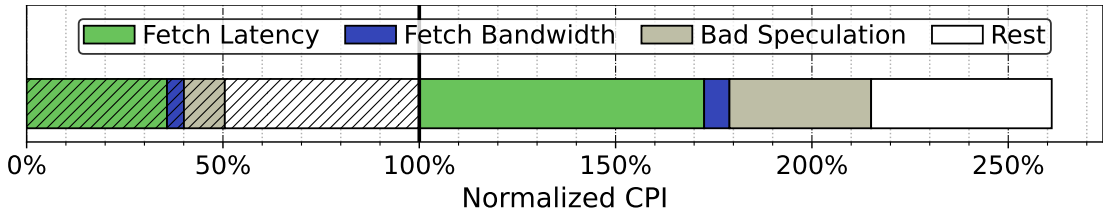


Figure 3.4: Average CPI in the interleaved setup normalized to the average CPI in the back-to-back execution. The striped part represents CPI in the back-to-back execution, the solid part — the extra CPI observed in the interleaved setup. *Rest* includes all other cycle categories except the front-end stall.

problem into focus. The graph shows that the vast majority of front-end stall cycles in both back-to-back and interleaved executions are due to fetch latency. On average, fetch bandwidth is responsible for only 5.7% of the additional stall cycles caused by interleaving, while bad speculation and fetch latency cause 30.2% and 64.1% of the additional stalls, respectively.

Figure 3.4 focuses on the front-end related stalls portion of the *Mean* bar for the interleaved execution from Figure 3.2. To identify the source of the extra stall cycles in the interleaved setup, the CPI stack in Figure 3.4 is normalized to the back-to-back execution. Our key observation is that with function interleaving, most of the extra stall cycles occur due to front-end inefficiencies. More specifically, the figures clearly point out fetch latency and bad speculation as key performance bottlenecks in the execution of serverless functions, responsible for, on average, 45% and 22% of all extra stall cycles in the interleaved setup, respectively.

In the remaining chapter, we drill down further into the largest source of stalls

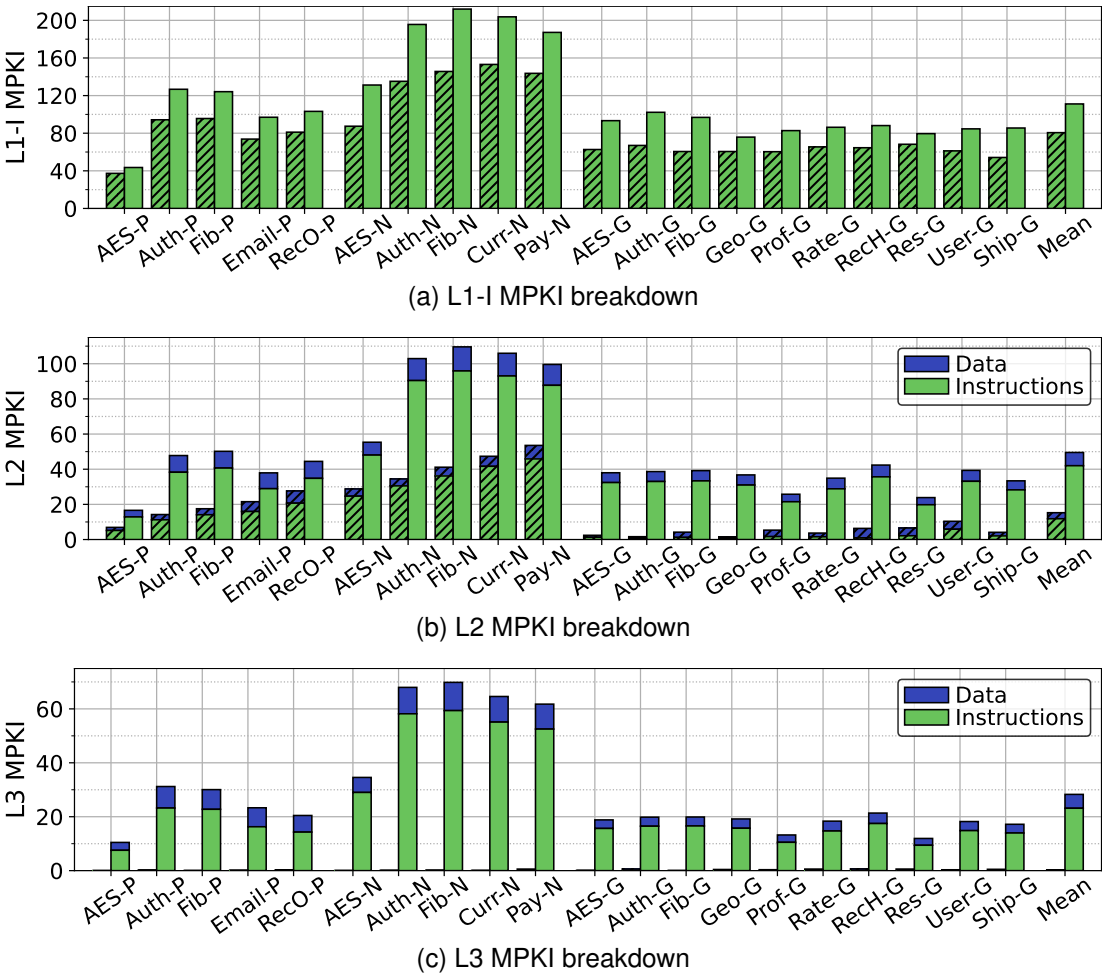


Figure 3.5: MPKI breakdowns of instruction cache and level two and three cache. The striped bars (on the left in each pair of bars): back-to-back execution, solid bars: interleaved execution.

(fetch latency) to understand its root causes. However, in chapter 5, we revisit front-end stalls and present a holistic analysis of the front-end bottlenecks.

3.4 The Story of Cache Misses

To identify what causes the additional fetch latency stalls, we examine instruction misses throughout the cache hierarchy and compare them to data misses. Figure 3.5 presents the for all three cache levels the MPKI breakdown. For brevity, the results for the L1 data cache are omitted. Noting that L1-I misses are consistently high in both back-to-back and interleaved executions, 80 and 111 MPKI respectively, we focus our study on L2 and L3 caches. Figure 3.5b shows the L2 MPKI for both instruction and

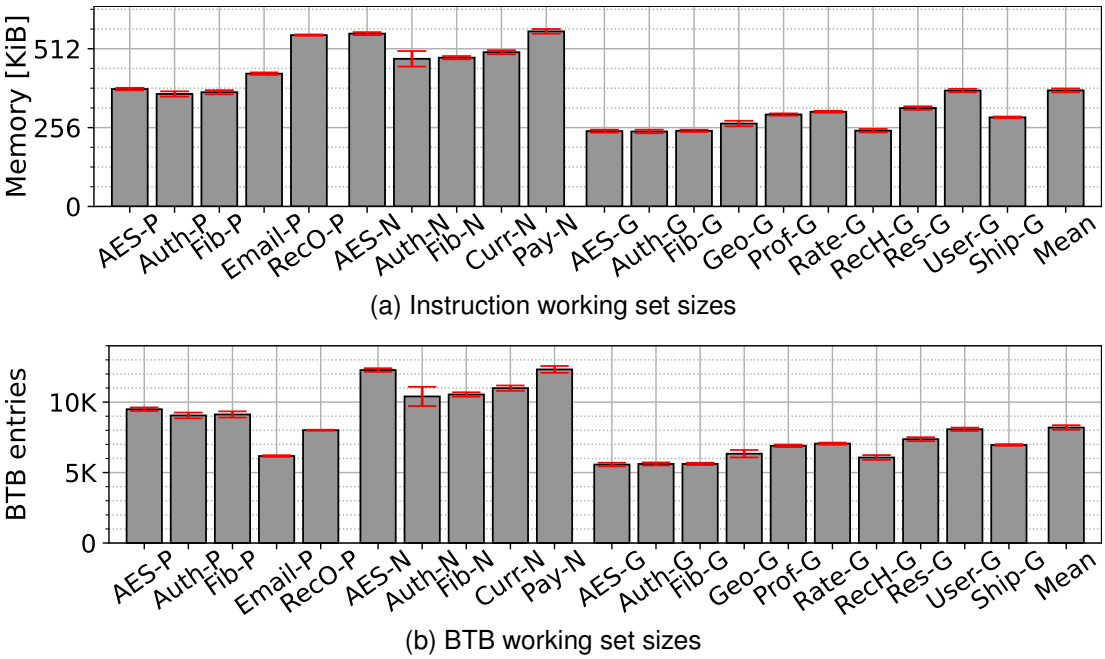
data references. We make several observations. First, we note that misses for instructions are more frequent than misses for data, which suggests that the instruction working set is larger than the data working set. Given that the in-order front-end can not overlap processing of instruction cache misses while the out-of-order back-end often can hide some of the latency of data cache misses, it is not surprising that the front-end is a more significant contributor to total stall cycles than the back-end (Figure 3.2). Second, the miss rates for back-to-back execution are with on average 15 MPKI, significantly lower than for the interleaved execution (avg. 49 MPKI). It shows that without other functions thrashing microarchitectural state, the Skylake core's 1.25MiB L2 cache can, at least partially, cache most of the combined instruction and data working sets. Meanwhile, in the interleaved setup, L2 miss rates are expected to be high due to the cold cache in the wake of interleaving.

We next shift our attention to the LLC (i.e., L3 cache), whose MPKI is shown in Figure 3.5c. The striking trend in the figure is that back-to-back executions have no LLC misses for instructions and very few misses for data, which is explained by the fact the working sets of the studied functions easily fit in the 54MB LLC of the evaluated server and that the back-to-back invocation pattern facilitates LLC residency. Meanwhile, the LLC misses for interleaved executions exceeds on average 20 MPKI, with several functions experiencing MPKIs in excess of 70. The majority of the misses are for instructions, which explains the high fraction of front-end related stall cycles in in the interleaved setup: each L1-I miss to the main memory leaves the core front-end starved of instructions for an extended period of time.

3.5 Severless Working Set Characteristics

Having identified long-latency misses for instructions as a key performance bottleneck in executing serverless functions, we examine the working sets of individual invocations of our suite to better understand why serverless workloads put such high pressure on the CPU front-end. For this analysis, we use the gem5 simulator [26, 103] to run the same set of serverless functions as in the hardware experiments (Section 3.3). We developed and open-sourced the vSwarm-μ framework [97], which enables us to run the entire software stack (including containerized function, 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 4.2.

Starting from a checkpoint, we trace the execution of 25 consecutive invocations



(a) Instruction working set sizes

(b) BTB working set sizes

Figure 3.6: Working sets of serverless functions for CPU front-end structures.

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 [3, 4]. Therefore, the BTB working set, which contains branches that are taken at least once, represents the branch working set.

3.5.1 Working Set Size

Figure 3.6 presents the average instruction working set (a) and branch working set (b) accumulated during one function invocation. Error bars indicate the range of recorded values for each function. The graphs show that despite their short execution times, serverless functions execute large amounts of code and a large number of unique branches relative to the 32 KiB instruction cache and 5K entry BTB found in Intel's Ice Lake server [72]. A single function invocation touches between 240–570 KiB of code memory and accumulates branch working sets ranging from 5.5K BTB entries (AES-G) to almost 13K BTB entries (Pay-N), with notably low variance for the vast majority of functions.

Our findings show that the front-end microarchitectural state required by a single serverless function may fit within existing CPU front-end structures. However, with thousands of functions interleaving their executions on a single server, a CPU is unable

(a) Instruction working set similarity

(b) BTB working set similarity

Figure 3.7: Working set similarities as distribution of Jaccard indices calculated from the 25 function invocations. The Jaccard index ranges from 0 (nothing in common) to 1 (identical), a higher value suggests that record and replay to be more likely to prefetch that function.

to retain the front-end microarchitectural state across invocations, which explains the observed front-end bottleneck under interleaved execution.

3.5.2 Working Set Commonality

Lastly, we study the *commonality* in the instruction and branch working sets across invocations. For this study, we compare the working sets (in terms of cache block or branch addresses, respectively) of each invocation with that of each other 24 invocations for a total of 253 pair comparisons. For each pair of invocations, we compute *commonality* as the Jaccard index [75], which is defined as the ratio between the intersection and the union of unique cache block or branch addresses that belong to working set of the pair of invocations, respectively.

Figure 3.7 presents the results of the commonality study for the instruction and BTB working sets. For all but one function (Rate-G), the mean instruction working set commonality among the 253 compared pairs of invocations exceeds 95%. Their commonality distributions range above 85%, with 3/4 of them not even falling below

90%. Only two functions show outliers in their 253 compared pairs, with their working sets having a commonality of less than 85%. Similar trends can be observed for the branch working sets; on average, 97% of the accessed branches are the same across invocations.

The results reveal that executing multiple invocations of the same function is highly common, resulting in almost exactly the same branches and cache blocks being accessed in every invocation. This high commonality can be explained by the fact that serverless functions execute a substantial amount of code to process the incoming request, traverse the communication fabric (in our case, gRPC), and execute kernel and runtime functionalities (Python, Node.js, or Go) before reaching the actual function code. This functionality is identical across invocations and mostly independent of the function input data.

3.6 Conclusion

In this chapter, we characterized serverless functions and demonstrate that their unique workload characteristics pose a particular challenge for modern CPUs. Many serverless functions have small memory footprints, short execution times and comparatively long IATs. Thousands of such function instances may run simultaneously on a cloud server, resulting in a very high degree of interleaving between two invocations of the same function. The interleaving obliterates on-chip microarchitectural state of the functions, resulting in a *lukewarm* execution with cold caches and a cold core. Lukewarm executions carry an average performance penalty of 162% compared to an execution with a fully warmed microarchitectural state. The single biggest source of performance overhead in a lukewarm execution is the core front-end, particularly fetch latency, which constitutes 64.1% of all additional stall cycles, on average.

Frequent L2 and LLC misses for instructions are a key contributor to the high fetch latency. The high on-chip miss rates for instructions in interleaved invocations of serverless functions can be explained by the large instruction footprints of individual invocations, commonly in the range of 240 KiB to over 550 KiB. At the same time, there exits high commonality in the instruction footprints of different invocations of the same function.

The high commonality reveals opportunities to address lukewarm execution, as it suggests that preserving the microarchitectural state across invocations can be an effective solution. The following two chapters present techniques that exploit this inside to

overcome performance degradation due to lukewarm execution. In chapter 4, we focus on overcoming the largest pain point, off-chip instruction misses, and in chapter 5, we present a holistic approach to warm up all front-end structures comprehensively.

Chapter 4

Jukebox: Preserving Microarchitectural State

Our characterization in chapter 3 demonstrates that the distinct workload characteristics of serverless functions result in severe performance degradation exceeding 2x. Aggressive function interleaving, caused by the short execution time of serverless functions combined with long Inter-Arrival Times (IATs), leads to an unprecedented frequency of context switches. The microarchitectural state, critical for achieving high efficiency, is obliterated between consecutive invocations of the same function, which we call lukewarm execution.

The working set study in Section 3.5 reveals a key insight: high commonality in executing serverless functions across invocations. This commonality suggests an opportunity – preserving the microarchitectural state (e.g., from caches, branch predictors) across invocations could effectively eliminate the performance bottlenecks. However, the same study also shows that while on-chip microarchitectural structures can hold the working set of a single function, they are orders of magnitude too small to accommodate the combined working sets of hundreds or thousands of interleaved executing functions[1]. Therefore, scaling up microarchitectural structures or attempting to store working sets on-chip would incur prohibitive costs in expensive chip area, which may not benefit other workloads. Thus, to overcome lukewarm execution, a lightweight mechanism is needed to preserve microarchitectural state across function invocations without requiring invasive hardware modifications or excessive on-chip area costs.

The Top-Down analysis in Section 3.3 reveals that the dominant source of perfor-

[1]While modern CPUs feature massive last-level caches, the size per core has remained relatively constant as core count has increased significantly [72]. More cores allow running more functions on the same machine, offsetting the increased LLC capacity.

mance degradation in serverless workloads is increased fetch latency, solely responsible for 45% of the additional stall cycles (Figure 3.4). This increase in fetch latency can be attributed to the absence of instructions on-chip, causing a high number of costly LLC misses, each stalling the front end for hundreds of cycles until memory responds.

In response, we present in the remaining chapter *Jukebox*, a technique to overcome long off-chip instruction misses in serverless workloads. The key features of Jukebox are its simplicity and effectiveness, eliminating 93.3% of off-chip instruction misses with low metadata cost and design complexity.

4.1 Jukebox Overview

Jukebox is an instruction prefetcher specifically designed to accelerate *lukewarm* executions of serverless functions. Based on the insights from Section 3.5.2 Jukebox exploits the insight of high commonality of instruction blocks across invocations by recording the working set of one invocation and replaying it whenever a new invocation to the same instance arrives.

Compared to state-of-the-art instruction prefetchers [15, 46, 85, 96] which target the L1-I, a unique feature of Jukebox is that it prefetches into the L2. This choice is motivated by two observations. First, the instruction footprints of individual invocations of the containerized functions generally stay within 600KB, a value much higher than a typical L1-I capacity of 32–64KB. However, such instruction footprints fit comfortably within the L2 capacities of today's server processors, including Intel Skylake and later [2, 14, 72], Amazon Graviton 2 and 3 [110, 158], and the AMD Zen 4 [121], all of which have L2 caches of 1MB. Secondly, prefetching into the large L2 significantly simplifies the prefetcher's design. With a small cache as a prefetch target, it is essential that prefetches arrive just in time to avoid being evicted (if they arrive too early) or not being useful (if late). With L2 as the prefetch target, an aggressive prefetcher can simply fill it at the start of execution and expect instructions to not be evicted in the duration of a short-running function. Prefetching into the L2 does sacrifice some performance compared to prefetching into the L1-I; however, since the latency of an L2 hit is approximately 10 cycles, while an LLC hit is typically over 30 cycles and an LLC miss is hundreds of cycles, the bulk of the opportunity in reducing stalls in a serverless environment comes from avoiding L2 misses. While Jukebox replays prefetches into the L2, its record logic sits at the L1-I, which enables recording of *virtual* addresses of instruction cache misses. Operating on virtual addresses is essential for the prefetcher

to work well with the virtual memory subsystem and not be impacted by, for instance, page migrations due to memory compaction [81].

We next discuss details of Jukebox, whose operation consists of two distinct phases: *record* and *replay*. The record phase begins as soon as the container running the function has been launched. The replay phase is triggered when the OS resumes the process of a function that has been suspended waiting for new invocations. Both record and replay phases are initiated by the OS initializing a pair of dedicated registers with a pointer to the memory region for Jukebox's metadata, similar to how the OS initializes the CR3 register holding the pointer to the root of the page table of a process.

4.1.1 Record

Jukebox records the stream of L2 misses for instructions using a spatio-temporal encoding that provides for a compact metadata footprint and facilitates timely prefetching. The main component tracking addresses to record is the *Code Region Reference Buffer (CRRB)*, a small fully-associative FIFO structure that is accessed using the virtual address of a code region. Each CRRB entry contains a pointer to a memory region *(region pointer)* and an *access vector* holding one bit per cache line within that region. The least significant bits of the pointer used to address individual bytes within the region are not included in a CRRB entry. The *region pointer* can address fixed-sized regions with size chosen at design time. We study a Jukebox configuration with a CRRB entry comprising of a 38-bit *region pointer* and a 16-bit *access vector* (assuming 48-bit virtual addresses and 64B cache lines), for a total of 54 bits per entry (see Section 4.3.1 for an analysis of preferred code region size).

Recording logic is outlined in Figure 4.1. Upon an L1-I miss, the request is forwarded to the L2 as usual. On an L2 hit, the Jukebox record mechanism takes no action, effectively filtering all L2 hits. If there was a miss in the L2, it is recorded by Jukebox when the miss finally returns to the L1-I. This is done by generating a lookup into the CRRB, where the virtual address of the corresponding code region is checked against the existing entries ①. The code region virtual address is generated by taking the most significant bits of the missed block's virtual address corresponding to a CRRB pointer (38 bits for the studied Jukebox design). If a matching entry is found, the prefetcher sets the n^{th} bit in the access vector of the found entry where n is the offset of the cache line within the code region. Otherwise, the oldest entry in the CRRB

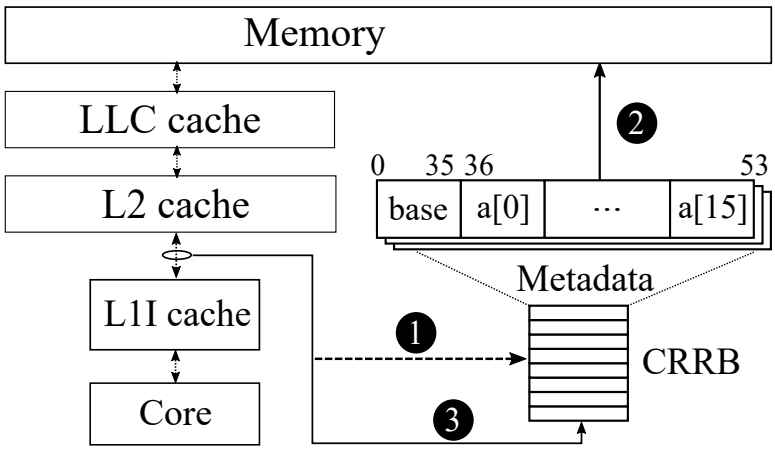


Figure 4.1: Jukebox record

is evicted ②, and a new entry is allocated with the n^{th} bit set ③. The evicted entry is written to memory, optionally bypassing the cache hierarchy since on-chip reuse of the metadata is not expected.

An entry evicted from the CRRB cannot be modified; thus, if an L2 instruction miss occurs to a region whose corresponding entry has already been pushed to memory, a new entry for the same region is created in the CRRB. As a result, a given code region might appear multiple times in the trace recorded by Jukebox. The effect of this design choice is that it increases the metadata footprint of Jukebox but simplifies the design, since evicted entries do not need to be brought back from memory.

There are two possible options for determining whether an L1-I miss also missed in the L2. The first option is to propagate the result of the L2 tag check back to the L1-I using a dedicated 1-bit signal. The second option is to measure the delay of an outstanding L1-I request and compare that to the expected L2 hit latency (e.g., using a timer at the L1-I MSHR [84].) Jukebox can work equally with either of these options.

The FIFO order of the entries in the recorded metadata directly encodes the temporal order of accesses at the chosen granularity. That is, the first metadata entry written to memory will encode the addresses of the cache lines in the first code region accessed after the function was invoked. This organization allows Jukebox to prefetch the entries in approximately the same order they are likely to be accessed, thus improving timeliness at replay time.

Note that because the recording is done in a region-based manner, the replay stream will first prefetch all of the indicated cache blocks from one code region before moving on to the next region. As a result, some reordering of individual cache blocks will happen at prefetch time compared to recording. However, recording at the region

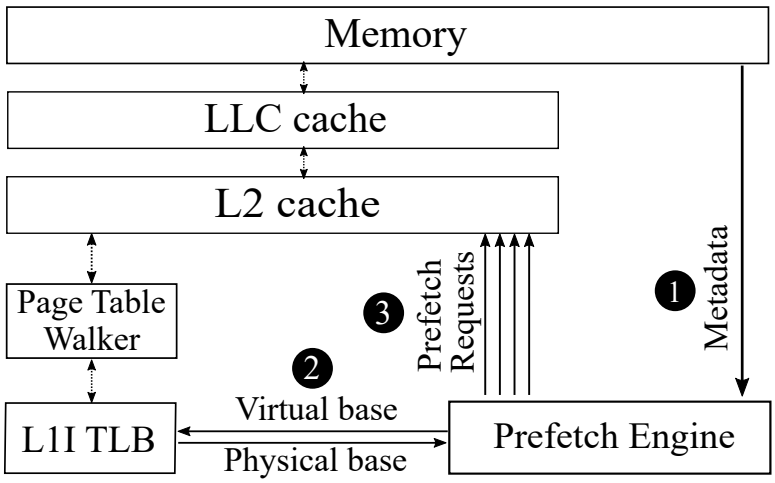


Figure 4.2: Jukebox replay

granularity enables a small metadata footprint, makes better use of address translation resources (a single lookup for all the blocks in a region), and reduces the number of DRAM activations due to spatial locality within a page.

The record phase is triggered by the OS by programming a pair of architecturally-exposed registers containing the base and limit of the metadata storage. The limit register is optional and lets the OS control the amount of metadata stored per function instance process.

4.1.2 Replay

The replay phase in Jukebox is triggered by the OS upon receiving a new function invocation. The OS triggers the replay by programming a pair of base and limit registers, similarly to how recording is triggered. The replay can be triggered by the OS's scheduler whenever the function instance thread is assigned to a core to process an invocation.

The replay phase is outlined in Figure 4.2. The prefetch engine starts the replay phase by issuing sequential reads starting from the beginning of the metadata region ①, reading the metadata in the same order it was written to memory. This enables prefetching of instruction cache blocks in the same temporal order as was recorded, albeit at page granularity.

The metadata entries are prefetched into a small FIFO inside the prefetch logic. Once the metadata entry is returned from memory, the prefetcher passes the base address of the code region to the I-TLB ②, triggering address translation like a normal code request. This serves two purposes: first, it ensures that Jukebox does not rely

on physical addresses that may change as a consequence of normal OS activity such as paging or memory compaction. Second, it effectively pre-populates the TLB with translations for code pages. As soon as the physical base address of the code region is known, using the information from the entry's access vector, the prefetch engine reconstructs full addresses of each of the accessed cache lines within the code region, and enqueues them in the L2 prefetch queue ③.

Once prefetch requests for all of the cache blocks encoded in an entry's access vector have been launched, the entry is retired from the FIFO. The next set of entries is fetched using a single 64B cache line read once the equivalent of 64B of data have been consumed from the FIFO.

4.1.3 Discussion

4.1.3.1 Metadata Memory Management

Upon a function instance's start (i.e., first invocation received by the host), the OS allocates two memory regions for the function instance process, each of which is contiguous in the physical space. These regions are used for bookkeeping of the Jukebox metadata of the function instance process. The OS associates the physical addresses of the two buffers with the PID of the function instance process. For example, in Linux, the addresses of the buffers can be stored in `task_struct`. Upon an invocation of a function, as part of assigning the function instance process to a core for execution, the scheduler consults the instance's `task_struct` and writes addresses of the buffers to the registers that define where the Jukebox metadata is written (at record) and where the metadata is fetched from (at replay). Once the invocation completes and the function instance process is descheduled, the values of buffer pointers are saved in the `task_struct`. A subsequent invocation received by the same instance thus replays the metadata from the memory region written by the previous invocation. Operating with the metadata buffers using physical addresses avoids the need for address translation while fetching/recording metadata, which (1) improves the timeliness of Jukebox prefetches and (2) does not cause contention for TLBs and hardware page walkers.

4.1.3.2 Hardware Overhead

Jukebox relies heavily on existing microarchitectural mechanisms for instruction prefetching, resulting in negligible hardware overhead. Jukebox requires a set of architecturally exposed registers to encode the boundaries of buffers for recording

and replaying metadata (4x48-bit registers for the start address and size of the record and replay trace). Additionally, it requires logic for the CRRB that supports fully associative search and FIFO replacement policy. Since short functions do not exhibit distant code reuse, a buffer of just 16 entries allows for high metadata compression (16 entries x 54 bits = 864 bits).

For creating and reading the trace, the record and replay logic comprises two 32-bit counters to stop upon reaching the trace limit. Finally, since the metadata is written and read in chunks of a whole cache line, two 512-bit buffers are needed to cache one cache line of metadata for the record and replay logic. The total hardware overhead of Jukebox is 2144 bits plus simple control logic, which corresponds to 0.81% of the 32KiB Ice Lake L1-I cache or approximately 0.0021% of the chip area of one Ice lake core[2].

4.1.3.3 Virtualization

Under virtualization, the guest OS triggers Jukebox record and replay. Jukebox metadata is stored in guest physical memory, i.e., as a part of the virtual machine state. Hence, in addition to lukewarm execution, Jukebox can accelerate the lengthy cold boots of serverless instances provided that a function snapshotting technique [41, 152, 164] is used and that the Jukebox metadata has been recorded before taking the snapshot.

4.1.3.4 Enabling Jukebox

Jukebox can be enabled for a particular thread upon its creation, similar to choosing a thread's scheduling priority by setting the corresponding attribute of the created thread [117]. For example, when the serverless runtime of a function instance starts a gRPC/HTTP server with a number of worker threads (or when spawning them at run time), it can enable Jukebox by setting the corresponding attribute before making a `pthread_create` [101] system call.

4.1.3.5 Generality

While Jukebox is particularly beneficial for lukewarm functions, it can accelerate start-up times for any memory-resident thread.

[2]Approximated base on annotated die shot photos of the Ice lake core `https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_(client)`

Core	
Architecture:	Ice lake-like, ISA: x86-64, dual-core, Freq.: 2.6 GHz
Fetch BW	16 bytes / cycle
BP Unit	BP: 64 KiB L-TAGE [145] + 5 KiB Bimodal 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[3], private, LRU, 10 MSHRs
L1-D Cache	48 KiB, 64 B line, 12-way, 4 cycles, private, LRU, 10 MSHRs
L2 Cache	1.25 MiB, 20-way, 13 cycles, private, LRU, 32 MSHRs
LLC	8 MiB, 16-way, 50 cycles, shared, non-inclusive, 32 MSHRs
Memory	DDR4 2400 MHz, 14 ns RCD, 14 ns RP, 14 ns CL
Jukebox	CRRB: 16 entries, Region size: 1KiB, 48KiB metadata size (24KiB record + 24KiB replay)

Table 4.1: Parameters of the simulated processor.

4.2 Methodology

4.2.1 Simulation Infrastructure

To evaluate Jukebox, we use gem5 v22.0.0.1 [26, 53, 103], a cycle-approximate full-system simulator configured to model the Intel Xeon Ice-Lake CPU used in the hardware studies in chapter 3. Considering industry trends toward much larger branch target buffers (BTBs) than in the recent past, we enlarge the BTB from 5 K entries in Ice Lake to 12 K entries as found in the latest Intel Xeon Sapphire Rapids CPU [14]. We find that overall trends and conclusions are not affected by this choice. Table 4.1 summarizes the modeled CPU parameters.

We run an identical software stack in simulation as we do on real hardware, i.e., the same OS and the same containers running gRPC servers. Before performing the measurements, we boot the system in functional mode (KVM core) and execute 20 000 invocations of each function, at which point we create a checkpoint of the system state.

[3]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 [62].

These checkpoints form the common starting state for all subsequent experiments. For the experiments, we switch to the detailed Out-of-Order core model and simulate 25 invocations in a cycle-approximate timing mode. To avoid interference from the client driving the functions, we create a two-machine simulation setup. The first machine runs the test client, which sends requests via gem5's Ethernet model to the second machine running the function. To simulate the effect of interleaving, we flush the microarchitectural structures of the simulated Ice Lake CPU and overwrite the branch predictor with a random state between two invocations[4].

4.2.2 Workloads

We evaluate Jukebox on the 20 serverless functions used in chapter 3 for our hardware characterizations and are listed in Table 3.1. We run the same software stack and function images (Ubuntu 20.04 with Linux kernel v5.4 and Docker v20.10 as the container host) in full end-to-end full-system simulations as used on the real hardware.

4.3 Evaluation

4.3.1 Parameterizing Jukebox

Recall from Section 4.1.1 that Jukebox uses a spatio-temporal encoding that exploits locality in code accesses by organizing its metadata as a sequence of entries, each corresponding to a spatial region. Each entry contains the upper bits of the address of the region and a bit vector, with one bit per cache line. Larger regions require a longer bit vector but may allow for fewer entries given sufficiently high spatial locality in the code. The entries are created in the CRRB, which coalesces accesses to the same region before the entry is written to the in-memory metadata storage. A larger CRRB may allow more accesses to the same region to be coalesced before an entry is evicted, resulting in a smaller metadata footprint at the cost of more on-chip storage and higher access energy.

To find the preferred code region size and CRRB size, we measure the size of the metadata required to store *all* of the entries produced by the Jukebox recording logic while a function executes. We do this for a range of code region sizes, from 128B to 8KiB, and three different CRRB buffer sizes: 8, 16 and 32 entries.

[4]Configurations and guidance on how to set up gem5 to run containerized serverless workloads are made available at `https://github.com/ease-lab/vSwarm-u`

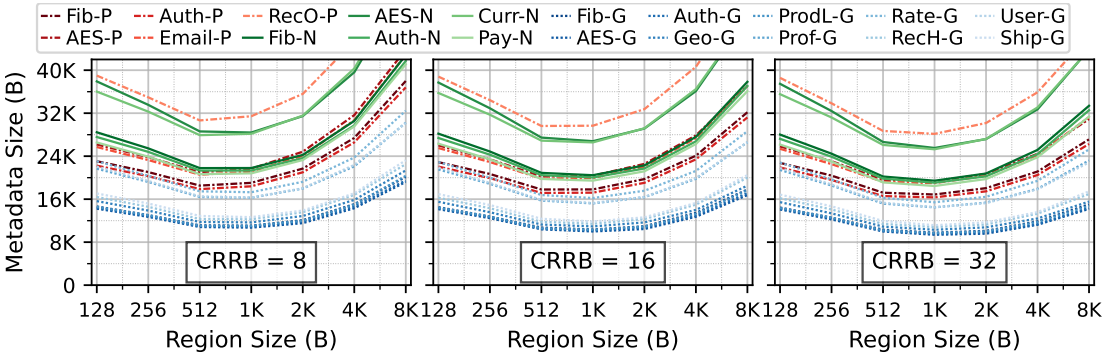


Figure 4.3: Sensitivity of Jukebox's metadata size to the code region size.

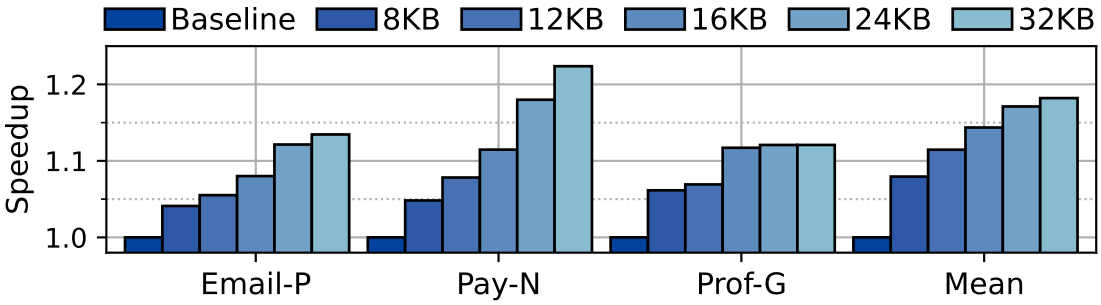


Figure 4.4: Speedup with Jukebox as a function of the size of its metadata storage compared to the baseline without Jukebox.

Figure 4.3 presents the results of the study. For the majority of the workloads, the metadata size reaches a minimum with the code region size of 1KiB — irrespective of the CRRB size — resulting in 10.8KiB to 31.5KiB metadata storage for an 8-entry CRRB. The two other CRRB sizes reveal very similar trends and modest sensitivity to the size of the CRRB. With a 16-entry CRRB, the metadata size is reduced by 6.4%, and for a 32-entry CRRB, by another 5.4%.

We next study the impact of limiting the size of Jukebox's metadata storage on its efficiency. Since we found in Figure 4.3 the most space-efficient code region size to be 1KB, we use this configuration and a 16-entry CRRB for this sensitivity study. Figure 4.4 shows the speedup Jukebox is achieved when constrained to various metadata storage capacities. In Figure 4.4, we plot only one representative function for each of the three implementation languages [5] together with the average across all 20 functions in our evaluation suite.

The figure shows that workloads with large working sets, e.g. Pay-N, tend to be more sensitive to the limited metadata size than workloads with small working sets, e.g.

[5] We found that the language in which the function is written is the single biggest determinant of a given function's runtime and Jukebox's efficacy.

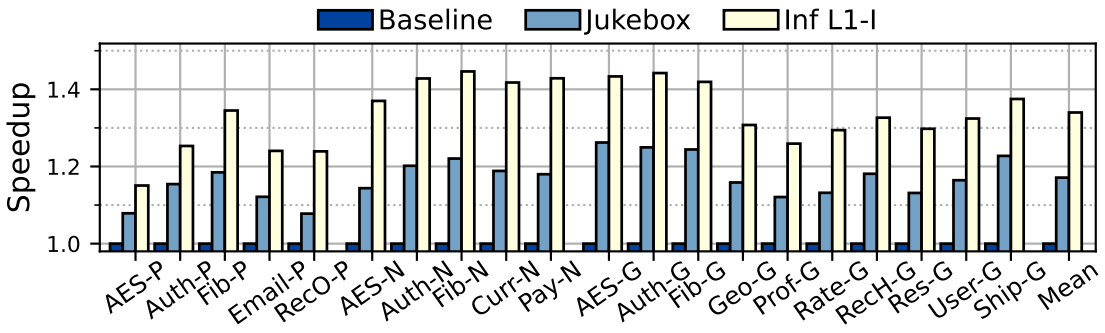


Figure 4.5: Performance results on the Skylake-like configuration.

ProdL-N. This is expected given that the metadata represents a compressed form of a function's working set. Note that Jukebox is designed to seamlessly extend to dynamic metadata sizes. For that a *metadata size* field needs to be added in the bookkeeping mechanism described in Section 4.1.3.1. When scheduling a thread, the OS sets up the size of the metadata of a function instance and assigns the addresses for record and replay metadata storage. We use the same size of metadata storage for all our workloads in further experiments.

As Figure 4.4 shows, on average, there is a little gain with increasing metadata storage beyond 24KiB. Thus, unless stated otherwise, we use a Jukebox configuration with 24KiB metadata storage, 1KiB code-region size, and 16-entry CRRB in the rest of the evaluation.

4.3.2 Performance

Figure 4.5 presents the main result of the evaluation. We compare three configurations: (1) the baseline, which represents a high degree of function interleaving; (2) Jukebox applied to the baseline setup; and (3) a perfect I-cache (Inf L1-I), which draws the maximum opportunity without any instruction misses. The baseline (1) is modelled by flushing all microarchitectural states in-between function invocations. For (3), we use an infinite-sized L1-I cache that maintains the complete instruction footprint a function accumulates over all simulated invocations.

The performance of the three evaluated configurations is shown in Figure 4.5 with the results normalized to the baseline. We find the maximum opportunity without any instruction misses boosts the performance of the studied functions by 34% on average (45% max on Auth-G). Jukebox delivers consistent speedups that correlates with the opportunity; that is, functions that have a larger difference in performance

between Perfect I-cache and the baseline enjoy larger speedups (e.g., Auth-G: 29.5% speedup with Jukebox), while the opposite is true for functions with a small difference between Perfect I-cache and the baseline (e.g., AES-P: 6.2% speedup with Jukebox). On average, Jukebox speeds up interleaved executions by 17.8%.

4.3.3 Miss Coverage

We next study Jukebox's ability to cover instruction misses. Since Jukebox prefetches into the L2 cache, we present fractions of L2 instruction misses in the baseline that are (1) covered, (2) not covered, and (3) overpredicted (i.e., prefetched but not referenced) by Jukebox.

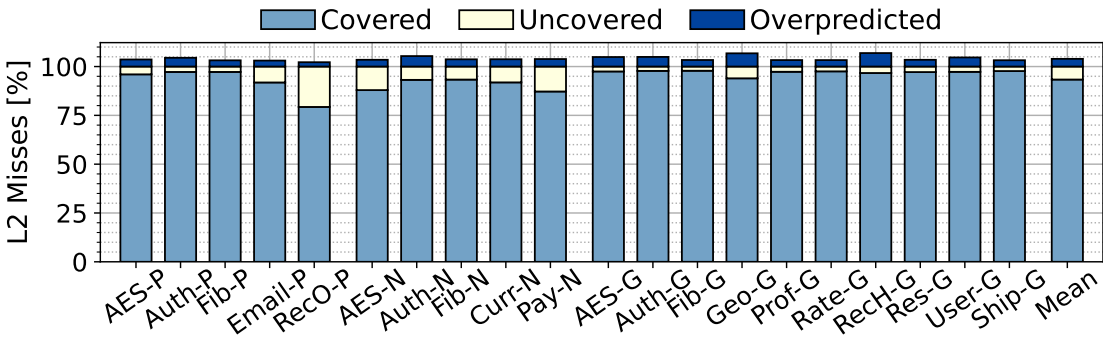
Figure 4.6a shows the result of this study. One can see that coverage correlates with the choice of a programming language; benchmarks that are written in Go show high Jukebox coverage (90–97%) while those written in Python and NodeJS exhibit lower coverage (79–96%). This can be explained by the fact that for the majority of the Go benchmarks, metadata fits into Jukebox's metadata storage, which is not the case for all Python and NodeJS benchmarks (see Figure 4.3). Across all benchmarks Jukebox covers 93.3% of L2 instruction misses on average.

Furthermore, the figure shows that Jukebox induces few wrong prefetches with an overprediction rate of just 5% (max. 7.3%). This result is anticipated by the high commonality in instruction footprints across invocations (Section 3.5.2). The high accuracy of Jukebox's prefetches affirm its record and replay approach to be highly effective in delivering the relevant instruction blocks on chip.

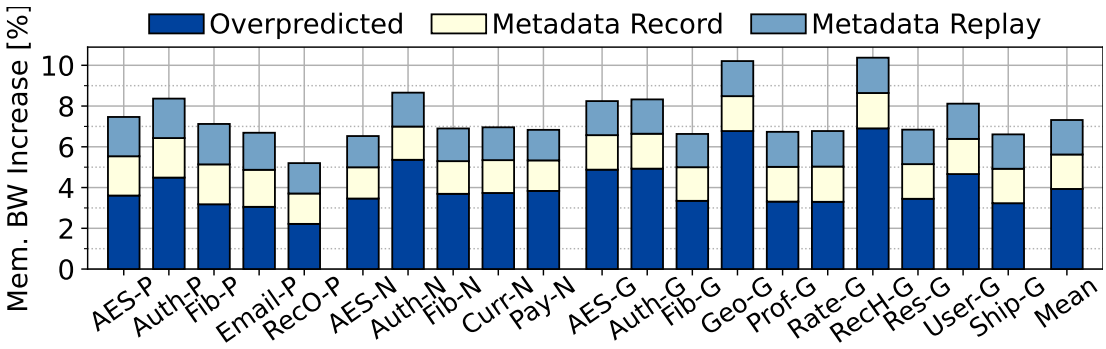
4.3.4 Memory Bandwidth

Figure 4.6b plots memory bandwidth usage of Jukebox normalized to the baseline. Memory bandwidth includes all requests issued to memory, which includes both instruction and data, demand and prefetches. Note that Jukebox does not change the amount of bandwidth consumed for correct timely prefetches. Overheads lie in overpredicted (i.e. unused) prefetches as well as metadata traffic associated with recording and replaying. Jukebox introduces a modest memory bandwidth overhead of 7.3% on average and 10.3% in the worst case. The overhead comprises 45% of Jukebox's metadata and 54% overpredicted traffic.

We observe that the memory bandwidth increase correlates well with the commonality of the instruction working sets across invocations for a certain workload (see



(a) Instruction miss coverage



(b) Bandwidth overhead

Figure 4.6: (a) Fractions of L2 instruction misses covered and overpredicted by Jukebox. (b) Memory bandwidth overhead by Jukebox. Both graphs are normalized to the number of L2 misses in the baseline).

Figure 3.7 in Section 3.5.2. For example, Go workloads tend to lower commonality, resulting in more prefetches being useless, reflected in the higher overpredicted fraction for those workloads. RecO-P has with 99.2% the highest commonality and also shows the smallest fraction of overpredicted prefetches. Outliers with a lower commonality, such as Auth-P, Auth-N, Geo-G, or RecH-G, present the highest number of useless prefetches.

4.3.5 Comparison to a State-of-the-Art Instruction Prefetcher

In this section, we compare Jukebox to a state-of-the-art instruction prefetcher, called PIF [46]. Recall from Section 2.3 PIF is a *temporal streaming*-based prefetcher, which works by recording and replaying the sequences of retired instruction addresses. Recording all instruction addresses allows PIF to be independent of variations in L1-I cache content and application's control flow, thus achieving good prefetch accuracy. To enable replay of instruction streams, PIF requires an *index*, which uses

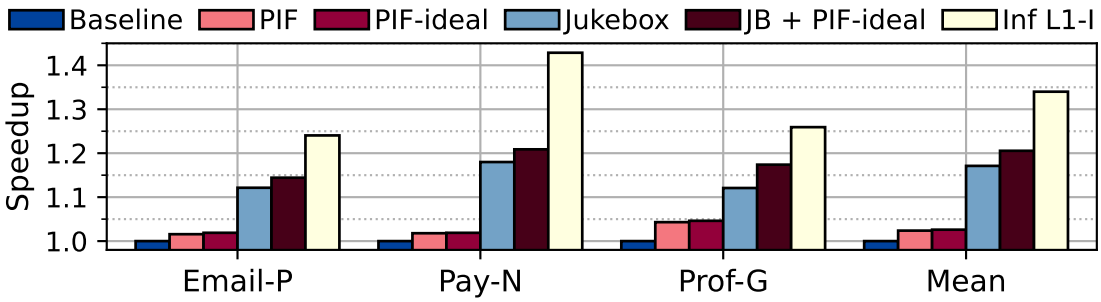


Figure 4.7: Comparison of performance with PIF and Jukebox.

an instruction address to find the most recent recorded stream that starts with that address.

By using the same parameters as in [46], we configure PIF with a 49KB index, 164KB of stream metadata storage, and an unrealistic single-cycle lookup latency for each of these components. Because PIF was designed for long-running traditional server workloads, it does not save its state across function invocations. To understand the best possible performance of PIF, we simulate another design, PIF-ideal, with an unlimited index and unlimited metadata storage that persist across function invocations.

Figure 4.7 plots the results of this study. We find that PIF delivers 2.4% speedup on average (5.6% max) while PIF-ideal boosts performance by 2.6% (5.8% max). Meanwhile, Jukebox with metadata size limited to 24KiB provides a 17.8% speedup, on average – a significant improvement over PIF and PIF-ideal. The reason for PIF's relatively poor efficacy can be explained by the fact that whenever the recorded stream differs from the actual access stream of the core, PIF stops prefetching and re-indexes to find the correct stream. Re-indexing prevents PIF from running far enough ahead of the core to cover the long latency of a main memory access.

The big picture is that PIF was designed to reduce L1-I misses for accesses expected to hit in the L2 or L3 caches. In contrast, lukewarm functions have instruction footprints in main memory, requiring a prefetcher to hide the associated high latency effectively. PIF must stop and re-index any time the actual control flow diverges from the prefetch stream. In contrast, Jukebox prefetches all instruction blocks recorded in its metadata without synchronizing with the core. By doing such bulk prefetching, Jukebox sacrifices the ability to prefetch into the small L1-I but achieves high instruction miss coverage in the L2 and L3.

The last bar shows the combined strengths of both designs — Jukebox's bulk off-chip prefetching and PIF's highly accurate L1 prefetching. Surprisingly, PIF achieves

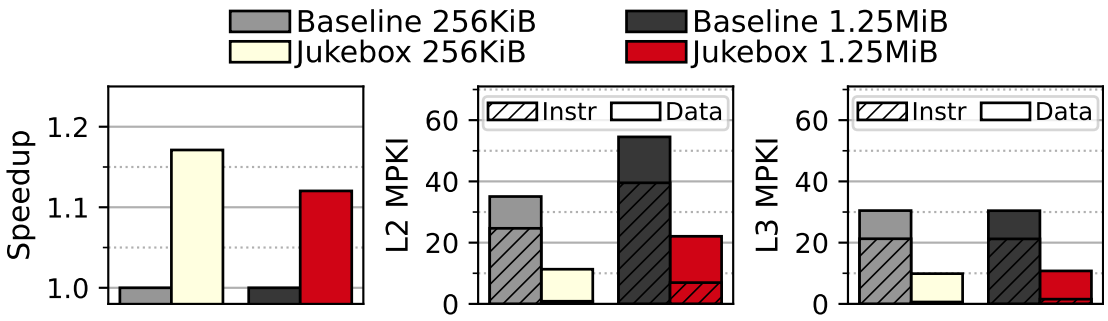


Figure 4.8: Jukebox on different cache sizes.

only a small improvement of 2.9% over Jukebox on average, leaving a large gap to a perfect L1-I. We explore this behavior in the following chapter and find that the cold state of the branch predictor results in a high number of mispredictions, causing frequent resets of PIF which hurts its efficacy.

4.3.6 Jukebox with a Smaller L2 Cache

A key feature of Jukebox is its decoupling from the core, unlocking aggressive off-chip prefetching and drastically reducing the design complexity. In fact, Jukebox's aggressiveness in replaying instructions into the L2 is only limited by the speed at which the trace can be read, and prefetches can be issued. This design point is highly effective for the Ice-Lake core because the instruction working sets of our serverless functions with a size of 240–570 KiB fit comfortably into its 1.25MiB L2 cache. While 1.25MiB is a standard cache size for modern server CPU's [2, 14, 121, 158, 168] Jukebox can be implemented on machines featuring smaller L2 caches where the working sets do not fit entirely. To understand Jukebox's efficiency in such a scenario, we model the same configuration as before, except we reduced the size of the L2 cache to 256KiB, representing the size of an Intel Broadwell CPU.

Figure 4.8 shows for both configurations the performance improvements of Jukebox as well as miss reduction for the L2 and L3 cache. Across our suite of serverless functions, we find that Jukebox delivers an average speedup of 12% on the 256KiB configuration. Noting that the speed-up is smaller than the 17.8% achieved on 1.25MiB L2 cache, we examine the cache miss rates for instructions in the two simulated platforms.

We observe that Jukebox's performance improvement is smaller with a 256KiB L2 — 12% on average — than with a 1.25MiB L2 cache. This difference can be explained by the number of L2 misses shown in the centre graph. While Jukebox can eliminate

almost all instruction misses for the 1.25MiB L2 (96.5% on average), it covers only 82.4% of the instruction misses with a 256KiB L2. Jukebox is too aggressive and evicts its own prefetches before they are consumed. Furthermore, with the smaller L2 cache, instructions compete with data, resulting in a high incidence of conflict misses, as indicated by the higher number of data misses for the 256KiB configuration.

While the graph shows Jukebox is less effective with a small L2, it remains highly effective at eliminating off-chip misses for instructions. These are the crucial misses to cover due to their excessively high latency. As shown in the right graph of Figure 4.8 reduces off-chip misses by 97.1% and 92.7% on average for the 256KiB and 1.25MiB L2 cache configuration, respectively.

To summarize, Jukebox is most effective in CPUs with a large L2, which have featured in recent server processors [2, 121, 158], yet also provides a tangible benefit in CPUs with a smaller L2.

4.4 Related Work

Context switches: Prior works examine the problem of fine-grained context switches in highly-consolidated virtual machines [35, 171]. These works focus on a setting where a single virtual machine occupies the entire CPU for multiple milliseconds, followed by a context switch to another VM. The problem solved in these works is restoring the entire multi-megabyte LLC state via prefetching by saving the address footprint of the LLC to main memory upon a context switch. The proposed designs suffer from large metadata overheads for high coverage and high bandwidth overheads associated with indiscriminate restoration of the entire LLC (in some cases more than doubling the amount of memory traffic compared to the no-prefetch baseline [35, 171]).

In contrast, Jukebox targets on-chip instruction misses by prefetching directly into the L2 cache using a minimal amount of metadata. Due to high instruction commonality across invocations of a given serverless function instance, Jukebox achieves high accuracy with low overprediction. While both Jukebox and prior works save prefetcher metadata in main memory, prior works save physical addresses, which are the only ones available at the LLC; in contrast, Jukebox saves virtual addresses, which makes Jukebox naturally compatible with a modern virtual memory manager that can move pages in memory (e.g., for memory compaction purposes).

Jevdjic et al. [78] record spatial footprints of data pages to reduce off-chip band-

width pressure for aggressive data prefetching. While Jukebox also uses the idea of footprints, it targets instruction prefetching with low metadata cost.

Ahn et al. also examines fine-grained context switches for virtualized systems and proposes a context-preservation technique that controls the LLC capacity available for each virtual machine, to preserve LLC working set across context switches [7]. Zhu et al. examine event-driven server-side applications and identify L1-I misses to be a major performance bottleneck [173]. The authors observe that instruction working sets of the studied applications fit in the LLC and propose a specialized cache replacement policy to preserve an instruction working in the LLC and augment it with a temporal prefetcher in the L1-I cache. Both of these techniques target settings where the instruction working set fits in the LLC, which is not the case for serverless functions with infrequent invocations and a huge degree of interleaving.

Sojkovic et al. [156] show that context switches are a main source of high tail latency in microservices and propose accelerating the context switch with hardware support. However, this work only replaces software instructions to store and restore the register file on a context switch and does not help with the cold microarchitectural state a context switch entails. Jukebox addresses cold microarchitectural state for instruction of serverless function and thus orthogonal to the proposed technique of [156].

Prefetching: There is a long history of works in instruction prefetching for server workloads. These papers fall into one of two categories: temporal streaming and BTB-directed. The former category records entire traces of instruction cache accesses or misses at the cache block granularity, resulting in metadata size of hundreds of kilobytes and requiring a complex indexing mechanism to find the correct metadata when the actual execution diverges from the recorded trace [46, 47, 85]. To store the metadata, existing temporal streaming proposals either use dedicated on-chip storage [46] or virtualize the metadata into the LLC [85]. In contrast, Jukebox uses aggressive filtering and spatial encoding, resulting in low metadata costs, does not require any indexing, prefetches into the L2 to further simplify the prefetcher design, and stores its metadata in memory to support thousands of warm functions.

The second category uses the BTB together with the branch predictor to identify upcoming control flow discontinuities to drive the instruction prefetch engine [96, 128]. This approach relies on a fully warmed-up BTB and branch predictor, which — as we will show in the next chapter — makes it fundamentally at odds with lukewarm executions that have to contend with a cold core.

4.5 Conclusion

The sharply contrasting workload characteristics of serverless functions present new challenges for modern CPUs. The cold microarchitectural state, resulting from short execution times and infrequent invocations, causes a severe front-end bottleneck.

In this chapter, we took the first step to overcome lukewarm execution and addressed long off-chip instruction misses — the dominant source of performance degradation in serverless workloads. We proposed Jukebox, a record-and-replay prefetcher specifically designed to accelerate lukewarm invocations. Jukebox exploits instruction commonality across invocations by recording the working set of a serverless function and storing the record in memory. When consecutive invocations occur, Jukebox uses the records to restore the working set into the L2 cache. The high commonality in the instruction working sets of serverless functions makes Jukebox highly effective in covering long off-chip instruction misses with a low metadata cost and low design complexity.

While Jukebox is simple and effective, it restores working sets only into the L2 cache and leaves other microarchitectural structures in a cold state. As our characterization (chapter 3) showed, lukewarm execution creates various bottlenecks, predominantly in the CPU front-end, including a high number of L1 instruction misses and branch mispredictions. Therefore, in the next chapter, we explore the remaining front-end bottlenecks holistically to provide a comprehensive solution for lukewarm front-ends.

Chapter 5

Warming Up a Cold Front-End with Ignite

Our results from Section 3.3 show the biggest bottlenecks of lukewarm execution are in the core front-end, particularly long off-chip instruction misses adversely hurt performance. While Jukebox — presented in the previous chapter — effectively eliminates off-chip instruction misses by prefetching instruction working sets in the L2 cache, it leaves other structures cold (including L1 instruction cache, BTB, and branch predictor). Therefore, in this chapter, we explore the following question:

Can existing front-end prefetchers capitalize upon Jukebox and warm up the remaining structures to overcome the lukewarm front-end?

To answer this question we evaluate the state-of-the-art front-end prefetcher Boomerang, which proactively fills the L1-I and the BTB [96], and combine it with Jukebox. We find Boomerang provides only a small speedup over Jukebox 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). As a result Boomerang falls considerably short of an ideal front-end that delivers 61% average speed-up over an aggressive next-line prefetcher.

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.

In short, this chapter makes the following contributions:

- We show that combining Jukebox with the state-of-the-art front-end prefetcher Boomerang [96] 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.

5.1 Front-End Prefetching on Lukewarm Invocations

In this section we study the performance of state-of-the-art microarchitectural front-end prefetchers on lukewarm invocations. We evaluate Jukebox, which mitigates off-chip misses for instructions for serverless, and Boomerang [96], a unified FDP instruction and BTB prefetcher. For clarity of exposition, we do not show results for a

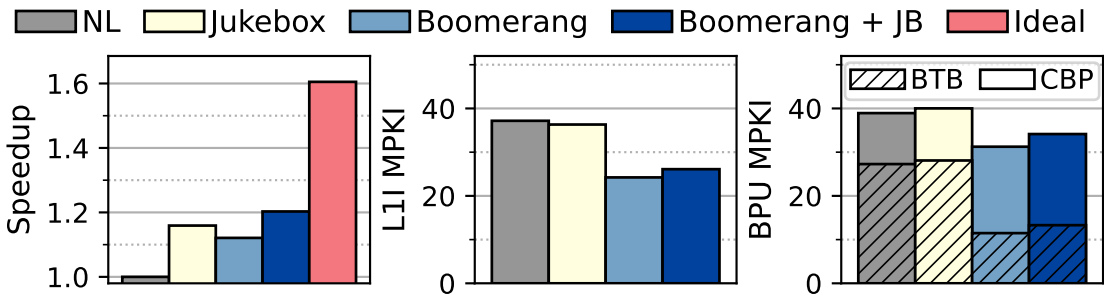


Figure 5.1: Performance, L1-I MPKI and BPU MPKI for various front-end configurations on lukewarm invocations.

temporal streaming prefetcher in this section, but note that our findings fully extend to that class of designs, which is demonstrated in Section 5.4.5. The same gem5 setup and Ice-Lake CPU model used throughout the work (outlined in Section 4.2.1) is used to evaluate the prefetchers in detailed cycle-accurate simulations.

5.1.1 Big Picture Results

Figure 5.1 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 5.1 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 5.1 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 5.1, both rates are high, with the average BPU miss rate exceeding 30 MPKI for both Boomerang and Boomerang+JB[1]. 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.

5.1.2 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 5.1).

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 avoid-

[1]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.

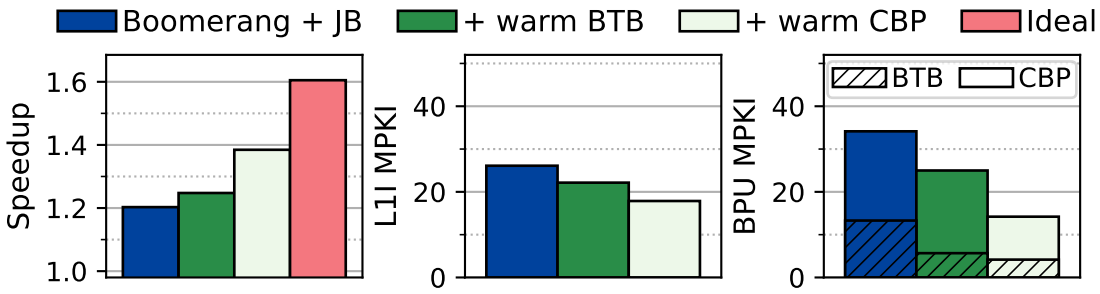


Figure 5.2: Sensitivity to BPU state

ing 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).

The results of the study are shown in Figure 5.2, 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.

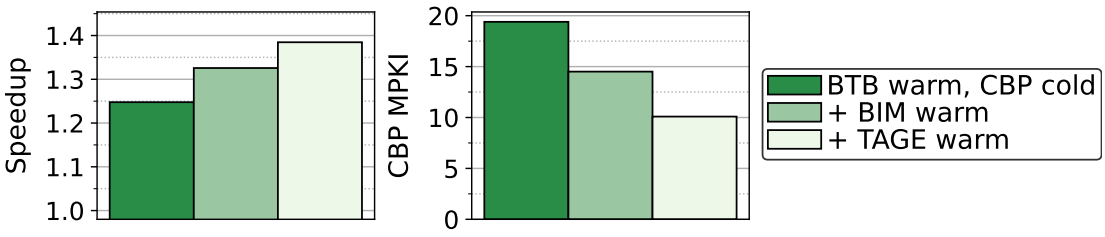


Figure 5.3: Sensitivity to the CBP state on Boomerang+JB with a warm BTB.

5.1.3 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[2]. 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 5.2. 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 5.3. 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 5.4 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 indi-

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

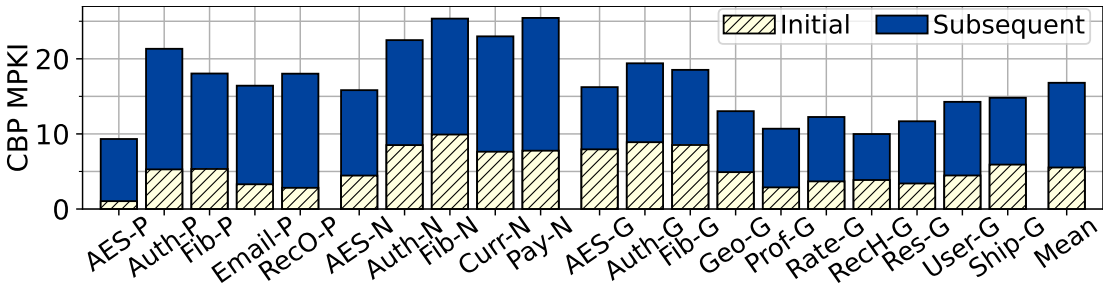


Figure 5.4: BPU MPKI of Boomerang+JB (warm BTB) split into initial (striped) and subsequent misprediction (solid).

cate 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 misspeculation.

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.

5.1.4 Putting it all Together

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

While we demonstrate in the previous chapter that Jukebox is effective 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.

5.2 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.,

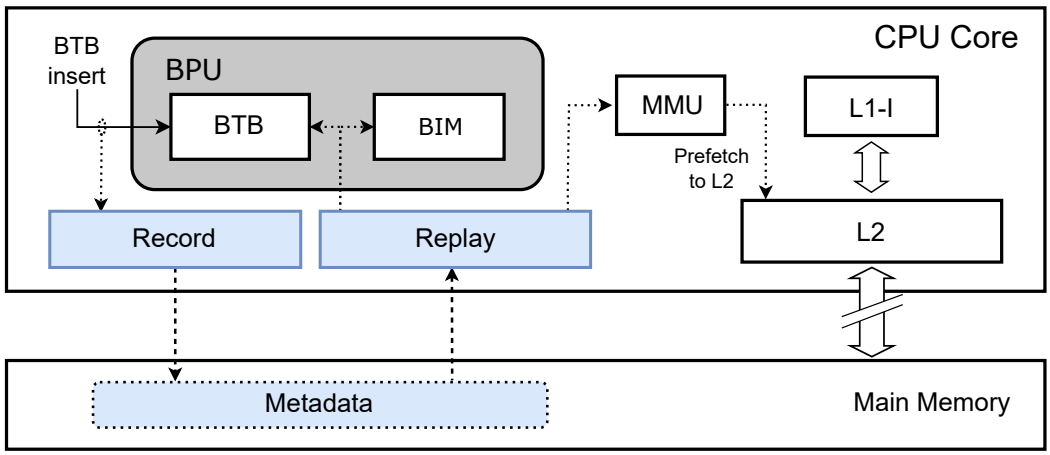


Figure 5.5: Ignite architecture overview. Blue shows new structures and metadata.

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 [3, 76]. 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 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 5.1.3).

Figure 5.5 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 [128] or Boomerang [96]), 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 [166, 173] 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 [166].

5.2.1 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 chapter 5, the BTB in modern processors only inserts taken branches. Furthermore, as a cold BTB can be expected when recording starts (see Section 3.5), 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

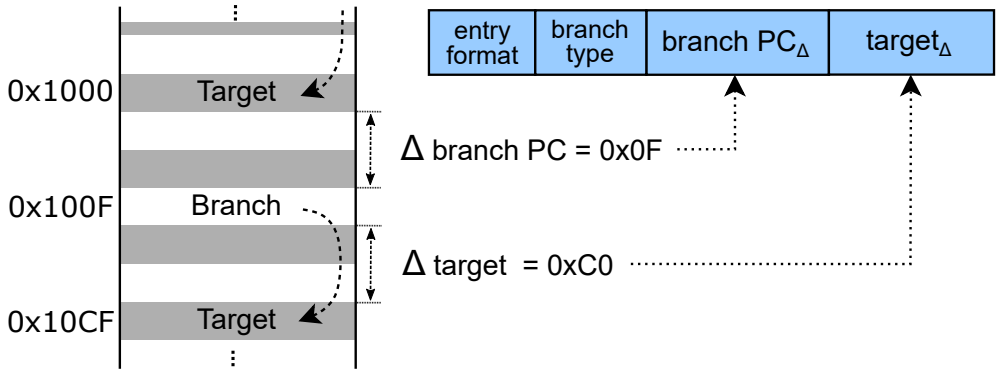


Figure 5.6: Ignite recording and metadata layout

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 [19, 153]. 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[3]. 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 5.6 visualizes the creation of metadata entry and its format.

5.2.2 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

[3]We empirically found 7 bits for branch PC delta and 21 bits for target delta to achieve the highest compression. Correspondingly the size of one Ignite entry is either 97 bits (full PC + target + type bit) or 29 bits (delta PC + target delta + type bit)

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 *restore-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 5.2.3) to capture a branch working set that evolves between invocations.

5.2.3 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, which is

described in more detail in Section 4.1.3.

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 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. Note that recording can be done at the same time as replaying. In that case, Ignite has to record BTB misses, and BTB hits on restored entries, i.e., usefully restored entries. It uses the restore bit to determine whether an entry was restored by Ignite or a normal BTB hit.

5.2.3.1 Hardware Overhead

Ignite reuses most of the hardware components from Jukebox (detailed in Section 4.1.3.2) and adds additional logic to restore BPU microarchitectural state. Specifically, it uses the same four architecturally exposed 48-bit registers to allow the OS to manage Ignite, two 32-bit counters to stop upon reaching the trace limit, and two 512-bit buffers needed to cache one cache line of metadata for the record and replay logic. Ignite does not need the CRRB from Jukebox. Instead, it uses two 96-bit registers to maintain the last metadata entry required to compute the deltas for the branch PC and target for compressing the metadata.

The most costly part of Ignite is the additional restore bit added to each BTB entry, which is used for throttling replay. For the 12K-entry Ice Lake BTB, the restore bits incur an additional 1.5KiB of hardware[4], corresponding to approximately a 1.5%

[4]As modern BTB typically have multiple levels, some metadata used to transfer and replace entries between levels is likely already available and could be used for Ignite. However, we consider the worst case where none of this information is available.

increase in BTB area (assuming a 48-bit target, 12-bit tag, 2-bit type, and 4 bits for validity and replacement information) and accounting for 90.4% of Ignite's overall hardware cost of 1.66KiB. While this is less than 0.013% of the Ice Lake core area[5], it suggests exploring less resource-intensive throttling mechanisms. Some ideas designed for prefetch throttling [68] might be useful for Ignite to make it even more lightweight.

To restore entries into the BTB and the CBP, Ignite must access both structures in parallel to the normal fetch operations. One approach is adding a second port to the BTB and CBP, enabling simultaneous access but introducing significant area and energy overhead [174][6]. Alternatively, Ignite can use the existing ports and arbitrate accesses to both structures. Specifically, whenever the fetch engine stalls and does not access the BTB or CBP, Ignite can use the normal port to place entries into both structures. This approach introduces minimal hardware overhead at the cost of potentially delayed restorations, which could impact prefetch distance. However, since the front end is completely cold upon function start and instructions must be fetched from memory, inducing hundreds of stall cycles, we argue that these stalls provide sufficient time for Ignite to restore the microarchitectural state. We leave a detailed analysis of this trade-off for future research.

5.2.4 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) [73] or Arm's branch record register (BRB) [17] 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 [16] Ignite does not increase the attack surface. Additionally, Ignite is compatible with side channel mitigations like Arm's BTB tagging feature (FEAT_CSV2) [16]. In a CPU featuring

[5] Approximated base on annotated die shot photos of the Ice lake core `https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_(client)`

[6] As modern BTBs are often already multiported [3, 123] to transfer information between different BTB levels, Ignite could piggy-back the existing mechanism. However, we consider the worst case with only one port available.

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.

5.3 Methodology

We evaluate Ignite on the same 20 distinct serverless functions and the same gem5 setup used throughout the work. For details on the workloads, we refer to Section 4.2.2 and for the gem5 configuration and simulator parameters to Section 4.2.1. Once configured with the parameters of the Intel Ice-Lake server CPU we evaluate the following prefetcher configurations:

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 [74][7]. FTQ: 32 entries; branch predictor bandwidth is double the fetch width [122]; branch predictor uses taken-only history [3, 4].

Boomerang: FDP augmented with the BTB filling mechanism as described in [96]. 6-cycle pre-decode latency, 16-entry BTB prefetch buffer.

Confluence: 8 K entry index and a 32 K entry history buffer [86]. 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.

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

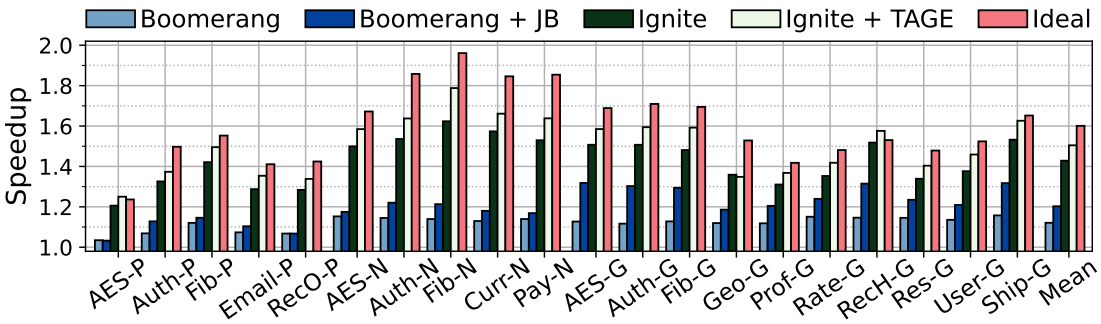


Figure 5.7: Performance results over next-line prefetcher.

5.4 Evaluation

5.4.1 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 [166], but it is useful for understanding the opportunity in restoring TAGE.

Figure 5.7 presents the results of the evaluation, normalized to our Baseline (NL). As reported in Section 5.1.1, 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 3.6b) 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.

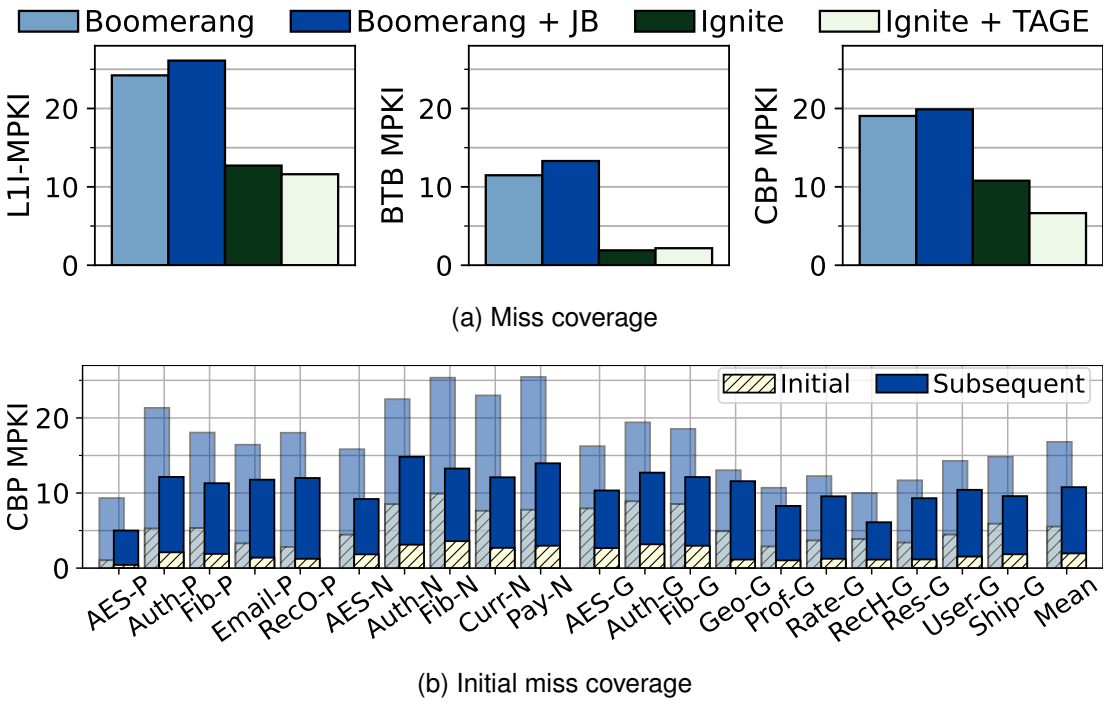


Figure 5.8: Ignites miss coverage (a) and detailed *initial* miss coverage (b). The initial missed coverage shows Ignite in the front and Boomerang+JB (warm BTB) in the back.

5.4.2 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 5.8a, 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 5.8a 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 5.8a, right, Ignite reduces the inci-

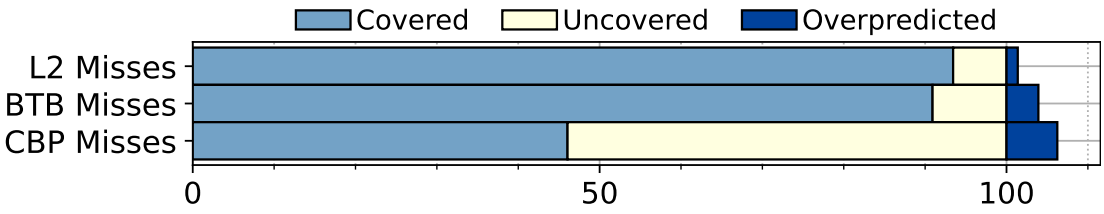


Figure 5.9: Ignites accuracy in restoring microarchitectural state.

dence of branch mispredictions by nearly half versus other front-end prefetchers — from 19 MPKI or more to just over 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 5.1.3, we split initial and subsequent mispredictions for Ignite and show the results Figure 5.8b. For ease of comparison, the previous results (Figure 5.4) 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 5.9 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.

5.4.3 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 (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.

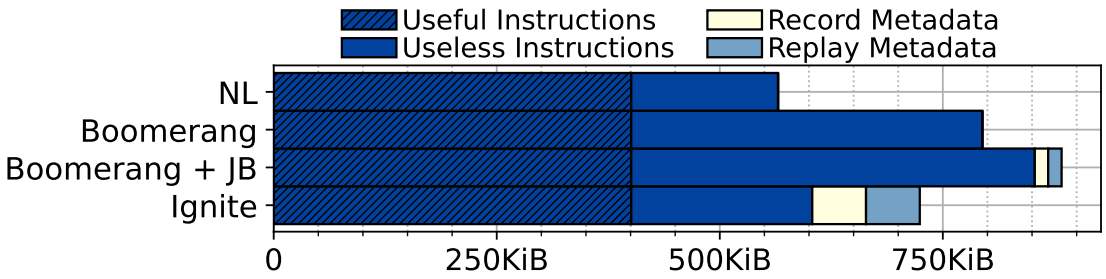


Figure 5.10: Ignite's impact on memory bandwidth.

Figure 5.10 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 5.10, 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.

5.4.4 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

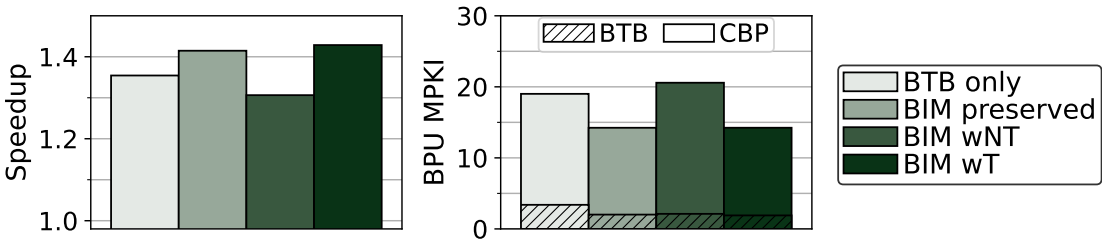


Figure 5.11: Comparison of different BIM initialization policies. *wNT*: weakly not-taken, (*wT*): weakly taken.

BIM entries together with inserting branches into the BTB to a *weakly not-taken* state (wNT) and a *weakly taken* (wT) state[8]. The latter policy — initializing inserted entries to wT — is used by Ignite.

In Figure 5.11, 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 5.1.3 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.

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

5.4.5 Temporal Streaming Prefetchers

So far, we have only considered FDP-based front-end prefetchers. We now examine temporal streaming prefetching (Section 2.3) 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 [86], a state-of-the-art unified temporal streaming prefetcher discussed in Section 2.3. 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 5.12.

As the figure shows, the general trends presented for Boomerang (an FDP-style prefetcher) in Figure 5.1 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 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 5.1.3 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.

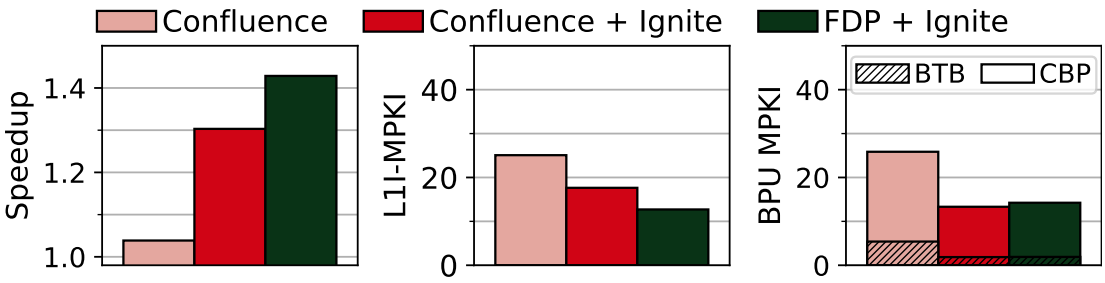


Figure 5.12: Evaluation of Temporal Streaming Prefetchers

5.5 Related Work

Serverless: Little work has been done on understanding microarchitectural implications of serverless computing. In one previous study, Shahrad et al. [149] 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.

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.[7] control LLC capacities available for individual virtual machines while others leverage record-and-replay to prefetch the entire LLC state upon context switch [35, 171]. 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 [166]. 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 [22, 89, 104], BTB entries [88] or branch prediction hints [90]. Each of these techniques requires architectural support and increases code size. Furthermore, prior techniques 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 [29, 120, 124]. But doing so does not help with BTB misses or branch mispre-

dictions.

5.6 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 chapter 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.

Chapter 6

Conclusions and Future Work

Serverless computing has emerged as a promising cloud execution model to bridge the gap between efficient resource management and ease of use for developers. Despite its potential to improve energy efficiency and reduce the carbon footprint of data centres through enhanced resource utilization, serverless computing represents a significant shift in how code is executed in the cloud. Characterized by short execution times, small memory footprints, and long inter-arrival times, serverless workloads differ fundamentally from traditional cloud workloads. The implications of these characteristics for CPU performance are not yet fully understood.

To address this knowledge gap and ensure the sustainability of serverless computing as an efficient cloud execution model, we revisit the objectives outlined in the introduction of this dissertation:

1. **Examine Performance Implications:** Develop a comprehensive understanding of the performance implications of serverless workload characteristics on modern CPUs.
2. **Understand Bottlenecks:** Identify bottlenecks and offer insights into their existence and potential solutions.
3. **Overcome Bottlenecks:** Design and evaluate solutions to address bottlenecks resulting from the mismatch in CPU designs and the serverless workload characteristics.
4. **Ensure Lightweightness:** The proposed solutions must be lightweight, non-disruptive, and easy to integrate with existing hardware and software to ensure sustainability.

In the remainder of this chapter, we summarize how the contributions of this thesis

address these objectives and conclude with a discussion of limitations and future work.

6.1 Summary of Contributions

6.1.1 Understanding Serverless Workloads

To address the first two objectives, we examined in chapter 3 a typical serverless scenario using an Intel Xeon Ice Lake server with numerous interleaved executing functions. The large number of co-located functions, with extremely short execution times, results in a high degree of interleaving and frequent context switching. Due to long intervals between invocations, functions often find their CPU microarchitectural state cold, a phenomenon we term *lukewarm execution*. This leads to a performance degradation of 2x or more compared to warm executions. A detailed Top-Down analysis identified the CPU front-end (i.e., instruction delivery, branch identification, and branch prediction) as the primary source of performance degradation, with a significant pain point being a high number of off-chip instruction misses.

We examined the instruction and branch working sets of functions and found significant commonality across invocations and a modest footprint, explaining the 2x performance difference between interleaved and non-interleaved scenarios. While high commonality provides high performance and efficiency for a single function — instruction and branch working sets are small enough to fit within fast on-chip structures — the combined working sets of hundreds of functions overwhelm the capacity of processor caches and branch predictor units by orders of magnitude. This results in many off-chip instruction misses, BTB misses, and a high branch misprediction rate, causing severe performance degradation.

6.1.2 Addressing Lukewarm Execution

In addressing the third and fourth objectives, we proposed in chapters 4 and 5 two lightweight techniques to mitigate the performance degradation of lukewarm serverless function execution:

Jukebox This technique addresses the largest source of performance degradation, off-chip instruction misses, by exploiting the high commonality of instruction working sets across function invocations. Jukebox records instruction working sets during one invocation and uses the records to issue prefetches during subsequent invocations. Its

key features of (1) recording and replaying instructions into the L2 cache, (2) storing records in the main memory, and (3) being decoupled from the core make Jukebox a simple and lightweight solution. Despite its lean design, Jukebox covers more than 90% of L2 instruction misses and shows a 17.8% average speed-up for lukewarm serverless functions with minimal hardware overhead and a small metadata cost.

Ignite: Building on Jukebox, Ignite provides a comprehensive solution to mitigate all remaining front-end inefficiencies caused by lukewarm execution, including high numbers of L1 instruction misses, BTB misses, and branch mispredictions. Ignite introduces a unified metadata format to densely store the control flow of a serverless function in memory. This metadata is used during subsequent invocations to restore the entire front-end state, including the BTB, CBP, and instruction cache. Ignite achieves a 43% average performance improvement and significantly reduces miss rates across all front-end structures compared to previous approaches, with low hardware cost and complexity.

6.2 Future Work

Jukebox and Ignite are lightweight, simple, and highly effective mechanisms for removing inefficiencies in lukewarm serverless function execution. However, several opportunities for future work could further improve the performance of serverless workloads:

6.2.1 Addressing Back-End Inefficiencies

While our characterization in chapter 3 highlights the CPU front-end as the main bottleneck for serverless workloads, the back-end also shows a significant increase in stall cycles — 171% compared to the non-interleaved scenario. Figure 3.5c reveals that data, in addition to instructions, experiences as well a high number of off-chip misses. Moreover, as Jukebox and Ignite mitigate front-end inefficiencies, an imbalance in the CPU pipeline is created, making back-end stalls more severe.

To fully address performance degradation due to lukewarm execution, future work could explore mechanisms to mitigate back-end inefficiencies. While conducting experiments for this dissertation, we found that — despite different inputs — not only instruction accesses but also data accesses have a high commonality across invoca-

tions [1]. This is likely because a large portion of execution time is spent in the communication and runtime layers (e.g., Python or NodeJS), which remain consistent across invocations. Thus, a record-and-replay mechanism similar to Jukebox could mitigate off-chip data misses.

An open question is whether Ignite could be extended to address back-end inefficiencies. Ignite's metadata could be expanded to include data addresses, but this would require two separate recording logics — one for the BTB and another for the L1-D cache.

Another consideration is the potential security concerns of a simple record-and-replay mechanism. Since input data is typically not scrubbed between invocations, replaying data from a previous invocation might leak sensitive information. Future work could explore mechanisms to secure the metadata.

6.2.2 Software Approach

Despite their lean designs, Jukebox and Ignite are hardware-based solutions that require additions to the CPU microarchitecture, increasing design complexity and the carbon footprint of the chip. Future work could explore software-based solutions, which would benefit from faster adoption and greater flexibility. Recent CPU generations already feature software prefetch instructions for both data and instructions [18, 73], making recording the working set efficiently the main challenge.

One approach could be to use hardware tracing mechanisms, such as Intel's last branch records (LBR) [73] or Arm's branch record buffer (BRB) [17], to record the control flow of a function. A post-processing tool could then extract the unique working set and store it in a compact format. Since tracing and post-processing add overhead, recording should be done only when the working set changes significantly, such as after JIT recompilations.

Serverless providers could integrate this feature transparently into their runtime environments by monitoring performance counters to trigger recording and post-processing when an increased number of off-chip misses are detected. Before or during function invocation, the runtime can spawn a helper thread that uses software prefetch instructions [22, 89, 104] to load the working sets into the L2 cache.

[1] The results of data access commonality are not shown in this thesis as it exceeds its scope.

6.2.3 Accelerate Context Switches

Although Jukebox and Ignite are specifically designed for serverless workloads, they could also benefit other workloads that experience high context switch rates. For example, these techniques may be promising for asynchronous or event-based programming models increasingly supported in languages such as Go [28], Python [126], or C++ [34].

However, key questions remain: (1) How much commonality exists between switching in and out of a context? (2) How should partially evicted working sets be handled?

Jukebox and Ignite are effective for serverless because the same functionality is executed during each invocation. In scenarios where the OS multiplexes multiple threads onto the same core, a thread's execution is paused when replaced by another thread and continues once switched back. It's unclear how much of the working set present before the switch is needed after the switch, and thus, a record-and-replay mechanism might not yield performance gains.

Due to the high degree of interleaving in serverless, the working set is typically cold upon every invocation, allowing Jukebox and Ignite to use simple logic to efficiently record the full working set of a serverless function. If these techniques are used to accelerate other applications, the degree of interleaving is likely less, and the microarchitecture may not be entirely cold. In this case, recording only L2 or BTB misses might not capture the full working set, reducing replay coverage.

Finding answers to those questions can be part of future work exploring the generalization of Jukebox and Ignite to other workloads and investigating their performance benefits.

6.3 Impact Beyond Severless

This work focuses on serverless computing, which exhibits frequent context switches and consistently cold microarchitectural state. The restoration mechanisms developed in Jukebox and Ignite address a significant performance challenge existing in various computing paradigms beyond serverless: how to efficiently warm up the microarchitectural state during frequent context switches.

Numerous application domains suffer from frequent context switches causing severe performance degradation, wasting billions in terms of cost and energy [24]. For

example, data center applications and web services process billions of micro-second events in real-time, with production traces from Google revealing some applications comprising up to 500 individual threads [99, 127] scheduled on the same CPU. Similarly, mobile applications suffer from frequent context switches, with typical switch periods ranging from 1ms to 100ms [157]. These switches occur as applications interact with accelerators and online services, causing the CPU to process other applications and background tasks during waiting periods. For instance, Vougioukas et al. documented a video camera app experiencing approximately 33.000 context switches per second, entailing a performance degradation of more than 15% [166]. The findings of this work can provide valuable insights for these domains to accelerate context switches and improve CPU performance.

Another promising application area is confidential computing, where this work could help reduce performance degradation caused by side-channel mitigation techniques [16, 166]. Currently, preventing data leakage across applications via side-channel attacks requires flushing all microarchitectural structures between context switches. While essential for security, this flushing significantly degrades performance, creating a challenging trade-off between security and user experience. Although Jukebox and Ignite cannot eliminate the need for flushing, their techniques could provide insights into mitigating some performance impacts by warming up flushed structures post-context switch and contribute to ongoing research efforts to better balance security and performance in modern computing.

The insights and techniques developed in this work transcend the specific context of serverless computing, offering a broader perspective on improving efficiency across diverse computing domains where frequent context switches pose significant performance challenges.

Bibliography

[1] 7-Zip LZMA Benchmark, "Intel Ice Lake," 2019. [Online]. Available: https://www.7-cpu.com/cpu/Ice_Lake.html

[2] 7-Zip LZMA Benchmark, "Intel Skylake," 2022. [Online]. Available: https://www.7-cpu.com/cpu/Skylake_X.html

[3] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito, "The IBM z15 high frequency mainframe branch predictor industrial product," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 27–39.

[4] I. Advanced Micro Devices, "Software optimization guide for the amd zen4 microarchitecture," Advanced Micro Devices, Inc., Cambridge, MA, USA, Tech. Rep., 2023.

[5] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, R. Bhagwan and G. Porter, Eds. USENIX Association, 2020, pp. 419–434.

[6] M. Aggar, "The it energy efficiency imperative," Microsoft Corporation, Tech. Rep., 2011. [Online]. Available: https://www.thegreengrid.org/en/system/files/ITEE_White_Paper.pdf

[7] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous CPU availability for consolidated systems," in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*. IEEE Computer Society, 2014, pp. 394–405.

[8] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*. IEEE Computer Society, 2003, p. 423.

[9] Amazon, "A Demo Running 4000 Firecracker MicroVMs," 2022. [Online]. Available: https://github.com/firecracker-microvm/firecracker-demo

[10] Amazon, "AWS Lambda Functions Now Scale 12 Times Faster When Handling High-Volume Requests," 2023. [Online]. Available: https://aws.amazon.com/blogs/aws/aws-lambda-functions-now-scale-12-times-faster-when-handling-high-volume-requests/

[11] Amazon, "AWS Lambda Pricing," 2023. [Online]. Available: https://aws.amazon.com/lambda/pricing/

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

[13] A. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, Apr. 2015.

[14] Andrei Frumusanu, "Golden Cove Microarchitecture (P-Core) Examined," 2021. [Online]. Available: https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3

[15] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 65–78.

[16] Arm, "Spectre-BHB: Speculative Target Reuse Attacks, Version 1.7," Arm Limited, Tech. Rep., 2022.

[17] Arm. (2023) Feature names in a-profile architecture. [Online]. Available: https://developer.arm.com/downloads/-/exploration-tools/feature-names-for-a-profile

[18] Arm. (2024) Prefetching with __builtin_prefetch. [Online]. Available: https://developer.arm.com/documentation/101458/2304/Optimize/Prefetching-with---builtin-prefetch

[19] T. Asheim, B. Grot, and R. Kumar, "BTB-X: A storage-effective BTB organization," *IEEE Comput. Archit. Lett.*, vol. 20, no. 2, pp. 134–137, 2021.

[20] Atlassian. (2023) Advantages of microservices. [Online]. Available: https://www.atlassian.com/microservices/cloud-computing/advantages-of-microservices

[21] S. W. S. Authors. (2022) Serverless workflow. [Online]. Available: https://serverlessworkflow.io/

[22] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, S. B. Manne, H. C. Hunter, and E. R. Altman, Eds. ACM, 2019, pp. 462–473.

[23] M. Azure. (2024) Azure functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[24] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, p. 48–54, Mar. 2017.

[25] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[26] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[27] 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*, M. L. Soffa and M. J. Irwin, Eds. ACM, 2009, pp. 313–324.

[28] S. Chakraborty. (2020) Goroutines in golang. [Online]. Available: https://golangdocs.com/goroutines-in-golang

[29] D. Chen, D. X. Li, and T. Moseley, "Autofdo: automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, B. Franke, Y. Wu, and F. Rastello, Eds. ACM, 2016, pp. 12–23.

[30] G. Cloud. (2024) Cloud functions. [Online]. Available: https://cloud.google.com/functions

[31] A. Cockroft. (2016) The evolution of microservices. [Online]. Available: https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference

[32] Colin Ian King, "Stress-ng," 2022. [Online]. Available: https://github.com/ColinIanKing/stress-ng

[33] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *1996 International Conference on Computer Design (ICCD '96), VLSI in Computers and Processors, October 7-9, 1996, Austin, TX, USA, Proceedings*. IEEE Computer Society, 1996, pp. 468–477.

[34] cppreference. (2024) Coroutines. [Online]. Available: https://en.cppreference.com/w/cpp/language/coroutines

[35] D. Daly and H. W. Cain, "Cache restoration for highly partitioned virtualized systems," in *18th IEEE International Symposium on High Performance*

Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012. IEEE Computer Society, 2012, pp. 225–234.

[36] Datadog, "The State of Serverless 2020," 2020. [Online]. Available: https://www.datadoghq.com/state-of-serverless-2020

[37] Datadog, "The State of Serverless 2021," 2021. [Online]. Available: https://www.datadoghq.com/state-of-serverless-2021/

[38] Datadog, "The State of Serverless 2022," 2022. [Online]. Available: https://www.datadoghq.com/state-of-serverless-2022/

[39] Datadog, "The State of Serverless 2023," 2022. [Online]. Available: https://www.datadoghq.com/state-of-serverless/

[40] M. Digital. (2024) Building a serverless application on google cloud to accelerate deployment time with reduced cost and complexity. [Online]. Available: https://medium.com/digital-mckinsey/building-a-serverless-application-on-google-cloud-to-accelerate-deployment-time-with-reduced-cost-467d1a82369d

[41] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 467–481.

[42] T. Economist. The world's most valuable resource is no longer oil, but data. [Online]. Available: https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data

[43] T. S. Edge. (2021) Find out how rapid development with serverless is like lego. [Online]. Available: https://theserverlessedge.com/find-out-how-rapid-development-with-serverless-is-like-lego/

[44] S. Eismann, J. Scheuner, E. V. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A review of serverless use cases and their characteristics," *CoRR*, vol. abs/2008.11110, 2020.

[45] 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 *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, T. Harris and M. L. Scott, Eds. ACM, 2012, pp. 37–48.

[46] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, C. Galuzzi, L. Carro, A. Moshovos, and M. Prvulovic, Eds. ACM, 2011, pp. 152–162.

[47] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008), November 8-12, 2008, Lake Como, Italy*. IEEE Computer Society, 2008, pp. 1–10.

[48] Flexera, "2024 state of the cloud," Flexera, Tech. Rep., 2024. [Online]. Available: https://info.flexera.com/CM-REPORT-State-of-the-Cloud-2024-Thanks

[49] A. Frumusanu. (2019) Arm announces neoverse n1 & e1 platforms & cpus: Enabling a huge jump in infrastructure performance. [Online]. Available: https://www.anandtech.com/show/13959/arm-announces-neoverse-n1-platform/2

[50] A. Frumusanu. (2019) Arm's new cortex-a77 cpu micro-architecture: Evolving performance. [Online]. Available: https://www.anandtech.com/show/14384/arm-announces-cortexa77-cpu-ip/2

[51] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 3–18.

[52] M. Garcia Bardon, P. Wuytens, L.-A. Ragnarsson, G. Mirabelli, D. Jang, G. Willems, A. Mallik, A. Spessot, J. Ryckaert, and B. Parvais, "Dtco including sustainability: Power-performance-area-cost-environmental score (ppace) analysis for logic technologies," in *2020 IEEE International Electron Devices Meeting (IEDM)*, 2020, pp. 41.4.1–41.4.4.

[53] gem5 developers. (2022) gem5. [Online]. Available: https://github.com/gem5/gem5/releases/tag/v22.0.0.1

[54] A. Gluck. (2020) Introducing domain-oriented microservice architecture. [Online]. Available: https://www.uber.com/en-GB/blog/microservice-architecture/

[55] Google, "Cloud Functions Pricing," 2022. [Online]. Available: https://cloud.google.com/functions/pricing

[56] Google. (2023) Configuring warmup requests to improve performance. [Online]. Available: https://cloud.google.com/appengine/docs/legacy/standard/python/configuring-warmup-requests

[57] Google Cloud, "Implementing SLOs," 2022. [Online]. Available: https://sre.google/workbook/implementing-slos

[58] GoogleCloudPlatform, "Online Boutique," 2022. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[59] Grand View Research, "Data center market size, share and trends analysis report by component, by type, by server rack density, by redundancy, by pue, by design, by tier level, by enterprise size, by end-use, by region, and segment forecasts, 2023 - 2030," Grand View Research, Tech. Rep., 2023. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/data-center-market-report

[60] Granulate. Granulate issues findings from state of cloud computing survey highlighting underutilization of it infrastructure. [Online]. Available: https://devops.com/granulate-issues-findings-from-state-of-cloud-computing-survey-highlighting-underutilization-of-it-infrastructure/

[61] B. Grayson, J. Rupley, G. D. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos CPU microarchitecture," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 40–51.

[62] U. D. C. A. R. Group. (2020) gem5 skylake config. [Online]. Available: https://github.com/darchr/gem5-skylake-config/blob/master/configuration-details.md

[63] gRPC Authors, "gRPC: A High-Performance, Open Source Universal RPC Framework," 2022. [Online]. Available: https://grpc.io

[64] U. Gupta, M. Elgamal, G. Hills, G. Wei, H. S. Lee, D. Brooks, and C. Wu, "ACT: designing sustainable computer systems with an architectural carbon modeling tool," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 784–799.

[65] U. Gupta, Y. G. Kim, S. Lee, J. Tse, H. S. Lee, G. Wei, D. Brooks, and C. Wu, "Chasing carbon: The elusive environmental footprint of computing," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 854–867.

[66] J. H. (2023) 4 Microservices Examples: Amazon, Netflix, Uber, and Etsy. [Online]. Available: https://blog.dreamfactory.com/microservices-examples/

[67] S. Hassani, G. Southern, and J. Renau, "Livesim: Going live with microarchitecture simulation," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 2016, pp. 606–617.

[68] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-side prefetch throttling: adaptive prefetching for high-performance many-core processors," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus,*

November 01-04, 2018, S. Evripidou, P. Stenström, and M. F. P. O'Boyle, Eds. ACM, 2018, pp. 28:1–28:11.

[69] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[70] C. H. House and R. L. Price, "The return map: Tracking product teams," *Harvard Business Review*, January 1991.

[71] IEA, "Digitalisation and energy," International Energy Agency (IEA), Tech. Rep., 2017. [Online]. Available: https://www.iea.org/reports/digitalisation-and-energy

[72] Intel, "Ice Lake SP," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/platforms/details/ice-lake-sp.html

[73] Intel. (2023) Intel 64 and ia-32 architectures software developer manuals. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[74] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*. IEEE, 2021, pp. 172–182.

[75] P. Jaccard, "The distribution of the flora in the alpine zone.1," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[76] C. Jacobi, A. Saporito, M. Recktenwald, A. Tsai, U. Mayer, M. M. Helms, A. Collura, P. Mak, R. J. Sonnelitter, M. A. Blake, T. Bronson, A. O'neill, and V. K. Papazova, "Design of the IBM z14 microprocessor," *IBM J. Res. Dev.*, vol. 62, no. 2/3, pp. 8:1–8:11, 2018.

[77] A. V. Jamet, G. Vavouliotis, D. A. Jiménez, L. Alvarez, and M. Casas, "A two level neural approach combining off-chip prediction with adaptive prefetch filtering," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 2024, pp. 528–542.

[78] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 404–415, jun 2013.

[79] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC 2013, Portland, OR, USA, September 22-24, 2013*. IEEE Computer Society, 2013, pp. 66–76.

[80] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *ASPLOS '21: 26th*

ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 152–166.

[81] Jonathan Corbet, "Memory compaction," 2010. [Online]. Available: https://lwn.net/Articles/368869

[82] N. Jones, "How to stop data centres from gobbling up the world's electricity," *Nature*, vol. 561, no. 7722, pp. 163–166, Sep. 2018.

[83] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, D. T. Marr and D. H. Albonesi, Eds. ACM, 2015, pp. 158–169.

[84] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001, Göteborg, Sweden, June 30-July 4, 2001*, P. Stenström, Ed. ACM, 2001, pp. 240–251.

[85] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: shared history instruction fetch for lean-core server processors," in *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*, M. K. Farrens and C. Kozyrakis, Eds. ACM, 2013, pp. 272–283.

[86] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: unified instruction supply for scale-out servers," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, M. Prvulovic, Ed. ACM, 2015, pp. 166–177.

[87] kernel.org, "perf: Linux profiling with performance counters," 2020. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[88] T. A. Khan, N. Brown, A. Sriraman, N. K. Soundararajan, R. Kumar, J. Devietti, S. Subramoney, G. A. Pokam, H. Litz, and B. Kasikci, "Twig: Profile-guided BTB prefetching for data center applications," in *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 2021, pp. 816–829.

[89] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-SPY: context-driven conditional instruction prefetching with coalescing," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 2020, pp. 146–159.

[90] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, "Whisper: Profile-guided branch misprediction elimination for data center applications," in *55th IEEE/ACM International Symposium*

on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022. IEEE, 2022, pp. 19–34.

[91] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, and K. Oyama, Eds. IEEE, 2019, pp. 502–504.

[92] J. Kim and K. Lee, "Practical cloud workloads for serverless faas," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 2019, p. 477.

[93] S. Kluyskens and L. Eeckhout, "Branch predictor warmup for sampled simulation through branch history matching," *Trans. High Perform. Embed. Archit. Compil.*, vol. 2, pp. 45–64, 2009.

[94] A. Kuhwilm. (2023) Understanding the costs of serverless architecture: Will it save you money? [Online]. Available: https://www.mongodb.com/blog/post/understanding-costs-serverless-architecture-save-money

[95] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 30–42.

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

[97] E. Lab. (2022) vswarm-u: Microarchitecture for serverless workloads. [Online]. Available: https://github.com/ease-lab/vSwarm-u

[98] V. Lannurien, L. D'Orazio, O. Barais, and J. Boukhobza, *Serverless Cloud Computing: State of the Art and Challenges*. Cham: Springer International Publishing, 2023, pp. 275–316. [Online]. Available: https://doi.org/10.1007/978-3-031-26633-1_11

[99] Y. Li, N. Lazarev, D. Koufaty, T. Yin, A. Anderson, Z. Zhang, G. E. Suh, K. Kaffes, and C. Delimitrou, "Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 922–936.

[100] C. Lin and S. J. Tarsa, "Branch prediction is not A solved problem: Measurements, opportunities, and future directions," in *IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019*. IEEE, 2019, pp. 228–238.

[101] Linux Foundation, “pthread_create – Linux manual page,” 2008. [Online]. Available: https://man7.org/linux/man-pages/man3/pthread_create.3.html

[102] LogicMonitor. (2023) What are microservices and why use them? [Online]. Available: https://www.logicmonitor.com/blog/what-are-microservices

[103] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, “The gem5 simulator: Version 20.0+,” *CoRR*, vol. abs/2007.03152, 2020.

[104] C. Luk and T. C. Mowry, “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 31, Dallas, Texas, USA, November 30 - December 2, 1998*, J. O. Bondi and J. Smith, Eds. ACM/IEEE Computer Society, 1998, pp. 182–194.

[105] Y. Luo, L. K. John, and L. Eeckhout, “Self-monitored adaptive cache warm-up for microprocessor simulation,” in *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2004), 27-29 October 2004, Foz do Iguacu, Brazil*. IEEE Computer Society, 2004, pp. 10–17.

[106] D. Meisner, B. T. Gold, and T. F. Wenisch, “Powernap: eliminating server idle power,” 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*, M. L. Soffa and M. J. Irwin, Eds. ACM, 2009, pp. 205–216.

[107] Mikhail Shilkov, “When Does Cold Start Happen on AWS Lambda?” 2021. [Online]. Available: https://mikhail.io/serverless/coldstarts/aws/intervals

[108] Mikhail Shilkov, “When Does Cold Start Happen on Azure Functions?” 2021. [Online]. Available: https://mikhail.io/serverless/coldstarts/azure/intervals

[109] Mikhail Shilkov, “When Does Cold Start Happen on Google Cloud Functions?” 2021. [Online]. Available: https://mikhail.io/serverless/coldstarts/gcp/intervals

[110] T. P. Morgan. (2021) Aws goes wide and deep with graviton3 server chip. [Online]. Available: https://www.nextplatform.com/2021/12/02/aws-goes-wide-and-deep-with-graviton3-server-chip/

[111] H. Mujtaba. (2022) Intel 14th gen meteor lake die shot pictured: Early sample gives first look at next-gen redwood cove & crestmont cores. [Online]. Available: https://wccftech.com/intel-14th-gen-meteor-lake-cpu-die-shot-redwood-cove-crestmont-cores-compute-tile/

[112] S. Naffziger, B. A. Stackhouse, T. Grutkowski, D. Josephson, J. Desai, E. Alon, and M. Horowitz, "The implementation of a 2-core, multi-threaded itanium family processor," *IEEE J. Solid State Circuits*, vol. 41, no. 1, pp. 197–209, 2006.

[113] M. Nair. (2017) How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play. [Online]. Available: https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b

[114] G. Neves. (2017) Keeping functions warm - how to fix aws lambda cold start issues. [Online]. Available: https://serverless.com/blog/keep-your-lambdas-warm

[115] N. Nikoleris, D. Eklov, and E. Hagersten, "Extending statistical cache models to support detailed pipeline simulators," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 2014, pp. 86–95.

[116] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, H. S. Gunawi and B. C. Reed, Eds. USENIX Association, 2018, pp. 57–70.

[117] Oracle Corporation, "Chapter 3 Thread Create Attributes," 2010. [Online]. Available: https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032j/index.html

[118] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 118–131.

[119] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*. IEEE Computer Society, 2016, pp. 122–132.

[120] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A practical binary optimizer for data centers and beyond," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, M. T. Kandemir, A. Jimborean, and T. Moseley, Eds. IEEE, 2019, pp. 2–14.

[121] Paul Lilly, "Leaked AMD Zen 4 Cache Upgrades Could Be Key In Competing With Alder Lake," 2021. [Online]. Available: https://hothardware.com/news/amd-zen-4-cache-key-competing-alder-lake

[122] A. Pellegrini, A. K. Tummala, J. Jalal, M. Werkheiser, A. Kona, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, and T. Ringe, "The arm neoverse N1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc," *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.

[123] A. Perais and R. Sheikh, "Branch target buffer organizations," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 240–253.

[124] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, B. N. Fischer, Ed. ACM, 1990, pp. 16–27.

[125] A. H. plc, "Arm neoverse v2 platform: Leadership performance and power efficiency for next-generation cloud computing, ml and hpc workloads," 2023. [Online]. Available: https://hc2023.hotchips.org/assets/program/conference/day1/CPU1/HC2023.Arm.MagnusBruce.v04.FINAL.pdf

[126] Python. (2024) Coroutines and tasks. [Online]. Available: https://docs.python.org/3/library/asyncio-task.html

[127] P. Ranganathan and V. Lee, "Advancing systems research with open-source google workload traces," 2022. [Online]. Available: https://cloud.google.com/blog/topics/research-and-perspectives/advancing-systems-research-with-open-source-google-workload-traces

[128] G. Reinman, B. Calder, and T. M. Austin, "Fetch directed instruction prefetching," in *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 32, Haifa, Israel, November 16-18, 1999*, R. Ronen, M. K. Farrens, and I. Y. Spillinger, Eds. ACM/IEEE Computer Society, 1999, pp. 16–27.

[129] E. M. Research, "Global serverless computing market outlook," Expert Market Research, Tech. Rep., 2024. [Online]. Available: https://www.expertmarketresearch.com/reports/serverless-computing-market

[130] R. Riedlinger, R. Arnold, L. Biro, B. Bowhill, J. Crop, K. Duda, E. S. Fetzer, O. Franza, T. Grutkowski, C. Little, C. Morganti, G. Moyer, A. Munch, M. Nagarajan, C. Parks, C. Poirier, B. Repasky, E. Roytman, T. Singh, and M. W. Stefaniw, "A 32 nm, 3.1 billion transistor, 12 wide issue itanium® processor for mission-critical servers," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 177–193, 2012.

[131] R. J. Riedlinger, R. Arnold, L. Biro, W. J. Bowhill, J. Crop, K. Duda, E. S. Fetzer, O. Franza, T. Grutkowski, C. Little, C. Morganti, G. Moyer, A. O. Munch, M. Nagarajan, C. Park, C. Poirier, B. Repasky, E. Roytman, T. Singh, and M. W. Stefaniw, "A 32 nm, 3.1 billion transistor, 12 wide issue itanium® processor for mission-critical servers," *IEEE J. Solid State Circuits*, vol. 47, no. 1, pp. 177–193, 2012.

[132] A. Ros and A. Jimborean, "The entangling instruction prefetcher," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 84–87, 2020.

[133] Y. Rybkin. (2023) Microservices Architecture Use Cases and Real-World Examples. [Online]. Available: https://codeit.us/blog/microservices-use-cases

[134] D. Sanchez, "Lecture notes: Branch prediction," 2021. [Online]. Available: http://csg.csail.mit.edu/6.823S21/Lectures/L10.pdf

[135] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: Characterization and optimization," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 757–770.

[136] D. Schall, A. Sandberg, and B. Grot, "Warming up a cold front-end with ignite," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*. ACM, 2023, pp. 254–267.

[137] S. Schirmer. (2016) Api-first architecture transformation at etsy. [Online]. Available: https://www.infoq.com/presentations/etsy-api/

[138] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: the next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[139] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: the next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[140] A. W. Services. (2020) Coca-cola freestyle launches touchless fountain experience in 100 days using aws lambda. [Online]. Available: https://aws.amazon.com/solutions/case-studies/coca-cola-freestyle/

[141] A. W. Services. (2022) Aws lambda functions powered by aws graviton2 processor. [Online]. Available: https://aws.amazon.com/blogs/aws/aws-lambda-functions-powered-by-aws-graviton2-processor-run-your-functions-on-arm-and-get-up-to-34-better-price-performance/

[142] A. W. Services. (2023) Use on-demand instance capacity for unpredictable workload usage. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/analytics-lens/best-practice-11.3---use-on-demand-instance-capacity-for-unpredictable-workload-usage..html

[143] A. W. Services. (2024) Aws lambda. [Online]. Available: https://aws.amazon.com/lambda/

[144] A. W. Services. (2024) Create a lambda function using a container image. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/images-create.html

[145] A. Seznec, "A 256 kbits l-tage branch predictor," *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, vol. 9, pp. 1–6, 2007.

[146] A. Seznec, "A 64 kbytes isl-tage branch predictor," in *JWAC-2: Championship Branch Prediction*, 2011.

[147] A. Seznec, "Tage-sc-l branch predictors," in *Proceedings of the 4th Championship Branch Prediction*, 2014.

[148] A. Seznec, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[149] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 1063–1075.

[150] M. Shahrad, R. Fonseca, I. Goiri, G. I. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 205–218.

[151] G. Shekhar. (2024) Microservices design patterns for high resiliency. [Online]. Available: https://dzone.com/articles/microservices-design-patterns-for-high-resiliency

[152] S. Shillaker and P. R. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 419–433.

[153] N. K. Soundararajan, P. Braun, T. A. Khan, B. Kasikci, H. Litz, and S. Subramoney, "Pdede: Partitioned, deduplicated, delta branch target buffer," in *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 2021, pp. 779–791.

[154] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 177–194.

[155] B. A. Stackhouse, B. S. Cherkauer, M. K. Gowan, P. E. Gronowski, and C. Lyles, "A 65nm 2-billion-transistor quad-core itanium® processor," in *2008 IEEE International Solid-State Circuits Conference, ISSCC 2008, Digest of Technical Papers, San Francisco, CA, USA, February 3-7, 2008*. IEEE, 2008, pp. 92–93.

[156] J. Stojkovic, C. Liu, M. Shahbaz, and J. Torrellas, "μmanycore: A cloud-native CPU for tail at scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, Y. Solihin and M. A. Heinrich, Eds. ACM, 2023, pp. 33:1–33:15.

[157] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 113–122.

[158] The Cloudflare Blog, "ARMs Race: Ampere Altra Takes on the AWS Graviton2," 2021. [Online]. Available: https://blog.cloudflare.com/arms-race-ampere-altra-takes-on-aws-graviton2

[159] The Cloudflare Blog, "The EPYC Journey Continues to Milan in Cloudflare's 11th Generation Edge Server," 2021. [Online]. Available: https://blog.cloudflare.com/the-epyc-journey-continues-to-milan-in-cloudflares-11th-generation-edge-server

[160] The Firecracker Authors, "Production host setup recommendations," 2022. [Online]. Available: https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md

[161] The University of Utah, "CloudLab Hardware," 2023. [Online]. Available: https://www.cloudlab.us/hardware.php

[162] Twitter. (2013) Decomposing twitter: Adventures in service-oriented architecture. [Online]. Available: https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture

[163] D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing tail latency in serverless clouds with stellar," in *IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021*. IEEE, 2021, pp. 51–62.

[164] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 559–572.

[165] G. Vavouliotis, L. Alvarez, B. Grot, D. A. Jiménez, and M. Casas, "Morrigan: A composite instruction TLB prefetcher," in *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 2021, pp. 1138–1153.

[166] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "BRB: mitigating branch predictor side-channels," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 466–477.

[167] K. A. Wang, R. Ho, and P. Wu, "Replayable execution optimized for page sharing for a managed runtime environment," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse, and C. Fetzer, Eds. ACM, 2019, pp. 39:1–39:16.

[168] WikiChip. (2023) Sunny cove - microarchitectures - intel. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

[169] J. Woodruff, D. Schall, M. F. P. O'Boyle, and C. Woodruff, "When does saving power save the planet?" in *Proceedings of the 2nd Workshop on Sustainable Computer Systems, HotCarbon 2023, Boston, MA, USA, 9 July 2023*, G. Porter, T. Anderson, A. A. Chien, T. Eilam, C. Josephson, and J. Park, Eds. ACM, 2023, pp. 20:1–20:6.

[170] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 2014, pp. 35–44.

[171] J. Zebchuk, H. W. Cain, X. Tong, V. Srinivasan, and A. Moshovos, "RECAP: A region-based cure for the common cold (cache)," in *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 2013, pp. 83–94.

[172] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *43rd*

ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016. IEEE Computer Society, 2016, pp. 456–468.

[173] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural implications of event-driven server-side web applications," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, M. Prvulovic, Ed. ACM, 2015, pp. 762–774.

[174] V. Zyuban and P. Kogge, "The energy complexity of register files," in *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*, 1998, pp. 305–310.