Truls Asheim

Analyzing and Optimizing Serverless Function Execution

Thesis for the degree of Philosophiae Doctor

Trondheim, November 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Innovation and Creativity

Abstract

Serverless computing, implemented as Function-as-a-Service (FaaS), is an emerging and rapidly growing could computing model. It attracts users thanks to its simple and flexible programming model, fine-grained billing model and, zero resource-provisioning overhead for the developer. In FaaS, the fundamental unit of computation is a function, defined as a stateless unit of code executed on-demand when triggered by an external event such as an HTTP request. To enable code reuse, FaaS functions are normally small with well-defined purpose and therefore they have a very short running time, often less than a second. Larger FaaS applications are built by composing multiple individual functions. To communicate, the functions often use high-latency network-backed Remote Procedure Calls (RPC) interfaces. This adds significant overhead to FaaS application execution, especially as the number of functions in an application increases and the communication latencies compound. The short execution time, transient resource allocation, and high communication overhead raise the question of how to build hardware and software platforms for efficiently executing FaaS applications. This thesis addresses this question from two research directions.

The first research direction investigates the hardware perspective of serverless computing. To that end, we first examine the impact of the short running time on FaaS functions on microarchitectural structures. The driving hypothesis is that, since FaaS functions run for a very short time, they do not give microarchitectural structures time to warm up sufficiently to perform optimally. Primarily, our analysis finds that the hypothesis holds only for functions with an extremely short running time (<1 ms). Since such functions are rare in real-world deployments, we conclude that microarchitectural warm-up delay is amortized for the vast majority of FaaS functions. Thus, developing microarchitectural optimizations explicitly targeting FaaS functions with a short running time is unlikely to be beneficial. However, our analysis additionally concludes that FaaS functions have large instruction working sets. This finding means that FaaS functions are affected by the widely established *front-end bottleneck* problem. This problem is caused by large instruction working sets overwhelming the capacities of the branch target buffer (BTB) and instruction caches of processors. A long line of research has established that instruction cache prefetching is an effective way to alleviate the frontend bottleneck. Fetch-directed prefetching (FDP) is a particularly attractive class of such instruction prefetchers. However, for FDP to be effective it requires a sufficiently large BTB. However, access latency, area, and power constraints preclude simply increasing the size of a conventionally designed BTB. Therefore, to increase BTB storage density we introduce BTB-X, an optimized BTB organization that is based on the critical observation that branch target offset lengths are unevenly distributed in server workloads. Therefore, BTB-X proposes to organize the BTB as an 8-way set-associative cache where each way is sized differently to accommodate the variably sized branch target offsets. BTB-X stores 2.24× more branches than a conventional BTB and 1.3× more branches than the state-of-the-art storage-optimized BTB design.

Following the second research direction, we focus on the software aspect of efficiently executing FaaS functions. One of the most significant overheads in FaaS function execution is the excessive latency of the network-backed RPC interfaces often used for inter-function

communication. Previous proposals aiming to alleviate this communication latency are unattractive since they sacrifice at least one of the essential properties of FaaS that makes the programming model appealing. To address this, we introduce CoFaaS, a fully automated software-based transformation for FaaS functions that does not sacrifice any of FaaS' essential properties. We observe that for functions written in different languages to communicate, the RPC interfaces exposed by each function must be well-defined. The critical insight exploited by CoFaaS is that these well-defined interfaces allow us to transform the implementation code of a function freely as long as its external interface is kept unchanged. This allows CoFaaS to retarget the FaaS functions comprising a FaaS application to run on a single WebAssembly runtime. By doing this, CoFaaS alleviates the inter-function communication overhead. CoFaaS reduces the application round-trip time by up to 6× and the inter-function communication and inter-function communication time by up to 100×.

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree of Philosophiae Doctor (PhD). The thesis is organized as a collection of papers, where Part I provides an overview of the work done as part of the thesis, and Part II includes the papers themselves. Apart from reformatting and minor visual adjustments, the contents of the papers are included unmodified. The work was supervised by Rakesh Kumar, and co-supervised by Magnus Jahre and Magnus Själander. Financing was provided by the Department of Computer Science.

Acknowledgements

First and foremost, I would like to express my gratitude to my main supervisor, Rakesh Kumar, for his continuous support over the years. He has been an inspirational figure throughout the thesis writing journey and I am grateful for him never stopping to believe in me, always being supportive of my ideas, and proposing numerous interesting opportunities that made my PhD journey what it was. Furthermore, I would like to thank my co-supervisors Magnus Jahre and Magnus Själander. In particular, my deepest gratitude goes to Magnus Jahre for his invaluable support, feedback, ideas, and encouragement during the final phases of the thesis process.

I would like to thank my colleagues for making my time at IDI a great one! In particular, I would like to mention Joseph, Björn, Fatemeh, Lukas,Roman, Bart, Pauline, Arthur, and Özlem for the good times, enjoyable conversations and entertaining lunches.

I would also like to thank my colleagues I met during my stay at the PARSA lab at EPFL at the beginning of my PhD. In particular, I would like to thank Babak Falsafi for accepting me for the opportunity. You were all highly inspirational and provided motivation and inspiration for continuing the PhD journey.

For igniting my interest in following a research path and giving me faith that I could do it, I owe my deepest gratitude to my former master's thesis advisors from the University of Copenhagen, Brian Vinter, and Kenneth Skovhede. Your support and encouragement were invaluable and I would not be here without you! Additionally, I would like to thank my other colleagues and friends from the now-defunct eScience group at the Niels Bohr Institute at the University of Copenhagen for lots of great times and fun Friday beers at Søernes Ølbar.

To my other friends from the Computer Science department at the University of Copenhagen, in particular Henrik, Ben, Oleks, Michael, Troels, Ida, Joachim, Christian, Martin, Morten, Klaus, Karl, and Alberthe – thank you so much for being a steady part of my life throughout the years, being a constant source of support and sharing countless wonderful experiences.

My most loving and most heartfelt gratitude and appreciation goes to Gala who is my greatest source of inspiration and support. Your unwavering love, support, and persistent belief in me was my greatest asset in this journey. Even during difficult times, you always encouraged me to keep going and reassured me that I could do it.

Finally, my warmest gratitude goes to my parents, Bente and Bjørn. and my brother, Mads, for their loving support and encouragement both to start and complete this journey and throughout my life. Long before starting this journey, you were my greatest role models, and sources of persistent inspiration, support, and love. I am forever grateful for the solid foundation you gave me in life. I am also deeply grateful for my uncle, Geir, and his valuable insights and support both in life and the thesis journey

Contents

Part I

Research Overview

Chapter 1

Introduction

1.1 Motivation

The emergence of cloud computing has driven a landslide change in the computing landscape where computing resources are increasingly centralized in the hands of a few global providers. The cloud computing revolution is driven by the developers' desire for simple, flexible, and scalable hosting with no up-front investment. Function-as-a-service (FaaS) computing, the core technology powering serverless computing, is a cloud programming model that is currently seeing rapid growth. FaaS promises simplicity, elasticity, and cost-efficiency by moving deployment and resource management decisions from the user to the provider. Currently, half of all organizations that rely on cloud computing deploy applications using the FaaS programming model [14]. In the FaaS model, the fundamental unit of computation is a *function*. A FaaS function is a stateless unit of code that is executed on demand when triggered by an external event, such as an incoming HTTP request or a timer tick. Thus, when developing a FaaS function, the developer only needs to implement its functionality, leaving all the deployment-level decisions to the provider. When an event associated with a FaaS function is triggered, the provider must allocate resources and spawn the target function in order to handle the request. When the FaaS function finishes its execution, it is torn down, and the resources allocated for its execution are freed, possibly after the expiry of a grace period to prevent repeatedly starting the function in case it receives requests with a short inter-arrival time. This ephemeral deployment structure makes FaaS fundamentally different from the conventional cloud computing model where computing resources are provisioned in advance by the developer and services run as persistent daemons. Another aspect that distinguishes FaaS functions from conventional cloud services is that their running time is often very short, with 67% of functions studied in [19] having a runtime in the order of seconds. Furthermore, the current trend indicates that functions are getting shorter with the median function execution time reducing by 2× from 2019 to 2020 [13].

The unique deployment model and runtime characteristics of FaaS functions call for an investigation into how FaaS interacts with current processors. Notably, performance-critical microarchitectural structures, such as caches, branch predictors, etc. are designed to exploit the temporal locality in the access stream. However, given the short execution times of FaaS functions, it is not clear if such microarchitectural structures get enough time to warm up before the function execution finishes. Further, though function execution times are usually short, we observe a large variance with some functions executing for less than a millisecond (ms) while others take a few seconds. This thesis aims to understand how the microarchitectural bottlenecks shift with the execution time variations and the large degree of function interleaving exhibited by real-world deployments.

One of the key outcomes of our FaaS microarchitectural bottleneck analysis is that the functions are mainly bottlenecked at the core frontend irrespective of whether they execute for less than a ms or a few seconds. Notice that this frontend bottleneck is a big performance limiter in conventional long-running server applications as well [22, 29]. Therefore, any mechanism to mitigate the frontend bottleneck is likely to improve the performance of

both the FaaS functions and conventional server applications. The research community has proposed a myriad of diverse methods to mitigate the front-end bottleneck [3, 5, 6, 20, 21, 30–33, 35–37, 43, 50, 51]. Further, prior work [27, 37] has shown that a fetch directed instruction prefetcher (FDIP), when driven by a large enough BTB, provides near ideal performance. However, a large BTB leads to significant storage costs with modern processors dedicating 100s of KBs for it [25]. This thesis aims to maximize BTB reach, without prohibitive area and storage overheads, to mitigate the frontend bottleneck.

The unique modularity, compositionality, and language-independence of FaaS functions make the FaaS computing model highly attractive to developers [57]. To maximize these benefits, developers are encouraged to keep each FaaS function small with a narrowly defined purpose. This also contributes to the aforementioned very short running time of FaaS functions [17, 39, 44, 46, 47]. Additionally, since providers charge for the CPU-time needed to execute a FaaS function, developers also have a strong monetary incentive to keep the running time of functions short [60]. Larger and more complex FaaS applications are built by composing multiple smaller FaaS functions. Since a FaaS function can be written in any language depending on what best fits its purpose, the functions comprising a FaaS application normally communicate using network-backed Remote Procedure Calls (RPC) interfaces [23]. In addition to facilitating language-independence, these loosely-coupled RPC-backed communication interfaces also make it possible to transparently replace parts of an application and rewrite it piece by piece without discarding tested and reliable components. Language independence is a critically important property of FaaS as 96% of large organizations that use FaaS deploy functions written in two or more different languages[14]. While this RPC-backed communication fabric enables many of the most attractive features of the FaaS model, it also introduces a considerable communication overhead that is highly detrimental to the performance of FaaS applications [55]. Due to the detrimental performance impact that the communication overhead has on FaaS functions, this thesis aims to reduce it by introducing a software-based transformation for FaaS functions.

1.2 Research overview

The interplay between hardware and software in computing systems is complex and multi-faceted. Software optimizations can only go so far until they are fundamentally limited by hardware restrictions. Likewise, hardware optimizations cannot alleviate fundamental overheads in software. Furthermore, introducing workload-tailored hardware optimizations is always a trade-off between the ensuing increase in complexity and the scope of workloads that benefit from the optimization.

In that setting, the guiding research statement of this thesis is to:

> **Identify the sources of inefficiencies in the FaaS application execution model, determine their root causes, and investigate hardware and software mechanisms to alleviate them.**

To that end, the research presented in this thesis takes two directions. The first research direction, RD-A, focuses on understanding the microarchitectural behavior of FaaS functions and leveraging this understanding to investigate microarchitectural features for optimizing their execution. The second research direction, RD-B, focuses on reducing the RPC-induced

inter-function communication overhead in FaaS applications by introducing a software-level transformation.

1.2.1 Research direction A: Hardware Optimizations for Serverless Computing

The first step in solving a problem is to gain an in-depth understanding of its characteristics. We begin approaching this research direction with a study that performs an in-depth characterization of how one of the unique features of FaaS functions, their very short running times, interact with current processors. Previous work has investigated the impact of FaaS on current processors and concluded that FaaS has a significant negative impact on current microarchitectures [44, 46]. In particular, Schall et al. [44] finds that microarchitectural state thrashing occurs due to the short-running time of FaaS functions and the heavy interleaving that occurs when many different functions are queued for execution on a single processor core. However, they stop short of investigating *why* FaaS functions have this impact on common microarchitectures. To address this, we investigate which properties of FaaS functions make them susceptible to the negative effects of microarchitectural state thrashing.

From this work, we learn that only functions with very short execution times (<1 ms) benefit from being executed on a processor with a warm microarchitectural state. In real-world workloads, such functions are uncommon [39, 47] and therefore, adding a specialized hardware optimization to accommodate them would not be worthwhile. Furthermore, the analysis shows that FaaS functions are heavily frontend-bound. This corroborates similar observations made in previous work [23, 44]. Thus, to improve execution performance in the general case for serverless functions we are strongly motivated to continue working on eliminating the front-end bottleneck.

To that end, we propose BTB-X, a storage-efficient Branch Target Buffer (BTB) (see Section 2.2) organization for servers. The starting point of this work is that Fetch Directed Instruction Prefetchers (FDIP) have superior performance when given sufficient BTB capacity [27]. Considering the vast branch working sets of contemporary server workloads, industry has responded by allocating large amounts of storage to the BTB [9, 40, 53] with resulting sizes reaching up to 500KB of on-chip storage [25]. This amount of space means that large chunks of chip area and power budget are dedicated to the BTB. Further, considering that instruction set sizes show no sign of slowing down with Google observing 20% annualized growth rates [29], it is infeasible to respond to this simply by increasing the BTB storage budget. BTB-X defies this trend by leveraging a key insight about the distribution of branch target offset sizes. A conventional BTB (Conv-BTB) stores the full target address of every branch it holds. We make a key observation that most branch targets are close to the branch instructions themselves. Therefore, we can achieve significant storage savings by storing branch target offsets instead of the full target addresses in the BTB. We further observe large variability in the target offset sizes, i.e., some offsets are much larger/smaller than others. Thus, a BTB design should be able to accommodate this variability to maximize storage efficiency. The resulting design improves the state of the art in terms of branch storage density, lookup latency, and power requirements. Since BTB-X increases the branch working set coverage of the BTB, it also improves instruction prefetching performance (see Section 2.2). Thereby, it helps alleviate the front-end bottleneck and improves the performance of, particularly, server workloads.

The motivating analysis is presented in Paper A1 (Chapter 5). The design of BTB-X evolved over time and is described in three papers. The first version of BTB-X is described in paper

A2 (Chapter 6), an improved version is presented in the short paper A3 (Chapter 7) and the final design is presented in the full length paper A4 (Chapter 8).

Paper A1 **Impact of Microarchitectural State Reuse on Serverless Functions**
Truls Asheim, Tanvir Ahmed Khan, Baris Kasicki and Rakesh Kumar
In: Proceedings of the Eighth International Workshop on Serverless Computing, 2022

Paper A2 **BTB-X: A Storage-Effective BTB Organization**
Truls Asheim, Boris Grot and Rakesh Kumar
In: IEEE Computer Architecture Letters, Volume: 20, Issue: 2, July-December 2021

Paper A3 **A Specialized BTB Organization for Servers**
Truls Asheim, Boris Grot and Rakesh Kumar
In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2022

Paper A4 **A Storage-Effective BTB Organization for Servers**
Truls Asheim, Boris Grot and Rakesh Kumar
In: IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2023

1.2.2 Research direction B: Software Approaches for Optimizing Serverless Computing

The second research direction investigates software optimizations for serverless computing. The software inefficiencies in serverless computing mainly have two origins. First of all, since resources for executing a FaaS function are allocated transiently, starting a FaaS function for the first time is marked by a significant latency known as the cold-start latency. A large body of previous work investigates this issue, for example [17, 56]. Another well-documented performance challenge in FaaS is the communication overhead experienced by FaaS applications [23, 55]. As mentioned previously, the loose coupling between the individual functions comprising a FaaS application means that slow, high-latency network-backed RPC interfaces are used for inter-function communication. Because of the impact of this problem, a large body of previous work has addressed this issue [7, 28, 34, 39, 48, 52]. These proposals are all effective at reducing the FaaS inter-function communication overhead to varying degrees. However, we observe that each of these approaches compromises with at least one of the critical properties that make FaaS attractive to developers. Some approaches require all functions in a FaaS application to be written in a specific language, compromising the developers' ability to freely mix and match implementation languages of functions. Other approaches require existing functions to be rewritten to use a specialized, non-standard API for communication. Finally, some approaches propose entirely new runtimes that can be difficult to integrate with existing FaaS computing platforms.

In response to this observation, we introduce CoFaaS, a software-based, fully automatic, transformation for FaaS applications that significantly reduces the inter-function communication overhead without sacrificing any of the essential properties that make FaaS attractive to developers. CoFaaS is based on the observation that an individual FaaS function must

have a formally defined external interface in order to preserve the modularity of FaaS and support composition with other functions. This means that, as long as we preserve a function's external interface, we can freely transform it. CoFaaS exploits this to retarget FaaS functions to a platform-neutral and language-independent bytecode language known as WebAssembly (Wasm) [4]. The CoFaaS transformation turns each FaaS function into a CoFaaS component. Multiple CoFaaS components can then be composed, yielding a complete application, that can be executed on a single Wasm runtime. Our evaluation shows that the CoFaaS transformation improves inter-function communication overheads by up to 100× without sacrificing any of FaaS' most attractive properties.

The complete CoFaaS design and implementation is described in Paper B1 (Chapter 9):

Paper B1 **CoFaaS: Automatic Transformation-based Consolidation of Serverless Functions**
Truls Asheim, Magnus Jahre and Rakesh Kumar
Submitted to: Eurosys 2024

1.3 Thesis Overview

This thesis is structured in two parts. Part I introduces the topic matter of the thesis and positions its contribution. Chapter 1 introduces the topic of the thesis, Chapter 2 provides background material that aides understanding and contextualizing the topics of the included paper, Chapter 3 summarizes the research contributions of the thesis in more details and finally Chapter 4 presents the thesis' conclusions. Part II contains the scientific publications that collectively describe the contributions of the thesis.

Chapter 2

Background

In this chapter, we introduce the contextual background that surrounds this thesis. In particular, we focus on microarchitectural optimizations specifically targeting data center workloads and we give an in-depth introduction to serverless computing, what makes it unique, and the challenges surrounding the execution of serverless functions.

2.1 Serverless Computing

Over the past couple of decades, cloud computing has revolutionized the way computing resources are managed. Traditionally, developers were in charge of manually provisioning the computing resources that they needed ahead of time. Further, following the traditional model for managing computing resources, practitioners needed to consider how to handle peak application demand. Scaling an entire hosting infrastructure only to meet peak demand would lead to massive amounts of wasted resources. On the other hand, not meeting peak demand leads to unacceptably poor application performance when it is needed the most. Furthermore, features such as distributing an application across diverse geographical locations were out of reach of all but the largest companies. The foundations of cloud computing first became publicly known when Google began to publish papers about their internal architecture starting from 2003 [8, 15, 24, 42]. These papers outlined the technological underpinnings of a large-scale elastic cloud computing service powered by a large aggregation of commodity computing hardware.

Since Amazon released the first publicly available cloud offering, EC2, in 2006 [42] the popularity of the cloud has exploded and a plethora of products have been released to meet practitioners' never-ending demand for flexible, infinitely scalable, maintenance-free and, cost-effective computing. Following this trend, serverless computing is a recent addition to the cloud computing landscape that is quickly gaining popularity. Serverless computing promises developers quick and easy application deployment without having to consider how the application is hosted. Serverless is somewhat of a misnomer since applications developed using this model are still executed on a server. The name refers to the fact that the developer is not in charge of deploying the application on a server, they simply provide the application code and from there they leave it up to the provider to decide how the code is executed. In other words, serverless abstracts away the notion of a server from the developer. The programming model that underpins serverless computing is known as Function-as-a-Service (FaaS). A FaaS function is a stateless unit of code that is executed on demand by the provider when it is needed to handle an external event, such as an HTTP request. FaaS functions are commonly provided by the user as a Docker [16] container image [14]. When an event targeting a FaaS function is received, the following sequence of events occurs:

1. The function is scheduled for execution on a host of available resources.

2. When the function gets to the end of the execution queue, the host executing the function must load the image containing the function.

3. The provider brings the image up in a lightweight virtual machine such as Firecracker or gVisor.

4. The function executes and returns its response.

5. The function is shut down after execution and the resources allocated to it are freed.

FaaS functions have a number of properties that make them unique compared to conventional cloud workloads. First of all, they generally have very short runtimes. A study performed on the Microsoft Azure cloud [47] showed that half of all FaaS functions executed within 1 second while 90% of all functions executed within 10 seconds. Another study [19] corroborated by concluding that 67% of functions studied ran for just a few seconds. Furthermore, the runtime of functions appears to be decreasing with the median function execution time reduced by 2× from 2019 to 2020 [13]. The reason for the sort execution time of FaaS functions is that developers are encouraged to keep functions simple and only perform a single purpose. This principle is derived from microservice architectures [23] and aims to ensure that FaaS functions remain reusable and composable. These two principles are essential, as larger, more complex FaaS applications are constructed by composing multiple smaller single-purpose functions. Another essential property that makes the FaaS programming model attractive to developers is language neutrality. This property enables FaaS functions to be written in any programming language and ensures that functions written in different programming languages can be composed into a single FaaS application. As mentioned in the introduction, language independence is a treasured property of FaaS as 96% of large organizations that use FaaS deploy functions written in two or more different languages[14].

To communicate, FaaS functions commonly use network-backed RPC interfaces. They do this for two reasons:

1. The network-backed interface allows a multi-function FaaS application to be distributed across more than one node

2. Network-backed RPC interfaces are language-independent and thus they allow FaaS functions written in different languages to communicate

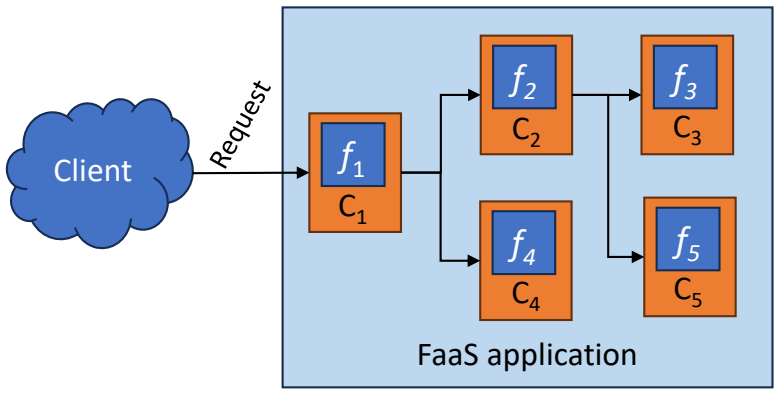


Figure 2.1: A schematic of the execution of a multi-function FaaS application.

Even though the network-backed RPC interface facilitates many of the advantages of the FaaS programming model they suffer from large communication overheads. In complex real-world FaaS applications, this communication overhead compounds to become the dominating source of execution overhead in FaaS applications [23, 55]. To illustrate why this is the case, we consider an example of a FaaS application composed of multiple FaaS functions in Figure 2.1. Each of the functions, $f_{1..5}$ are deployed in a separate container $c_{1..5}$ and they communicate via a network-backed RPC interface. The arrows denote the dependencies within the application. When the client issues a request to f_1 , the function subsequently issues a request to f_2 that again issues a request to f_5. Thus, we pay the communication latency penalty at least twice when issuing a request to this function. Keeping in mind that the latency of an RPC call to another function can be in excess of 1 ms, it becomes clear how this communication overhead can cause significant performance deterioration for FaaS applications. Further, the example we show here is comparatively small. Real-world FaaS applications can be composed of tens or hundreds of individual FaaS applications.

2.2 Instruction Delivery in the Datacenter

Datacenter workloads are distinguished by having massive multi-megabyte instruction footprints. These workloads generally have multi-layered software architectures with deep dependency stacks. Consider, for example, a server hosting a website. Processing a request to the website involves invoking the web server, the interpreter of the backend language powering the website, and a database server. All of these components also involve a large amount of system calls that perform context switches and execute a significant amount of kernel code [2, 22]. These large code footprints overwhelm the capabilities of current processor frontends causing a well-documented phenomenon known as the frontend bottleneck [6, 22, 29, 35–37, 51].

The frontend bottleneck occurs when the frontend of the processor is unable to deliver a continuous stream of instructions to the backend. Instruction delivery interruptions from the frontend can occur for two reasons: 1) the frontend mispredicts the next instruction due to a branch misprediction or a branch target miss or 2) the frontend encounters an instruction cache miss.Recovering from either of these events causes significant performance deterioration by causing long pipeline stalls that prevent the processor from making progress for tens of cycles. When a branch misprediction occurs, the instruction stream must be resteered onto the right-path execution. Doing this requires a pipeline flush, exposing the core to a pipeline fill latency of tens of cycles. L1-I cache misses are even more expensive exposing the core to the cache fill delay occurring while the right-path instructions are fetched into the instruction cache.

Traditionally, the instruction cache prefetching capabilities of processor frontends consisted of only next-line prefetching (NLP) [49]. In NLP, the next instruction cache block is prefetched unconditionally. While this technique provides miss coverage for linear code execution, it is incapable of covering the discontinuous control flows with large spatial footprints that are common in server processors. To address this, researchers have proposed numerous hardware mechanisms for instruction prefetching. We generally subdivide these into two categories: temporal streaming and fetch-directed prefetching. We describe these in the following two sections.

2.2.1 Temporal Streaming

Fundamentally, temporal streaming-based prefetching exploits the repetitiveness of program control flow. The intuition behind this is that even if a program can perform several operations, the instruction sequence executed to perform a particular operation is highly repetitive. This principle enables instruction prefetching through a record and replay approach. Essentially, this approach records streams of instructions and subsequently replays them to generate prefetching targets when a starting instruction is accessed. Ferdman et al. [21] proposed Temporal Instruction Fetch Streaming (TIFS), the first prefetcher to exploit this principle. While TIFS is effective at avoiding instruction cache misses it requires a massive amount of metadata storage to save the recorded instruction access streams. Proactive Instruction Fetch [20] improves on both the performance and metadata storage requirements of TIFS by recording compacted retire-order instruction streams. Still, it requires large amounts of metadata storage of 200KB per core.

To summarize, prefetchers based on temporal streaming are highly effective at eliminating L1-I misses but they do so at the cost of requiring large amounts of metadata storage and complex dedicated logic.

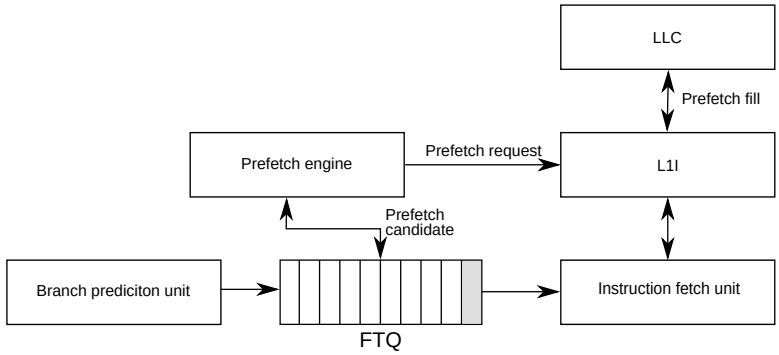


Figure 2.2: A schematic of the FDIP architecture.

2.2.2 Fetch-directed Prefetching

Reinman, Calder, and Austin [43] proposed a prefetching mechanism known as Fetch-Directed Instruction Prefetching (FDIP). The key insight of FDIP is that the BTB captures program control flow by storing branches and their targets. Therefore, it already contains the metadata needed to identify program control flow divergences and perform instruction prefetching accordingly. A schematic of FDIP is shown in Figure 2.2. FDIP decouples the branch prediction unit from the fetch unit using a dedicated queue known as the Fetch Target Queue (FTQ). This decoupling allows the branch prediction unit to run ahead of program execution and fill the FTQ with locations of the future predicted control flow. Thus, the targets in the FTQ are suitable as instruction cache prefetch candidates since they correspond to the future control flow of the program being executed. In spite of FDIP's alluring simplicity and absence of dedicated metadata storage, conventional wisdom long held that FDIP was incapable of effectively covering instruction cache misses in commercial server workloads. This was considered the case for two reasons: 1) Realistically sized BTBs are too small to capture the entire branch working set of server workloads. This causes frequent BTB misses

and the resulting frequent pipeline flushes prevent the prefetching from working optimally. 2) Branch misprediction probability compounds geometrically with the number of branches predicted. Thus, it is infeasible that future control flow predictions generated by the branch predictor would stay on the right path for long enough to cover the large LLC fetch delay encountered in modern processors [21].

Kumar et al. [37] introduced Boomerang, an instruction prefetching approach that challenged the conventionally held wisdom claiming the inadequacy of FDIP prefetchers. Boomerang takes a twofold approach: Firstly, it pre-decodes blocks fetched into the instruction cache to identify and extract the branches they contain. The extracted branches are used to pro-actively pre-fill the BTB. Secondly, Boomerang leverages a basic-block[1] based BTB [54]. Since more than one branch may have targets that lie within the same basic block, this BTB organization increases the BTB storage density and reduces BTB misses. Shotgun [35, 36] improves the miss coverage of Boomerang further by introducing an optimized BTB organization consisting of three structures. The U-BTB stores unconditional branches, the Return Instruction Buffer RIB) tracks return instructions, and the C-BTB stores conditional branches. The key insight behind this BTB organization is that unconditional branches tend to jump far from their origin as they typically represent function calls, possibly spanning different libraries. Conditional branches, on the other hand, tend to have targets close to their origin since they typically represent control flow within a function, such as an `if` block or a loop. Furthermore, program control flow often follows a pattern where a (long) unconditional jump precedes a sequence of multiple (short) conditional jumps. To effectively track the code footprints referenced by conditional jumps, each entry of the U-BTB tracks the spatial footprints of instructions accessed by conditional jumps around the target and return addresses of the unconditional jump. When an unconditional branch in the U-BTB is predicted taken, the code referenced by the associated spatial footprints is prefetched and the contained conditional branches are decoded and pre-filled into the C-BTB. The Shotgun BTB organization significantly improves the storage density and accuracy of the BTB by allowing it to track a larger portion of the branch working set. This improves the efficiency and accuracy of FDIP.

These results strongly support the viability of FDIP instruction prefetchers and disprove the conventional wisdom deeming them unsuitable for covering the massive instruction footprints of server workloads. This view is supported by a recently presented industrial perspective from ARM. Ishii et al. [27] presents an FDIP prefetching mechanism and argues for its viability in commercial processors because it requires no additional metadata and has a minimal implementation complexity overhead. Notably, their mechanism outperforms the winner of the recent Instruction Prefetching Championship (IPC-1) [1]. Based on this, they argue that their FDIP implementation should be considered the new baseline that instruction prefetching research proposals are compared against. Further, they strongly emphasize the importance of ensuring sufficient BTB capacity in modern processors as FDIP-based prefetchers are superior from both a performance and implementation perspective when coupled with a sufficiently large BTB.

[1] A basic-block is here defined as a sequence of straight-line instructions ending with a branch. We note that this definition differs from the conventional basic-block definition covering straight-line code starting from an instruction targeted by a branch and ending with a branch instruction.

Chapter 3

Research Contributions

The research contributions of this thesis are presented in five articles. The first four (Chapters 5, 6, 7 and 8) are published and the final paper (Chapter 9) is pending review at the time of thesis submission. In this chapter, we summarize the contributions of each paper and contextualize the papers' prior work. The rest of this chapter summarizes our research contributions starting from the research directions introduced in Section 1.2.

3.1 Research Direction A: Hardware approaches to optimizing FaaS

We present our results in two parts where the first partI (Section 3.1.1) covers the background study, presented in Paper A1 (Chapter 5) and the second part (Section 3.1.2) describes BTB-X, the proposal for a redesigned BTB proposal covered in Papers A2, A3 and A4 (Chapters 6, 7 and 8).

3.1.1 Understanding FaaS

Motivation. Function-as-a-service (FaaS) computing is a rapidly growing cloud computing model that challenges fundamental assumptions about the design of computing systems. In FaaS, the fundamental unit of computation is a function, often with a very short execution time. A FaaS function is loaded and spawned on demand when it is needed to respond to an event. When the FaaS function has finished execution, it is shut down and the resources allocated to its execution are freed. This forces providers to co-schedule and queue thousands of different functions for execution on the same processor cores to maximize server utilization. This causes heavy execution interleaving and means that if, for example. functions A and B are executed in the sequence ABA, function A will not find much or any of its microarchitectural state left behind the second time it is executed. The thrashing of the microarchitectural state caused by interleaved execution means that most invocations of FaaS functions happen from a cold microarchitectural state. Recent work has noted this and reported that the interleaved execution of FaaS functions causes significant performance deterioration [44, 46]. However, no prior work investigates which specific properties FaaS functions have that make them particularly vulnerable to the impact of state thrashing. Is it, for example, dependent on the function code footprint size or the point where a function has a sufficiently long execution time to amortize the warm-up delay of the microarchitectural state?

Approach. To address this gap in the research, we evaluate FaaS functions executed on a processor with warm and cold microarchitectural states. We use a representative suite of FaaS functions consisting of both real-world and synthetic workloads. Using synthetic workloads allows us to modify specific parameters of interest in a controlled way to determine their impact. To measure the impact of executing the functions on a warm and cold microarchitectural state we execute the functions in two modes: back-to-back and interleaved. In back-to-back execution, the functions are executed repeatedly in a tight loop. In the interleaved execution case, each invocation of a function is interleaved with an invocation of a *trhasher* function

that thrashes all existing microarchitectural state. The thrasher function achieves this by performing two actions: 1) it invokes the x86 WBINVD instruction that writes back and invalidates the entire cache hierarchy of the processor package [12, Chapter 6] and 2) it runs a function that executes a long sequence of branches to thrash the state of the BTB and the branch predictor. We measure microarchitectural parameters using perf [18] and use the Top-Down methodology [59] for identifying specific microarchitectural bottlenecks. For both the back-to-back and interleaved executions we measured both the wall-clock running time of the functions and relevant microarchitectural parameters.

To support our findings (presented below) we measure the code footprint of each of the FaaS functions that we use in our analysis. Doing this using a full microarchitectural simulator is a slow and tedious process. Further, it is impossible to perform this analysis statically since the dynamic instruction trace of a program is only known at runtime. Therefore, we use a method described in [58] that uses branch-record traces and a cache simulator to estimate the code footprint of a program. The branch-record traces show the branches taken by a program during execution. By feeding the instruction code regions demanded by the targets of the branches in the trace through a cache simulator, we can measure how big the cache has to be to fit the entire code footprint of a program. Specifically, the method increase the size of the simulated cache until the observed miss-rate reaches zero. When this is the case, the entire code footprint of the program fits inside the cache.

Key Results. Our results show that two properties of a FaaS function impact how sensitive it is to being executed on a cold microarchitectural state: Its execution time and its code footprint. For example, the shortest running function that we evaluate (0.25 ms) shows a slowdown of 17× when executed on a processor with a cold microarchitectural state. The longest running functions are largely unaffected by microarchitectural state thrashing. Since 99% of real-world FaaS invocations run for more than 1 ms [47], our results indicate that the warm-up latency of microarchitectural structures is amortized for the vast majority of FaaS functions.

Another important observation we make from our experiments is that the real-world FaaS functions we evaluate are overwhelmingly front-end bound. This is in line with similar observations made for general server workloads [6, 22, 29] and corroborates similar observations made for FaaS functions specifically [44]. This strongly motivates further research aiming to alleviate the front-end bottleneck to optimize the execution of FaaS functions.

3.1.2 A storage-efficient BTB organization for servers

Motivation. The front-end bottleneck is a well-known and widely documented problem affecting processors executing server workloads [2, 6, 22, 29]. Compared to other workload classes, server workloads pose particular challenges for processor frontends due to their very large instruction footprints. These footprints quickly overwhelm the capacities of the L1-I and the Branch Target Buffer (BTB). To handle this problem, computer architects have proposed a large number of different approaches. In particular, L1-I prefetchers are a highly effective way to alleviate the front-end bottleneck as they help avoid the fill-latency that follows an L1-I miss. As detailed in Section 2.2, a particularly effective class of instruction prefetchers is known as Fetch Directed Instruction Prefetching (FDIP) [43]. Since FDIP relies on the BTB to predict the control flow of a program and identify prefetch candidates, having a BTB with sufficient capacity is crucial. By simply scaling the design of a conventional BTB design, the capacity required for holding realistic branch working sets of contemporary server workloads

quickly becomes infeasible. To address this, previous work observed that all branch targets within a page share the same numbers and only differ in page offsets. Thus the page number can be deduplicated by only storing it once in a dedicated table and then replacing its use in a branch target address with a pointer to that table [45]. Soundararajan et al. [50] extend this concept further by observing that several branch page numbers share a common region number which can be additionally deduplicated. The limitation of these approaches is that they add a level of indirection to BTB lookups, adding complexity and increasing the BTB access latency.

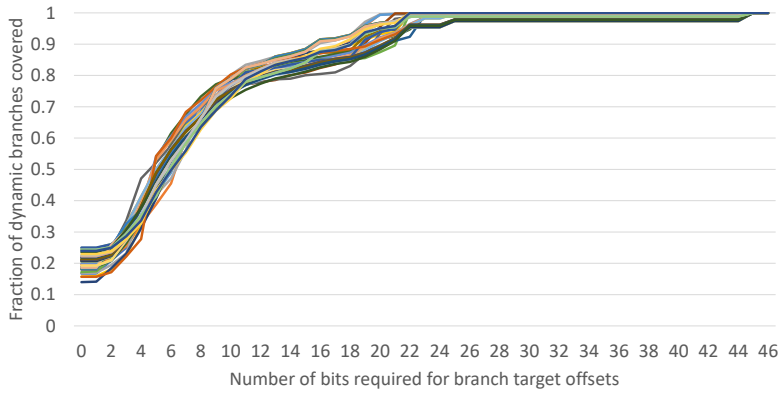


Figure 3.1: Distribution of branch target offsets in the IPC-1 [1] workload traces

Approach. To gain a deeper understanding of the characteristics of branch targets, we performed an analysis of the branch target offsets in a large number of real-world workload traces released for the first Instruction Prefetching Championship (IPC-1) [1]. A branch target offset is the relative distance from a branch to its target. From this analysis, shown in Figure 3.1, we observe that the vast majority of branches have targets very close to their origin. For example, using just 19 bits we can represent the target offsets of 90% of all branches. This is significantly less than the 46 bits[1] required to store the full target address.

Based on this observation we propose BTB-X, a reorganized BTB that uses differently sized ways to match the distribution of target offset sizes that we observed. The full design of BTB-X is shown in Figure 3.2. To accommodate branch targets that require the full address size, we use a small conventional BTB called BTB-XC. When accessing the BTB, BTB-X and BTB-XC are looked up in parallel. Performing a BTB-X lookup is similar to performing a lookup in a conventional BTB. The key difference is that the BTB-X lookup yields a target offset rather than the full target address. Thus, the target offset must be concatenated with the branch PC to obtain the full target address. When allocating an entry, BTB-X stores the target offset in the first free way where it will fit.

We implement and evaluate BTB-X using the Champsim simulator. Champsim is a trace-based simulator used for microarchitectural studies [10]. For evaluation, we use the aforementioned workload traces released as part of the IPC-1 championship. The traces were provided by Qualcomm and include both server and client workloads. For the BTB-X design to apply to all workloads, the branch target offset size distribution that guided its design

[1]This is assuming an Arm-64 architecture where all instructions are 4 bytes long. Thus, we save the last two offset bits of the full 48-bit virtual address space.

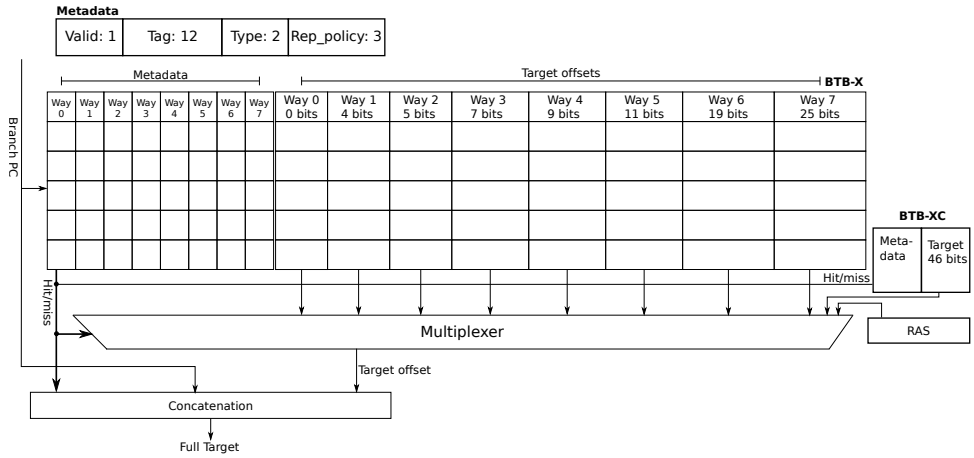


Figure 3.2: The design of BTB-X.

must be shown to be widely applicable. To ensure this, we measure the branch target offset size distributions of the more than 750 traces provided by Qualcomm as part of the First Championship Value Prediction CVP-1 and traces from 6 known server applications. These results show nearly identical branch target offset size distributions. Finally, we measure the energy requirements and access latencies of BTB-X, the state-of-the-art BTB design, and a conventional BTB using Cacti 7.0 [38] at the 22 nm technology node.

Key Results. Our evaluation shows that BTB-X stores 2.24× more branches than a conventional BTB organization. Furthermore, when compared to the state-of-the-art BTB design PDede, BTB-X stores 1.24× and 1.34× more branches for storage budgets of 0.9KB and 59KB respectively. This increase in branch storage density translates to a significant decrease in BTB MPKI. Specifically, BTB-X lowers BTB MPKI to 9.3 compared to 25 for a conventional BTB. When looking at the overall performance gain of introducing an FDIP instruction prefetcher using different BTB configurations we see that FDIP produces geomean speedups of 24%, 33%, and 38% for Conv-BTB, PDede, and BTB-X, respectively. Estimates of the power requirements of BTB-X show that reading and writing require 8.5 pJ and 11.4 pJ respectively. This compares favorably to the up to 9.4 pJ and 19.5 pJ required by PDede and 13.2 pJ 25.2 pJ by Conv-BTB respectively. The access latency of BTB-X is 0.33 ns compared to 0.36 ns and 0.47 ms for Conv-BTB and PDede respectively.

We see that BTB-X outperforms the state-of-the-art BTB organization PDede while, at the same time, proposing a simpler design that avoids BTB access indirections.

3.2 Research direciton B: Software optimizations for serverless computing

Motivation. The network-backed RPC interfaces commonly used to facilitate inter-function communications in FaaS applications are a key facilitator of the loose coupling between functions that give FaaS many of its attractive benefits [23]. The RPC interface allows FaaS functions written in different languages to communicate with each other and it allows functions to be swapped out at runtime as long as their external interfaces are preserved. Furthermore,

since RPC protocols can use a networked transport, FaaS functions communicating over RPC can be transparently distributed across physical nodes. The major disadvantage of these network-backed RPC interlaces is that they induce a massive inter-function communication latency. Where the latency of a local function call in a monolith application is in the order of a few nanoseconds, RPC-backed function calls between FaaS functions in a FaaS application can take several milliseconds to execute – a difference of several orders of magnitude. When two FaaS functions scheduled for execution on different nodes need to communicate, using a networked RPC interface for communication is necessary. However, when two FaaS functions are scheduled on the same node, using the RPC interface for communication only adds overhead without giving any benefits. To tackle this issue, prior work has proposed diverse mechanisms aiming to reduce the inter-function communication overhead of FaaS [7, 28, 34, 39, 48, 52]. Common for all of these approaches, however, is that they violate one or more of the key properties of FaaS that make the programming model attractive to developers (see Section 2.1).

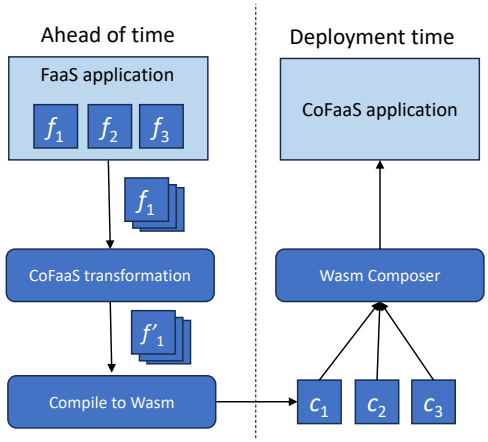


Figure 3.3: The compilation workflow required when using CoFaaS.

Approach. We propose CoFaaS, a software transformation of FaaS applications that eliminates the networked inter-function communication overhead. CoFaaS is unique in the solution space as it eliminates the FaaS communication overhead without sacrificing any of the key properties that make FaaS attractive to developers. The CoFaaS transformation is fully automatic and can be applied to existing FaaS applications without requiring manual modification of existing functions.

For FaaS functions to communicate, their public interfaces must be specified using language-independent Interface Definition Languages (IDL). For example, a popular IDL used in FaaS applications is Protobuf. The IDL definitions written in Protobuf [41] are used by Google's gRPC [26] framework. CoFaaS makes the key insight that, is that as long as we keep the public interface unchanged, we can transform the function implementation freely. Furthermore, since IDLs are declarative, they can be easily mapped to other IDLs. We exploit this to retarget FaaS functions run on a single WebAssembly (Wasm) [4] runtime. Wasm is a language-independent portable bytecode format with wide platform support. For the FaaS functions to co-exist on a single Wasm runtime, we transform the FaaS functions into a Wasm component. Wasm components [11], like FaaS functions, use a language-independent

IDL to define their public interface and since each Wasm component exists in its own isolated memory space, we can simply transform the IDL of the FaaS function to the IDL of Wasm and compile the functions to Wasm. The final point to consider is that the original functions are written to communicate over a network-backed RPC interface. To address this we do two things: First, we provide a stub library that provides API-level compatibility with the original gRPC library but instead of calling other functions using a networked RPC call, it uses the Wasm interface to call functions running on the same Wasm runtime.

For evaluating CoFaaS we use a two-function producer-consumer FaaS application that works as follows: A client issues a request to the producer function, then the producer function generates a data payload of m bytes and transmits this payload to the consumer function. Finally, the request from the producer to the consumer is repeated n times. By changing values for m and n evaluation we can estimate a) to what extent the latency reduction CoFaaS achieves is amortized by the data transfer happening between the consumer and producer and b) we can estimate the impact of CoFaaS on more complex applications consisting of more functions by varying the number of intra-application requests. We compare two versions of the application, one written in Go and one written in Rust.

Key Results. We evaluate CoFaaS using a two-function FaaS application that CoFaaS reduces the round-trip request time of the FaaS application that we evaluate by 6×. For a 1 kilobyte payload, the latency of inter-function requests is reduced by up to 100×. For larger payload sizes, the achieved speedup diminishes but remains significant, with a 2× speedup being achieved for the largest evaluated payload size of 512 kilobytes. We note that the median inter-function request payload size observed in real-world FaaS traces is 8 kilobytes [39], a size comfortably within the range where CoFaaS gives performance improvements For this size, CoFaaS gives a 20× speedup. Further, the round-trip time speedup increases when we increase the number of intra-application requests. This indicates that the relative advantage of CoFaaS increases with application complexity.

Introducing CoFaaS in a production environment adds only minimal overhead. This is because transforming a FaaS function into a composable CoFaaS component is simply another compilation step. The operation needed to compose CoFaaS components into an application is instantaneous and can easily be done at deployment time. The process of applying the CoFaaS transformation to a FaaS application is shown in Figure 3.3.

Chapter 4

Concluding Remarks

Function-as-a-Service (FaaS) is a highly attractive programming model due to its ease of deployment, zero provisioning overhead for developers and its ability to compose applications from functions written in multiple different programming languages. However, the short-running nature of FaaS functions challenges the assumptions underpinning the design of conventional microarchitectures. Additionally, the language-independence, modularity and deployment flexibility of FaaS functions are enabled by the use of high-overhead, network-backed RPC interfaces for inter-function communications. These interfaces add significant overhead to inter-function communication in FaaS applications which cause significant performance deterioration. In this thesis, we have analyzed and optimized the execution of serverless functions from both the hardware and software perspective. In doing so, we made the following key contributions to each of our research directions.

RD-A. We began exploring this direction by answering a key question about FaaS functions: How long does a function need to run before the warm-up delay of microarchitectural structures is amortized? Our analysis shows that only functions that run for a very short time 1 ms) see performance deterioration as a result of microarchitectural warm-up latency. We furthermore observe that this effect is exacerbated for FaaS functions with a large instruction footprint and that the functions in general have a large instruction footprint. The latter observation corroborates results from previous work and means that FaaS functions are affected by the front-end bottleneck in line with general server workloads. To address this we introduce BTB-X, an optimized BTB organization that stores branch target offsets instead of full branch targets. To improve storage density, BTB-X exploits the uneven distribution of branch offset sizes in workloads. The resulting design significantly improves BTB storage capacity without increasing storage requirements. This, in turn, improves the performance of instruction cache prefetchers that rely on BTB storage capacity.

RD-B. To address the excessive communication overhead in FaaS applications we introduce CoFaaS, a fully automated and software-based transformation of FaaS applications. Previous work targeting the same problem can effectively reduce the communication overhead. However, all previously proposed optimization sacrifices at least one of the essential properties that make the FaaS programming model attractive, for example by requiring that all functions in a FaaS application be written in the same language. CoFaaS practically eliminates the inter-function communication overhead without sacrificing any of FaaS' essential properties. The key insight exploited by CoFaaS is that we can use the well-defined RPC interfaces of FaaS functions to make code transformations that alleviate the inter-function communication overhead. As CoFaaS' mechanism of action consolidates multiple FaaS functions onto a single WebAssembly runtime, it also intrinsically addresses the performance deterioration that we observed for very short-running functions while working on RD-A.

Bibliography

[1] *1st Instruction Prefetching Championship*. https://research.ece.ncsu.edu/ipc/. 2020 (cit. on pp. 13, 17).

[2] Anastassia Ailamaki et al. "DBMSs on a Modern Processor: Where Does Time Go?" In: *Proceedings of the 25th International Conference on Very Large Data Bases*. 1999 (cit. on pp. 11, 16).

[3] Samira Mirbagher Ajorpaz et al. "Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018 (cit. on p. 4).

[4] Andreas Rossberg (editor). *WebAssembly Core Specification*. Tech. rep. Version 2.0. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. W3C, Apr. 19, 2022 (cit. on pp. 7, 19).

[5] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. "Divide and conquer frontend bottleneck". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020 (cit. on p. 4).

[6] Grant Ayers et al. "AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. Phoenix, Arizona: Association for Computing Machinery, 2019 (cit. on pp. 4, 11, 16).

[7] Daniel Barcelona-Pons et al. "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures". In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019 (cit. on pp. 6, 19).

[8] Luiz Andre Barroso, Jeffrey Dean, and Urs Hölzle. "Web Search for a Planet: the Google Cluster Architecture". In: *IEEE Micro* 23 (2003) (cit. on p. 9).

[9] James Bonanno et al. "Two level bulk preload branch prediction". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013 (cit. on p. 5).

[10] *ChampSim Simulator*. https://github.com/ChampSim/ChampSim. 2023 (cit. on p. 17).

[11] *Componet Model design and specification*. https://archive.ph/jHHgn. Accessed: 2023-10-18. 2023 (cit. on p. 19).

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2, Instruction set reference*. Version 325383-081US. Sept. 2023 (cit. on p. 16).

[13] Datalog. *The state of serverless*. https://www.datadoghq.com/state-of-serverless-2021/. 2021 (cit. on pp. 3, 10).

[14] Datalog. *The state of serverless*. https://www.datadoghq.com/state-of-serverless-2022/. 2022 (cit. on pp. 3, 4, 9, 10).

[15] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008) (cit. on p. 9).

[16] Docker. *Docker*. `https://docker.com/`. Accessed: 2023-11-06. 2923 (cit. on p. 9).

[17] Dong Du et al. "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020 (cit. on pp. 4, 6).

[18] Jake Edge. *Perfcounters added to the mainline*. `https://lwn.net/Articles/339361/`. July 1, 2009 (cit. on p. 16).

[19] Simon Eismann et al. "A Review of Serverless Use Cases and Their Characteristics". In: *CoRR* abs/2008.11110 (2020) (cit. on pp. 3, 10).

[20] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. "Proactive Instruction Fetch". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*. ACM Press, 2011 (cit. on pp. 4, 12).

[21] Michael Ferdman et al. "Temporal instruction fetch streaming". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. Nov. 2008 (cit. on pp. 4, 12, 13).

[22] Michael Ferdman et al. "Clearing the Clouds". In: *ACM SIGARCH Computer Architecture News* 40.1 (Apr. 2012) (cit. on pp. 3, 11, 16).

[23] Yu Gan et al. "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 '19. Providence, RI, USA: ACM, 2019 (cit. on pp. 4–6, 10, 11, 18).

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003 (cit. on p. 9).

[25] Brian Grayson et al. "Evolution of the Samsung Exynos CPU Microarchitecture". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020 (cit. on pp. 4, 5).

[26] gRPC. *A high performance, open source universal RPC framework*. `https://grpc.io/`. Accessed: 2023-10-19. 2923 (cit. on p. 19).

[27] Yasuo Ishii et al. "Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective". In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021 (cit. on pp. 4, 5, 13).

[28] Zhipeng Jia and Emmett Witchel. "Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021 (cit. on pp. 6, 19).

[29] Svilen Kanev et al. "Profiling a warehouse-scale computer". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*. 2015 (cit. on pp. 3, 5, 11, 16).

[30] Cansu Kaynak, Boris Grot, and Babak Falsafi. "SHIFT. shared history instruction fetch for lean-core server processors". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46*. 2013 (cit. on p. 4).

[31] Cansu Kaynak, Boris Grot, and Babak Falsafi. "Confluence". In: *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48* (2015) (cit. on p. 4).

[32] Tanvir Khan et al. "I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2020 (cit. on p. 4).

[33] Tanvir Ahmed Khan et al. "Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications". In: *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*. ISCA 2021. June 2021 (cit. on p. 4).

[34] Swaroop Kotni et al. "Faastlane: Accelerating Function-as-a-Service Workflows". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021 (cit. on pp. 6, 19).

[35] Rakesh Kumar and Boris Grot. "Shooting Down The Server Front-End Bottleneck". 2020 (cit. on pp. 4, 11, 13).

[36] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. "Blasting Through the Front-End Bottleneck With Shotgun". In: *ACM SIGPLAN Notices* 53.2 (Mar. 2018) (cit. on pp. 4, 11, 13).

[37] Rakesh Kumar et al. "Boomerang: a Metadata-Free Architecture for Control Flow Delivery". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2017 (cit. on pp. 4, 11, 13).

[38] Sheng Li et al. "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2011 (cit. on p. 18).

[39] Ashraf Mahgoub et al. "Wisefuse: Workload Characterization and Dag Transformation for Serverless Workflows". In: *Proc. ACM Meas. Anal. Comput. Syst.* 6.2 (June 2022) (cit. on pp. 4–6, 19, 20).

[40] Andrea Pellegrini et al. "The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC". In: *IEEE Micro* 40.2 (2020) (cit. on p. 5).

[41] Protobuf. *Protocol Buffers*. `https://protobuf.dev/`. Accessed: 2023-10-19. 2923 (cit. on p. 19).

[42] Ling Qian et al. "Cloud computing: An overview". In: *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*. Springer. 2009 (cit. on p. 9).

[43] G. Reinman, B. Calder, and T. Austin. "Fetch Directed Instruction Prefetching". In: *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Comput. Soc, 1999 (cit. on pp. 4, 12, 16).

[44] David Schall et al. "Lukewarm Serverless Functions: Characterization and Optimization". In: *Proceeding of the 49st Annual International Symposium on Computer Architecuture*. ISCA '22. New York, New York, USA: IEEE Press, 2022 (cit. on pp. 4, 5, 15, 16).

[45] André Seznec. “Don’t Use the Page Number, but a Pointer to It”. In: *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. ISCA ’96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996 (cit. on p. 17).

[46] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. “Architectural Implications of Function-as-a-Service Computing”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: Association for Computing Machinery, 2019 (cit. on pp. 4, 5, 15).

[47] Mohammad Shahrad et al. “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020 (cit. on pp. 4, 5, 10, 16).

[48] Simon Shillaker and Peter Pietzuch. “Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020 (cit. on pp. 6, 19).

[49] A.J. Smith. “Sequential Program Prefetching in Memory Hierarchies”. In: *Computer* 11.12 (1978) (cit. on p. 11).

[50] Niranjan K Soundararajan et al. “Pdede: Partitioned, deduplicated, delta branch target buffer”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021 (cit. on pp. 4, 17).

[51] Lawrence Spracklen, Yuan Chou, and Santosh G. Abraham. “Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications”. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. HPCA ’05. USA: IEEE Computer Society, 2005 (cit. on pp. 4, 11).

[52] Vikram Sreekanti et al. “Cloudburst: Stateful Functions-As-A-service”. In: *Proc. VLDB Endow.* 13.12 (July 2020) (cit. on pp. 6, 19).

[53] David Suggs, Mahesh Subramony, and Dan Bouvier. “The AMD “Zen 2” Processor”. In: *IEEE Micro* 40.2 (2020) (cit. on p. 5).

[54] Tse-Yu Yeh and Y. N. Patt. “A Comprehensive Instruction Fetch Mechanism For A Processor Supporting Speculative Execution”. In: *[1992] Proceedings the 25th Annual International Symposium on Microarchitecture MICRO 25*. Dec. 1992 (cit. on p. 13).

[55] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. “Analyzing Tail Latency in Serverless Clouds with STeLLAR”. In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 2021 (cit. on pp. 4, 6, 11).

[56] Dmitrii Ustiugov et al. “Benchmarking, Analysis, and Optimization of Serverless Function Snapshots”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’21)*. ACM, 2021 (cit. on p. 6).

[57] Christopher L. Williams et al. “The Growing Need for Microservices in Bioinformatics”. In: *Journal of Pathology Informatics* 7.1 (2016) (cit. on p. 4).

[58] S.C. Woo et al. “The SPLASH-2 programs: characterization and methodological considerations”. In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. 1995 (cit. on p. 16).

[59] 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)*. Mar. 2014 (cit. on p. 16).

[60] Tianyi Yu et al. "Characterizing Serverless Platforms with Serverlessbench". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020 (cit. on p. 4).

Part II

Publications

Chapter 5

Paper A1 – Impact of Microarchitectural State Reuse on Serverless Functions

Authors

Truls Asheim, Tanvir Ahmed Khan, Baris Kasicki and Rakesh Kumar

Published in

Proceedings of the Eighth International Workshop on Serverless Computing, 2022

Impact of Microarchitectural State Reuse on Serverless Functions

Truls Asheim[1], Tanvir Ahmed Khan[2], Baris Kasicki[2,3], Rakesh Kumar[1]

1) Norwegian University of Science and Technology, Norway
2) University of Michigan, USA
3) Google, USA

Abstract

Serverless computing has seen rapid growth in the past few years due to its seamless scalability and zero resource provisioning overhead for developers. In serverless, applications are composed of a set of very short-running functions which are invoked in response to events such as HTTP requests. For better resource utilization, cloud providers interleave the execution of thousands of serverless functions on a single server.

Recent work argues that this interleaved execution and short run-times cause the serverless functions to perform poorly on modern processors. This is because interleaved execution thrashes the microarchitectural state of a function, thus forcing its subsequent execution to start from a cold state. Further, due to their short-running nature, serverless functions are unable to amortize the warm-up latency of microarchitectural structures, meaning that most the function execution happen from cold state.

In this work, we analyze a function's performance sensitivity to microarchitectural state thrashing induced by interleaved execution. Unlike prior work, our analysis reveals that not all functions experience performance degradation because of microarchitectural state thrashing. The two dominating factors that dictate the impact of thrashing on function performance are function execution time and code footprint. For example, we observe that only the functions with short execution times (< 1 ms) show performance degradation due to thrashing and that this degradation is exacerbated for functions with large code footprints.

5.1 Introduction

Serverless computing (or Function-as-a-Service (FaaS)) is emerging as a prominent cloud computing model. Applications developed for the serverless model are structured using one or more stateless *functions* that are invoked in response to specific external events such as an HTTP request or a timer trigger. The underlying design philosophy of serverless applications

is descendant from microservices. As such, serverless application design heavily emphasizes modularity to enable compositionality and reusability of functions. This means that the execution time of most serverless functions is very short, often as low as a few milliseconds. However, unlike microservice applications, the resources needed for executing a serverless function are transiently allocated. This enables significant cost savings as the user only needs to pay for hosting resources when they are used.

Though function execution times are very short, the majority of functions are invoked very infrequently, often leaving a few seconds or minutes between two consecutive invocations. Thus, to improve resource utilization, cloud providers are forced to co-schedule thousands of functions on each server. However, the downside of such high degree of interleaving is the long startup delay of booting new function instances because a server can keep only a certain number of recently invoked function instances in warmed-up state [8, 9, 14]. Consequently, prior research has aimed at reducing this startup delay to improve the performance of serverless functions [4, 13]. The key idea is to quickly load an execution-ready image of the function into the main memory of the system.

Another downside of function interleaving, as reported by recent work [10, 11], is that interleaved execution thrashes the microarchitectural states of functions. This means that when a function is invoked it does not find any (or much) of its microarchitectural state from its last execution in the microarchitectural structures such as caches, branch predictors, etc. This is because the interleaved functions evict this state as they bring their own microarchitectural state in these structures. Further, prior work [11] also reports that the short execution time of serverless functions prevent them from amortizing the warm-up latency of microarchitectural structures. Consequently, the majority of function executions happen with cold microarchitectural state. As a result, serverless functions show poor performance.

This work analyzes the factors that make performance of serverless functions sensitive to interleaved execution induced microarchitectural state thrashing. We analyze both real-world serverless functions as well as synthetic functions with a wide range of execution times (from 0.25 ms to 1.1 s) and different implementation languages. Synthetic functions give us better control over function properties such as execution time, code and data footprints etc. Our results reveal that not all functions show performance degradation due to interleaving induced state thrashing; rather it depends on function properties. Our studies further identify function execution time and code footprint to be the two dominating factors that dictate the impact of thrashing on function performance. The execution time is a particularly interesting factor because real-world deployments report high variability in the execution time of different functions. For example, a study [12] reported that 50% of functions on the Azure cloud completed in less than 1s. Another study [3] found that 50% of functions deployed on AWS Lambda in 2020 completed in 60ms or less. However, the same study noted a decreasing trend in function execution time as 50% of functions completed within 130ms in 2019.

In our study, we find that only very short running functions (median runtime < 1 ms) are adversely affected by being executed on a cold microarchitectural state and that this trend is exacerbated proportionally with increasing function instruction working sizes. For functions with longer execution times (> 50 ms), we find that the performance deterioration caused by interleaved execution is small. This suggests that the microarchitectural state warm-up latency is amortized and the most of function execution happens from warmed-up state.

Table 5.1: The functions used for characterization.

Name	Language	Description
autocomplete	NodeJS	Returns a list of autocomplete candidates.
sentiment_analysis	Python	Identifies the sentiment of a text.
deltablue	Python	Pure-python compute benchmark.
markdown2html	Python	Converts markdown to HTML. Relies heavily on the sha256 implementation in the OpenSSL library.
json_dumps	Python	Serializes a Python dict as JSON
img-resize	NodeJS, libraries	Produces resized versions of images. All of the heavy-lifting in this function is done by native image libraries such as libpng and libjpeg
ocr-img	NodeJS, C++	Invokes Tessarect to OCR an image
dynamichtml	NodeJS	Generates HTML from a template
fib_js_NNN	NodeJS	calculates the NNN'th fibonacci number and returns it
fib_py_NNN	Python	Same as fib_js but implemented in Python.
footprint_NN	Python, C	Synthetic function that claims a code footprint of NN KB.

5.2 Methodology

5.2.1 Experimental setup

We perform our experiments on a server with a 4-core 3.8GHz Intel Xeon E3-1275 v6 (Kaby Lake) processor. The processor has 8MB of shared LLC, 256KB private L2, and 32KB each of private L1-I and L1-D cache and has 64GB of DRAM, SSD drives and both the host system and the application containers run Fedora Linux 36. SMT and Turbo Boost were disabled during the experiments.

Each of the target functions are run inside a Podman container which is pinned to a single processor core. The functions are wrapped by a gRPC server that invokes the function on request. Since the process executing the functions runs as a daemon, as opposed to running as multiple processes, the microarchitectural state is shared across contiguous invocations of the same function. The functions are invoked by a client, implemented in Go, that issues gRPC requests. Each gRPC request only triggers a single invocation and thus, where required by the experiment, the client is configured to repeatedly issue requests for a set amount of time. The client is pinned to a different processor core than the functions. Before statistics collection starts, the functions are warmed by repeated invocations for 30 seconds. This is done to ensure that caches in the execution path are warmed up and that JIT compiled functions had time to reach a fixpoint. Then, the experiments are run for 300 seconds with data collection. The exception to this is the instruction working set estimation (Section 5.3.3) which is run only for 30 seconds due to the large amount of data collected. All of the invocations of a single benchmark use the same input data.

To simulate how a function is affected by interleaved execution, we invoke a *thrasher* process between subsequent invocations of a function. The container hosting the thrasher

process is pinned to the same processor core as the function and is invoked by the client through a gRPC request after every invocation of the function. Upon invocation, the thrasher performs two operations. First, it fills the BTB and branch predictor with garbage using a process [3] that executes a long series of conditional and unconditional jumps. Then, to clear caches, it invokes the WBINVD x86 instruction that writes back and invalidates the entire cache hierarchy (L1 to LLC).

To avoid unintentionally collecting data from the thrasher process we configure `perf` to filter events not belonging to the `cgroup` of the container running the function.

The evaluated functions are executed in two different configurations. In both cases, function executions happen as fast as possible depending on the function. To distinguish these, we use a consistent naming scheme when presenting our results where the experiment configuration is indicated by a suffix added to the benchmark name as follows:

Back-to-back (*benchmark_name* suffix *-none*) Each benchmark is repeatedly invoked by back-to-back requests.

Thrashing (*benchmark_name* suffix *-thrashing*) The thrasher (as described above) is invoked after each invocation of the function.

5.2.2 Function description

To perform our experiments, we use a mix of representative and synthetic functions written in Python and NodeJS, the two most popular application runtimes for serverless applications [3]. A description of the functions and their implementation is provided in Table 5.1. The functions used are mainly derived from the FaaSProfiler framework [11] but we use a custom setup for invoking the functions. As mentioned, we also introduce two synthetic functions to the suite in addition to the representative functions. One, *fib_js* and *fib_py* calculates the Nth Fibonacci number and the other, *footprint*, hogs a configurable instruction working set while it is running. *Footprint* achieves this by executing a sequence of jumps to randomized locations in its instruction working set. The Fibonacci calculation function is implemented in both Python and NodeJS allowing us to directly compare the runtime behavior of these two platforms. Since the runtime characteristics of the synthetic functions change depending on their invocation parameters, we use them to corroborate hypotheses derived from the behavior of the representative functions. The observed execution times of the functions are shown in Table 5.2.

5.3 Measurements

This section presents the measurements we made with our functions. For each of these measurements, we compare back-to-back and interleaved function execution and discuss the consequences.

5.3.1 Where does time go?

The invocation machinery of a FaaS function is complex and multi-layered as it involves multiple components that are not directly related to a function's core functionality. To understand the potential of these components to impact the function execution behaviour, we analyze their contribution to a request's overall processing time, Figure 5.1. The sampled

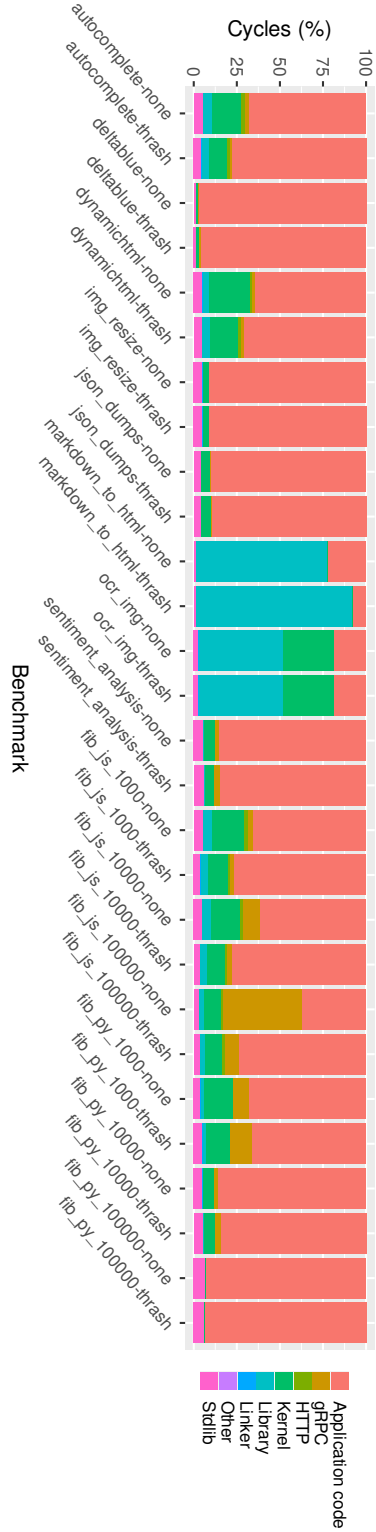


Figure 5.1: The breakdown of where executed cycles are spent divided into categories.

cycles are categorized primarily based on the Dynamic Shared Object (DSO, e.g. a shared library or executable) they originated from and, in some cases, secondarily subdivided based on the function that was executing when the cycle was sampled. This subdivision is necessary to correctly decompose NodeJS functions as they depend on platform-native libraries executed as JIT-generated code that do not appear as separate DSOs.

The categories shown in Figure 5.1 were chosen to represent the majority of the execution time. They are defined as follows:

Application Code The core part of the functionality of the function.

gRPC Cycles spent in the gRPC and Protobuf libraries.

HTTP Cycles spent processing HTTP requests.

Kernel Cycles spent in the kernel. Note that we do not trace cycles in this category back to the component that triggered their execution.

Library Cycles spent in libraries that are directly related to the core functionality of the function and only invoked from application code. For example, an image processing library is counted in this category whereas glibc is not.

Linker Cycles spent in the dynamic linker.

Stdlib Calls to C and C++ standard libraries. Like the kernel category, we do not identify who made the standard library calls.

In general, application or application-specific library code dominates the CPU cycle distribution regardless of the function and execution mode used. The principal pattern that emerges is that the functions with the shortest execution times (cf. Table 5.2) spend relatively more time in the function invocation machinery. This is not surprising considering that, unlike functions with longer execution times, they are unable to amortize the invocation overhead.

Next we look at how the cycle distribution changes when comparing back-to-back invocations of the functions to interleaved invocations. Examining this change in distribution informs us about how the various components involved in the function execution lifecycle are affected by the interference from the thrasher. If a component takes up relatively more cycles in the interleaved execution case, it means that the component is disproportionally negatively affected by the thrasher. Our results show that there are significant differences in the degree to which different functions are affected that largely depend on the function execution time. For example, in *autocomplete* and *dynamichtml*, which have very small execution time, we observe an increase in the fraction of cycles spent in application code with interleaved execution. In contrast, long running functions such as *img_resize* and *ocr_img* do not show much difference in cycle distribution.

Now, we demonstrate the significance of interleaved execution on functions with a short execution time from another perspective by comparing the elapsed per-invocation wall-clock time between the two invocation modes. Figure 5.2 shows the difference in request round trip time between back-to-back and interleaved executions as measured from the client. For the majority of the functions there is no significant difference in the execution time between the interleaved and back-to-back executions. We only see marked increases in execution time for very short functions with < 1 ms median execution time. The most extreme example is the 17× increase in execution time of the *fib_js_1000* function.

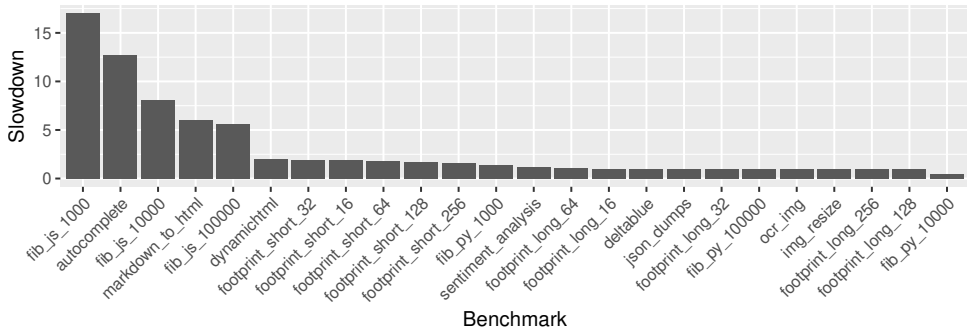


Figure 5.2: The wall-time slowdown encountered when running functions interleaved instead of back-to-back.

Table 5.2: Percentiles of the observed running times for the functions in ms.

Benchmark	50%	90%	95%	99%
autocomplete	0.26	0.29	0.31	0.38
deltablue	8.82	9.56	9.65	15.10
dynamichtml	0.47	0.51	0.54	0.74
img_resize	747.69	793.62	799.36	803.95
json_dumps	14.54	14.63	14.65	14.90
markdown_to_html	38.24	38.43	38.78	38.92
ocr_img	1164.65	1177.91	1179.57	1180.89
sentiment_analysis	2.03	2.14	2.17	2.23
fib_js_1000	0.25	0.27	0.29	0.34
fib_js_10000	0.34	0.43	0.47	0.63
fib_js_100000	0.48	0.59	0.64	0.86
fib_py_1000	0.52	0.60	0.62	0.66
fib_py_10000	1.82	2.11	2.28	4.49
fib_py_100000	97.74	98.37	98.40	99.15
footprint_long_16	10.52	10.58	10.61	10.70
footprint_long_32	20.75	20.83	20.86	21.00
footprint_long_64	52.93	53.04	53.16	53.36
footprint_long_128	121.25	121.41	121.64	121.95
footprint_long_256	260.76	262.45	262.65	262.82
footprint_short_16	0.30	0.35	0.37	0.40
footprint_short_32	0.32	0.37	0.39	0.42
footprint_short_64	0.37	0.43	0.44	0.48
footprint_short_128	0.49	0.54	0.55	0.59
footprint_short_256	0.74	0.80	0.81	0.84

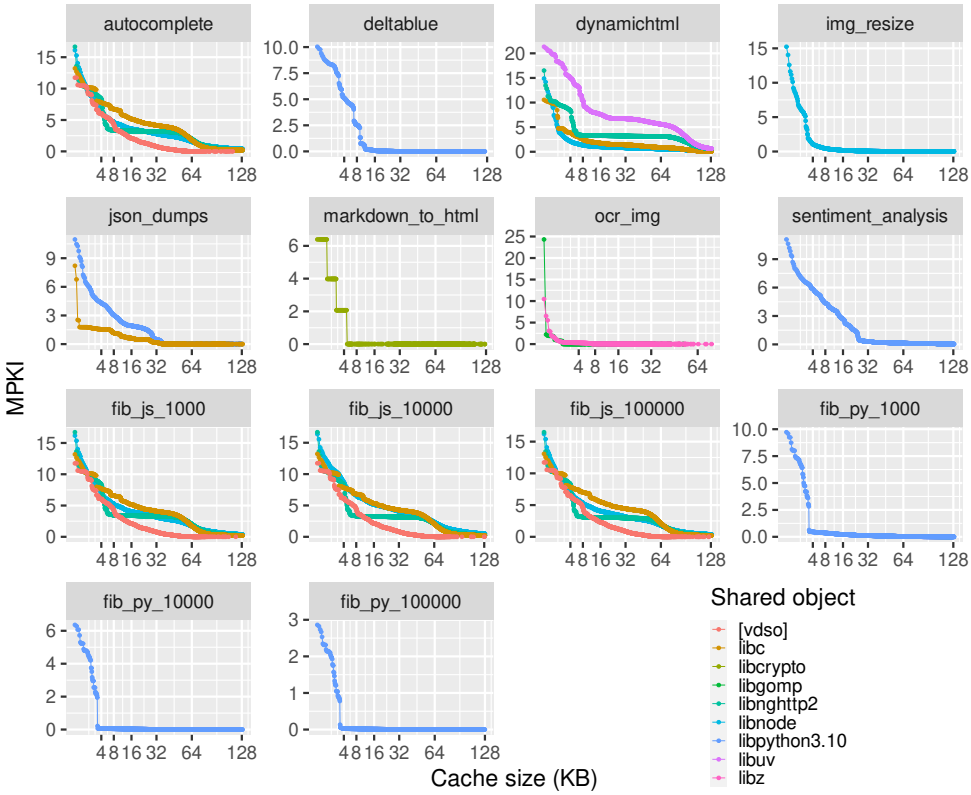


Figure 5.3: The estimated instruction working set sizes of the functions.

Finally, we also observe that the execution time alone does not always explain a function's sensitivity to interleaved execution. Compare, for example, *dynamichtml* to *fib_py_1000* which have very similar execution times as depicted in Table 5.2. For the *dynamichtml* function, we see that more cycles are spent on application code in the interleaved execution case, whereas the characteristics of *fib_py_1000* are largely unchanged. A reason for this behaviour can be that other function properties such as code and data footprints, control flow behaviour, etc. influence performance sensitivity to interleaved execution. As there can be a large number of function properties, we first analyze the ones that have the largest impact on function performance, in the next section, and then analyze the impact of thrashing on them.

5.3.2 Microarchitectural analysis

To analyze the microarchitectural behaviour of our functions, we use the established Top-Down methodology [16]. The Top-Down methodology uses performance counters to estimate the fraction of pipeline slots that are stalled due to bottlenecks in specific parts of the processor. The top level of the analysis is broken down into four categories: *Retiring*, *Backend_Bound*, *Bad_Speculation*, and *Frontend_Bound*. The *Retiring* category covers slots containing retired instructions, that is, instructions that completed and committed their result. As such, this is the desirable category of Top-Down and we want to maximize the number of pipeline slots that fall in this category. *Backend_Bound* denotes slots that are stalled due to the execution units

of the processor backend being unable to accept additional instructions. The *Bad_Speculation* category denotes slots that are stalled due to incorrect speculations, for example, branch mispredictions. Finally, the *Frontend_Bound* category contains slots that are stalled due to the frontend's inability to supply the backend with instructions at a sufficiently high rate. The identified bottlenecks can be broken down in a hierarchical fashion making it possible to identify a specific microarchitectural structure that is put under stress by the evaluated function.

The cycles per instruction (CPI) stacks resulting from the Top-Down analysis are shown in Figure 5.4. The CPI stack visualizes the contributions of the individual bottlenecks to the overall performance of the function. From the results, we see that there is a strong correlation between the execution time of a function and the instructions per cycle (IPC) rates that they achieve: shorter running functions achieve lower IPC rates (i.e., high CPI). Additionally, when comparing back-to-back and interleaved executions, the functions with low IPC also show the largest relative performance degradation in the interleaved execution mode.

The figure also implies that short running functions show low IPC because of the front-end bottleneck, i.e., they are *Frontend_Bound*. These results corroborate prior work which also found serverless functions and conventional server applications to be *Frontend_Bound* and proposed diverse mechanisms to mitigate this bottleneck [1, 2, 5–7, 10]. Further, prior work [10] also reported that instruction cache (L1-I) misses are the principal reason for this front-end bottleneck in serverless functions. This finding suggests that the instruction working set size of the functions also has the most significant impact on their performance. To show the instruction working set of a function impacts its performance sensitivity to interleaved execution, we estimate the instruction working set of our functions in the next section.

5.3.3 Estimating instruction working set

Motivated by the Top-Down analysis results, we now estimate the instruction working set size of our functions to understand how it impacts function performance. To perform the estimation, we use a method described in [15] adapted to work with traces gathered using Intel PT. The result is shown in Figure 5.3 giving the Cumulative Distribution Functions for the functions of the estimated MPKI rate of each function's shared objects depending on the instruction cache size. When the MPKI for a cache size reaches zero, it means that the entire instruction working set of the function fits the cache. Therefore, this cache size corresponds to the instruction working set of the function. For clarity, we only show the CDFs for shared objects that combined contribute 99% of the executed instructions or the single shared object that alone contribute more than 99% of executed instructions.

With these results, we can now shed further light on the data presented in Section 5.3.1 and Section 5.3.2. Comparing the *fib_py_1000* and *fib_js_10000* functions using the data from Figure 5.3, we see that the NodeJS version of the function has a significantly larger instruction working set than the Python version. The instruction working set of the Python version of the function is less than 4KB while the NodeJS implementation requires more than 64KB, a 16× difference. This observed correlation remains consistent across all of the measured functions.

Next, we look at how the instruction working set of a function affects its sensitivity to interleaved execution. Looking at the synthetic footprint functions (right side of Figure 5.4) and comparing them to *fib_py_1000* we see that, again, only functions with a short execution time are affected by interleaved execution. Additionally, for short-running functions a

large instruction working set exacerbates the impact of interleaved execution. On the other hand, the performance of longer-running functions are unaffected by interleaved execution regardless of their instruction working set. The conclusion of this is that only functions with a very short execution time *and* and a large instruction working set see a performance degradation because of interleaved function execution.

Finally, our results highlight the importance of choice of programming language and runtime environment for application performance. For example, looking at *fib_js* and *fib_py* functions in Figure 5.4, the NodeJS implementation show significantly worse CPI. However, computing the 100,000th Fibonacci number takes 181× longer using the Python implementation than with the NodeJS implementation as shown in Table 5.2. This observation is far from novel but it highlights the magnitude of the gains that are possible by making purely application-level changes.

5.3.4 Category-wise performance

Lastly, we discuss the performance of the functions broken down by category. We divide the executed cycles into the same categories as in Section 5.3.1. The results are shown in Figure 5.5. The purpose of this experiment is to assess if particular parts of the application exhibits worse performance than the average. For the back-to-back executions, no particular component diverges significantly from the average performance of the function. For the interleaved execution, a different pattern emerges. Again, only the functions with the shortest execution times are affected but the components taking the smallest part of the total execution time are disproportionally affected. For example the Linker component contributes only a negligible fraction of the executed cycles (see Figure 5.1) of the *autocomplete* function and in the back-to-back execution scenario it exhibits the same performance as the application as a whole, around 2 CPI. However, in the interleaved execution scenario, its performance deteriorates more than 2×. Meanwhile, the performance of the Application Code category deteriorates slightly, from 2 to 2.5 CPI, ending up just below the overall function performance of 2.9 CPI (as seen in Figure 5.4). The same pattern can be observed for the *dynamichtml*, *fib_js_1000* and *fib_js_10000* functions.

5.4 Discussion

The results of our study shows that functions with very short execution times (< 1 ms) benefit significantly from being executed on a processor with a warm microarchitectural state. However, as noted in the introduction, such short functions are quite uncommon in real-world applications. Additionally, even if the microarchitectural efficiency of these functions are improved using targeted optimizations the running time of the actual function may still only constitute a small fraction of the total round trip time of a function invocation. Meanwhile, larger functions, whose execution time contribute significantly to the total request round trip time, will not see any benefit from targeted microarchitectural optimizations.

These observations motivates research into high-level approaches for improving the execution time of the shortest functions. For example, functions that are executed in sequence as part of an application graph could be dynamically compiled together into a single component. Such an approach would preserve the function compositionality and modularity that are essential to the serverless application model while, at the same time, eliminate the overhead arising from the execution of a large number of short functions on both the microarchitectural and system level.

However, it is important to note that even if the execution overhead of serverless application graphs can be reduced by dynamically compiling functions into a single component, a large number of serverless applications consist only of a single function [3]. Furthermore, many of these requests are interactive, that is, the user that made the request expects an immediate response. The functions responding to such requests have an inherently short running time and are therefore likely to still see significant benefit from targeted microarchitectural optimizations.

5.5 Conclusion

This paper aimed to identify properties of serverless functions that predicts if a function is likely to benefit from warm microarchitectural state. To do this, we evaluated a suite of both real-world and synthetic functions to identify their per-invocation execution times and their instruction working set sizes. Subsequently we interleaved the function executions with a process that thrashes the microarchitectural state of the previously invoked function. By comparing the performance of the back-to-back and interleaved function executions across several metrics we identified key properties that makes a function likely to benefit from being executed from a warm microarchitectural state. We found that only the functions with a very short execution time (< 1 ms) and large instruction working sets are negatively affected by being interleaved with the thrasher. However, functions with longer execution times (> 50 ms) were not adversely affected by the microarchitectural state thrashing.

5.6 References

[1] Truls Asheim, Boris Grot, and Rakesh Kumar. "BTB-X: A Storage-Effective BTB Organization". In: *IEEE Computer Architecture Letters* 20.2 (2021), pp. 134–137 (cit. on p. 40).

[2] Truls Asheim, Boris Grot, and Rakesh Kumar. "A Specialized BTB Organization for Servers". In: *Proceedings of the 31st International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. Chicago, IL, USA, 2022 (cit. on p. 40).

[3] Datalog. *The state of serverless*. `https://www.datadoghq.com/state-of-serverless-2021/`. 2021 (cit. on pp. 33, 35, 42).

[4] Dong Du et al. "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 467–481 (cit. on p. 33).

[5] Tanvir Ahmed Khan et al. "Twig: Profile-Guided BTB Prefetching for Data Center Applications". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 816–829 (cit. on p. 40).

[6] Rakesh Kumar, Boris Grot, and Vijay 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 '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 30–42 (cit. on p. 40).

[7] Rakesh Kumar et al. “Boomerang: A Metadata-Free Architecture for Control Flow Delivery”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 493–504 (cit. on p. 40).

[8] H. Lee, K. Satyam, and G. Fox. “Evaluation of Production Serverless Computing Environments”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. July 2018, pp. 442–450 (cit. on p. 33).

[9] W. Lloyd et al. “Serverless Computing: An Investigation of Factors Influencing Microservice Performance”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. Apr. 2018, pp. 159–169 (cit. on p. 33).

[10] David Schall et al. “Lukewarm Serverless Functions: Characterization and Optimization”. In: *Proceeding of the 49st Annual International Symposium on Computer Architecuture*. ISCA ’22. New York, New York, USA: IEEE Press, 2022, ?? (Cit. on pp. 33, 40).

[11] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. “Architectural Implications of Function-as-a-Service Computing”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1063–1075 (cit. on pp. 33, 35).

[12] Mohammad Shahrad et al. “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 205–218 (cit. on p. 33).

[13] Dmitrii Ustiugov et al. “Benchmarking, Analysis, and Optimization of Serverless Function Snapshots”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’21)* (cit. on p. 33).

[14] Liang Wang et al. “Peeking Behind the Curtains of Serverless Platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 133–146 (cit. on p. 33).

[15] S.C. Woo et al. “The SPLASH-2 programs: characterization and methodological considerations”. In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 24–36 (cit. on p. 40).

[16] 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)*. Mar. 2014, pp. 35–44 (cit. on p. 39).

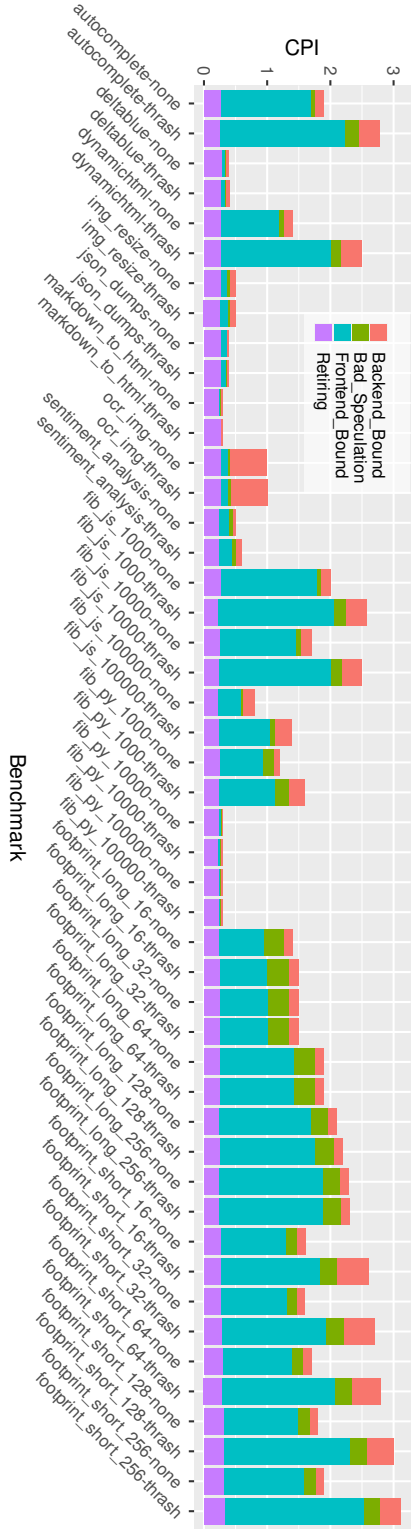


Figure 5.4: CPI stack for the functions broken down based on the contribution from each bottleneck category.

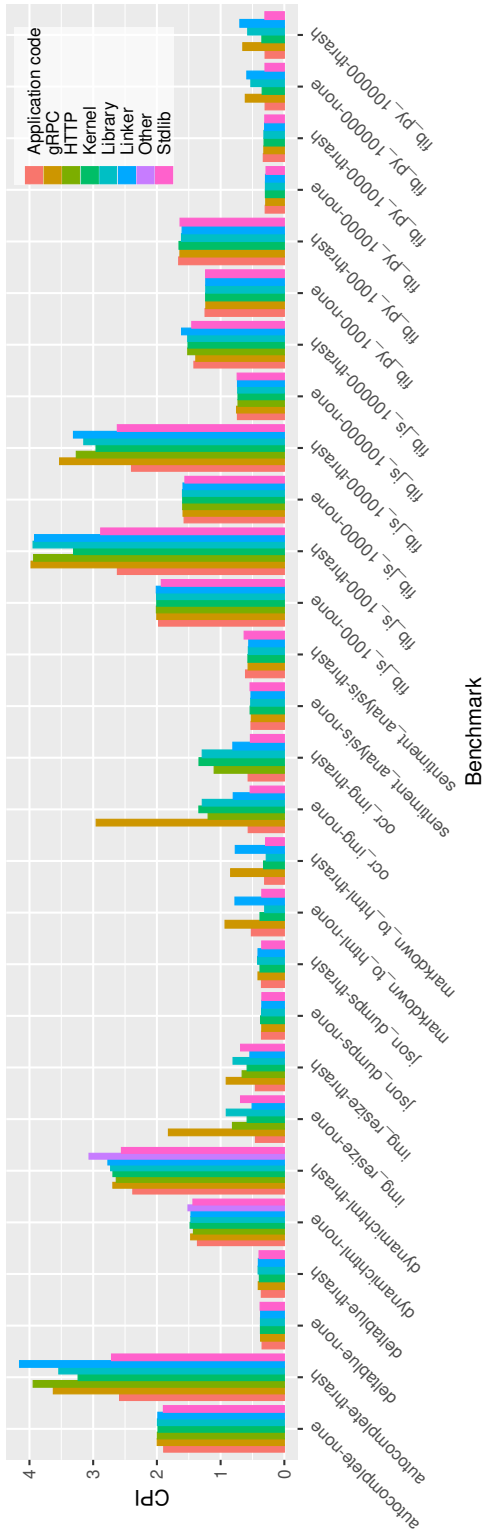


Figure 5.5: The CPI for the different components of the application.

Chapter 6

Paper A2 – BTB-X: A Storage-Effective BTB Organization

Authors

Truls Asheim, Boris Grot and Rakesh Kumar

Published in

IEEE Computer Architecture Letters, Volume: 20, Issue: 2, July-December 2021

BTB-X: A Storage-Effective BTB Organization

Truls Asheim[1], Boris Grot[2], Rakesh Kumar[1]
1) Norwegian University of Science and Technology, Norway
2) University of Edinburgh, UK

Abstract

Many contemporary applications feature multi-megabyte instruction footprints that overwhelm the capacity of branch target buffers (BTB) and instruction caches (L1-I), causing frequent front-end stalls that inevitably hurt performance. BTB is crucial for performance as it enables the front-end to accurately resolve the upcoming execution path and steer instruction fetch appropriately. Moreover, it also enables highly effective fetch-directed instruction prefetching that can eliminate many L1-I misses. For these reasons, commercial processors allocate vast amounts of storage capacity to BTBs. This work aims to reduce BTB storage requirements by optimizing the organization of BTB entries. Our key insight is that today's BTBs store the full target address for each branch, yet the vast majority of dynamic branches have short offsets requiring just a handful of bits to encode. Based on this insight, we organize the BTB as an ensemble of smaller BTBs, each storing offsets within a particular range. Doing so enables a dramatic reduction in storage for target addresses. We also compress tags to reduce the tag storage cost. Our final design, called BTB-X, uses an ensemble of five BTBs with compressed tags that enables it to track 2.8x more branches than a conventional BTB with the same storage budget.

6.1 Introduction

Contemporary server applications feature massive instruction footprints stemming from deeply layered software stacks. These footprints may far exceed the capacity of the branch target buffer (BTB) and instruction cache (L1-I), resulting in the so-called front-end bottleneck. BTB misses may lead to wrong-path execution, triggering a pipeline flush when misspeculation is detected. Such pipeline flushes not only throw away tens of cycles of work but also expose the fill latency of the pipeline. Similarly, L1-I misses cause the core front-end to stall for tens of cycles while the miss is being served from lower-level caches.

BTB stands at the center of a high-performance core front end for three key reasons: it determines the instruction stream being fetched, it identifies branchs for the branch predictor,

and it affects the L1-I hit rate. Specifically, by identifying control flow divergences, the BTB ensures that the branch predictor can make predictions for upcoming conditional branches. For predicted-taken and unconditional branches, the BTB supplies targets to which instruction fetch should be redirected. Finally, the BTB together with the direction predictor enables an important class of instruction prefetchers called fetch-directed instruction prefetchers (FDIP) [6, 7, 9], which rely on the BTB to discover L1-I prefetch candidates.

Considering the criticality of capturing the large branch working sets of modern workloads, commercial CPUs feature BTBs with colossal capacities, a trend also observed by [5]. Thus, IBM z-series processors [3], AMD Zen-2 [11], and ARM Neoverse N1 [8] feature 24K-entry, 8.5K-entry, and 6K-entry BTBs. With each BTB entry requiring 10 bytes or more (Section 6.2), BTB storage costs can easily reach into tens and even hundreds of KBs. Indeed, the Samsung Exynos M6 mobile processor allocates a staggering 529KB of on-chip storage to BTBs [4]. While such massive BTBs are effective at capturing branch working sets, they do so at staggering area costs.

This work seeks to reduce BTB storage requirements by increasing its *branch density*, defined as branches per KB of storage. To that end, we aim to reorganize individual BTB entries to minimize their storage cost. Our key insight is that branch offsets, defined as delta between the address of the branch instruction and that of its target, are unequally distributed but tend to require significantly fewer bits to represent than full target addresses. Our analysis reveals that 37% of dynamic branches require only 7 bits or fewer for offset encoding, while a meager 1% of branches need 25 bits or more to store their offsets.

Based on this insight, we propose to store offsets in the BTB rather than full target addresses, which can be up to 64 bits long depending on the size of virtual address space. To accommodate the varied distribution of branch offsets, we partition the BTB into several smaller BTBs, each storing only those branches whose target offsets can be encoded with a certain number of bits. Because the target field accounts for over half of each entry's storage budget in a conventional BTB (Figure 6.1), this optimization brings significant storage savings. We further observe that the tag field is the second-largest contributor to each BTB entry's storage requirement. To reduce this cost, we propose compressing the tags through the use of hashing.

Our final design, called *BTB-X*, uses an ensemble of five BTBs, each with 16-bit tags. The BTBs differ only in the number of bits they allocate for branch target offsets. Our evaluation shows that BTB-X can track over 2.8x more branches than a conventional BTB with the same storage budget. Conversely, BTB-X can accommodate the same number of branches as existing BTBs while requiring 2.8x less storage.

6.2 Background

6.2.1 Branch Target Buffer (BTB)

BTB is used in the core front-end to identify whether a program counter (PC) corresponds to a branch instruction before the instruction itself is even fetched. As depicted in Figure 6.1, each BTB entry is composed of *tag*, *type*, and *target* fields. BTB is indexed with the lower order PC bits and *tag* field of the indexed entry is compared with the higher order PC bits. A match indicates that the PC belongs to a branch instruction. The *type* field of the indexed BTB entry determines whether the branch is a call, return, conditional, or unconditional branch. The branch type determines whether the branch direction (*taken/not taken*) needs to be predicted and where its target address is found. Call, return, and unconditional branches

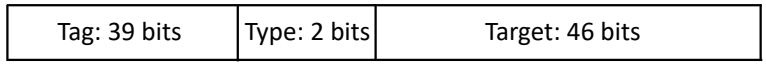


Figure 6.1: BTB entry composition in a conventional BTB.

are always *taken*, whereas for conditional branches, a direction predictor is used to predict their direction. If the branch is predicted to be taken, *target* field in the BTB entry provides the address for the next instruction, except for returns. This is because a given function can be called from different call sites; as such, the return address is call-site dependent. Therefore, a return address stack (RAS) is typically employed to record return addresses at call-sites. On a function call, the call instruction pushes the return address to RAS, which is later popped by the corresponding return instruction.

6.2.2 The cost of a BTB miss

A BTB miss for a branch instruction means that the branch is undetected and the front-end continues to fetch instructions sequentially. Whether or not the sequential path is the correct one depends on the actual direction of the missed branch. Unless the missed branch is a conditional branch that is not taken, the sequential path is incorrect. When the wrong path is eventually detected by the core, all the instructions after the branch that missed in the BTB are flushed, fetch is redirected to the branch target and pipeline is filled with correct-path instructions. BTB misses are thus highly deleterious to performance as they result in a loss of tens of cycles of work and expose the pipeline fill latency.

6.2.3 BTB's role in instruction prefetching

Fetch-directed instruction prefetchers are a class of powerful L1-I prefetchers that intrinsically rely on a BTB. These prefetchers are highly effective and, when coupled with a sufficiently large BTB, outperform the winner of the recently-concluded Instruction Prefetching Championship [1], as reported by Ishii et al. [5]. Variants of these prefetchers have been adopted in commercial products, for example in IBM z15 [10], ARM Neoverse N1 [8] etc.

Figure 6.2 shows a canonical organization of a fetch-directed instruction prefetcher (FDIP) [9]. As originally proposed, FDIP decouples the branch-prediction unit and the fetch engine via the *fetch target queue (FTQ)*. This decoupling allows the branch prediction unit to run ahead of the fetch engine and discover prefetch candidates by predicting the control flow far into the future. With FDIP, each cycle, the branch prediction unit identifies and predicts branches to anticipate upcoming execution path and inserts corresponding instruction addresses into the FTQ. Consequently, the FTQ contains a stream of anticipated instruction addresses to be fetched by the core. The prefetch engine scans the FTQ to identify prefetch candidates and issue prefetch requests.

For FDIP to be effective, the BTB needs to accommodate the branch working set, otherwise frequent BTB misses will cause FDIP to prefetch the wrong path as FTQ will be filled with wrong path instruction addresses. This is one of the key reasons why commercial processors deploy massive BTBs, as also observed by [5].

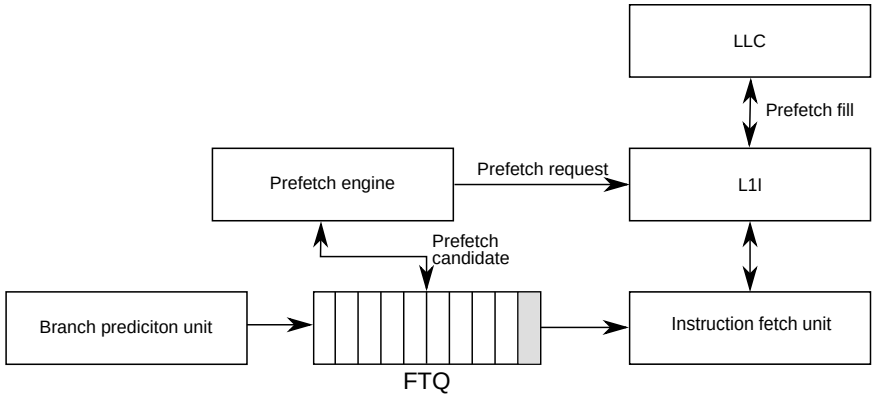


Figure 6.2: FDIP microarchitecture

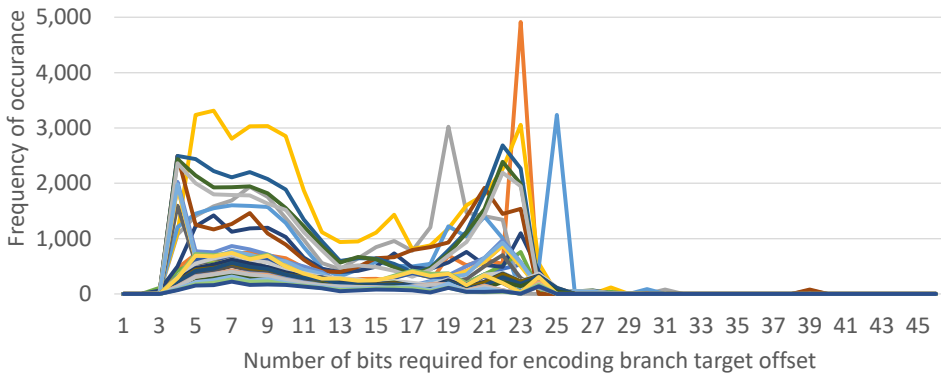


Figure 6.3: Distribution of branch target offsets.

6.3 BTB-X

To reduce the overall storage cost, this work seeks to minimize the storage requirements of the costliest fields making up each BTB entry, i.e. target and tag, through two ideas: partitioning and hashing.

6.3.1 Partitioned BTB

As Figure 6.1 shows, the largest contributor to storage cost is the target field, which stores the branch target address. For instance, in the ARM v8 ISA, which uses a 32-bit fixed length instruction encoding, the target address is 46 bits long with a 48-bit virtual address space. Our key insight is that targets of most branches lie relatively close in the virtual address space to the branch itself. As a result, encoding the *distance* to the target, in the form of an offset from the branch instruction, instead of a full target address, can provides drastic storage savings.

Figure 6.3 plots the distribution of offsets in the branch working sets of our workload traces. Offsets are calculated in instruction words, which are 32 bits in the ARM v8 ISA. The data includes both conditional and unconditional branches; hence, it comprehensively covers the full branch working set. The X-axis shows the number of bits required to encode the offset,

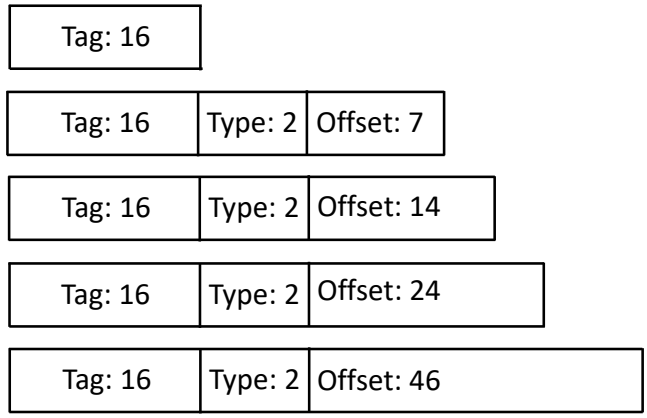


Figure 6.4: BTB entry composition for BTB-X partitions.

while the Y-axis plots the frequency of occurrence. Note that, in addition to bits for encoding the offset, an additional bit is required for the direction of the offset (forward/backward).

As the figure shows, short offsets dominate the distribution with 37% of branches requiring only seven bits or fewer for their offsets. A further 30% of branches only require between 8 and 14-bits to represent their offsets. The reason why such a high fraction of offsets is short is that conditional branches dominate the dynamic branch working set, and they tend to have short offsets [6]. This is because conditional branches generally guide the control flow only inside a function; meanwhile, software engineering principles favor small functions, thus restricting conditional branch offsets to short distances.

Perhaps surprisingly, Figure 6.3 also shows that very few branches require a large number of bits to encode their offset. Indeed, a meagre 1% of branches requires 25 bits or more for their offset encoding. The sum of these results indicates that reserving space for the full 46-bit target address results in an appalling under-utilization of BTB storage, since 99% of branches need at most half the number of bits needed to represent the full target address if offsets are used instead.

Based on these insights, we propose to partition a single logical BTB into multiple physically-separate BTBs. The BTBs differ amongst themselves only in the size of the offset. When the branch prediction unit queries an address, all BTB partitions are accessed in parallel, hence presenting a logical equivalent of a monolithic BTB. If the core queries the BTB with n addresses per cycles, each BTB-X partition must be accessed with all n addresses.

Figure 6.4 shows the BTB partitions used by our proposed BTB organization, called BTB-X. It uses five different BTBs with offset field sizes of 0, 7, 14, 24 and 46 bits. The BTB with no offset field (i.e., 0-bit offset) tracks only return instructions. Recall from Section 6.2 that return instructions read their target address from RAS; as such, there is no need to allocate space for targets of returns in the BTB. Further, as all instructions in this BTB are returns, it does not require the branch *type* field either. Other branches are allocated entries in one of the remaining four BTBs based on the minimum number of bits required to encode their offsets. For example, if a branch requires 10 bits for encoding its target offset, it is allocated an entry in the BTB with target offset field size of 14 bits.

We further make use of the data in Figure 6.3 to size each of the BTBs. Because very few branches require more than 24 bits to encode their target offsets, the BTB with the 46-bit offset field is allocated the fewest entries. Meanwhile, the BTBs corresponding to 7-, 14-, and

Table 6.1: Microarchitectural parameters

Core	6-wide OoO, 128-entry FTQ, 128 reservation stations, 352-entry ROB, 128-entry load queue, 72-entry store queue
Branch Predictor	Hashed Perceptron
L1-I	32 KB, 8-way, 4 cycle latency, 8 MSHRs
L1-D	48 KB, 12-way, 5 cycle latency, 16 MSHRs
L2	512 KB, 8-way, 14/15 cycle latency, 32 MSHRs
LLC	2MB, 16-way, 34/35 cycle latency, 64 MSHRs

24-bit offset are allocated a similar number of entries, as the frequency of 1-7 bit, 8-14 bit, and 15-24 bit offsets is about same – 37%, 30% and 32% respectively.

6.3.2 Tag Compression

Tags comprise the second largest source of storage overhead in each BTB entry, requiring 39 bits in the baseline design. To further reduce the storage requirement, BTB-X uses a compressed 16-bit tag in all of its BTBs. Our compression scheme maintains the 8 low-order bits same as in the full tag. The remaining bits of the full tag are folded, using the XOR operator, in blocks of eight to compute the 8 higher-order bits for the compressed tag. As our evaluation shows, the performance impact of this scheme is negligible as the hashing function (folded XOR) preserves most of the entropy found in the high-order bits.

6.3.3 Applicability to Basic-Block-Based BTBs

While this work describes BTB-X in the context of an instruction-based BTB organization (i.e., the BTB is accessed using individual instruction addresses), our insights and design are equally applicable to basic-block-based BTBs (BB-BTBs) [6, 7, 9]. BB-BTBs are similar to instruction-based ones but are accessed using a basic-block address. Because existing BB-BTB designs store full branch targets and offsets, they would benefit from optimizations described in this work.

6.4 Evaluation

We use ChampSim [2], an open-source trace-driven simulator, to evaluate the efficacy of BTB-X on server and client workload traces from IPC-1 [1]. We warm up microarchitectural structures for 50M instructions and collect statistics over the next 50M. The microarchitectural parameters for the modeled processor are listed in Table 6.1.

6.4.1 Storage Breakdown

The storage requirements for a conventional BTB for different number of BTB entries are presented in Table 6.2 assuming a 48-bit virtual address space. We increase the number of sets in the BTB to increase the number of entries while keeping the associativity same (8-way). Notice that the entry size reduces by one bit while doubling the number of entries. This is because the tag size reduces as more bits are needed to index the BTB.

Table 6.3 presents the allocation of the storage budget among the five BTB-X partitions. For this analysis, the storage budget is capped at that of a 1K-entry conventional BTB. As the table shows, the partition for 46-bit offsets gets the smallest amount of storage as very few

Table 6.2: Storage breakdown for conventional BTB

Entries	Organization	Entry size (bits)	Total (bytes)
1K	128-set, 8-way	87	10.875K
2K	256-set, 8-way	86	21.5K
4K	512-set, 8-way	85	42.5K
8K	1024-set, 8-way	84	84K
16K	2048-set, 8-way	83	166K

Table 6.3: Storage breakdown for BTB-X. The storage budget is comparable to that of a 1K-entry conventional BTB.

Partition	Entry size	Entries	Storage
0-bit offset	16-bits	768	1.5KB
7-bit offset	25-bits	768	2.34KB
14-bit offset	32-bits	640	2.5KB
24-bit offset	42-bits	640	3.28KB
46-bit offset	64-bits	80	0.625KB
Total		2,896	10.25KB

Table 6.4: Storage and entries in conventional BTB and BTB-X

Conventional BTB		BTB-X	
Storage	Entries	Storage	Entries
10.875KB	1K	10.25KB	2,896
21.5KB	2K	20.5KB	5,792
42.5KB	4K	41KB	11,584
84KB	8K	82KB	23,168
166KB	16K	164KB	46,336

branches need to be allocated there. Meanwhile, the remaining partitions get relatively more storage with a roughly similar number of entries in each partition.

When presented with a larger storage budget, we follow the same strategy for scaling up BTB-X as for a conventional BTB. Thus, we double the number of sets in each BTB partition to double the capacity while maintaining the associativity (i.e., 0-bit and 7-bit offset partitions are 6-way, others are 5-way).

Table 6.4 shows the number of entries that a conventional BTB and BTB-X can accommodate for several storage budgets. As is evident from the table, for a given storage budget, BTB-X can store about 2.8x more entries than the conventional BTB. Note that since the number of sets have to be a power of 2, we are not able to precisely match the storage of conventional BTB and BTB-X – the conventional BTB gets a slightly higher storage.

6.4.2 Performance

To assess the effectiveness of BTB-X, we compare its performance to that of a conventional BTB across different storage budgets. Recall from Section 6.2 that a larger BTB can deliver two distinct benefits: 1) reduce the incidence of pipeline flushes by detecting branches in the upcoming control flow and 2) facilitate instruction prefetching when coupled with FDIP. Thus, we compare the performance gains achieved by the two competing BTB designs by evaluating them with FDIP.

Figure 6.5 presents the performance gains obtained on server and client traces. Each bar in the figure shows the contribution to performance of having fewer pipeline flushes and from better instruction prefetching stemming from larger BTB capacities. The results are normalized to the performance of a core with a 1K-entry conventional BTB (10.875KB storage budget) and no instruction prefetching.

As the figure shows, BTB-X provides significantly higher overall performance than the conventional BTB for equal storage budgets of up to several tens of kilobytes. The performance advantage of BTB-X is particularly pronounced on server traces whose large instruction footprints pressure the BTB and L1-I. For instance, BTB-X provides 63% performance gain over the baseline compared to 38% of conventional BTB with 21.5KB storage budget. At large BTB storage budgets, the branch working sets of many workloads start to fit in the available BTB capacity, at which point the performance gap between the two designs diminishes.

A key take-away from the figure is that BTB-X provides same or higher performance than the conventional BTB even when BTB-X is given just half the storage budget of its conventional counterpart. For example, in Figure 6.5a, the conventional BTB improves performance by 38% with a 21.5KB budget whereas BTB-X provides a 44% improvement with just 10.875KB of storage. The reason for this phenomenon is that BTB-X accommodates 2.8x more entries than a conventional BTB of equal storage budget; thus, halving BTB-X's budget still gives a capacity advantage over the conventional design.

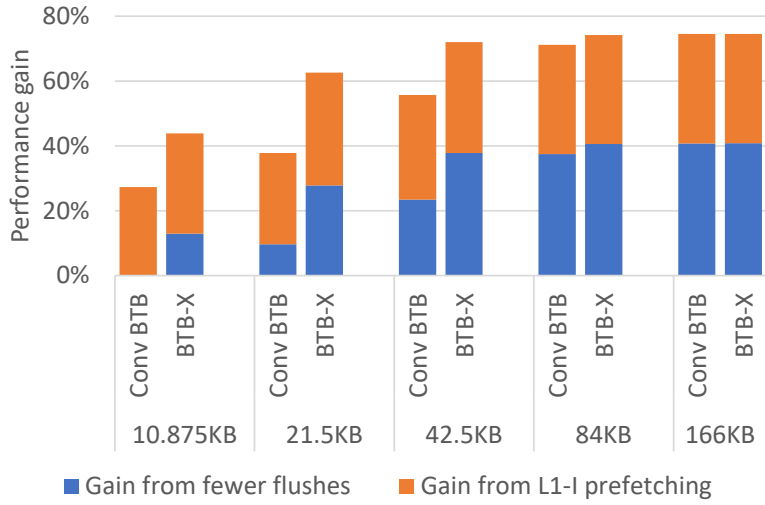
Ignoring instruction prefetching and looking exclusively at performance gains stemming from reduced pipeline flushes, the trends are similar to above. For storage budgets of up to several tens of KBs, BTB-X outperforms a conventional BTB even with half of the latter's storage budget. For instance, Figure 6.5a (blue segments of the bars) shows that BTB-X provides 13% gain with a 10.875KB budget whereas a conventional BTB with twice the budget (21.5KB) gains only 10%.

6.4.3 Impact of Tag Compression

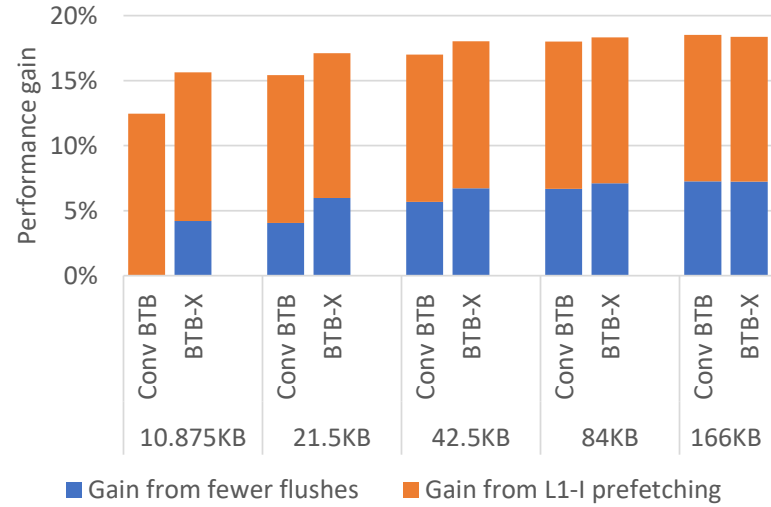
For assessing the performance loss due to compressed tags, we compare the performance of BTB-X with 16-bit tags versus full tags for the smallest BTB size (10.875 KB). We focus on the smallest BTB as it is likely to suffer the highest degree of aliasing due to tag compression. Our results show that, full tags provide 38.21% performance gain, geo-mean across server and client traces, over the baseline compared to 38.16% with compressed tags, a difference of only 0.05%. This indicates that our tag compression scheme is able to preserve the entropy of higher-order bits.

6.5 Conclusion

The multi-megabyte instruction footprints of contemporary server applications cause frequent BTB and L1-I misses, which have become major performance limiters. Because BTB capacity greatly affects front-end performance in terms of flush rate and the efficacy of fetch-directed instruction prefetching, commercial products allocate tens to hundreds of KBs of storage to BTBs. To reduce the BTB storage requirements, this paper introduced an optimized BTB organization. The proposed design, BTB-X, leverages our insight that branch target offsets vary but tend to be much shorter than full target addresses. BTB-X uses an ensemble of five BTBs, each storing offsets of a different length, and also compresses the tags to track 2.8x more branches than a conventional BTB with an equal storage budget.



(a) Server workloads.



(b) Client workloads.

Figure 6.5: Performance gain for conventional BTB and BTB-X (both with FDIP) on (a) **server** and (b) **client** traces. Baseline is no-prefetch 1K-entry conventional BTB. X-axis is storage for a 1K-, 2K-, 4K-, 8K-, and 16K-entry conventional BTB.

Acknowledgments

This work is partially supported through the Research Council of Norway (NFR) grant 302279 to NTNU and by UK EPSRC grant EP/M001202/1 to the University of Edinburgh.

6.6 References

[1] In: https://research.ece.ncsu.edu/ipc/ (cit. on pp. 50, 53).

[2] In: https://github.com/ChampSim/ChampSim (cit. on p. 53).

[3] J. Bonanno et al. "Two level bulk preload branch prediction". In: *HPCA*. 2013 (cit. on p. 49).

[4] Brian Grayson et al. "Evolution of the Samsung Exynos CPU Microarchitecture". In: *ISCA*. Virtual Event, 2020 (cit. on p. 49).

[5] Yasuo Ishii et al. "Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective". In: *ISPASS 2021*. (Cit. on pp. 49, 50).

[6] Rakesh Kumar et al. "Boomerang: a Metadata-Free Architecture for Control Flow Delivery". In: *HPCA*. 2017 (cit. on pp. 49, 52, 53).

[7] Rakesh Kumar et al. "Blasting through the Front-End Bottleneck with Shotgun". In: *ASPLOS*. 2018 (cit. on pp. 49, 53).

[8] A. Pellegrini et al. "The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC". In: *IEEE Micro*. 2020 (cit. on pp. 49, 50).

[9] G. Reinman et al. "Fetch Directed Instruction Prefetching". In: *MICRO*. 1999 (cit. on pp. 49, 50, 53).

[10] Anthony Saporito. "The IBM z15 processor chip set". In: *Hot Chips*. 2020 (cit. on p. 50).

[11] D. Suggs et al. "The AMD "Zen 2" Processor". In: *IEEE Micro*. 2020 (cit. on p. 49).

Chapter 7

Paper A3 – A Specialized BTB Organization for Servers

Authors

Truls Asheim, Boris Grot and Rakesh Kumar

Published in

Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2022

A Specialized BTB Organization for Servers

Truls Asheim[1], Boris Grot[2], Rakesh Kumar[1]
1) Norwegian University of Science and Technology, Norway
2) University of Edinburgh, UK

Abstract

Contemporary server applications feature massive instruction footprints stemming from deeply layered software stacks. These footprints far exceed the capacity of the branch target buffer (BTB) and instruction cache (L1-I), resulting in the so-called front-end bottleneck. BTB misses may lead to wrong-path execution, triggering a pipeline flush when misspeculation is detected. Such pipeline flushes not only throw away tens of cycles of work but also expose the fill latency of the pipeline. Similarly, L1-I misses cause the core front-end to stall for tens of cycles while the miss is being served from lower-level caches.

7.1 Introduction and Motivation

Contemporary server applications feature massive instruction footprints stemming from deeply layered software stacks. These footprints far exceed the capacity of the branch target buffer (BTB) and instruction cache (L1-I), resulting in the so-called front-end bottleneck. BTB misses may lead to wrong-path execution, triggering a pipeline flush when misspeculation is detected. Such pipeline flushes not only throw away tens of cycles of work but also expose the fill latency of the pipeline. Similarly, L1-I misses cause the core front-end to stall for tens of cycles while the miss is being served from lower-level caches.

BTB stands at the center of a high-performance core front end for three key reasons: it determines the instruction stream being fetched, it identifies branches for the branch predictor, and it affects the L1-I hit rate. Specifically, by identifying control flow divergences, the BTB ensures that the branch predictor can make predictions for upcoming conditional branches. For predicted-taken and unconditional branches, the BTB supplies targets to which instruction fetch should be redirected. Finally, the BTB together with the direction predictor enables an important class of instruction prefetchers called fetch-directed instruction prefetchers (FDIP) [7, 8, 10], which rely on the BTB to discover L1-I prefetch candidates.

Considering the criticality of capturing the large branch working sets of modern workloads, commercial CPUs feature BTBs with colossal capacities. Thus, IBM z-series processors [2], AMD Zen-2 [12], and ARM Neoverse N1 [9] feature 24K-entry, 8.5K-entry, and 6K-entry BTBs. With each BTB entry requiring 8 bytes or more, BTB storage costs can easily reach into tens and even hundreds of KBs. Indeed, the Samsung Exynos M6 mobile processor allocates a staggering 529KB of on-chip storage to BTBs [4]. Not only the BTB storage cost is

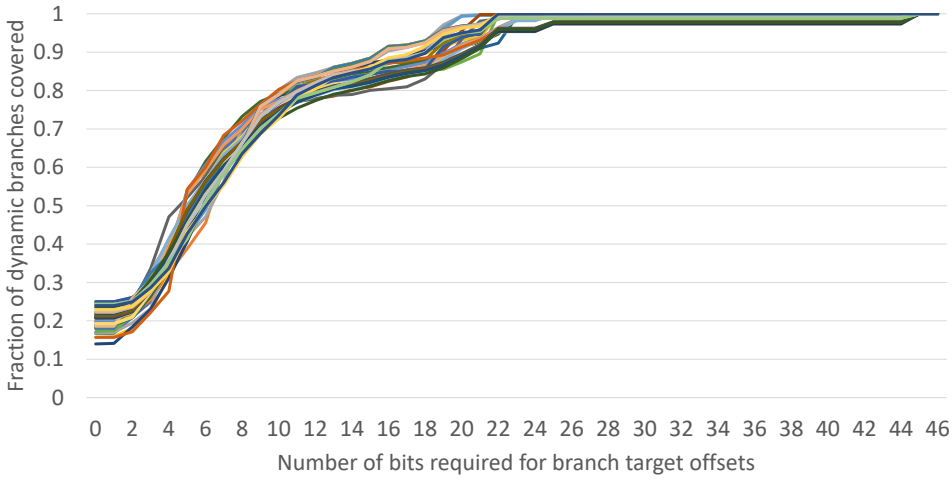


Figure 7.1: Distribution of branch target offsets.

high, it is increasing at a rapid pace. For example, the Samsung Exynos BTB storage budget increased nearly six fold (98.9KB to 561.5KB) from M2 to M6, over a period of about eight years [4]. While such massive BTBs are effective at capturing branch working sets, they do so at staggering area costs.

As the instruction footprints of server applications continue to expand, a trend also reflected in Google Web Search workload whose instruction footprint is growing at annualized rate of 27% [6], the BTB sizes and their storage overheads are destined to increase in future. Therefore, there is an urgent need to investigate storage-effective BTB organizations to combat the front-end bottleneck without necessitating prohibitive area budgets.

7.2 Key Insights

We analyze conventional and state-of-the-art BTB organizations and observe that the branch targets are the single largest contributor to BTB storage cost. Further, we analyze the number of bits required for branch target *offsets* to assess if storing the offsets, instead of the full or compressed targets, can reduce BTB storage requirements.

Figure 7.1 plots the distribution of branch target offsets in the branch working sets of our workloads. The data includes both conditional and unconditional branches; hence, it comprehensively covers the full branch working set. The X-axis shows the number of bits required to store offsets, while the Y-axis plots the fraction of dynamic branches covered.

As the figure shows, short offsets dominate the distribution with 54% of branches requiring only six bits or fewer for their offsets. A further 22% of branches only require between 7 and 10-bits to represent their offsets. The reason why such a high fraction of offsets is short is that conditional branches dominate the dynamic branch working set, and they tend to have short offsets [7]. This is because conditional branches generally guide the control flow only inside a function; meanwhile, software engineering principles favor small functions, thus restricting conditional branch target offsets to short distances. Furthermore, return instructions get their target from the return address stack (RAS), thus they do not need to store any target bits in BTBs. Therefore, for the purpose of this analysis, we assume 0-bit offsets for return instructions.

Perhaps surprisingly, Figure 7.1 also shows that very few branches require a large number of bits for their offset. Indeed, a meagre 1% of branches requires more than 25 bits for their offsets. The sum of these results indicates that reserving space for the full 46-bit target address results in an appalling under-utilization of BTB storage, since 99% of branches need at most half the number of bits needed to represent the full target address if offsets are used instead.

We gain two key insights from this analysis:

Key Insight 1 The targets of most branches lie relatively close in the virtual address space to the branch itself. As a result, storing the *distance* to the target, in the form of an offset from the branch instruction can provide drastic storage savings.

Key Insight 2 The target offset sizes are unevenly distributed with 0-6 bits, 7-10 bits, and 11-25 bits required to encode the offsets of 54%, 22% and 23% of branches respectively. Therefore, a single size offset field cannot provide storage optimal solution.

7.3 BTB-X

Based on these insights, we propose a new BTB design, called BTB-X, that stores target offsets instead of full or compressed target addresses. To accommodate the uneven distribution of target offsets, we size different ways of a set associative BTB-X to hold different sized target offsets. A branch is allocated to a way whose offset field is at least as large as the number of bit required to store the target offset. We use an 8-way set associative BTB-X and leverage the data in Figure 7.1 to appropriately size the offset field of each way such that each way covers about 12.5% dynamically executed branches. Figure 7.1 shows that, on average, 0-, 4-, 5-, 7-, 9-, 11-, 19-, and 25-bit offsets cover about 20%, 36%, 46%, 61%, 72%, 79%, 90%, and 99% dynamic branches. Therefore, we size the 8-ways of BTB-X ways to hold 0-, 4-, 5-, 7-, 9-, 11-, 19-, and 25-bit target offsets respectively. Notice that about 20% of dynamic branches that require 0-bits for their offset are return instructions that read their target from RAS. Therefore, way-0 of BTB-X does not feature any storage for target offsets. Though return instruction do not get their target from BTB, they still need to be allocated BTB entries so that the branch prediction unit can identify them and pick their target from RAS while generating instruction stream to be fetched.

BTB-X covers 99% of the dynamically executed branches and we employ a very small conventional direct-mapped BTB, called BTB-XC, that stores full target addresses for the remaining 1% branches. Reserving a way in BTB-X for such branches would unnecessarily increase the storage requirements as these branches require much fewer entries than the number of sets in BTB-X. Indeed, based on our analysis, we size BTB-XC to store 64x fewer entries than BTB-X, i,e, 8x fewer entries than the number of sets in BTB-X.

7.4 Evaluation

We use ChampSim [3], an open-source trace-driven simulator, to evaluate the efficacy of BTB-X on server workload traces provided for the first Instruction Prefetching Championship (IPC-1) [1]. The modeled processor resembles Intel Sunny Cove [5]. We compare the storage requirements and performance of BTB-X against a conventional BTB design (Conv-BTB) that stores full target addresses, and also against the state-of-the-art BTB design, called PDede [11], which stores compressed targets.

Table 7.1: Number of branches in different BTB designs.

Storage	BTB-X (+ BTB-XC)	PDede	Conv-BTB
0.9KB	256 (+ 4)	210	116
1.8KB	512 (+ 8)	415	232
3.6KB	1K (+ 16)	820	464
7.25KB	2K (+ 32)	1617	928
14.5KB	4K (+ 64)	3190	1856
29KB	8K (+ 128)	6292	3712
58KB	16K (+ 256)	12405	7424

7.4.1 Storage comparison

Table 7.1 presents the number of branches the different BTB organizations (BTB-X, PDede, and Conv-BTB) can track at different storage budgets. The storage budgets shown are BTB-X storage required for storing 256, 512, 1K, 2K, 4K, 8K, and 16K branches. Our calculations assume a 48-bit virtual address space. As the table shows BTB-X stores significantly more branches than any other BTB organizations. Concretely, it stores 2.24x more branches than a conventional BTB organization. Compared to PDede, BTB-X stores 1.24x more branches at 0.9KB storage budget and 1.34x more branches at 58KB storage budget. BTB-X's advantage over PDede increases with storage budget because PDede entry size increases with the number of branches PDede can accommodate.

7.4.2 Performance comparison

Figure 7.2 presents the performance gains obtained by different BTB designs on a set of server workloads. The results are normalized to the performance of Conv-BTB with 0.9KB storage budget. Instruction prefetching (FDIP) is enabled in all designs including the baseline. As the figure shows BTB-X provides significantly higher performance than the Conv-BTB and PDede for equal storage budgets of up to 29KB and 14.5KB respectively. For instance, BTB-X provides 45% performance gain over the baseline compared to 38% and 27% of PDede and Conv-BTB, respectively, at 14.5KB budget. At large BTB storage budgets, the branch working sets of many workloads start to fit in the available BTB capacity, at which point the performance gap between BTB-X and the other two designs diminishes. A key take-away from this figure is that BTB-X outperforms the conventional BTB even when it is given just half the storage budget of its conventional counterpart. For example, in Figure 7.2, the Conv-BTB improves performance by 27% with a 14.5KB budget whereas BTB-X provides a 31% improvement with just 7.25KB of storage. The reason for this behaviour is that BTB-X accommodates 2.24x more entries than Conv-BTB of equal storage budget; thus, halving BTB-X's budget still gives a slight capacity advantage over Conv-BTB.

7.5 Conclusion

As BTB capacity greatly affects front-end performance, commercial products allocate tens to hundreds of KBs of storage to BTBs. We propose a storage-effective BTB organization, called BTB-X, that stores target offsets in place of full target addresses and employs differently sized BTB-ways for storing offsets of different lengths, thus drastically reducing BTB storage requirements.

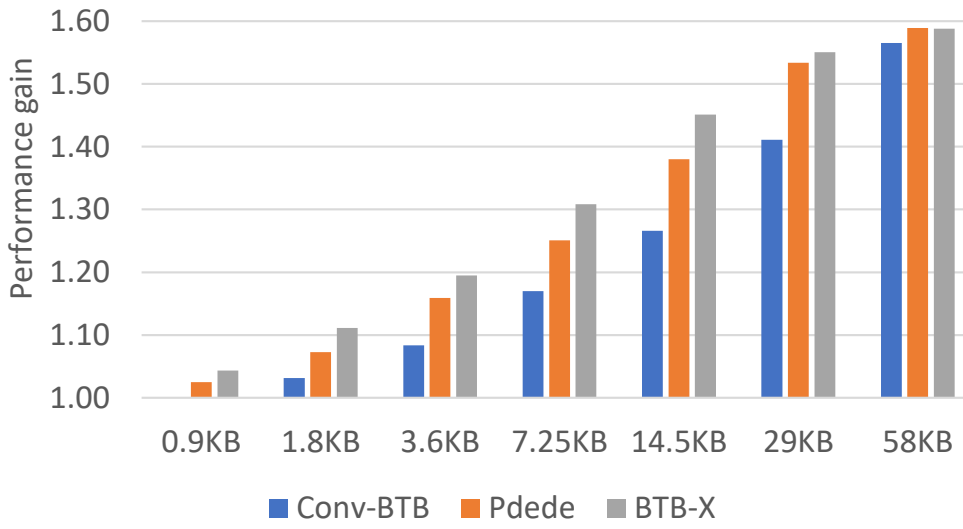


Figure 7.2: Performance comparison of different BTB designs.

7.6 References

[1] *1st Instruction Prefetching Championship*. https://research.ece.ncsu.edu/ipc/. . (Cit. on p. 62).

[2] James Bonanno et al. "Two level bulk preload branch prediction". In: *HPCA'13*. 2013 (cit. on p. 60).

[3] *ChampSim Simulator*. https://github.com/ChampSim/ChampSim. . (Cit. on p. 62).

[4] B. Grayson et al. "Evolution of the Samsung Exynos CPU Microarchitecture". In: *ISCA'20*. 2020 (cit. on pp. 60, 61).

[5] *Ice Lake processors*. https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3. . (Cit. on p. 62).

[6] Svilen Kanev et al. "Profiling a Warehouse-Scale Computer". In: *ISCA'15*. 2015 (cit. on p. 61).

[7] Rakesh Kumar et al. "Boomerang: A Metadata-Free Architecture for Control Flow Delivery". In: *HPCA'17*. 2017 (cit. on pp. 60, 61).

[8] Rakesh Kumar et al. "Blasting through the Front-End Bottleneck with Shotgun". In: *ASPLOS'18*. 2018 (cit. on p. 60).

[9] Andrea Pellegrini et al. "The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC". In: *IEEE Micro* 40.2 (2020) (cit. on p. 60).

[10] G. Reinman et al. "Fetch directed instruction prefetching". In: *MICRO'99*. 1999 (cit. on p. 60).

[11] Niranjan K Soundararajan et al. "PDede: Partitioned, Deduplicated, Delta Branch Target Buffer". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 779–791 (cit. on p. 62).

[12] David Suggs et al. “The AMD “Zen 2” Processor”. In: *IEEE Micro* 40.2 (2020) (cit. on p. 60).

Chapter 8

Paper A4 – A Storage-Effective BTB Organization for Servers

Authors

Truls Asheim, Boris Grot and Rakesh Kumar

Published in

IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2023

A Storage-Effective BTB Organization for Servers

Truls Asheim[1], Boris Grot[2], Rakesh Kumar[1]
1) Norwegian University of Science and Technology, Norway
2) University of Edinburgh, UK

Abstract

Many contemporary applications feature multi-megabyte instruction footprints that overwhelm the capacity of branch target buffers (BTB) and instruction caches (L1-I), causing frequent front-end stalls that inevitably hurt performance. BTB capacity is crucial for performance as a sufficiently large BTB enables the front-end to accurately resolve the upcoming execution path and steer instruction fetch appropriately. Moreover, it also enables highly effective fetch-directed instruction prefetching that can eliminate a large portion of L1-I misses. For these reasons, commercial processors allocate vast amounts of storage capacity to BTBs.

This work aims to reduce BTB storage requirements by optimizing the organization of BTB entries. Our key insight is that storing branch target offsets, instead of full or compressed targets, can drastically reduce BTB storage cost as the vast majority of dynamic branches have short offsets requiring just a handful of bits to encode. Based on this insight, we size the ways of a set associative BTB to hold different number of target offset bits such that each way stores offsets within a particular range. Doing so enables a dramatic reduction in storage for target addresses. Our final design, called BTB-X, uses an 8-way set associative BTB with differently sized ways that enables it to track about 2.24x more branches than a conventional BTB and 1.3x more branches than a storage-optimized state-of-the-art BTB organization, called PDede, with the same storage budget.

8.1 Introduction

Contemporary server applications feature massive instruction footprints stemming from deeply layered software stacks. These footprints may far exceed the capacity of the branch target buffer (BTB) and instruction cache (L1-I), resulting in the so-called front-end bottleneck. BTB misses may lead to wrong-path execution, triggering a pipeline flush when misspeculation is detected. Such pipeline flushes not only throw away tens of cycles of work but also expose the fill latency of the pipeline. Similarly, L1-I misses cause the core front-end to stall for tens of cycles while the miss is being served from lower-level caches.

BTB stands at the center of a high-performance core front end for three key reasons: it determines the instruction stream to be fetched, it identifies branches for the branch direction predictor, and it affects the L1-I hit rate. Specifically, by identifying control flow divergences, the BTB ensures that the branch direction predictor can make predictions for upcoming conditional branches. For predicted-taken and unconditional branches, the BTB supplies targets to which instruction fetch should be redirected. Finally, the BTB together with the direction predictor enables an important class of instruction prefetchers called fetch-directed instruction prefetchers (FDIP) [29–31, 42], which rely on the BTB to discover L1-I prefetch candidates.

Considering the criticality of capturing the large branch working sets of modern workloads, commercial CPUs feature BTBs with colossal capacities, a trend also observed by [23]. With each BTB entry potentially requiring 8 bytes or more (Section 8.2), BTB storage costs can easily reach into tens and even hundreds of KBs. Indeed, the Samsung Exynos M6 mobile processor allocates a staggering 529KB of on-chip storage to BTBs [20]. Not only the BTB storage cost is high, it is increasing at a rapid pace. For example, the Samsung Exynos BTB storage budget increased nearly six fold (98.9KB to 561.5KB) from M2 to M6, over a period of about eight years [20]. While such massive BTBs are effective at capturing branch working sets, they do so at staggering area costs.

To reduce the BTB storage cost, prior work [46–48] has focused on compressing the branch targets as they account for the majority of BTB storage budget as shown in Figure 8.1. Concretely, Seznec [46, 47] observes that all branch targets within a page share the same page number and BTB storage requirements can be significantly reduced by storing the page number only once per page instead of once per target. To exploit this observation, he partitions the BTB in two structures, Main-BTB and Page-BTB, each storing different portions of branch targets. The Main-BTB stores the page offset and a pointer to the Page-BTB entry that stores the page number. The state-of-the-art BTB organization, PDede [48], further observes that target addresses span significantly fewer *regions* than pages, where a region is a group of contiguous pages. Therefore, it partitions the BTB even further and introduces a Region-BTB that lowers page number storage cost as the region number for all pages inside a region is stored only once. By storing page/region numbers only once for all branches in a page/region, these BTBs avoid information duplication, thus reducing storage requirements.

Though, these designs significantly reduce BTB storage requirements, they introduce several complexities that increase their access latency and power requirements. First, these designs introduce a level of indirection, i.e., on a BTB access, Main-BTB is accessed first to get the pointers to the Page-BTB and Region-BTB and only then these BTBs can be accessed. This sequential access, Main-BTB followed by Page/Region-BTBs, increases the overall BTB access latency which either requires a two-cycle BTB lookup or a longer clock period. Both of these alternatives incur a performance penalty. Second, on allocating a new BTB entry, Page-BTB and Region-BTB need to be searched to check if the page/region number for the target address is already present or not. As the page/region number can be anywhere in Page/Region-BTB, a fully-associative associative search is required [47] which increases BTB power requirements. An alternative is to restrict the locations where a page/region number can be stored in Page/Region-BTB [48]; however, it increase the likelihood of conflict misses.

This work seeks to reduce BTB storage requirements without increasing BTB complexity. To that end, we propose to store target offsets, defined as delta between the address of the branch instruction and that of its target, instead of full or compressed (i.e., page offset, page number, and region number) targets. Our key insight is that target offsets are unevenly distributed but tend to require significantly fewer bits to represent than full and even compressed

Figure 8.1: Composition of an entry in conventional BTB. The numbers are the number of bits required to encode each field.

target addresses. Our analysis reveals that 54% of dynamic branches require only 6 bits or fewer for offset encoding, while a meager 1% of branches need 25 bits or more to store their offsets.

Based on this insight, we propose to store target offsets in the BTB rather than compressed or full target addresses, which can be up to 64 bits long depending on the size of virtual address space. To accommodate the varied distribution of target offsets, we size different ways of a set associative BTB to hold different number of offset bits such that each way stores only those branches whose target offsets can be encoded with a certain number of bits. In doing so, we not only significantly reduce BTB storage requirements but also avoid the complexities, indirection and fully-associative searches of the state-of-the-art BTB designs.

This paper introduces BTB-X, a simple yet highly storage-effective BTB organization, that incarnates our idea of storing target offsets. BTB-X is a set associative BTB with its ways sized to store different sized target offsets. Our evaluation shows that BTB-X can track about 2.24x more branches than a conventional BTB storing full targets and about 1.3x more branches than PDede, a state-of-the-art BTB organization, with the same storage budget. Conversely, BTB-X can accommodate the same number of branches as conventional BTB and PDede while requiring 2.24x and 1.3x less storage. This work makes the following key contributions:

- We show that storing branch target offsets, instead of full or compressed target addresses, can provide drastic BTB storage savings because about 54% of branches require only 6 bits or fewer to encode their offsets. A further 22% of branches require between 7 and 10 bits.
- We show that the target offset sizes are unevenly distributed with 0-6 bits, 7-10 bits, and 11-25 bits required to encode the offsets of 54%, 22% and 23% of branches respectively. Therefore, a single size offset field cannot provide storage optimal solution.
- We introduce BTB-X, a simple and highly storage-effective BTB organization, that stores target offsets instead of targets themselves. Furthermore, BTB-X ways are sized to hold different sized target offsets.
- We demonstrate that, with the same storage budget, BTB-X can accommodate about 2.24x and 1.3x more branches than a conventional BTB and PDede, a state-of-the-art BTB. Our evaluation further shows that BTB-X outperforms the conventional BTB even when provisioned with just half the storage budget.

8.2 Background and Motivation

Branch prediction unit predicts the program control flow and supplies a stream of instruction addresses/program counters (PCs) on the predicted path to the fetch unit which fetches the corresponding instructions to feed the rest of the core. As branch instructions disturb the otherwise sequential control flow, the branch prediction unit needs to identify them to

predict the upcoming control flow. However, whether an instruction is a branch or not can only be determined after it has been fetched and decoded. To avoid the latency of fetching and decoding instructions before generating next PCs, the branch prediction unit employs a special hardware structure, called branch target buffer (BTB), to identify branch instructions solely from their PCs before the instructions themselves are even fetched.

8.2.1 Branch Target Buffer (BTB)

Figure 8.1 presents the conventional BTB organization. Each BTB entry consists of *valid*, *tag*, *type*, *target*, and *rep_policy* fields. Figure 8.1 also shows the typical number of bits needed for these fields. The *tag* field usually stores only a partial tag, which is generated by hashing the full tag, to reduce storage cost while introducing minimal aliasing. The number of bits for *target* field depends on the size of virtual address space and instruction set architecture (ISA). We assume a 48-bit virtual address space and ARMv8 ISA to calculate target field size in Figure 8.1. As ARMv8 instructions are always 32-bits and 4-byte aligned, the least significant two bits of a PC are always zeros. Therefore, we only need 46-bits for the *target* field. The *valid* bit indicates whether the entry contains valid information or not, while *rep_policy* bits choose one of the existing branches for eviction when a new branch is inserted in the BTB.

To check whether a PC corresponds to a branch instruction, the BTB is indexed, i.e. accessed, with the low order PC bits. The high order PC bits are hashed, using the same function that is used to generate partial tags, and compared to the *tag* field of the indexed BTB entry. A match indicates that the PC belongs to a branch.

A branch instruction simply implies the presence of a potential control flow divergence point in program execution. However, whether or not the divergence actually happens depends on the type of branch, i.e. call, return, conditional, or unconditional branch, which is stored in the *type* field of a BTB entry. Call, return, and unconditional branches always cause control flow divergence as they are always *taken*. Conditional branches, in contrast, are not always taken and a direction predictor is used to predict their direction.

If a branch is predicted to be taken, the *target* field in the BTB entry provides the address for the next instruction, except for returns. This is because the return address is call-site dependent and a given function can be called from different call sites. Therefore, a return address stack (RAS) is typically employed to record return addresses at call-sites. On a function call, the call instruction pushes the return address to RAS, which is later popped by the corresponding return instruction.

8.2.2 The cost of a BTB miss

A BTB miss for a branch instruction means that the branch is undetected and the front-end continues to fetch instructions sequentially. Whether or not the sequential path is the correct one depends on the actual direction of the missed branch. Unless the missed branch is a conditional branch that is not taken, the sequential path is incorrect. When the wrong path is eventually detected by the core, all the instructions after the branch that missed in the BTB are flushed, fetch is redirected to the branch target and pipeline is filled with correct-path instructions. BTB misses are thus highly deleterious to performance as they result in a loss of tens of cycles of work and expose the pipeline fill latency.

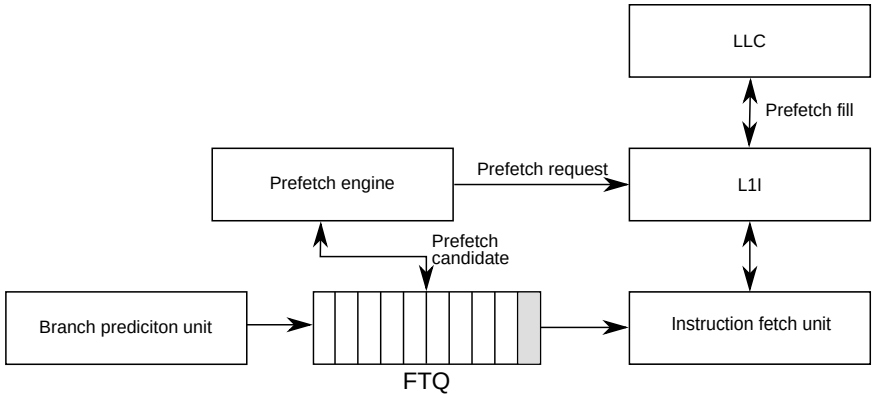


Figure 8.2: FDIP microarchitecture

8.2.3 BTB's role in instruction prefetching

Fetch-directed instruction prefetchers are a class of powerful L1-I prefetchers that intrinsically rely on BTB to identify prefetch candidates. These prefetchers are highly effective and, when coupled with a sufficiently large BTB, outperform the winner of the recently-concluded Instruction Prefetching Championship [1] and approach the performance of an ideal L1-I, as reported by Ishii et al. [23]. Variants of these prefetchers have been adopted in commercial products, for example in IBM z15 [44], ARM Neoverse N1 [40] etc.

Figure 8.2 shows a canonical organization of a fetch-directed instruction prefetcher (FDIP) [42]. As originally proposed, FDIP decouples the branch-prediction unit and the fetch engine via the *fetch target queue (FTQ)*. This decoupling allows the branch prediction unit to run ahead of the fetch engine and discover prefetch candidates by predicting the control flow far into the future. With FDIP, each cycle, the branch prediction unit identifies and predicts branches to anticipate upcoming execution path and inserts corresponding instruction addresses into the FTQ. Consequently, the FTQ contains a stream of anticipated instruction addresses to be fetched by the core. The prefetch engine scans the FTQ to identify prefetch candidates and issue prefetch requests.

For FDIP to be effective, the BTB needs to accommodate the branch working set, otherwise frequent BTB misses will cause FDIP to prefetch the wrong path as FTQ will be filled with wrong path instruction addresses. This is one of the key reasons why commercial processors deploy massive BTBs, as also observed by [23]. These massive BTBs incur astronomical storage overheads. Also, not only the BTB storage overhead is high, it is increasing at a rapid pace. For example, Table 8.1, presents the BTB storage cost in several generations of Samsung Exynos processors. As the table shows, the BTB storage cost nearly doubled in each generation, except between M4 and M5. Overall, the storage cost increased nearly six fold (98.9KB to 561.5KB) from M1 to M6, over a period of about eight years [20].

As the instruction footprints of server applications continue to expand, a trend also reflected in Google Web Search workload whose instruction footprint is growing at annualized rate of 27% [24], the BTB sizes and their storage overheads are destined to increase in future. Therefore, there is an urgent need to investigate storage-effective BTB organizations to combat the front-end bottleneck without necessitating prohibitive area budgets.

Table 8.1: BTB storage cost in Samsung Exynos processors

CPU	BTB Storage
M1/M2	98.9KB
M3	175.8KB
M4	288.0KB
M5	310.8KB
M6	561.5KB

Bit position	48	...	9	8	7	6	5	4	3	2	1
Branch PC	0	...	1	0	1	1	0	1	0	0	0
Branch Target	0	...	1	0	1	1	1	1	0	0	0
Target Offset		...					1	1	0	0	0

Figure 8.3: Branch target offset example

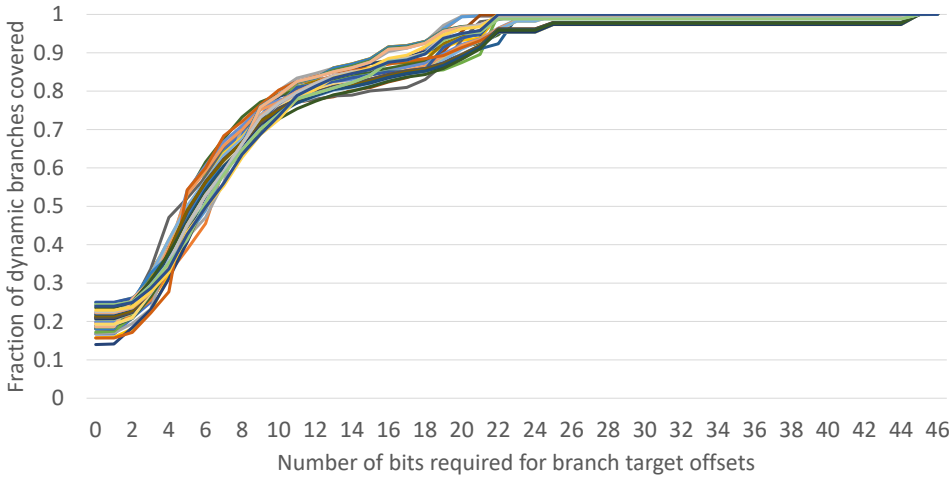


Figure 8.4: Distribution of branch target offsets in different workloads.

8.3 Branch Target Distance Analysis

The storage cost of branch targets accounts for a major fraction of BTB storage requirements. For example, in a conventional BTB, as depicted in Figure 8.1, the target field accounts for about 72% (46 of 64 bits) of the total BTB storage requirements. We analyze the number of bits required for branch target *offsets* to assess if storing the offsets, instead of the full or compressed targets, can reduce BTB storage requirements. We define the target offset as the n least significant bits of target address, with n being the position of most significant bit that differs among branch PC and target. As an example, for the branch PC and target shown in Figure 8.3, the most significant bit that differs among them is at position five, whereas all bits at positions higher than five are same. Therefore, the target offset for this branch PC and target pair is '11000', i.e., the target bits from position 5 to 1. Also, as our modelled ARM v8 ISA aligns instructions at 4-byte boundaries, the two least significant bits of a target are always zeros. Therefore, we only need to store '110' as offset in the BTB.

An advantage of defining an offset as n lower order target bits, instead of the numerical

distance between branch PC and target (i.e., target - PC), is that the full target can be recovered by simply concatenating the shifted branch PC with the offset retrieved from BTB. In contrast, using numerical distance as offset would require a 48-bit adder to recover the full target from offset.

Figure 8.4 plots the distribution of branch target offsets in the branch working sets of our workloads. The data includes both conditional and unconditional branches; hence, it comprehensively covers the full branch working set. The X-axis shows the number of bits required to store offsets, while the Y-axis plots the fraction of dynamic branches covered.

As the figure shows, short offsets dominate the distribution with 54% of branches requiring only six bits or fewer for their offsets. A further 22% of branches only require between 7 and 10-bits to represent their offsets. The reason why such a high fraction of offsets is short is that conditional branches dominate the dynamic branch working set, and they tend to have short offsets [31]. This is because conditional branches generally guide the control flow only inside a function; meanwhile, software engineering principles favor small functions, thus restricting conditional branch target offsets to short distances. Furthermore, as discussed in Section 8.2, return instructions get their target from RAS, thus they do not need to store any target bits in BTBs. Therefore, for the purpose of this analysis, we assume 0-bit offsets for return instructions.

Perhaps surprisingly, Figure 8.4 also shows that very few branches require a large number of bits for their offset. Indeed, a meagre 1% of branches requires more than 25 bits for their offsets. The sum of these results indicates that reserving space for the full 46-bit target address results in an appalling under-utilization of BTB storage, since 99% of branches need at most half the number of bits needed to represent the full target address if offsets are used instead.

We gain two key insights from this analysis:

Key Insight 1 The targets of most branches lie relatively close in the virtual address space to the branch itself. As a result, storing the *distance* to the target, in the form of an offset from the branch instruction can provide drastic storage savings.

Key Insight 2 The target offset sizes are unevenly distributed with 0-6 bits, 7-10 bits, and 11-25 bits required to encode the offsets of 54%, 22% and 23% of branches respectively. Therefore, a single size offset field cannot provide storage optimal solution.

8.4 State-of-the-art BTBs and Their Limitations

Prior work has proposed several BTB organizations that aim to reduce the storage cost by compressing branch targets. This section presents the most representative BTB organizations and analyzes their limitations.

8.4.1 Reduced BTB

Seznec [46] made a critical observation that all branch targets within a page share the same page number and only differ in page offsets. Thus, storing full target addresses in a BTB results in massive duplication of page numbers and wastage of storage capacity. To eliminate this duplication, Seznec proposed Reduced BTB (R-BTB), a variant of which was also used in ITTAGE [47]. The key innovation of R-BTB is to store a pointer to the page number rather than storing the page number itself in BTB.

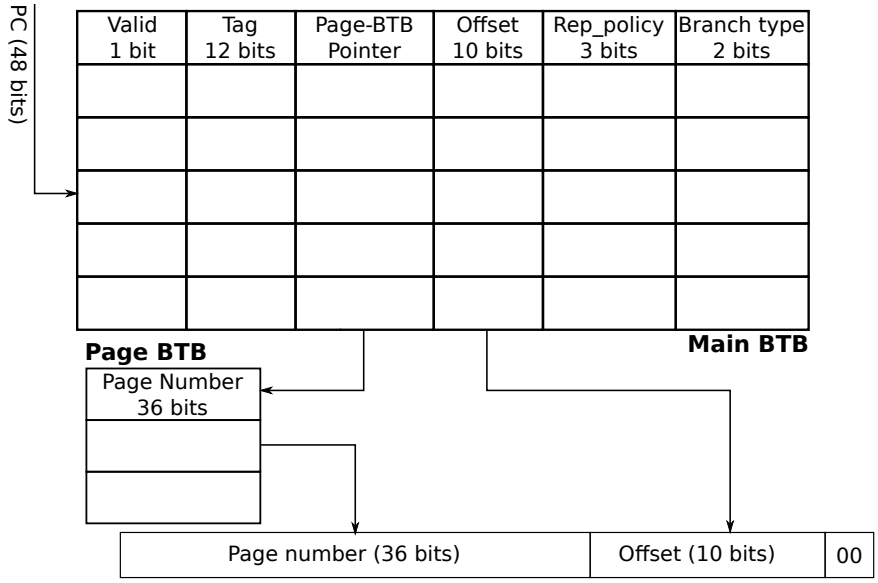


Figure 8.5: Reduced BTB Organization.

Figure 8.5 presents the logical organization of R-BTB and the composition of its entries. R-BTB is composed of two partitions: Main-BTB and Page-BTB. For each branch target, apart from page offset, Main-BTB stores a pointer to the page number. The page number itself is stored in Page-BTB. If two or more branches have their targets in the same page, their Main-BTB entries will hold pointer to the same Page-BTB entry. As the number of pages is significantly smaller than number of branch targets, fewer bits are needed to hold Page-BTB pointers than page numbers themselves. Consequently, by storing a page number only once in Page-BTB, R-BTB avoids duplication and reduces storage requirements.

8.4.2 PDede

PDede [48] is the state-of-the-art BTB organization that comes with three different variants. Figure 8.6 depicts the most storage effective and best performing PDede variant, called PDede-Multi Entry Size. It improves over R-BTB in two aspects. First, it reduces the cost of storing page numbers in the Page-BTB. PDede observes that server applications, due to their large instruction footprints, touch a large number of pages thus increasing Page-BTB storage requirements. They further observe that, as different libraries get dynamically mapped to different locations in address space, the pages tend to form spatial *regions*, where a region consists of multiple contiguous pages. Just like branch targets inside a page share the same page number, the page numbers inside a region share the same region number. To eliminate the duplication of region numbers, as shown in Figure 8.6 PDede introduces a Region-BTB which stores the region number while Main-BTB stores a pointer to it just like it stores a pointer to page BTB.

Second, for the same-page branches, i.e., when the branch and its target are in the same page, PDede does not store page/region numbers as they can be recovered from branch PC. PDede reserves half of the ways in a set associate BTB for same-page branches. As the ways

reserved for same-page branches do not need to store Page-BTB and Region-BTB pointers, as shown in Figure 8.7, PDede achieves additional storage savings.

8.4.3 Limitations of the state-of-the-art

Though R-BTB and PDede achieve significant storage saving by avoiding page and region number duplication, they increase BTB complexity by introducing a level of indirection and associative searches in Page- and Region-BTB. These complexities lead to increased access latency and power requirements. In addition, the state-of-the-art BTB designs are suboptimal in utilizing the available storage budget.

8.4.3.1 Indirection

As Figures 8.5 and 8.6 show, the access to Main-BTB only provides a part of the target address, i.e., page offset. The other parts have to be retrieved from Page-BTB and Region-BTB. Also, Page- and Region-BTB cannot be accessed in parallel with Main-BTB because the Main-BTB access provide the pointers to them. As a result, the sequential Main-BTB and Page-/Region-BTB accesses increase the overall BTB access latency. This additional latency either enforces a two-cycle BTB lookup or necessitates a longer clock period. Both of these alternatives are detrimental to performance. PDede does avoid this indirection penalty to some extent because the same-page branches do not need to access Page-/Region-BTB rather they get their page and region number from the branch PC itself. However, the different-page branches, i.e., where branch PC and target address lie in different pages, do need to pay the indirection penalty.

8.4.3.2 Associative searches

On allocating a new BTB entry, all BTB partitions (Main-BTB, Page-BTB, and Region-BTB) may need to be updated. The replacement policy chooses the entry to be replaced in Main-BTB. However, Page- and Region-BTB need to be searched to check if the page/region number for the incoming target is already present or not. As the page/region number can be present anywhere in the Page/Region-BTB, ITTAGE [47] uses fully-associative searches which increase the power requirements especially when the number of entries grows. PDede (partially) solves this limitation by restricting the number of entries where a page number can reside for a given branch target to 16. However, limiting the number of entries increases the likelihood of conflict misses.

8.4.3.3 Suboptimal storage utilization

Though R-BTB and PDede significantly improve storage utilization over conventional BTB, they still miss plenty of opportunity. This is because, in case of R-BTB, it uses a fixed size target representation for all branches, i.e., a 10-bit offset plus a fixed sized page-BTB pointer. In contrast, our analysis of Section 8.3 shows a large variance in target offset sizes that naturally makes the single sized organization of R-BTB storage inefficient as it needs to be sized for the largest target offset. For example, as shown by Figure 8.4, 54% of target offsets fit in six or fewer bits; however, R-BTB needs to use all 10+ bits for these branches, thus resulting in a high storage under-utilization. PDede provides slightly better storage utilization than R-BTB as it has differently sized entries for Same-page and Different-page branches. However,

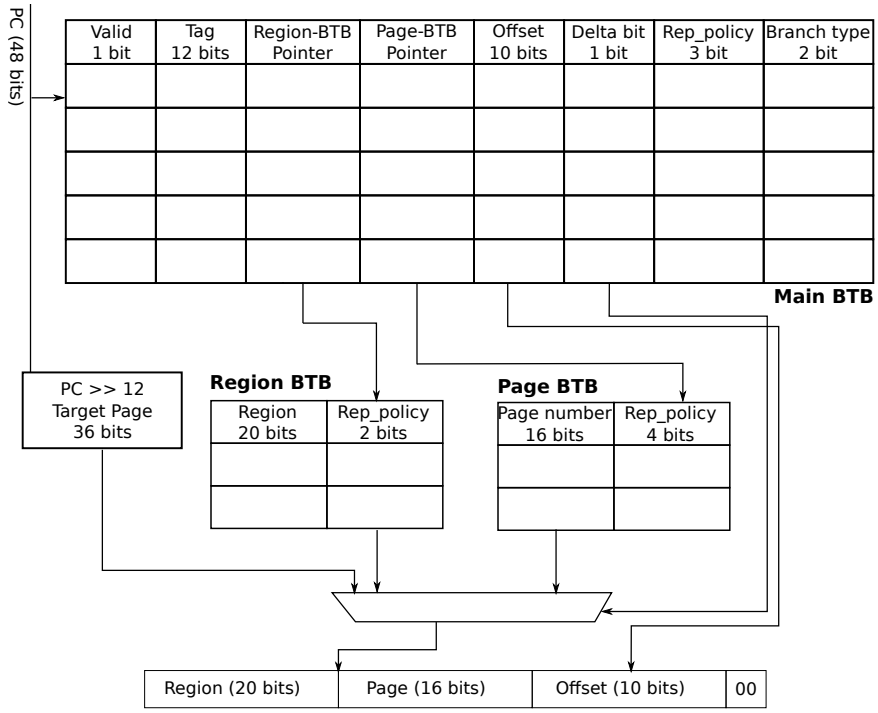


Figure 8.6: PDede BTB Organization.

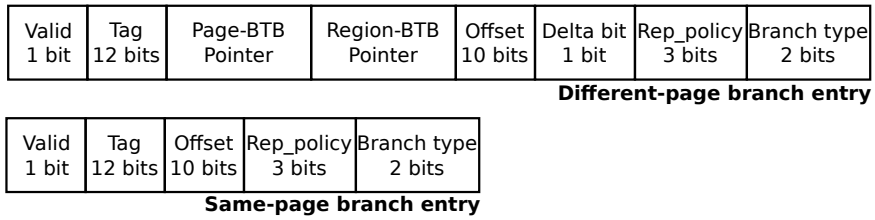


Figure 8.7: Different- and same-page PDede entry composition.

BTB entries of only two different sizes, i.e. Same-page and Different-page, are not enough to capture the large offset size variance (Figure 8.4) observed in server applications.

8.5 BTB-X

BTB-X is a simple and storage-effective BTB organization. Building on the insights gained in Section 8.3, it stores target offsets, instead of full targets, to minimize storage requirements while also accounting for the large variance in target offset sizes. Its microarchitecture and entry/set composition are shown in Figure 8.8.

Figure 8.8: BTB-X organization and entry/set composition.

8.5.1 BTB-X Organization

The offset field in BTB-X needs to accommodate the uneven distribution of target offset sizes as observed in Section 8.3. In principle, this field should be sized such that the BTB can store the largest target offset. However, as the largest offset can be nearly as large as the full target, sizing the *offset* field this way would nearly eliminate the potential storage saving from storing offsets. An attractive alternative is to size the offset field such that it can store the majority of offsets. Looking at Figure 8.4, an *offset* field of 25-bits would capture more than 99% of branch target offsets as they requires 25-bits or fewer. However, there are two major drawbacks to this scheme. First, it still leads to poor storage utilization. This is because, as Figure 8.4 implies, 54% of branches would waste more than three quarters of *offset* storage capacity as they require only 6-bits or fewer for their offsets. Another 25% of branches would waste nearly half of the offset storage as their offsets fit in 7-12 bits. Second, all branches that need more than 25 bits for their offsets can not fit in the BTB and will always cause BTB misses. Though, as there are very few such branches (< 1%), their impact is likely to be small.

To minimize the storage under-utilization, we size different ways of a set associative BTB-X to hold different sized target offsets. A branch is allocated to a way whose offset field is at least as large as the number of bit required to store the target offset. We use an 8-way set associative BTB-X and leverage the data in Figure 8.4 to appropriately size the offset field of each way such that each way covers about 12.5% dynamically executed branches. Figure 8.4 shows that, on average, 0-, 4-, 5-, 7-, 9-, 11-, 19-, and 25-bit offsets cover about 20%, 36%, 46%, 61%, 72%, 79%, 90%, and 99% dynamic branches. Therefore, we size the 8-ways of BTB-X ways to hold 0-, 4-, 5-, 7-, 9-, 11-, 19-, and 25-bit target offsets respectively. Notice that about 20% of dynamic branches that require 0-bits for their offset are return instructions that read their target from RAS, as discussed in Section 8.2. Therefore, way-0 of BTB-X does not feature any storage for target offsets. Though return instruction do not get their target from BTB, they still need to be allocated BTB entries so that the branch prediction unit can identify them and pick their target from RAS while generating instruction stream to be fetched.

BTB-X covers 99% of the dynamically executed branches and we employ a very small conventional direct-mapped BTB, called BTB-XC, that stores full target addresses for the remaining 1% branches. Reserving a way in BTB-X for such branches would unnecessarily increase the storage requirements as these branches require much fewer entries than the number of sets in BTB-X. Indeed, based on our analysis, we size BTB-XC to store 64x fewer entries than BTB-X, i,e, 8x fewer entries than the number of sets in BTB-X.

8.5.2 Accessing BTB-X

8.5.2.1 BTB-X Lookup

A BTB-X lookup is very similar to a conventional BTB lookup as shown in Figure 8.8. It is accessed with the index bits of a PC and all eight ways are looked up in parallel. Also, BTB-XC is looked up in parallel with BTB-X. The main difference between a conventional BTB lookup and BTB-X lookup is that a BTB-X lookup provides target offset, rather than full target address, if the lookup hits in way-1 to way-7. Thus, target offset needs to be concatenated with branch PC to get the full target address. The number of bits to be concatenated from branch PC depends on the BTB-X way in which the lookup hits. For example, a hit in way-1 provides 4 lower order bits of target while the rest needs to be concatenated from branch PC. Further, a hit in way-0 implies that the full target is in RAS, while a hit in BTB-XC provides the full target address.

Table 8.2: Microarchitectural parameters

Parameter	Value
Fetch	6-wide, 128-instruction FTQ
Branch Predictor	Hashed Perceptron
Return address stack	64 entries
Scheduler	128 entries
Re-order buffer	352 entries
Load queue	128 entries
Store queue	72 entries
L1-I	32 KB, 8-way, 4 cycle latency, 8 MSHRs
L1-D	48 KB, 12-way, 5 cycle latency, 16 MSHRs
L2	512 KB, 8-way, 14/15 cycle latency, 32 MSHRs
LLC	2MB, 16-way, 34/35 cycle latency, 64 MSHRs

8.5.2.2 BTB-X Allocation

As with any existing BTB organization, BTB-X entries are allocated (or updated) as branch instructions retire. The number of bits required to represent a branch target offset determines the way(s) where a branch can be allocated an entry. For example, return instructions can be allocated entries in any of the ways, based on replacement policy's decision, as they have no offset and can fit in all ways. Other branches have fewer ways where they can be allocated entries that are determined by the minimum number of bits required to store their offsets. For example, if a branch requires 20 bits for its target offset, it cannot be allocated in way-0 to way-6.

BTB-X uses a slightly modified least recently used (LRU) replacement policy. Concretely, we modify it to compare the LRU counters of only the entries that can accommodate the target offset and replace the one that is least recently used among them. All other aspect of LRU, such as counter updates, stay exactly the same as in baseline policy.

8.6 Evaluation

8.6.1 Methodology

We use ChampSim [12] to evaluate the efficacy of BTB-X on server and client workload traces provided by Qualcomm for the first Instruction Prefetching Championship (IPC-1) [1]. We warm up microarchitectural structures for 50M instructions and collect statistics over the next 50M. The microarchitectural parameters for the modeled processor, resembling Intel Sunny Cove [14], are listed in Table 8.2.

We improved two important aspects of Champsim to evaluate the baseline, state-of-the-art, and proposed BTB organizations. First, being a trace-driven simulator, Champsim detects branches by consulting the information available in the traces, rather than looking up a BTB. This essentially translates to Champsim using an ideal BTB. Therefore, we first implement a realistic conventional BTB (Conv-BTB), presented in Section 8.2, in Champsim.

Table 8.3: BTB-X storage requirements. The numbers in parentheses are for BTB-XC.

Entries	Sets	Set size	Storage
256(4)	32(4)	224(64)-bits	0.9KB
512(8)	64(8)	224(64)-bits	1.8KB
1K(16)	128(16)	224(64)-bits	3.6KB
2K(32)	256(32)	224(64)-bits	7.25KB
4K(64)	512(64)	224(64)-bits	14.5KB
8K(128)	1024(128)	224(64)-bits	29KB
16K(256)	2048(256)	224(64)-bits	58KB

Second, Champsim resolves all branches in execute stage, i.e., branch mispredictions are detected and the fetch is resteered to correct path only when a mispredicted branch instruction reaches the execute stage. Such branch resolution overestimates the misprediction penalty of unconditional direct branches. This is because such branches can be resolved in the decode stage (hence, fetch can be resteered sooner) as they are always taken, thus the PC of the next instruction can be compared to the target encoded in the branch instruction to detect mispredictions. Further, taken conditional branches that miss in BTB but are correctly predicted by the direction predictor, can also be resolved in the decode stage. To do so, the fetch stage passes the direction prediction for all instructions, despite BTB hit/miss, to decode stage. If decode identifies a branch that missed in the BTB but predicted taken by the direction predictor, it resteers the fetch to the target encoded in the branch instruction, thus reducing BTB miss penalty. Given that the direction predictors are highly accurate, this optimization reduces average BTB miss penalty. Overall, we improve branch resolution so that the unconditional direct branches and the taken conditional branches that miss in BTB are resolved in the decode stage. Finally, BTB is updated at commit stage by only the taken branches (both conditional and unconditional).

8.6.2 Storage breakdown

We first assess the number of branches different BTB organizations (Conv-BTB, PDede, and BTB-X) can accommodate in a given storage budget compared to each other. We use storage budgets required for storing 256, 512, 1K, 2K, 4K, 8K, and 16K branches in BTB-X as presented in Table 8.3. Our calculations assume a 48-bit virtual address space and BTB-X entry compositions presented in Figure 8.8. To double the number of entries in BTB-X, we double the number of sets while keeping the associativity same. Notice that Table 8.3 presents set size instead of entry size. This is because BTB-X features different sized entries in different ways; however, the set size remains constant.

Table 8.4 presents the number of branches the different BTB organizations can track at different storage budgets. PDede distributes the overall BTB storage budget among its Main-BTB, Page-BTB, and Region-BTB. We follow the distribution used by its inventors[48] to allocate the budget among different PDede BTBs as shown in Table 8.4. Accordingly, for 29KB storage budget, we configure PDede to use 1K Page-BTB entries and about 6K Main-BTB entries. While halving the storage budget to lower values, we halve the number of entries in the Main-BTB as well as the Page-BTB. Halving the number of Page-BTB entries reduces the number of bits required to store Page-BTB pointer in the Main-BTB. Thus, the Main-BTB entry size reduces with the reduction in storage budget. Further, we use four Region-BTB entries across all storage budgets, so Region-BTB requires a fixed storage of

Table 8.4: Number of branches in different BTB organizations at various storage budgets.

Storage	BTB-X + BTB-XC	PDede				Conv-BTB	
	Branches	Page-BTB budget	Main-BTB budget	Entry Size	**Branches**	Entry Size	**Branches**
0.9KB	256 + 4	0.078KB	0.817KB	32-bits	210	64-bits	116
1.8KB	512 + 8	0.156KB	1.645KB	32.5-bits	415	64-bits	232
3.6KB	1K + 16	0.312KB	3.3KB	33-bits	820	64-bits	464
7.25KB	2K + 32	0.625KB	6.6KB	33.5-bits	1617	64-bits	928
14.5KB	4K + 64	1.25KB	13.2KB	34-bits	3190	64-bits	1856
29KB	8K + 128	2.5KB	26.5KB	34.5-bits	6292	64-bits	3712
58KB	16K + 256	5KB	53KB	35-bits	12405	64-bits	7424

0.0107KB. Also recall that PDede reserves half of the ways in a set for same-page branches while the other half can store both same-page and different-page branches. Therefore, its entries are of two different sizes. The PDede entry size shown in Table 8.4 is the average of two sizes.

As the table shows BTB-X stores significantly more branches than any other BTB organizations. Concretely, it stores 2.24x more branches than a conventional BTB organization. Compared to PDede, BTB-X stores 1.24x more branches at 0.9KB storage budget and 1.34x more branches at 58KB storage budget. BTB-X's advantage over PDede increases with storage budget because PDede entries require more bits at higher budgets to accommodate larger Page-BTB pointers.

8.6.3 BTB MPKI

To understand the advantage of higher BTB-X branch density, we measure misses per 1000 instructions (MPKI) that different BTB organizations incur on client and server workloads. Since BTB misses for not-taken branches do not hurt performance, we only consider the BTB misses for taken branches. For this analysis, we assume a BTB storage budget of 14.5KB that corresponds to 4160-, 3190-, and 1856-entries in BTB-X, PDede and Conv-BTB respectively. The results are presented in Figure 8.9.

As the figure shows, server workloads experience significantly higher MPKI compared to client workload due their massive instruction and branch footprints. The figure also shows that BTB-X provides a much lower MPKI compared to both conventional BTB and PDede especially on server workload. Concretely, on average, conventional BTB incurs 25 MPKI on server workload as it stores the least amount of branches among the three organization for a given storage budget. PDede is able to lower the MPKI to 13.7 while BTB-X brings it further down to 9.5. The advantage of BTB-X over other organizations is particularly evident on very high MPKI workloads, i.e., server_023 to server_035, where it provides much lower MPKI compared to conventional BTB and PDede.

8.6.4 Performance

To assess how the reduced MPKI translates to performance, we compare the performance of the three BTB organizations on client and server workloads. Recall from Section 8.2 that a larger BTB delivers two distinct benefits: 1) it reduces the incidence of pipeline flushes by detecting branches in the upcoming control flow and 2) it facilitates instruction prefetching when coupled with FDIP. Thus, we compare the performance gains achieved by the three BTB organizations by evaluating them with FDIP.

Figure 8.10 presents the performance gains obtained on server and client traces. The results are normalized to the performance of the Conv-BTB without any instruction prefetching. The figure shows three bars for each workload. The first bar presents performance gain achieved by Conv-BTB when copuled with FDIP. The second and third bars present the performance gains achieved by PDede and BTB-X respectively. The PDede and BTB-X bars divide the performance gain into contributions from fewer pipeline flushes and from better instruction prefetching stemming from capturing more branches in the BTB.

Looking at the overall performance gain with instruction prefetcher (FDIP), the figure shows that BTB-X provides a geometric mean gain of 39% over baseline on server workloads. In comparison, PDede and Conv-BTB deliver a performance gain of only 33% and 24% on these workloads. Looking at individual workloads, BTB-X comprehensively outperforms

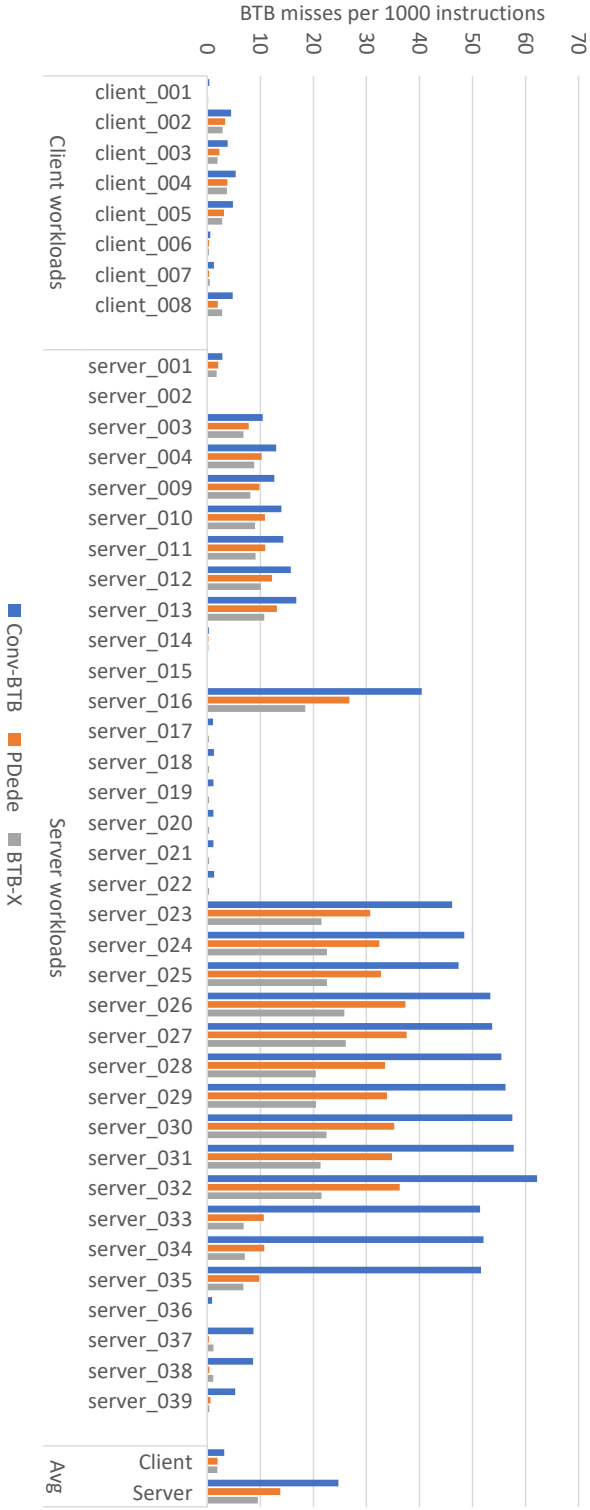


Figure 8.9: BTB MPKI experienced by different BTB organizations.

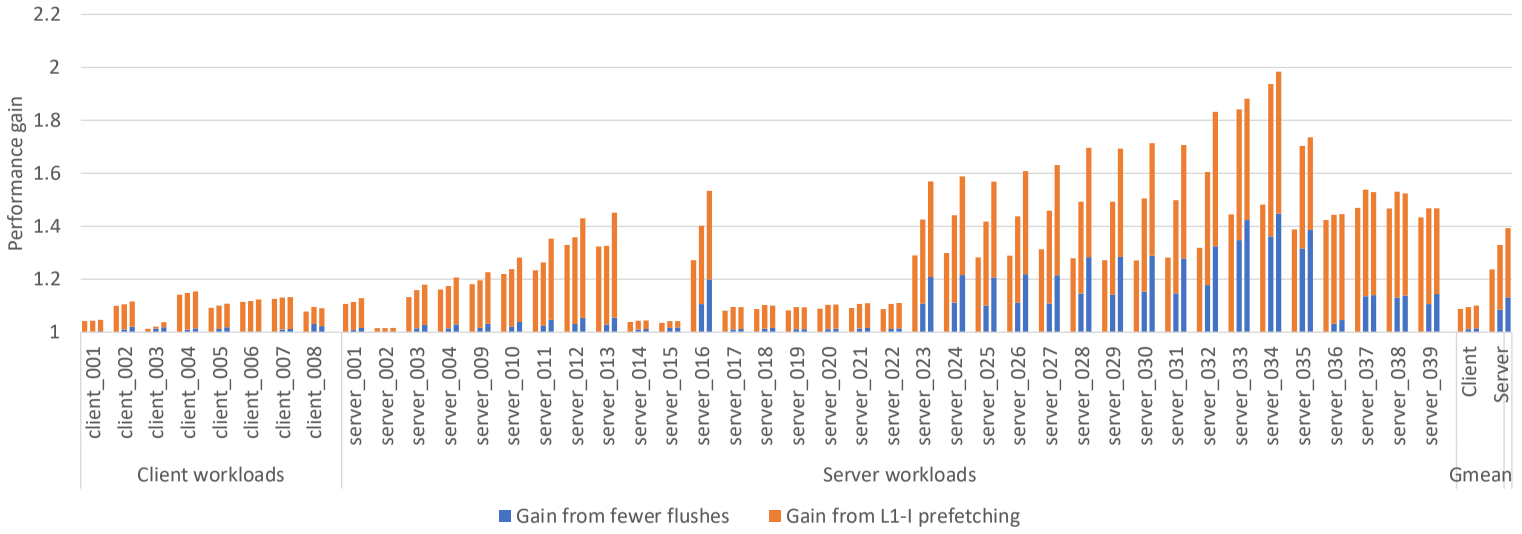


Figure 8.10: Performance gain obtained by conventional BTB (with FDIP), PDede and BTB-X (with and without FDIP) over the conventional BTB without FDIP. The three bars for each workload correspond to Conv-BTB, PDede, and BTB-X respectively.

Table 8.5: Energy requirements of different BTB designs.

BTB	Access Type	Energy (Per access)	#Accesses	Energy (Total)
Conv-BTB	Read	13.2pJ	1.60E+08	2122µJ
	Write	25.2pJ	4.36E+06	110µJ
	Total Energy			**2232µJ**
PDede	Main-BTB Read	8.4pJ	1.24E+08	1047µJ
	Main-BTB Write	12.5pJ	5.74E+05	7µJ
	Page-BTB Read	0.9pJ	2.01E+06	2µJ
	Page-BTB Write	0.8pJ	2.04E+04	0.02µJ
	Page-BTB Search	6.2pJ	2.14E+05	2µJ
	Total Energy			**1058µJ**
BTB-X	Read	8.5pJ	1.16E+08	994µJ
	Write	11.4pJ	4.03E+05	5µJ
	Total Energy			**999µJ**

PDede and Conv-BTB on server_023 to sever_32. For example, on server_032, BTB-X provides 83% speedup over baseline whereas PDede and Conv-BTB achieve only 60% and 32% performance gain. This is because the branch working set of these workloads starts to fit in BTB-X due to its higher branch capacity. As a result, BTB MPKI lowers which not only reduces pipeline flushes and but also keeps FDIP on correct prefetch path for longer intervals.

Looking at the results without instruction prefetcher, Figure 8.10 shows that BTB-X provides 13% performance gain over the baseline Conv-BTB whereas PDede is achieves 8% gain. On individual workloads, BTB-X achieves significantly high gain over Conv-BTB and PDede on workloads from server_23 to server_32 even without FDIP. Figure 8.10 also shows the FDIP by itself performs better with more number of BTB entries. For example, on server_32 FDIP with Conv-BTB provides 32% performance gain. With PDede, the performance gain from prefetching increases to 42% and with BTB-X it further increases to 51%.

These results show that by accommodating more branches in a given storage budget, BTB-X not only reduces pipeline flushes but also improves instruction prefetching, both lead to better performance.

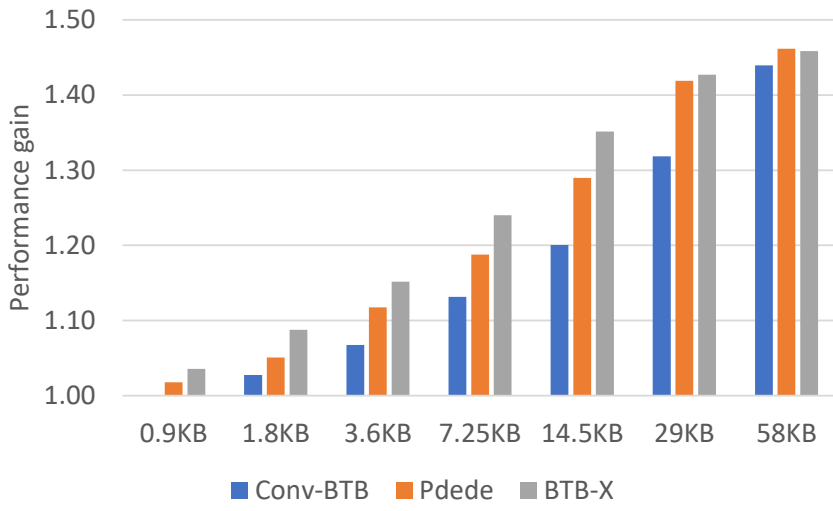
Finally, Figure 8.10 shows that all three BTB organizations perform similar on client workloads. This is because their branch working sets mostly fit in the baseline Conv-BTB and the additional entries in PDede and BTB-X do not bring much performance benefit.

8.6.5 Energy and delay analysis

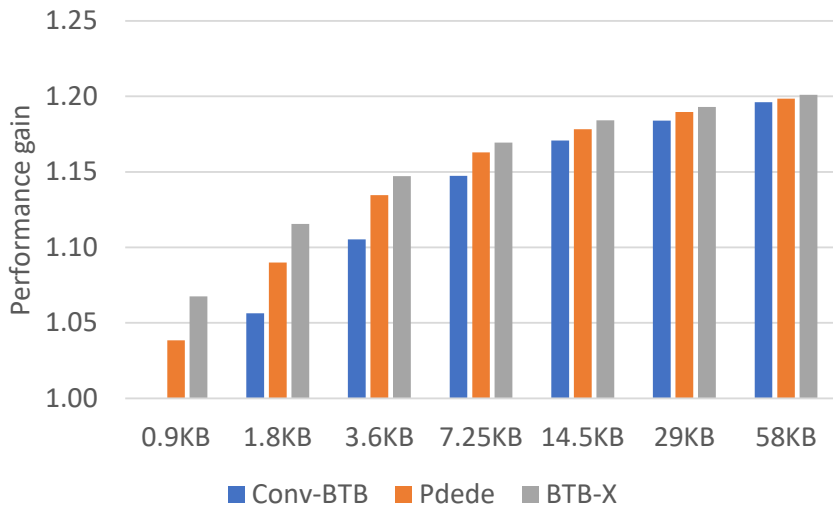
We use Cacti 7.0 [34] to analyze the energy requirements and access latencies of Conv-BTB, PDede, and BTB-X at 22 nm, which is the most recent technology node supported by Cacti. For this analysis we assume the same storage budget, i.e. 14.5KB, as used in Section 8.6.4 for the performance analysis.

8.6.5.1 Energy requirements

Table 8.5 shows the per access read and write energy requirements of different BTB designs. As the table shows, BTB-X and PDede's Main-BTB incur very similar per access read and write energy cost. However, in addition to Main-BTB, PDede also needs to access Page-BTB



(a) Server workloads



(b) Client workloads

Figure 8.11: Performance gains for conventional BTB, PDede, and BTB-X on (a) **server** and (b) **client** workloads over a conventional BTB with 0.9KB storage budget. X-axis label is storage requirements of 256-, 512-, 1K-, 2K-, 4K-, 8K-, and 16K-entry BTB-X.

for different-page branches, i.e., the branches that have their targets in a different page than the branches themselves. Further, the Page-BTB needs to be searched on a BTB write to check if the target page number is already in Page-BTB or not. Consequently, PDede's per access read and write energy for different page branches reaches 9.3 pico Joules (pJ) and 19.5 pJ, respectively, compared to 8.5pJ and 11.4 pJ of BTB-X. PDede also features a Region-BTB; however, its energy requirements are negligible and, thus, not shown in Table 8.5. Finally, Conv-BTB's per access energy cost is significantly higher than BTB-X as its each read and write access requires 13.2pJ and 25.2pJ respectively.

Table 8.5 also shows the number of read/write accesses, averaged across the workloads, and the total energy consumption. Despite very similar per access energy cost, PDede' Main-BTB consumes considerably higher energy than BTB-X. This is because PDede often goes on the wrong execution path due to its higher MPKI. These additional wrong path BTB accesses, reflected in higher BTB reads in Table 8.5, result in higher energy consumption. Further, PDede needs to handle more BTB writes than BTB-X because it holds fewer branches, which results in frequent replacements. Thus, the total energy consumption of PDede reaches 1058µJ compared to 999µJ of BTB-X. Finally, the energy requirements of Conv-BTB are significantly higher, 2232µJ, than BTB-X because of higher per access energy and higher number of total accesses.

Overall, this analysis shows that BTB-X not only delivers better performance than PDede but also consumes less energy, thus providing much better energy efficiency.

8.6.5.2 Access Latency

Our analysis shows that the Conv-BTB requires about 0.36ns to complete an access. As discussed in Section 8.4.2, PDede's access latency is the sum of Main-BTB and Page-BTB access latencies as these two structures are accessed sequentially. Our analysis shows that the Main-BTB and Page-BTB accesses require 0.34ns and 0.13ns, respectively, thus resulting in an overall PDede access latency of 0.47ns which is considerably higher than Conv-BTB latency. To address this, PDede employs multi-cycle BTB accesses: the Main-BTB is accessed in the first cycle, and the Page-BTB is accessed in the next cycle only if the branch is predicted to be taken and it's target is in a different page than the branch. Thus, the same page branches need one cycle and the taken different page branches need two cycles to get their target address from PDede. Finally, our analysis shows that a BTB-X access takes only 0.33ns. In summary, this analysis shows that BTB-X provides better storage efficiency without any adverse effects on the access latency.

8.6.6 Performance variation with BTB storage budget

To further understand the performance advantage of BTB-X over PDede and Conv-BTB, we compare their performances across different storage budgets. Figure 8.11 presents the performance gains obtained on server and client workloads. The results are normalized to the performance of Conv-BTB with 0.9KB storage budget. Instruction prefetching is enabled in all designs including baseline.

As the figure shows, on server workloads, BTB-X provides significantly higher performance than the Conv-BTB and PDede for equal storage budgets of up to 29KB and 14.5KB respectively. The performance advantage of BTB-X is pronounced on server traces whose large instruction footprints pressure the BTB and L1-I. For instance, BTB-X provides 35% performance gain over the baseline compared to 29% and 20% of PDede and Conv-BTB

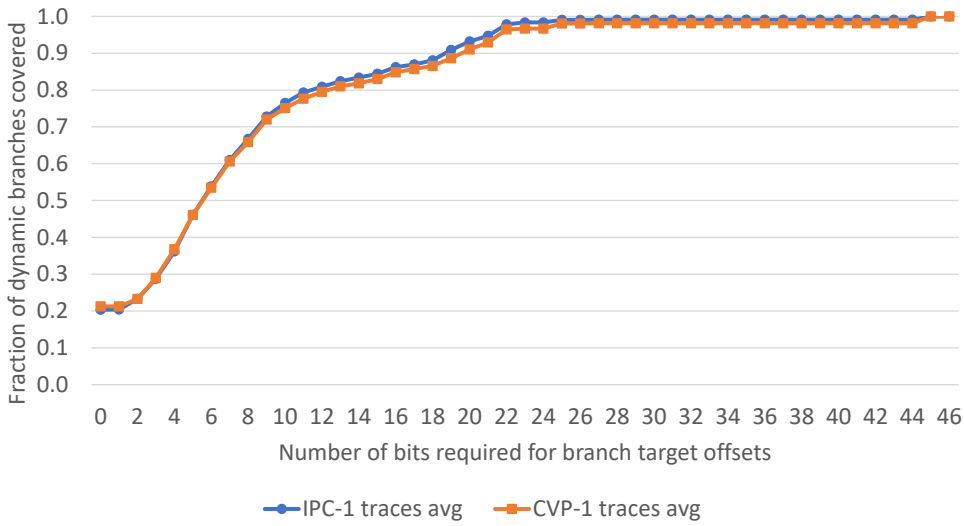


Figure 8.12: Target offset distribution in CVP-1 and IPC traces.

respectively at 14.5KB budget. At large BTB storage budgets, the branch working sets of many workloads start to fit in the available BTB capacity, at which point the performance gap between BTB-X and the other two designs diminishes. Also, the performance gap between the three BTB organizations levels off earlier on client trace due to their smaller instruction working sets.

A key take-away from this figure is that BTB-X outperforms the conventional BTB even when it is given just half the storage budget of its conventional counterpart. For example, in Figure 8.11a, the Conv-BTB improves performance by 20% with a 14.5KB budget whereas BTB-X provides a 24% improvement with just 7.25KB. The reason for this phenomenon is that BTB-X accommodates 2.24x more entries than Conv-BTB of equal storage budget; thus, halving BTB-X's budget still gives a slight capacity advantage over Conv-BTB.

8.6.7 Analyzing target offset distribution in more workloads

We study the target offset distribution in 750+ Qualcomm server traces that were provided for the first Championship Value Prediction(CVP-1) [19]. The results, presented in Figure 8.12, show that their offset distribution is very similar to the distribution in IPC-1 traces presented in Figure 8.4. This study confirms that such an offset distribution is a consequence of how the applications are written and the resulting control-flow behavior. As discussed in Section 8.3, such offset distribution stems from the fact that the conditional branches dominate dynamic branch working set and they tend to have short offsets. This is because conditional branches guide the control-flow inside functions, and software engineering principles favor small functions. Consequently, short offsets dominate the branch offset distribution.

In addition to CVP-1 traces, we analyze five more server applications - Wordpress [53], Mediawiki [52], and Drupal [51] from Facebook's HHVM OSS-performance benchmarks [15], Kafka [50] from Java DaCapo [8], and Finagle-HTTP [49] from Java Renaissance [41]. Further, these applications are compiled to x86 (CVP-1 and IPC-1 traces are compiled to Arm64)

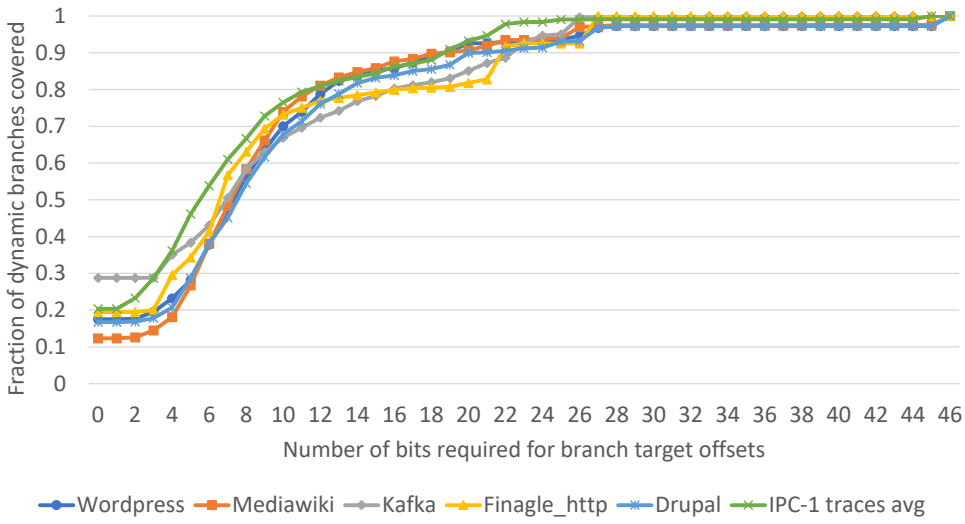


Figure 8.13: Target offset distribution in x86 compiled server applications and Arm64 IPC traces.

which also enables us to assess the impact of ISA on target offset distribution. The results presented in Figure 8.13 show that the offset distribution in these applications is also very similar to that in IPC-1 traces. The only difference is that x86 traces require slightly larger offsets (1 or 2-bits more) to achieve a similar dynamic branch coverage as the Arm64 (CVP-1 and IPC-1) traces. For example, 6-bit offsets cover about 54% branches in Arm64 traces, whereas x86 offsets need 8-bits to achieve 58% branch coverage. This is because x86 offsets specify the distance between branch PC and target in number of *bytes* because x86 instruction are variable size. In contrast, Arm64 offsets specify this distance in number of *instructions* because all instructions are 4-bytes, thus saving 2 offset bits.

As BTB-X needs to store slightly larger offsets for x86 than Arm64, we reassess its storage advantage over PDede and Conv-BTB for x86 architectures. As each way in 8-way BTB-X needs to cover about 12.5% of branches, we size its ways to store offset of 0-, 5-, 6-, 7-, 9-, 12-, 20-, and 27-bits based on the offset distribution in x86 applications shown in Figure 8.13. Thus, each set needs 86-bits for offsets compared to 80-bit in Arm64. Consequently, BTB-X's storage advantage is slightly lower for x86 than Arm64. However, BTB-X still stores 2.18x more branches than Conv-BTB for x86 (2.24x for Arm). Compared to PDede, BTB-X stores 1.21x more branches (1.24x for Arm64) at 0.9KB storage budget and 1.31x more branches (1.34x for Arm64) at 58KB storage budget. (Section 8.6.2 presents this analysis of Arm64 traces.)

8.7 Related work

8.7.1 Mitigating BTB misses

BTB was first disclosed by Losq [35] and was further expanded by Lee et al [32]. Since BTB lies on the critical path for instruction delivery, there has been several proposals to increase its effectiveness. Instead of accessing BTB with the PC of each individual instruction, Yeh et al.

[54] proposed to access it with basic-block address and store not only the target but also the fall-through address in the BTB. In case the branch is predicted to be not taken, the fall-through address is used, after fetching the current basic-block, as the next PC for both instruction fetch as well as for the next BTB access. The advantage of such a BTB organization over the conventional BTB organization is that it reduces BTB bandwidth and power requirements as a single access provides the next control flow divergence point, whereas the conventional organization requires as many accesses as the number of instructions until the next branch. Whereas the initial proposal on basic-block-based BTB [54] stores full fall-through address, the later work [43] proposed to store the delta between two sequential basic-block addresses. Fagin [16] proposed to use the BTB storage more effectively by storing only the partial tags. To further amortize the tag storage cost, some designs proposed to share a BTB entry among multiple branches that reside in the same cache block [2, 26]. Though these BTB designs aim to improve different aspects of BTB management, they all share a common trait, i.e., they store full target addresses. Thus, the key idea of BTB-X can be applied to all of these BTB designs to reduce their target storage cost.

Prior work [4, 5, 22, 46–48] has also explored mechanisms to reduce the storage cost of branch targets. Seznec [46, 47] proposed to break the target address into page number and offset; and store a pointer to the page number, along with the page offset, in the BTB while the page number itself is stored in a separate structure. It reduces the storage cost as a pointer to page number is smaller than the page number itself, and the page number for all the targets in a page is stored only once. Hoogerbrugge [22] proposed to size some of the entries in a set for storing small target offsets, thus reducing BTB storage requirements.

The state-of-the-art BTB design, PDede [48], combines these two ideas to address their individual limitations. Concretely, Sezenc's design is sub-optimal for same-page branches as it unnecessarily stores (pointer to) their target page number even though it is same as the page number of their branch PCs. In contrast, Hoogerbrugge's design is sub-optimal for inter-page branches as it stores their full targets. Inspired from Hoogerbrugge's design, PDede sizes some entries in a set to store same-page branch targets; and similar to Seznec's design, for inter-page branches, it stores pointers to page numbers instead of page numbers themselves. PDede further reduces the inter-page target storage cost by dividing the page number into page- and region-number. However, as it is based on Seznec's design, it also has to pay the addition latency cost of indirection between main-BTB and the page-/region-BTB. Micro BTB [21], proposes a flexible BTB entry structure where each entry can store either one branch, if its offset is large, or two branches if their offsets are small. We show that all these designs are sub-optimal in exploiting the storage optimization opportunity presented by the uneven branch offset distribution. BTB-X not only captures this opportunity but also avoids the BTB indirection of the state-of-the-art.

Apart from optimizing BTB organization, prior work [9, 11, 28, 30, 31] has also explored BTB prefilling/prefetching to mitigate BTB misses. The state-of-the-art in BTB prefetching is a profile guided software prefetcher, called Twig [28]. It analyzes an application's execution profile to identify critical BTB misses and then injects software prefetch instructions. The prefetch instruction takes compressed branch PC and target as operands and its execution fills this information in BTB. These prefetching techniques are complementary to BTB organization and, thus, can be used along with BTB-X.

8.7.2 Mitigating L1-I misses

As L1-I misses continue to be a major performance limiter in server applications[6, 24, 45], prior work has proposed both hardware and software mechanisms to mitigate L1-I misses. On the hardware side, state-of-the-art temporal stream prefetchers [17, 18] record the L1-I miss/access history and replay it to discover prefetch candidates. While such prefetchers are highly effective, their huge metadata storage cost renders them impractical despite recent attempts to address this weakness [25, 26]. Fetch-directed prefetchers use in-core structures (BTB and branch direction predictor) to run ahead of the fetch unit to find prefetch candidates. While the early work [42] focused on L1-I prefetching only, the state-of-the-art fetch-directed prefetchers [30, 31] also prefill into the BTB.

Several purely-software based approaches to instruction prefetching and improving the L1-I capacity has also been proposed [3, 7, 13, 27, 33, 36–39]. These methods use data from application profiling to perform either compile-time, link-time or post-link time optimizations. Since these methods are software-only they will benefit from the increased BTB capacity provided by the BTB-X organization.

8.8 Conclusion

The multi-megabyte instruction footprints of contemporary server applications cause frequent BTB and L1-I misses, which have become major performance limiters. Because BTB capacity greatly affects front-end performance by dictating pipeline flush rate and the efficacy of fetch-directed instruction prefetching, commercial products allocate tens to hundreds of KBs of storage to BTBs. We observe that the single largest contributor to the BTB storage cost is the cost of storing branch target. We further observe that BTB storage cost can be drastically reduced by storing target offsets instead of full or even compressed targets. This is because targets of most branches lie relatively close to the branches themselves and our analysis shows that more than 99% of offsets can be represented with at most half the bits required to store the full targets. Based on these observations, we propose a storage-effective BTB organization, called BTB-X, that stores target offsets in place of target address. Furthermore, BTB-X, an 8-way set associative BTB, uses differently sized ways with each storing offsets of a different length, thus accounting for the uneven distribution of offset lengths. Overall, BTB-X is capable of storing about 2.24x more branches than a conventional BTB and 1.3x more branches than a state-of-the-art BTB organization within the same storage budget.

Acknowledgements

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

8.9 References

[1] *1st Instruction Prefetching Championship*. `https://research.ece.ncsu.edu/ipc/` (cit. on pp. 72, 80).

[2] *AMD Software Optimization Guide. Section 2.8.1.2*. `https://www.amd.com/system/files/TechDocs/56665.zip` (cit. on p. 91).

[3] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. "Call graph prefetching for database applications". In: *ACM Transactions on Computer Systems (TOCS)* 21.4 (2003), pp. 412–444 (cit. on p. 92).

[4] Truls Asheim, Boris Grot, and Rakesh Kumar. "BTB-X: A Storage-Effective BTB Organization". In: *IEEE Computer Architecture Letters* 20.2 (2021), pp. 134–137 (cit. on p. 91).

[5] Truls Asheim, Boris Grot, and Rakesh Kumar. "A Specialized BTB Organization for Servers". In: *Proceedings of the 31st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2022 (cit. on p. 91).

[6] Truls Asheim et al. "Impact of Microarchitectural State Reuse on Serverless Functions". In: *Proceedings of the Eighth International Workshop on Serverless Computing*. WoSC '22. Quebec, Quebec City, Canada: Association for Computing Machinery, 2022, pp. 7–12 (cit. on p. 92).

[7] Grant Ayers et al. "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers". In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019, pp. 462–473 (cit. on p. 92).

[8] Stephen M Blackburn et al. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2006, pp. 169–190 (cit. on p. 89).

[9] J. Bonanno et al. "Two Level Bulk Preload Branch Prediction". In: *International Symposium on High-Performance Computer Architecture*. 2013, pp. 71–82 (cit. on p. 91).

[10] *BTB-X GitHub Ripository*. `https://github.com/rakeshdhakla/ChampSim-master-BTBX` (cit. on p. 97).

[11] Ioana Burcea and Andreas Moshovos. "Phantom-BTB: a virtualized branch target buffer design". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*. 2009, pp. 313–324 (cit. on p. 91).

[12] *ChampSim Simulator*. `https://github.com/ChampSim/ChampSim` (cit. on p. 80).

[13] Dehao Chen, Tipp Moseley, and David Xinliang Li. "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications". In: *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2016, pp. 12–23 (cit. on p. 92).

[14] *Examining Intel's Ice Lake processors: Taking a bite of the Sunny Cove microarchitecture*. `https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3` (cit. on p. 80).

[15] *facebookarchive/oss-performance: Scripts for benchmarking various php implementations when running open source software*. `https://github.com/facebookarchive/oss-performance`. 2022 (cit. on p. 89).

[16] Barry Fagin. "Partial resolution in branch target buffers". In: *IEEE Transactions on Computers* 46.10 (1997), pp. 1142–1145 (cit. on p. 91).

[17] Michael Ferdman et al. "Temporal Instruction Fetch Streaming". In: *International Symposium on Microarchitecture*. 2008 (cit. on p. 92).

[18] Michael Ferdman et al. “Proactive Instruction Fetch”. In: *International Symposium on Microarchitecture*. 2011 (cit. on p. 92).

[19] *First Championship Value Prediction*. `https://www.microarch.org/cvp1/cvp1online/contestants.html` (cit. on p. 89).

[20] Brian Grayson et al. “Evolution of the Samsung Exynos CPU Microarchitecture”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 40–51 (cit. on pp. 69, 72).

[21] Vishal Gupta and Biswabandan Panda. “Micro BTB: A High Performance and Storage Efficient Last-Level Branch Target Buffer for Servers”. In: *Proceedings of the 19th ACM International Conference on Computing Frontiers*. CF ’22. Turin, Italy: Association for Computing Machinery, 2022, pp. 12–20 (cit. on p. 91).

[22] Jan Hoogerbrugge. “Cost-Efficient Branch Target Buffers”. In: *Euro-Par 2000 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 950–959 (cit. on p. 91).

[23] Yasuo Ishii et al. “Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective”. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pp. 172–182 (cit. on pp. 69, 72).

[24] Svilen Kanev et al. “Profiling a Warehouse-Scale Computer”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA ’15. Portland, Oregon: Association for Computing Machinery, 2015, pp. 158–169 (cit. on pp. 72, 92).

[25] Cansu Kaynak et al. “SHIFT: Shared History Instruction Fetch for Lean-core Server Processors”. In: *International Symposium on Microarchitecture*. 2013 (cit. on p. 92).

[26] Cansu Kaynak, Boris Grot, and Babak Falsafi. “Confluence: Unified instruction supply for scale-out servers”. In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 166–177 (cit. on pp. 91, 92).

[27] Tanvir Ahmed Khan et al. “I-spy: Context-driven conditional instruction prefetching with coalescing”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 146–159 (cit. on p. 92).

[28] Tanvir Ahmed Khan et al. “Twig: Profile-Guided BTB Prefetching for Data Center Applications”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 816–829 (cit. on p. 91).

[29] Rakesh Kumar and Boris Grot. “Shooting Down the Server Front-End Bottleneck”. In: *ACM Trans. Comput. Syst.* 38.3–4 (Jan. 2022) (cit. on p. 69).

[30] Rakesh Kumar, Boris Grot, and Vijay 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 ’18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 30–42 (cit. on pp. 69, 91, 92).

[31] Rakesh Kumar et al. “Boomerang: A Metadata-Free Architecture for Control Flow Delivery”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 493–504 (cit. on pp. 69, 74, 91, 92).

[32] Lee and Smith. “Branch Prediction Strategies and Branch Target Buffer Design”. In: *Computer* 17.1 (1984), pp. 6–22 (cit. on p. 90).

[33] David Xinliang Li, Raksit Ashok, and Robert Hundt. "Lightweight feedback-directed cross-module optimization". In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 2010, pp. 53–61 (cit. on p. 92).

[34] Sheng Li et al. "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2011, pp. 694–701 (cit. on p. 86).

[35] J. Losq. ""Generalized history table for branch prediction (in pipeline computers)". In: *IBM Tech. Disclosure Bull* 1 (1982) (cit. on p. 90).

[36] C-K Luk et al. "Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 15–26 (cit. on p. 92).

[37] Chi-Keung Luk and Todd C Mowry. "Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors". In: *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE. 1998, pp. 182–193 (cit. on p. 92).

[38] Guilherme Ottoni and Bertrand Maher. "Optimizing function placement for large-scale data-center applications". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 233–244 (cit. on p. 92).

[39] Maksim Panchenko et al. "Bolt: a practical binary optimizer for data centers and beyond". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 2–14 (cit. on p. 92).

[40] Andrea Pellegrini et al. "The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC". In: *IEEE Micro* 40.2 (2020), pp. 53–62 (cit. on p. 72).

[41] Aleksandar Prokopec et al. "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". In: *Programming Language Design and Implementation*. 2019 (cit. on p. 89).

[42] G. Reinman, B. Calder, and T. Austin. "Fetch directed instruction prefetching". In: *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. 1999, pp. 16–27 (cit. on pp. 69, 72, 92).

[43] Glenn Reinman, Todd Austin, and Brad Calder. "A Scalable Front-End Architecture for Fast Instruction Delivery". In: *Proceedings of the 26th Annual International Symposium on Computer Architecture*. ISCA '99. Atlanta, Georgia, USA: IEEE Computer Society, 1999, pp. 234–245 (cit. on p. 91).

[44] Anthony Saporito. "The IBM z15 processor chip set". In: *Hot Chips*. 2020 (cit. on p. 72).

[45] David Schall et al. "Lukewarm Serverless Functions: Characterization and Optimization". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 757–770 (cit. on p. 92).

[46] André Seznec. "Don't Use the Page Number, but a Pointer to It". In: *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. ISCA '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 104–113 (cit. on pp. 69, 74, 91).

[47] André Seznec. "A 64-Kbytes ITTAGE indirect branch predictor". In: *J. Instruction-Level Parallelism* (2011) (cit. on pp. 69, 74, 76, 91).

[48] Niranjan K Soundararajan et al. "PDede: Partitioned, Deduplicated, Delta Branch Target Buffer". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 779–791 (cit. on pp. 69, 75, 81, 91).

[49] *Twitter finagle*. `https://twitter.github.io/finagle/` (cit. on p. 89).

[50] Wikipedia contributors. *Apache Kafka — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Apache_Kafka&oldid=988898935`. 2022 (cit. on p. 89).

[51] Wikipedia contributors. *Drupal — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Drupal&oldid=989582664`. 2022 (cit. on p. 89).

[52] Wikipedia contributors. *MediaWiki — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=MediaWiki&oldid=989993176`. 2022 (cit. on p. 89).

[53] Wikipedia contributors. *WordPress — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=WordPress&oldid=977243718`. 2022 (cit. on p. 89).

[54] Tse-Yu Yeh and Yale N. Patt. "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution". In: *Proceedings of the 25th Annual International Symposium on Microarchitecture*. MICRO 25. Portland, Oregon, USA: IEEE Computer Society Press, 1992, pp. 129–139 (cit. on p. 91).

Appendix 8.A Artifact Appendix

8.A.1 Abstract

We implement BTB-X in Champsim simulator. Our artifacts provide the following: 1) BTB-X implementation in Champsim, 2) Link to workload traces, 3) Scripts for generating configuration files, launching simulations, and collecting results, and 4) Excel file for plotting the most important results. We identify three key results for artifact evaluation: a) Branch Target Offset distribution (Figure 4), b) BTB MPKI reduction (Figure 9), and c) Performance improvement (Figure 10).

8.A.2 Meta-information

- **Compilation** Tested with GCC 8.5.0. It should also work with other recent GCC versions.
- **Code/Workloads** Download code/workloads from the provided link.
- **Experiments** Modify the provided scripts (as described below) to run simulations.
- **Metrics** IPC, BTB MPKI, Branch Target Offset distribution.
- **Time needed to run experiments** Less than 30 minutes when running all traces in parallel.
- **Plotting graphs** Excel file, *BTBX_artifact_results.xlsx*, is provided to plot graphs.

8.A.3 Access to artifacts

- **Code** Download BTB-X implementation from [10].
- **Workloads** The workloads can be downloaded from

  `https://drive.google.com/file/d/1qs8t8-YWc7lLoYbjbH_d3lf1xdoYBznf/view?usp=sharing`

  Place the workloads in <*Path_to_code/dpc3_traces/*.
- **Excel file** For plotting the graphs, download excel file, *BTBX_artifact_results.xlsx*, from [10].

8.A.4 System requirements

Any hardware capable of running Champsim is sufficient. SLURM is recommended to run simulation on a cluster. The scripts are written in bash.

8.A.5 Experiment workflow

- **Compilation** Champsim needs to be compiled with three BTB designs (convBTB, pdede, and BTBX) and two instruction prefetchers (no, fdip). Follow the instructions at [10] to compile the code.

  Important note on compilation IFETCH_BUFFER needs to be 128 entries when compiling with "fdip" prefetcher and "FETCH_WIDTH*2" entries when compiling with "no" prefetcher. This is because of how instruction fetch is implemented in baseline Champsim. IFETCH_BUFFER size is defined in line 63 of <*Path_to_code*>*/inc/ooo_cpu.h*.
- **Generating configuration files** Go to directory <*Path_to_code*>*/launch/scripts/*. In the script file *createConfig.sh*, point PATH_TO_CHAMPSIM to <Path_to_code>. Run this script (*./createConfig.sh*) to generate config files needed by Champsim.
- **Running simulations**

  Running all workloads: Go to the directory <*Path_to_code*>*/launch/*. In script file *launch.sh*, replace the line <cluster_launch_command_here>(line 64) with the command to run experiments on your cluster. A sample command is given that runs experiments on our cluster. Running this script (*./launch.sh*) will run simulations, and the stats will be stored in directory <*Path_to_code*>*/results_50M/*.

  Running a single workload: An example command to run simulation for a single workload is provided at [10].

8.A.6 Results

- **Collecting results** Go to <*Path_to_code*>*/collectStats/*. Run the script *getResults.sh*, and it will collect results from all workloads and save them in a file *all_res*.
- **Plotting Results** Download the *all_res* file. Open the provided excel file *BTBX_artifact_results.xlsx*. Click on "Data" in MS-Excel top menu bar. Click on "Refresh All" in "Queries and Connections" ribbon, go to the folder where you stored *all_res* and double click on *all_res*. Now "Offset Distribution", "MPKI", and "Performance" sheets in the excel file should have plots for Figure 4, Figure 9, and Figure 10 respectively.

Chapter 9

Paper B1 – CoFaaS: Automatic Transformation-based Consolidation of Serverless Functions

Authors

Truls Asheim, Magnus Jahre and Rakesh Kumar

Submitted to

Eurosys 2024

CoFaaS: Automatic Transformation-based Consolidation of Serverless Functions

Truls Asheim[1], Magnus Jahre[1], Rakesh Kumar[1]
1) Norwegian University of Science and Technology, Norway

Abstract

The attractive property of decoupling deployment decisions from application development has led to fast adoption of the Function-as-a-Service (FaaS) cloud computing model. FaaS applications are typically highly modular with each function having a specific purpose and well-defined interface as this enables code reuse, simplifies maintenance, improves testability and provides language independence. To enable these attractive features, FaaS applications often use Remote Procedure Call (RPC) interfaces for inter-function communication — which comes at the cost of orders of magnitude higher latency than native function calls. Prior works that alleviate this overhead are unattractive because they either require non-standard APIs, only support a single language, or rely on futuristic languages or runtimes.

Our key insight is that we can exploit the well-defined RPC interfaces to perform code transformations that alleviate inter-function communication overhead. We hence propose CoFaaS which leverages this insight to consolidate FaaS functions on top of a WebAssembly runtime and thereby avoid accessing the network layer when functions are deployed on the same compute node. Our evaluation shows that this strategy is highly effective and reports that CoFaaS reduces inter-function communication latency by up to 100× and application-level request round-trip time by up to 6×.

9.1 Introduction

Function-as-a-Service (FaaS) computing is becoming an increasingly popular cloud computing model that simplifies the cloud application lifecycle by moving critical decisions about application deployment from the developer to the provider. The unit of composition in the FaaS model is a *function*. A FaaS function is stateless and therefore produces a deterministic mapping from input to output. When a function is deployed, it is configured to be triggered based on specific events, such as an incoming HTTP request. In the FaaS model, building complex functions that perform multiple tasks are generally discouraged. Instead, the FaaS computing model encourages assigning each function a single well-defined purpose and interface as this promotes code reuse, simplifies maintenance and enhances testability [25]. Larger

FaaS *applications* are built by composing individual functions. FaaS adaptation is increasing in industry and is currently used by 50% of organizations that rely on cloud computing technologies [8].

A key advantage of the FaaS computing model is that applications can be composed of functions written in any language. Language independence is attractive for three reasons. Firstly, it allows a function to be written in the language that supports the purpose of the function in the best possible way. Secondly, it allows applications to be gradually rewritten and ported to other languages function by function. Finally, it allows applications to continue relying on tested and stable units of functionality even if the implementation language of other parts of the application changes.

When developing traditional monolithic applications, combining languages is typically cumbersome because it requires crufty low-level foreign function interfaces that are complex and error-prone to use. In FaaS, programming language interoperability is achieved by design. This is because FaaS functions use highly abstract and language-independent Remote Procedure Call (RPC) interfaces to communicate across a network fabric, for example Google's gRPC [11]. Enabling functions to communicate, regardless of language and internal implementation details, fundamentally requires that external interfaces are specified in a formal, language-independent and declarative manner. Modern RPC interfaces hence provide Interface Definition Languages (IDLs) to specify the contract of each function, i.e., encoding the specifics of the calls that the function supports. Code generators then take the IDL definition as input and outputs a concrete function interface.

Unfortunately, FaaS' attractive properties of simplified deployment, modular design and language-independence come with a severe performance penalty, in part due to its reliance on RPC for inter-function communication. Whereas a local function call in a monolithic application incurs a latency of less than a single microsecond, an RPC call in a FaaS application can incur a latency of several milliseconds — a difference of several orders of magnitude. When a client issues an RPC request, the request is first serialized to the RPC wire format, then the serialized request is send across the network and on the receiver's side, the request is deserialized and processed. Each of these steps take time, however, the primary contributor to the overhead of issuing an RPC request comes from accessing the network transport layer.

Alleviating inter-function communication overheads in FaaS applications is hence critically important, and a rich body of prior work has investigated this issue [5, 12, 13, 15, 20, 21]. While these proposals effectively reduce communication latencies, they do this by relinquishing one or more of the properties that make FaaS attractive in the first place. In particular, they either 1) provide a non-standard API demanding that existing functions need to be rewritten, 2) restrict FaaS applications to be implemented in a single language or 3) propose entirely new FaaS programming models or runtime environments that do not readily integrate with the current cloud computing platforms.

There is hence a need for an approach that reduces inter-function communication overhead while using standard APIs, retaining language independence *and* being easily integratable with current cloud computing platforms — and our goal in this work is to provide such a system. Towards that end, we leverage that inter-function RPC communication interfaces are statically defined. More specifically, the developer specifies the interface of each function using an established IDL which means that we have significant leeway in how to generate function interfaces while respecting the IDL specification. In particular, accessing the network layer — which we demonstrate is the key contributor to inter-function communication overhead — is entirely unnecessary when the functions are deployed on the same compute node.

We hence propose CoFaaS which automatically consolidates FaaS functions — while respecting the semantics of the function's IDL specification — and thereby completely avoids accessing the network layer when functions are scheduled on the same node. More specifically, CoFaaS first transforms the IDL description used by the target RPC interface to WebAssembly Interface Types (WIT). This enables us to leverage the WebAssembly (Wasm) ecosystem and co-locate FaaS functions on a single Wasm runtime; each function is hosted in its own container to maintain isolation. In this way, CoFaaS uses the Wasm runtime to provide inter-function communication and thereby avoids accessing the network layer when functions are deployed on the same node. Our evaluation shows that this strategy yields significant speedups. CoFaaS reduces inter-function communication latency by up to 100× and application-level request round-trip time by up to 6×. The overhead of applying CoFaaS in production scenarios is minimal because it is fully automatic and only slightly increases compilation and deployment times (see Section 9.4 for details).

To summarize, we present the following key contributions:

- We observe that the well-defined interface definitions of existing FaaS applications provide significant leeway in how to generate function interfaces and that this, in turn, can be leveraged to implement powerful, automated transformations of FaaS applications.
- We exploit the above insight to design CoFaaS, the first automatic function transformation framework that significantly reduces inter-function communication latencies while being standard API compliant, language-independent *and* compatible with current cloud computing platforms.
- We demonstrate that CoFaaS yields significant speedups, i.e., it reduces inter-function message latency by up to 100× and application request round-trip time by up to 6×.

9.2 Background

9.2.1 The FaaS Function Communication Latency

The FaaS computing model provides several advantages to the developer. One powerful advantage is hosting flexibility and that decisions about how to host an application is entirely left to the provider. This saves valuable developer time as managing hosting resources can be a demanding task. Furthermore, to unlock many of the advantages offered by the FaaS computing model, FaaS applications need to be highly modular. This means that the functionality of a fully-fledged FaaS application should be backed by composing multiple smaller FaaS functions. While, in theory, a developer could choose to deploy a FaaS application consisting of only a single function containing all of the required functionality, this would relinquish many of the advantages that the FaaS model has to offer. Most notably, to patch or upgrade a single part of an application, the entire application would need to redeployed, a potentially risky operation. If the FaaS application was developed using a modular architecture consisting of several functions, the application can be updated function by function. Additionally, the loose coupling of FaaS functions means that a function can be written in any language at the discretion of the developer.

This modularity, however, comes with significant overheads. Figure 9.1 shows a schematic of a common deployment scenario for a FaaS application. Each function, marked $f_{1..5}$ is hosted inside a separate container, marked $c_{1..5}$, and the communication between them is

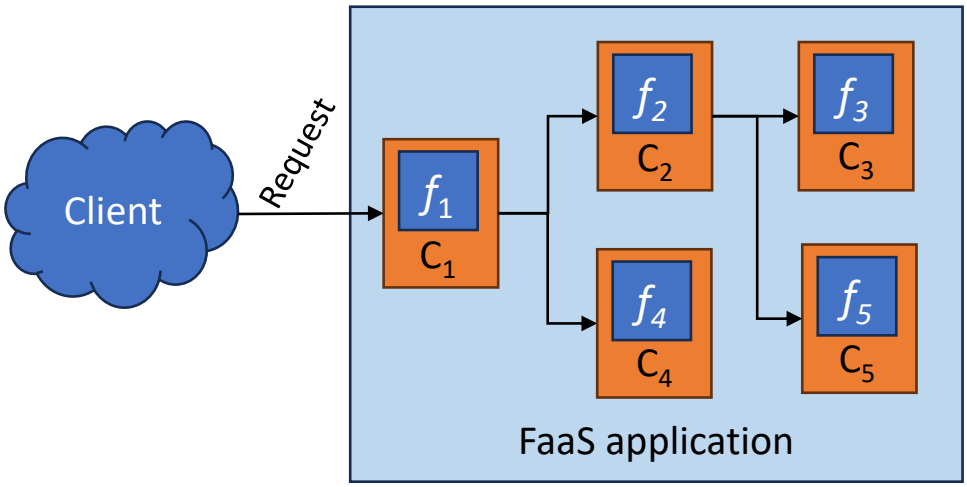


Figure 9.1: A schematic of a FaaS application conventionally deployed showing how an applicaiton is composed of multiple functions that each are hosted in their own container.

done using a network-backed RPC protocol. The use of a network-backed RPC interface in this case facilitates the loose coupling between functions. The main disadvantage, however, is that networked requests have high latency. In our example, whenever the first function f_1 in the FaaS application receives a request it subsequently calls its dependencies in order to perform its tasks. Since each inter-function call comes with the network overhead, examine, for example, the path from f_1 to f_5 in which we pay the RPC penalty twice. For complex FaaS applications, we therefore easily end up paying the RPC penalty multiple times to fulfill a request. Real-world applications can grow significantly more complex, in some cases containing more than 40 functions [10]. Further, applications can perform multiple inter-function requests per operation which causes communication overheads to add up.

This strongly motivates the importance of mitigating this communication latency as the aggregated latencies in complex FaaS applications can quickly grow out of control, significantly harming the performance of real-world applications [10]. Further, it is important to note that accessing the network layer is only strictly necessary for two functions running on different nodes. When possible, Providers prefer to schedule two functions on the same node in order to minimize their communication latency. In this case, using networked communication and thereby inducing the associated overhead is completely unnecessary.

9.2.2 WebAssembly Components

WebAssembly (Wasm) [3] is a portable bytecode format that originally targeted delivering high-performance compiled applications to web browsers. However, since none of Wasm's features explicitly targets web pages, it is also suitable as a general purpose bytecode format. The introduction of the WebAssembly System Interfaces (WASI) gives Wasm POSIX-like capabilities and allows Wasm programs to run outside the browser. A particularly appealing property of Wasm is the increasingly large number of programming languages that supports it as a target.

The WebAssembly Component Model [7] is a recent addition to the Wasm ecosystem that turns a compiled Wasm module into a portable, embedded and composable *component*

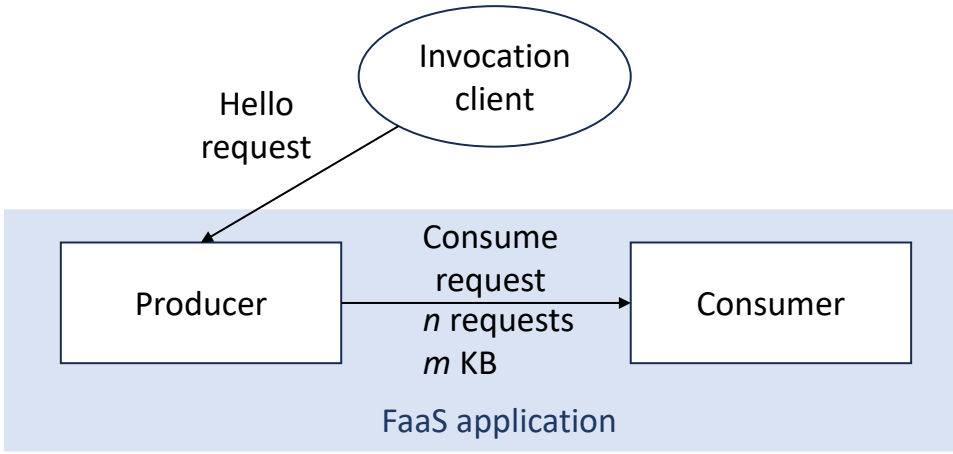


Figure 9.2: Schematic of the FaaS application used for evaluating CoFaaS

with a formally defined external API. A goal of the component model is therefore to allow developers to build language-independent applications that are composed of several isolated components.

Following this definition, it becomes clear that the scope and purpose of a Wasm Component in this regard is the same as a FaaS function. The Wasm Component model even comes with its own dedicated IDL, known as WebAssembly Interface Types (WIT). A key observation, in this regard, is that IDL's that are used to specify the interfaces of FaaS functions, such as Protobuf for gRPC, are easily mapped to WIT.

Currently programs compiled from Rust, C/C++, Go and Java are supported by the Wasm component model, but there is no limitation that prevents further languages from being supported [26].

9.3 CoFaaS

In this section, we discuss the design and implementation details of CoFaaS and describe how a FaaS function is transformed. We start by describing how the IDL mapping is performed, then we introduce the concept of a CoFaaS *component* and finally we discuss the combined transformation workflow.

To describe CoFaaS functionality, we will use a simple two-function FaaS application, named Prodcon, that we will also revisit in the evaluation section (Section 9.4). The application is shown in Figure 9.2. The functionality of this application is simple: when the Producer function receives a request from the Invocation Client it sends one or more requests to the Consumer function. The size of the request can be varied and it does not need to have any particular structure.

9.3.1 IDL mapping

As mentioned in introduction, a FaaS function must have a contract that specifies how it interacts with the surrounding world. For this purpose, it is common to use Interface Definition Languages that formally defines a function's inputs and outputs. An IDL is purely

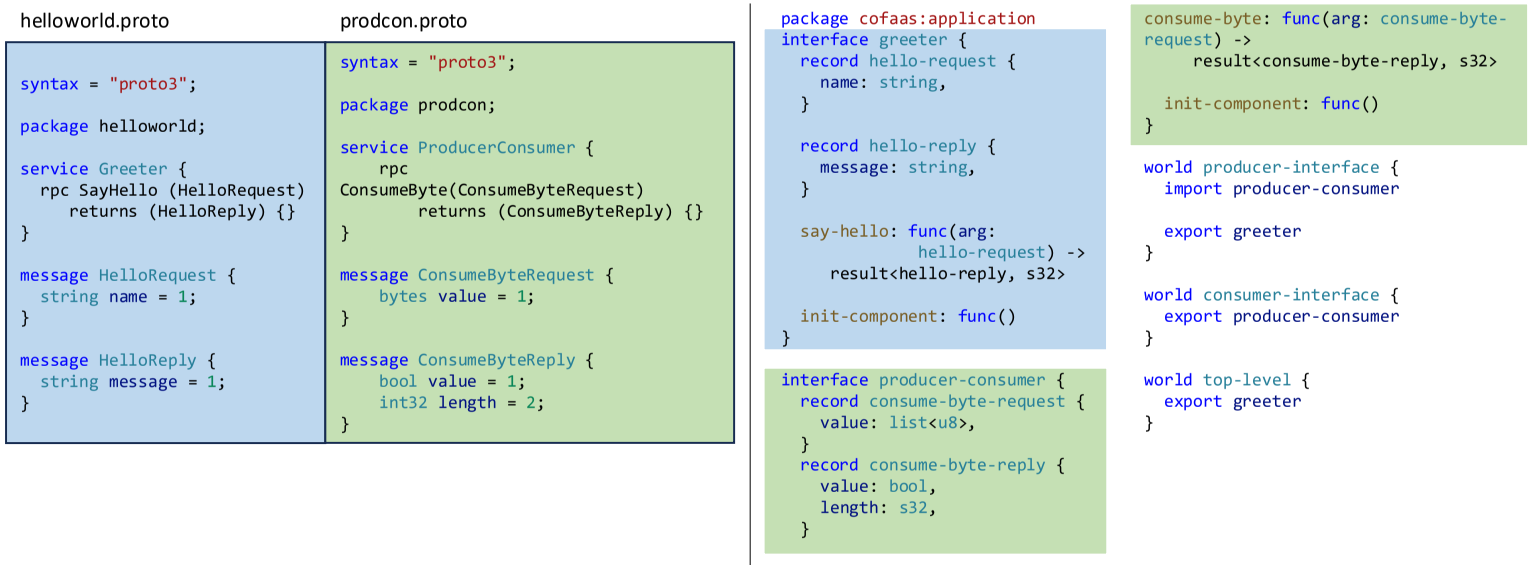


Figure 9.3: Example showing how gRPC (left) is transformed to the Wasm Interface Type (WIT) (right).

declarative, a key reason for their language-independence, and therefore they rely on code generators to be turned into a usable interface.

In practice, several different IDLs are in use by real-world FaaS applications, two notable examples being Protobuf [18], used by Google's gRPC [11], and Apache Thrift [4]. Currently, CoFaaS only supports gRPC but there is no fundamental limitation preventing other IDLs from being supported. In the following, we will detail how the transformation from Protobuf to Wasm's WIT is done.

Figure 9.3 shows an example of how two gRPC interfaces, marked by the blue and green shades respectively, are transformed into their equivalent WIT definitions. Referring to the Prodcon application of Figure 9.2, the Producer function *exports* the `helloworld` protocol and *imports* the `prodcon` protocol. The Consumer function only exports the `prodcon` protocol. The interfaces exported by a function can be called by other functions whereas the interfaces imported by a function can be used to call another function exporting the same interface.

We will start by describing the structure of the `prodcon` gRPC interface. The interface consists of two *messages* and one *service*. In gRPC terms, a message is equivalent to a `struct` in C. For example, looking at the `prodcon` protocol in Figure 9.3 we see that it defines two messages: `ConsumeByte-Request` and `ConsumeByteResponse`. Additionally, it defines a single service `ProducerConsumer` that defines a single method named `ConsumeByte`. To put this in the terminology of a conventional program, a function exporting the `prodcon` interface will expose a public API containing the `ConsumeByte` function that takes a struct of type `Consume-ByteRequest` as an argument and returns a struct of type `ConsumeByteReply`.

Next, we describe how the WIT interface on the right is generated. As an input to the process, the developer currently needs to add a metadata file to each FaaS function that specifies the import and export protocols of the function. This is not strictly necessary and can be easily replaced by a static code analysis step that infers this information. The WIT generator takes all of the protocols used by functions in the applications and does the following: a) it maps each protocol to a WIT *interface* and b) it generates a WIT *world* for each function in the application. Further, it generates a world called `top-level` that defines the public interface of the whole application. A WIT world represents the entire public interface of a function, as such, we see that the world corresponding to the producer function, `producer-interface`, imports the `producer-consumer` interface enabling the Producer function is able to call the Consumer function. It exports the `greeter` interface as this is the external interface of the application.

For each interface generated the WIT notion of a `record` corresponds to a `message` in gRPC and services are represented as a sequence of function definitions. We also note the addition of a `init-component` function. This method calls the `main` method of the original FaaS functions to perform necessary initialization of the function. When a CoFaaS-transformed application is loaded, all of its comprising functions chain-calls their `init-component` methods.

9.3.2 FaaS Function to CoFaaS Component

In the FaaS computing model, functions are the fundamental building block that applications are composed of. The CoFaaS transformation takes each FaaS function and transforms it into a different, but analogous, building block known as a CoFaaS component. In this section, we describe how a FaaS function is turned into a CoFaaS component and how CoFaaS components can be recomposed into different application in the same way as FaaS functions. An overview of the transformation process is given in Figure 9.4. The left side of the figure shows the high-level overview of the CoFaaS transformation, particularly how the

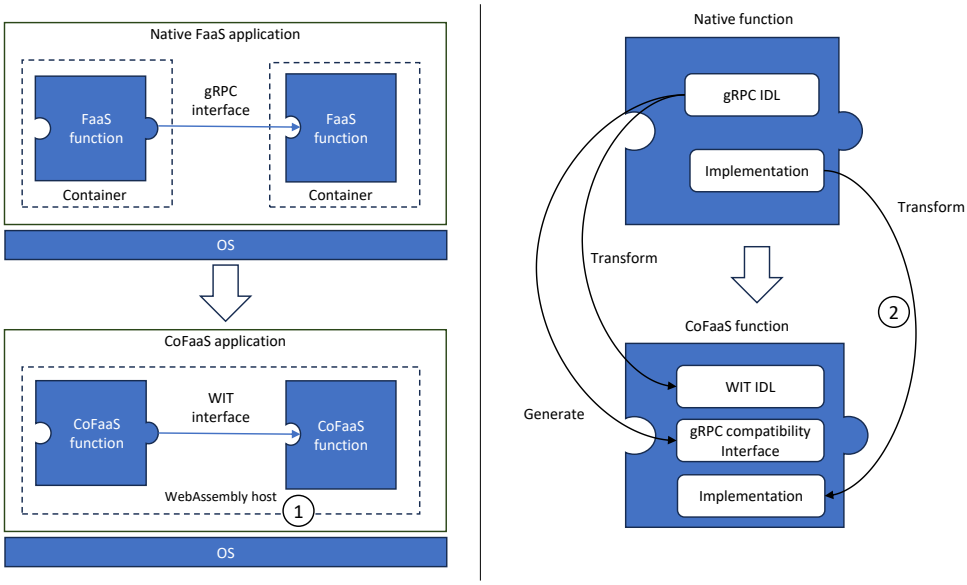


Figure 9.4: Overview of the CoFaaS transformation process.

inter-function communication interface is transformed and how the functions are changed from running in a separate container to sharing a runtime on the same WebAssembly host. The right side of the function shows how the individual functions are transformed. The functions are represented as puzzle pieces to emphasize their composability. In the rest of this section, we will refer to the circled numbers in the figure to explain each step.

Generating a CoFaaS component is equivalent to a separate compilation step. A standard FaaS function contains code compiled to a binary that is then packed in a container that is brought up on nodes as needed. When generating a CoFaaS component, the compilation step is preceded by CoFaaS code generators and code transformers. The resulting code is then compiled to a CoFaaS component. Crucially, this step can be performed ahead-of-time in a similar way as a normal FaaS function is turned into a container. Once the CoFaaS components are built, generating a CoFaaS application by composing components is fast, and can be done on the fly on a host at deployment time. We revisit this in Section 9.3.4.

Next, we describe how the implementation code of the original function is transformed ②. To ensure that we preserve the semantics of the original functions, CoFaaS seeks to change the implementation code of the original functions as little as possible. However, since the original code of the FaaS functions 1) expects to run as a separate server process that listens to a network request and 2) use the API generated by the gRPC code generator for communicating with the outside world, we can not entirely avoid minor changes to the application code.

The default code generator for gRPC produces code that uses networked RPC calls for communicating with other functions. When transforming a FaaS function to a CoFaaS component, we need to eliminate and replace this RPC-backed communication. At the same time, since we want to avoid changing the original implementation code as much as possible, we also need to maintain compatibility with the API produced by the gRPC code generator. To achieve this, we implement a custom gRPC code generator that generates code conforming

to the same API as the gRPC-generated code but replaces network-backed RPC calls with local function calls. Additionally, the interface code that we generate contains a small wrapper that translates between the data structures passed from the Wasm component call and the gRPC-defined data structures that are used by the implementation code.

Finally, we need to deal with redundant library calls. For example, a function using gRPC for communication uses the following code for initializing a connection to a server.

```
conn, err := grpc.Dial(addr, grpc.WithBlock(), [...])
[...]
client := pb_client.NewProducerConsumerClient(conn)
```

The resulting `client` object holds a interface that can be used to communicate with the Consumer process in Figure 9.2. The first call to `grpc.Dial` initializes a network connection that is used as the transport for the RPC call. In our case, initialing this networking connection is not needed. Now, recall again that we want to avoid modifying the original implementation code as much as possible. Therefore, rewriting the implementation code to remove the call to `grpc.Dial` is not an option. To handle this, we therefore introduce our own stubbed gRPC library replacement that provides a simple `nop` implementation of the `Dial` function defined as follows

```
func Dial(target string, opts ...interface{})
        (*ClientConn, error) {
    return &ClientConn{}, nil
}
```

This library stubbing technique allows us to change the behavior of the function without directly transforming critical parts of its code. If we performed such transformation automatically, it would be difficult to ensure that we do not break the behavior of existing code in the process. At the same time, we avoid executing redundant and/or undesired operations present in the original code. Making the rewritten code use our stubbed library is simple: we simply change the gRPC import of the client code to use our library instead. We also provide a stubbed version of the built-in Go `net` library.

Currently, we always replace imports of these libraries with our stubbed versions. For cases where all gRPC calls are transformed to local CoFaaS calls this works well. However, this is not always desirable. Functions may want to open network connections for other reasons. To support this, it is trivial to add a code analysis step to the code transformation that determines which libraries specific gRPC calls are related to and only replace libraries that are related to the gRPC calls that we want to turn into local CoFaaS calls.

The final step involved is to generate the Wasm host wrapper ① that hosts a Wasm runtime and will load and run the CoFaaS application. This host wrapper exposes a gRPC interface corresponding to the public interface of the FaaS application. When requests are received by this wrapper, they are transformed into a call to the WIT bindings of the first function of the CoFaaS application.

9.3.3 Putting it all together

In summary, following are the steps for applying the CoFaaS transformation to a FaaS application. We emphasize that all of these steps are completely automated.

1. The WIT code generator is invoked to transform the gRPC protocols used by the FaaS functions to a WIT interface describing the entire application Section 9.3.1

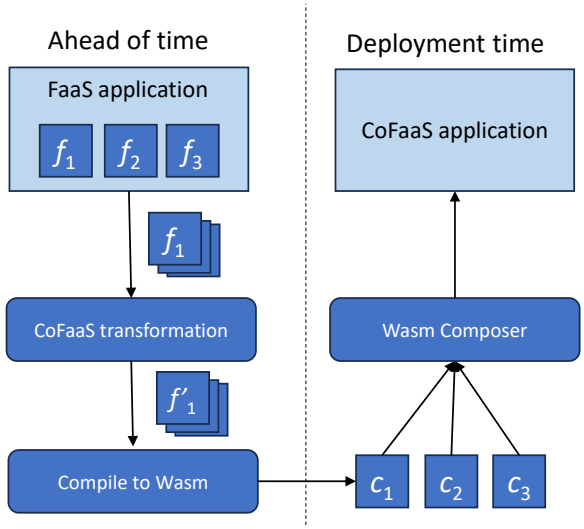


Figure 9.5: The workflow of applying the CoFaaS transformation to a FaaS application.

2. For every FaaS function, we generate bindings for its corresponding WIT world and use our custom gRPC code generators to generate a compatibility layer between the gRPC API used by the function implementation and the WIT bindings used for communicating within the CoFaaS application

3. Then, we apply a set of transformations to the implementation code of the function. These transformations make the code use the replacement gRPC API that we generate and make the code use our stub libraries to disable undesired functionality

4. Finally, we generate the Wasm host wrapper that exposes the external interface of the FaaS application over gRPC and loads and runs the CoFaaS application.

To add support for a new language in CoFaaS, only steps 2 and 3 above need to be re-implemented. Steps 1 and 4 are generic and doesn't change regardless of the function implementation language. Current, CoFaaS supports applying this transformation to functions written in Go automatically. For Rust, we manually rewrite function code in a way that is identical to the automated process.

9.3.4 CoFaaS in Practice

As previously mentioned, applying the CoFaaS transformation to a FaaS application involves several steps. Some of these can be can be done of time as part of application development and packaging while others can be performed at deployment time. The workflow is depicted in Figure 9.5. The first step is to take the functions comprising a FaaS application and apply the CoFaaS transformation to them. The mechanics of this transformation is described earlier in this section. The output of the CoFaaS transformation is the source code needed to create a CoFaaS component. By compiling the transformed functions to Wasm we turn them into CoFaaS components, labeled $c1..3$ in the figure. The final step is to feed the CoFaaS component through the Wasm compositor that yields the resulting CoFaaS application, a single Wasm

binary containing the entire application. An important detail here is that the "Ahead of time" steps are, like compilation, potentially quite time consuming. The "Deployment time" composition step, on the other hand, is fast and can be performed instantaneously. This means that we can re-use CoFaaS components in several applications by dynamically re-composing them on deployment. In this way, CoFaaS maintains the flexibility of native FaaS deployments.

9.4 Evaluation

9.4.1 Methodology

The benchmarks were executed on an Intel Xeon E3-1275 v6 with 4 HT cores clocked at 3.8 GHz with 64 GB of RAM and SSD drives running Fedora 37. We disabled frequency scaling, turbo boost and swap space during the experiments.

We are using a two-process configuration, depicted in Figure 9.2. For every client request, the producer process sends n requests of m KB to the consumer process. We evaluate values of 1, 10 and 20 for n and, where possible, values from 2^0 to 2^9 for m. CoFaaS only optimizes the inter-function communication latency, i.e., the latency occurring when the producer process in Figure 9.2 calls the consumer process. Therefore, adjusting the value of n (request repetitions) allows us to estimate the performance impact of CoFaaS on larger applications that perform a variable number of inter-function requests. We refer to requests occurring within the application as *inter-function* requests. The benchmarked application and invocation client were adapted from [24].

We evaluate two functionally equivalent implementations of the application shown in Figure 9.2 written in the languages Go [22] and Rust [19], respectively. The key distinction between the two languages is that Go uses a Garbage Collector (GC) for memory management whereas Rust inserts compile-time instructions to handle memory. Go therefore relies on a managed runtime for execution. Since languages using managed runtimes have different performance characteristics and platform requirements than unmanaged languages, our experiments demonstrate that the CoFaaS is effective in both cases.

Compilers, with Wasm targets other than web browsers, need to support the WebAssembly System Interface (WASI). For Rust, Wasm support is generally good and its compiler supports a usable WASI target. However, the WASI target of the mainline Go compiler is currently not supported by the Wasm component model. Instead, we are limited to using the TinyGo compiler which primarily targets small embedded devices. For this reason, it is positioned at a different design point than a compiler targeting general-purpose systems. In particular, its garbage collector implementation is optimized for code size rather than speed causing its GC performance to trail the mainline Go compiler [23]. Because of this, our example application implemented in Go is heavily penalized when executed on Wasm. Therefore, we disable GC for our Go benchmarks. Naturally, this limits how long we can run our benchmarks for and the payload sizes we can use. When running on Wasm, we are further limited by the current Wasm specification only supporting 32-bit pointers [3]. In our case, this means that we can only evaluate our Go application with a payload size of up to 16KB, as the experiment otherwise runs out of memory. These limitations are unrelated to CoFaaS, and, at the time of writing, efforts are underway to alleviate both of these limitations [16, 17].

For the round-trip-time benchmarks (Section 9.4.2), we invoke 6,000 requests back-to-back for every configuration. We chose this number of requests to get a representative number of data points while not exceeding memory capacity in the non-GC configurations.

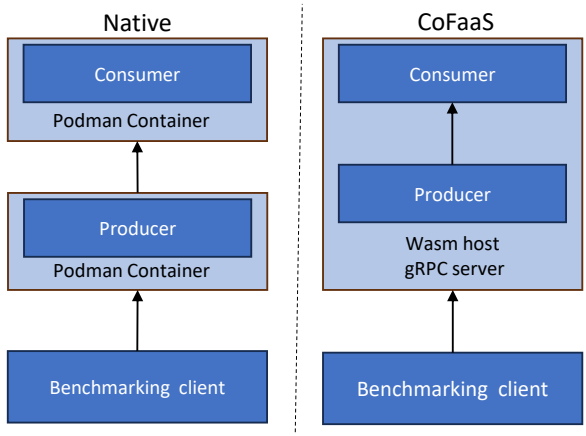


Figure 9.6: The deployment configurations for the native and the CoFaaS benchmarks.

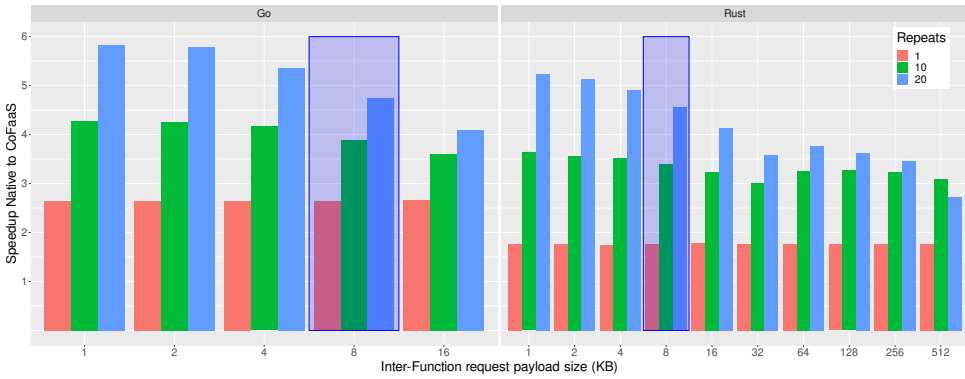


Figure 9.7: The round-trip latency of issuing a request to our example application.

For evaluating the latency of inter-function requests, we report the mean of 100 inter-function requests and we repeat each experiment 100 times.

To mimic a common real-world deployment scenario, each native function is deployed in a separate Podman container. The CoFaaS benchmarks are run inside a Wasm host written in Rust that embeds the Wasmtime [2] runtime. This host loads the CoFaaS component and externally exposes the gRPC interface of the Producer function (using the Tonic gRPC library). When the host receives an external gRPC request, it translates and proxies the request to the CoFaaS application. This experiential setup is depicted in Figure 9.6. Finally, we point out that we run our native FaaS application on the same machine without any other functions running at the same time. We consider this a highly optimized baseline compared to, for example, deploying our functions to a public cloud. The speedups achieved by CoFaaS could therefore increase further if a more realistic baseline was used.

9.4.2 Round-trip Latency

We begin our evaluation of CoFaaS' performance by showing measurements of the request round-trip latency. The round-trip latency is defined by the time it takes before the requesting

client receives the reply to a request that it sent. The speedups achieved by the CoFaaS optimized application over the native baseline are shown in Figure 9.7. The blue boxes mark the median request payload size as observed from traces of real-world FaaS deployments on the Azure cloud [15] and is hence a notable data point. We run the Rust benchmarks with payload sizes between 1 and 512 whereas the Go benchmarks are run for sizes between 1 and 16. Due to the GC limitations outlined in Section 9.4.1, we are unable to run the Go benchmark for the full range of payload sizes. In both cases, we perform 1, 10 and 20 inter-function calls per external request.

Figure 9.7 shows that the payload size does not impact configurations that issue only a single inter-function call, i.e., CoFaaS consistently yields speedups of 2.75× and 1.75× for Go and Rust, respectively. The reason is that a single inter-function call accounts for a moderate but significant fraction of the complete request round-trip time. Thus, as we increase the number of inter-function calls, the achieved speedup also increases. Peak speedups are achieved when an inter-function 1KB request is repeated 20 times, yielding a 5.9× speedup for Go and 5.2× for Rust. This shows that the beneficial impact of CoFaaS is larger for applications that perform a lot of inter-function requests.

A general trend is that speedups increase with smaller payload sizes. The intuition behind this is that regardless of how we make the request, we need to copy data from the producer process to the consumer. The performance of this underlying memory copy operation is limited by the hardware and therefore limits the performacne opportunity available for CoFaaS. To support this observation, we provide the raw latency numbers in Figure 9.8. Here, we see that with increasing payload sizes, the time needed by the native invocation and CoFaaS converges. Since the objective of CoFaaS is to reduce the *latency* of requests, rather than the transfer rate, dealing with such larger requests is outside the scope of CoFaaS. Furthermore, as previously stated, an analysis of Azure FaaS traces showed that the median request payload size is a mere 8KB, well within the range where CoFaaS gives performance improvements. Further, to our knowledge, it is uncommon to move large amounts of data using RPC requests in FaaS applications. In such cases, using external storage to transfer data between the functions is preferred. Still, we emphasize that even if the relative speedups yielded by CoFaaS decrease with increasing request sizes, the absolute reduction in round-trip times is still significant. For example, for the 512KB payload size the round-trip-time decreases from 20ms to 7ms; a significant improvement with a potentially big impact on application responsiveness.

9.4.3 Inter-Function Request Latency

Next, we narrow the scope of our investigation to the latency of inter-function requests; the primary target of CoFaaS. Figure 9.9 shows the speedups of the CoFaaS optimized application over the native baseline when measuring the latency of a single request. Compared to the results in Figure 9.7, the speedups are significantly higher as this measurement bypasses constant factors involved in every request issued to the application.

This result cements that CoFaaS is highly effective at reducing the latency of inter-function calls in serverless applications. For the Rust application with a 1KB request payload size, we see a 100× speedup. Similar to our round-trip time evaluation, we see diminishing returns when increasing payload sizes. We can understand this trend using the same intuition as before; for larger payload sizes, the time needed to transfer the request payload data becomes dominating regardless of the transfer method used.

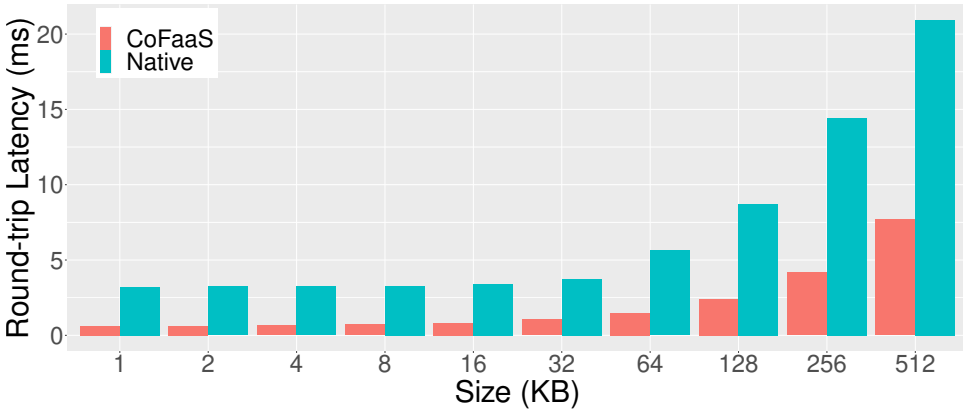


Figure 9.8: Latency (ms) of the Rust application performing 20 inter-function requests when running natively and using the CoFaaS transformation

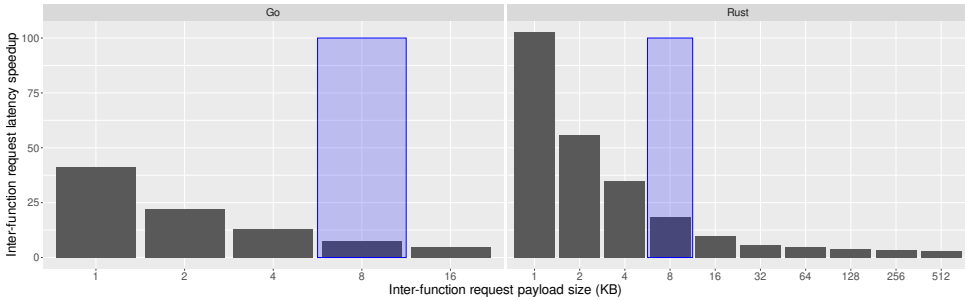


Figure 9.9: The speedups of a single request.

9.4.4 Impact of Go Garbage Collection

Recall that we are evaluating the application implemented in Go with GC disabled as described in Section 9.4.1. To ensure that we are not giving the native Go functions an unfair advantage by running them without GC, we examine the impact of disabling GC on the native Go application. The results are presented in Figure 9.10. For smaller payload sizes ($\leq$ 8KB) the impact is negligible and for a small advantage of 1.2× to the largest payload size repeated 20 times. From this, we conclude that our decision to disable Go's GC does not skew the results we presented in Section 9.4.2 and Section 9.4.3

9.4.5 Transformation Performance

Applying the CoFaaS transformation to our application written in Go currently takes 1s. As this step comes before compilation, it has only minimal impact on the time needed to compile the application. The more interesting observation, is that composing CoFaaS components into an application takes just 20ms. This is important because, as stated in Section 9.3.4, compilation can be done ahead of time. Composing CoFaaS functions into a CoFaaS application, however, may be done at deployment time where any additional overheads are much more concerning.

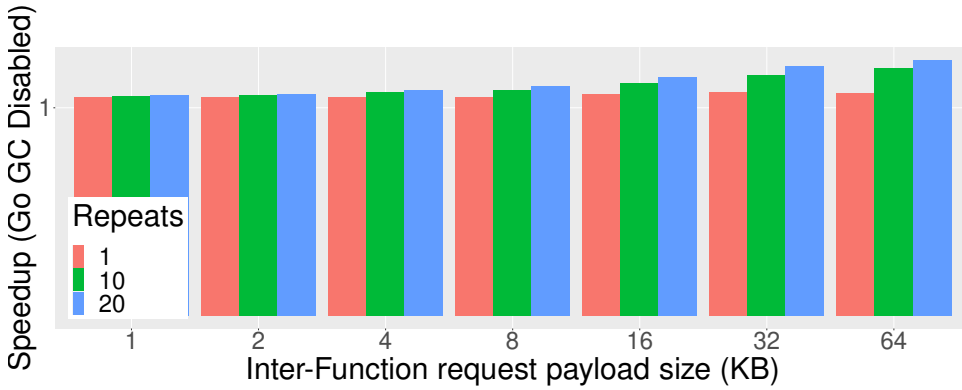


Figure 9.10: The speedup achieved by disabling the Go garbage collector.

Table 9.1: The total sizes of containers and binaries needed to run our application.

Language	CoFaaS	Native	Decrease
Go	24.5 MB	224.0 MB	9.1×
Rust	26.3 MB	127.1 MB	4.1×

9.4.6 Binary Size Implications

Another benefit of CoFaaS is that it reduces the size of the compiled application binaries. While not a primary objective of this work, smaller binaries are beneficial in deployments since they decrease the time and bandwidth needed to store, transfer and bring up an application on a node. Table 9.1 shows the reductions. The given sizes include the compiled application and any containers and hosts needed to run them. Thus, the sizes given for the native application include the size of the statically compiled function binary and the footprint of the container it is packed in. The CoFaaS sizes include the Wasm binary containing the CoFaaS optimized and the statically compiled Wasm host runtime needed to run it. Reducing application image sizes can help reduce the cold-start latency of the application, a much-discussed and central challenge of FaaS deployments (e.g., [9]).

9.5 Discussion

9.5.1 Security implications

The security of a CoFaaS-transformed application is equivalent to the native FaaS application because the WebAssembly component model gives a separate memory space to each component and no other resources are shared. Therefore, the components do not have direct access to the memory spaces of each other, thus providing isolation. This means that the fundamental security assumptions of an application are not changed by applying the CoFaaS transformation [6].

9.5.2 Scalability

We now discuss an important aspect of the FaaS computing model: the ability to elastically scale additional function instances as needed and execute multiple smaller functions in parallel. In FaaS, this is easily achieved and is a side-effect of the function-level granularity in FaaS and the RPC-level function communication. The idea is, that if we have a computational bottleneck in a FaaS application, we can easily spawn additional instances of a function to handle the load. Additionally, we can distribute functions across several nodes as the network-backed RPC interfaces used for communication naturally enable this.

CoFaaS does not currently support function-level scaling in FaaS applications. This is due to several reasons. First of all, Wasm currently has no support for threads. Secondly, we do not support to optionally maintain networked coupling of FaaS functions when multi-node processing is needed. It is important to note that none of these limitations are fundamental to CoFaaS. Efforts are under way to add supports for threads to Wasm. This will allow CoFaaS to orchestrate functions to run in parallel when this is desirable. Furthermore, optionally keeping the network coupling of functions in some cases is simply a matter of additional engineering.

9.6 Related Work

9.6.1 Communication Latency Reduction

In recent years, academic research has presented numerous proposals that aim to change the way FaaS functions communicate with the intention of reducing their communication latency. Surveying this work reveals several design properties to be considered:

- **Code modification.** Can unmodified FaaS functions continue to be used or do they need to be rewritten to be compatible with an improved execution model?
- **Language independence.** Is the cross-language interoperability of FaaS functions preserved?
- **Data transfer.** How is data transfer handled? Is information passed directly between functions or uploaded to an external location?
- **Function isolation.** Is function isolation maintained or can unintentional data sharing occur?

While CoFaaS retains all of these properties, we will now explain that prior work that alleviates inter-function communication latencies [5, 12, 13, 15, 20, 21] falls short with respect to at least one of these properties.

Kotni et al. [13] proposes FaastLane that seeks to reduce the communication latency by replacing Serverless Workflow [27] descriptions with a workload compositor that transfers the results of preceding functions to the input of succeeding functions. In a native execution, each component in the serverless workflow description is executed in a separate container causing excessive inter-function communication latency. The FaastLane approach optimizes application communication latency by generating a static Python orchestration that executes the functions and weaves their inputs and outputs together. Normally, the security implications of this transformation would be quite severe, moving functions that were previously running in separate containers onto the same interpreter. To alleviate this, FaastLane patches the Python

interpreter with a custom memory allocator that constraints memory access to the function that made the allocation using Intel's Memory Protection keys. While FaastLane is able to work on unmodified code, the design has a number of significant drawbacks. Particularly, since their implementation is Python-centric it only works with FaaS functions written in Python. Furthermore, its security model is implemented by modifying a critical component of the Python interpreter, potentially introducing bugs while also adding code maintenance. By leveraging the language-independence and strong isolation guarantees offered by the Wasm component model, CoFaaS avoids these issues.

FaaS Track [5] proposes a custom data store and programming model that allows FaaS applications to communicate more efficiently. While their method does reduce communication latency, it is constrained to a single language *and* requires existing functions to be rewritten. Thus, it, unlike CoFaaS, breaks two of the properties that make the FaaS computing model attractive.

The custom runtimes introduced by [12, 20] enable low-latency inter-function communication across functions written in different languages. However, they do this by introducing custom APIs that applications are required to implement. This requires existing code to be rewritten. Furthermore, their implementation are highly complex and fails to leverage state-of-the-art developments.

WiseFuse [15] propose several different transformations of FaaS application workflows. One of these, called fusion, merges several functions into one by combining dependencies and weaving the inputs and outputs of user-defined entry point functions together. While this design effectively removes communication overhead, it is limited to acting on functions written in a single programming language. Furthermore, it handles security by allowing users to specify that functions handling sensitive data should not be fused. This approach is highly error prone as a user could simply forget to appropriately flag a function or be unaware that their application is transformed in a way that changes isolation assumptions.

9.6.2 Communication Orchestration

A proposal that takes a different approach is SAND [1]. Instead of fundamentally changing how functions communicate, SAND introduces a hierarchical message bus where a high-performance local message bus runs on each host and a slower global message bus moves messages between hosts. Since SAND maintains the function container as an unmodifiable unit, it only changes how data is moved between the containers; it hence does not require code modification, retains language independence and provides function isolation. However, preserving the function container as an unmodifiable unit means that SAND still has to access the network layer on inter-function communication and it is hence unable to address the main source of overhead targeted by CoFaaS.

SONIC [14] also alters how functions communicate with each other but does so by introducing a custom data store. This approach enables unmodified functions to continue to function but like SAND, unblocking the highest reductions in communication latencies require that functions are more tightly coupled.

Cloudburst [21] leverages an auto-scaling key-value store called Anna to implement a platform for executing stateful FaaS functions. As with the other proposals described in this section, Cloudburst leave room for improvement of communication latencies. Furthermore, they introduce a custom API that require functions to be modified and their API is only implemented for Python.

9.7 Conclusion

We have now presented CoFaaS which exploits the well-defined RPC interfaces of FaaS applications to automatically consolidate functions on top of a Wasm runtime — thereby alleviating inter-function communication overhead by not accessing the network layer when functions are scheduled on the same compute node. Our evaluation showed that CoFaaS is highly effective and reduces inter-function communication latency by up to 100× and application-level request round-trip time by up to 6×.

9.8 References

[1] Istemi Ekin Akkus et al. "SAND: Towards High-Performance Serverless Computing". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 923–935 (cit. on p. 116).

[2] Bytecode Alliance. *Wasmtime: A fast and secure runtime for WebAssembly*. `https://wasmtime.dev/`. Accessed: 2023-10-18 (cit. on p. 111).

[3] Andreas Rossberg (editor). *WebAssembly Core Specification*. Tech. rep. Version 2.0. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. W3C, Apr. 19, 2022 (cit. on pp. 103, 110).

[4] *Apache Thrift*. `https://thrift.apache.org/`. Accessed: 2023-10-19 (cit. on p. 106).

[5] Daniel Barcelona-Pons et al. "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures". In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 41–54 (cit. on pp. 101, 115, 116).

[6] *Component model desing goals*. `https://archive.ph/swiyS`. Accessed: 2023-10-19 (cit. on p. 114).

[7] *Componet Model design and specification*. `https://archive.ph/jHHgn`. Accessed: 2023-10-18 (cit. on p. 103).

[8] Datalog. *The state of serverless*. `https://www.datadoghq.com/state-of-serverless-2022/`. 2022 (cit. on p. 101).

[9] Dong Du et al. "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 467–481 (cit. on p. 114).

[10] Yu Gan et al. "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 '19. Providence, RI, USA: ACM, 2019, pp. 3–18 (cit. on p. 103).

[11] gRPC. *A high performance, open source universal RPC framework*. `https://grpc.io/`. Accessed: 2023-10-19 (cit. on pp. 101, 106).

[12] Zhipeng Jia and Emmett Witchel. "Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 152–166 (cit. on pp. 101, 115, 116).

[13] Swaroop Kotni et al. "Faastlane: Accelerating Function-as-a-Service Workflows". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 805–820 (cit. on pp. 101, 115).

[14] Ashraf Mahgoub et al. "SONIC: Application-aware Data Passing for Chained Serverless Applications". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 285–301 (cit. on p. 116).

[15] Ashraf Mahgoub et al. "Wisefuse: Workload Characterization and Dag Transformation for Serverless Workflows". In: *Proc. ACM Meas. Anal. Comput. Syst.* 6.2 (June 2022) (cit. on pp. 101, 112, 115, 116).

[16] Mossaka. *wit-bindgen github issue 499: Go bindgen todos*. `https://archive.ph/mjnRV`. Accessed: 2023-10-18 (cit. on p. 110).

[17] WebAssembly Propsals. *Memory64 Proposal for WebAssembly*. `https://archive.ph/wpvrp`. Accessed: 2023-10-18 (cit. on p. 110).

[18] Protobuf. *Protocol Buffers*. `https://protobuf.dev/`. Accessed: 2023-10-19 (cit. on p. 106).

[19] *Rust Programming Language*. `https://rust-lang.org/`. Accessed: 2023-10-19 (cit. on p. 110).

[20] Simon Shillaker and Peter Pietzuch. "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 419–433 (cit. on pp. 101, 115, 116).

[21] Vikram Sreekanti et al. "Cloudburst: Stateful Functions-As-A-service". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2438–2452 (cit. on pp. 101, 115, 116).

[22] *The Go Programming Language*. `https://go.dev/`. Accessed: 2023-10-19 (cit. on p. 110).

[23] TinyGo. *Go language features*. `https://archive.ph/AFlUi`. Accessed: 2023-10-14 (cit. on p. 110).

[24] Dmitrii Ustiugov et al. "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM, 2021 (cit. on p. 110).

[25] Christopher L. Williams et al. "The Growing Need for Microservices in Bioinformatics". In: *Journal of Pathology Informatics* 7.1 (2016), p. 45 (cit. on p. 100).

[26] *wit-bindgen readme*. `https://archive.ph/wjeV2`. Accessed: 2023-10-19 (cit. on p. 104).

[27] Serverelss Workflow. *Specify low-code, event-driven workflow orchestrations*. `https://serverlessworkflow.io/`. Accessed: 2023-10-18 (cit. on p. 115).