NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Benchmarking Support for RISC-V CPUs in Serverless Computing

Georgios T. Pournaras

Supervisor: **Vasileios Karakostas,** Assistant Professor

ATHENS

SEPTEMBER 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υποστήριξη Αξιολόγησης RISC-V Επεξεργαστών στην Υπολογιστική χωρίς Εξυπηρετητή

Γεώργιος Θ. Πουρνάρας

Επιβλέπων: **Βασίλειος Καρακώστας,** Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2024

BSc THESIS

Benchmarking Support for RISC-V CPUs in Serverless Computing

Georgios T. Pournaras
S.N.: 1115201800162

SUPERVISOR: **Vasileios Karakostas,** Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υποστήριξη Αξιολόγησης RISC-V Επεξεργαστών στην Υπολογιστική χωρίς Εξυπηρετητή

Γεώργιος Θ. Πουρνάρας
Α.Μ.: 1115201800162

ΕΠΙΒΛΕΠΩΝ: **Βασίλειος Καρακώστας,** Επίκουρος Καθηγητής

ABSTRACT

Serverless computing has emerged as a competitive cloud computing paradigm. At the same time, the open-source RISC-V ISA has gained a lot interest and the first RISC-V systems have already started to appear in the server market for datacenter and cloud computing. The combination of these two computing trends necessitates the performance assessment of the impact of the RISC-V ISA and relevant processor implementations when executing serverless workloads, particularly with respect to well-established ISAs and processor designs. However, currently, there is no benchmarking support for systematically evaluating serverless workloads on RISC-V systems. The goal of this thesis is to bridge this gap in benchmarking support across the layers of the computing stack, from the microarchitecture up to the application. We rely on vHive's vSwarm that is a recently proposed serverless benchmark suite, and on vSwarm-u that provides infrastructure for executing serverless workloads in the gem5 microarchitecural simulator. We port several workloads from vSwarm to the RISC-V ISA enabling their execution on RISC-V systems. We also enhance vSwarm-u to enable the execution of those serverless workloads on simulated RISC-V CPUs using gem5. To achieve our goal, we address several challenges that mostly stem from the immaturity of the RISC-V software ecosystem. To demonstrate the usefulness of the enhanced benchmarking infrastructure, we evaluate the execution of the ported serverless workloads on a simulated RISC-V out-of-order multicore system. We also compare its execution with an equivalent x86 system. Our evaluation results highlight the important performance trade-off of cold vs warm execution for serverless workloads. Overall, our contributions pave the way for further experimentation with serverless workloads on RISC-V platforms, as well as for further comparison across various ISAs and processor microarchitectural parameters.

SUBJECT AREA: Computer architecture

KEYWORDS: Serverless computing, benchmarking, RISC-V, software porting, vSwarm, docker, gem5, microservices

ΠΕΡΙΛΗΨΗ

Η υπολογιστική χωρίς εξυπηρετητή (serverless computing) εχει ξεχωρίσει ως ένα ανταγωνιστικό μοντέλο εκτέλεσης εφαρμογών στο υπολογιστικό νέφος (cloud computing). Συγχρόνως, η ανοιχτού κώδικα RISC-V αρχιτεκτονική έχει συγκεντρώσει αρκετό ενδιαφέρον και τα πρώτα RISC-V συστήματα έχουν ήδη αρχίσει να εμφανίζονται στην αγορά διακομιστών για κέντρα δεδομένων. Ο συνδυασμός αυτών των δύο τάσεων στην υπολογιστική απαιτεί την αξιολόγηση της απόδοσης του αντίκτυπου της αρχιτεκτονικής RISC-V και των σχετικών υλοποιήσεων επεξεργαστών κατά την εκτέλεση serverless φορτίων εργασίας, ιδιαίτερα σε σύγκριση με τις καθιερωμένες αρχιτεκτονικές και μοντέλα επεξεργαστών. Ωστόσο, προς το παρόν, δεν υπάρχουν μετροπρογράμματα (benchmarks) που επιτρέπουν τη συστηματική αξιολόγηση των serverless φορτίων εργασίας σε συστήματα RISC-V. Στόχος αυτής της πτυχιακής εργασίας είναι να γεφυρώσει αυτό το χάσμα στην υποστήριξη αξιολόγησης μέσω μετροπρογραμμάτων όλων των επιπέδων της υπολογιστικής στοίβας, από τη μικροαρχιτεκτονική μέχρι τις εφαρμογές. Βασιζόμαστε στη vSwarm σουίτα του vHive, η οποία είναι μια πρόσφατα προτεινόμενη σουίτα μετροπρογραμμάτων για serverless περιβάλλοντα εκτέλεσης, καθώς και στο vSwarm-u framework που παρέχει υποδομή για την εκτέλεση serverless φορτίων εργασίας στον μικροαρχιτεκτονικό προσωμοιωτή gem5. Μεταφέρουμε διάφορες εφαρμογής από τη σουίτα vSwarm στην αρχιτεκτονική RISC-V ISA, επιτρέποντας την εκτέλεσή τους σε συστήματα RISC-V. Επιπλέον, επεκτείνουμε το vSwarm-u ώστε να επιτρέπεται η εκτέλεση αυτών των serverless εφαρμογών σε προσομοιούμενους RISC-V επεξεργαστές μέσω του gem5. Για να πετύχουμε τον στόχο μας, αντιμετωπίσαμε διάφορες προκλήσεις που προέρχονται κυρίως από την ανωριμότητα του οικοσυστήματος λογισμικού για RISC-V. Για να επιδείξουμε τη χρησιμότητα της ενισχυμένης υποδομής αξιολόγησης που παρέχουμε, εκτελούμε τις serverless εφαρμογές σε ένα προσομοιωμένο πολυπύρηνο RISC-V σύστημα που αποτελείται από επεξεργαστές εκτέλεσης εντολών εκτός σειράς (out-of-order). Επίσης, συγκρίνουμε την εκτέλεσή τους με ένα αντίστοιχο σύστημα αρχιτεκτονικής x86. Τα αποτελέσματα της αξιολόγησής μας αναδεικνύουν το σημαντικό ζήτημα απόδοσης μεταξύ "κρύας" και "ζεστής" εκτέλεσης που προκύπτει για serverless εφαρμογές. Συνολικά, οι συνεισφορές μας ανοίγουν το δρόμο για περαιτέρω πειραματισμούς με serverless φορτία εργασίας σε πλατφόρμες RISC-V, καθώς και για περαιτέρω συγκρίσεις μεταξύ διαφόρων ISA και μικροαρχιτεκτονικών παραμέτρων των επεξεργαστών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική υπολογιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Υπολογιστική χωρίς εξυπηρετητή, αξιολόγηση μέσω μετροπρογραμμάτων, RISC-V, μετατροπή λογισμικού, vSwarm, docker, gem5, μικροϋπηρεσίες

To my parents Mantha and Thomas and my siblings Kostas, Aris, Angeliki as well as their families for their love and support all those years

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Assistant Professor Vasileios Karakostas for coming up with the topic of the thesis. This thesis would not have been possible without his guidance and assistance.

I would also like to express my gratitude to Professor Dimitrios Gizopoulos and researcher George Papadimitriou for their insightful contributions and feedback during the course of the research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis was completed at the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens under the guidance of Asst. Prof. Vasileios Karakostas and the valuable contributions of Prof. Dimitris Gizopoulos and Dr. George Papadimitriou.

1. INTRODUCTION

Serverless computing has gained a lot of interest recently [1, 23, 5, 27], as it provides important benefits for both the users and the cloud service providers. The end users pay only for the resources they use, while the cloud providers collocate multiple workloads on the same physical servers to maximize the profit from the invested resources [58, 62]. However, the performance challenges of serverless computing require careful consideration, as they severely affect the profit of cloud service providers [38, 5, 10]. Because of the short-living nature of serverless workloads, prior works have shown that the performance of serverless workloads can be significantly affected by the chosen architectural and microarchitectural parameters of the system's processor [57, 3] and have proposed mechanisms for improving performance [52, 69, 53].

At the same time, the interest in the RISC-V ISA has increased significantly, thanks to its open-source nature that fosters processing sovereignty and the growing support from system and software vendors. While CISC-based systems have dominated the server market so far, cloud service providers have started to adopt RISC-based (Arm) systems in their production clusters [42] and more recently to introduce RISC-V platforms. For example, SiFive [59] and Ventana [63] provide high-performance RISC-V data center-class CPUs, while Scaleway [51] launched a range of RISC-V servers in the cloud on February 2024. Hence, having relevant benchmarking support is critical for enabling further research and development towards its successful adaptation.

1.1 Motivation

Despite the growing interest in serverless computing and the RISC-V ISA, there is currently no benchmarking support that combines these two technologies together and that allows the systematic analysis and evaluation of the architectural and microarchitectural parameters of RISC-V systems in serverless computing. It is well known that different ISAs offer different trade-offs with respect to performance, power, and energy efficiency [7]. Having such benchmarking support would be crucial to quantify the potential of RISC-V for serverless computing concerning other ISAs. In addition, prior works have shown that the unique nature of serverless workloads exposes novel opportunities for microarchitectural optimizations [57, 3, 52, 69, 53]. Such benchmarking support would allow the evaluation of the impact of different microarchitectural designs for RISC-V systems.

While there are multiple benchmark suites for serverless computing [30, 31, 71, 33, 24, 11, 66], they lack support for the RISC-V ISA. Only a few prior works [61, 60, 16] have used individual serverless workloads on RISC-V-based systems. Howerver, those works have only considered few individual serverless workloads but none of them has ported or used any existing benchmark suite for exploring the impact of architectural and microarchitectural choices. In addition, those works did not consider the impact of the containerization/virtualization layers that play a critical performance role in the serverless computing stack. Finally, they do not provide support for running serverless workloads in microarchitectural simulators, e.g., gem5, for the RISC-V ISA.

1.2 Goal & Approach

The goal of this thesis is to bridge the gap in benchmarking support between serverless computing and the RISC-V ISA. To meet this goal, we enhance existing infrastructure that supports serverless benchmarking along two axes. We first port serverless benchmarks to the RISC-V ISA. This enables the experimentation on real platforms that are equipped with RISC-V processors. Second, we enhance the benchmarking methodology to support the evaluation of the serverless benchmarks on RISC-V CPUs using microarchitectural simulation. This allows for the evaluation of the impact of various microarchitectural components on the execution of serverless workloads, as well as for the direct comparison of different ISAs for this emerging cloud computing paradigm.

More specifically, we base our approach on vSwarm [66] and vSwarm-u [67]. vSwarm is a benchmark suite for serverless systems that currently provides support for x86 and Arm platforms. vSwarm provides plenty of benchmarks including: standalone functions (i.e., Fibonacci, Authentication, Aes-Cipher), an online-shop suite, and a hotel booking collection, among others. vSwarm-u provides the necessary infrastructure (i.e., scripts, configuration files, and experimental methodology) for executing the aforementioned workloads in the gem5 simulator [19] with x86 and Arm CPUs. We extend vSwarm and vSwarm-u to enable the evaluation of serverless workloads on RISC-V CPUs using the gem5 simulator.

Accomplishing our goal was not an easy task, as we faced numerous challenges due to various software dependencies and the immaturity of the RISC-V software ecosystem. First of all, we had to create a proper development environment based on QEMU [43] for porting the serverless workloads. Some well known software tools, e.g., Docker [14], were not available through packet managers for the RISC-V ISA, so we had to build and install them using their source codes. Still, building software components was not always sufficient. For example, we were unable to compile the gRPC [26] module for RISC-V which is necessary for running python containers, due to another software dependency. Similarly, the Hotel application from vSwarm depends on the MongoDB database. However, MongoDB has not been ported to RISC-V. We overcame this obstacle by modifying the application to use Apache Cassandra instead as an alternative NoSQL database [49]. Finally, to enable the execution of the serverless workloads in gem5 we had to create a custom Linux kernel image that allows the execution of docker containers. Overall, in spite of these challenges we successfully managed to port the workloads to the RISC-V ISA and provide support for executing them in gem5.

To validate our porting effort and demonstrate the usefulness of our enhanced benchmarking infrastructure, we evaluate the execution of the ported serverless workloads on a simulated RISC-V out-of-order multicore system using gem5. Our results highlight the important performance trade-off of cold vs warm execution for serverless workloads, and its correlation with various microarchitectural statistics, such as L1 data/instruction misses and L2 misses, among others. In addition, we compare the performance of the ported serverless workloads on a RISC-V system with respect to a equivalent x86 system with the same microarchitectural characteristics using the same system setup (i.e., identical gem5 version, Ubuntu version, Linux kernel version and similar configuration). Our results show that all the ported benchmarks run faster in the RISC-V system than in the x86 one, whereas the cold execution time in the RISC-V system is faster for some workloads than the corresponding warm execution time in the x86 system. The main reason for this performance difference is the fact that the execution of the functions in the RISC-V simulated platform resulted in significantly fewer executed instructions than the execution of

the functions in the x86 simulated platform.

1.3 Thesis Contributions

In summary, the main contributions of this thesis are:

- We port the vSwarm benchmarks in the RISC-V ISA addressing several challenges due to the immaturity of the software ecosystem.
- We extend the vSwarm-u infrastructure for executing the benchmarks on the gem5 simulator.
- We use the developed infrastructure to comprehensively evaluate the impact of cold vs warm execution on RISC-V systems.
- We compare the performance of RISC-V and x86 microprocessors in the context of serverless computing.

1.4 Organization

The rest of this document is organized as follows. Section 2 provides background information regarding serverless computing, the RISC-V ISA, and the emulation and simulation tools that we use in this work. Section 3 describes our effort towards enabling benchmarking support for serverless computing for RISC-V CPUs. Section 4 describes our evaluation methodology and experimental results. Section 5 summarizes prior works on benchmarking serverless applications and the impact of microarchitecture on their execution. Finally, Section 6 concludes this thesis and provides directions for future work.

2. BACKGROUND

In this section we provide background information regarding the two main trends in computing that this thesis bridges by providing relevant benchmarking support, i.e., serverless computing (also known as Function-as-a-service or FaaS) and the RISC-V ISA. We also describe briefly the two tools that we use in this thesis for emulating and simulating RISC-V systems, i.e., QEMU and gem5, respectively.

2.1 Serverless Computing

Serverless Computing was first introduced in 2008 by Google's "Google App Engine" [22] and gained mass-market appeal in when Amazon released AWS Lamda [45]. Serveless Computing is a cloud computing paradigm in which an application is a set of many event driven functions that require minimal resources. Serverless computing is similar to microservices as they are both part of modern cloud-native architectures and aim to break down monolithic applications into smaller, more manageable components. The key feature of serverless computing is the scalability that provides. Developers run code inside containers in response to specific events or requests without specifying or managing the infrastructure required to run the code. Furthermore, the users only pay for the execution time of their functions, because a Pay-as-you-go model is applied. This model motivates the user to implement even smaller functions, in the order of a few hundreds of milliseconds [28], which facilitates the provider's task to quickly shutdown and initiate functions minimizing resource waste.

Regarding performance, one of the most critical metrics in the execution of serverless functions is the execution latency. That latency is heavily affected by the state of the function on the system. Functions can have three states: Running, Waiting, and Dead. When a function is in the dead state, it does not occupy any memory or resources on the server and its execution requires costly initialization steps (i.e., booting the function). When a function is in the waiting state, it means that it is still present in the memory of the server and occupies resources. The term *cold execution* refers to the first invocation of a function that is currently in dead state. In contrast, subsequent executions of a function that is in the waiting state are called *warm executions* and are much faster. The difference in performance between cold and warm execution can be significant and even affect the billing policy of the providers [62]. Furthermore, because of the function's short execution time, initializing/booting a function from the dead state may dominate the total execution time. To reduce as much as possible the cold execution of functions, the provider selects the criteria that will change a function's state from waiting to dead and vise-versa. The goal is to keep functions that will soon be needed again present in memory and to deactivate the ones that will probably used very far in the future.

In addition, prior works [52, 53] have shown that, the repetitive execution of thousands short-lived functions in the same core in combination with relative sparse time intervals between the invocations, make subsequent function callings unable to capitalize on the microarchtectural state of previous callings. In an ideal scenario where every execution happens consecutively, one can witness exceptional low execution times. In reality, the execution of other functions in between cause the thrashing of caches and the microarchitectural state, leading every invocation to *lukewarm execution*, i.e., to behave as if it was called for the first time.

2.2 The RISC-V Instruction Set Architecture

Unlike closed-source processor architectures, RISC-V [47] represents an open-source instruction set architecture (ISA) utilized for the development of custom processors aimed at a broad spectrum of end applications. Initially designed at the University of California, Berkeley, RISC-V ISA stands as the fifth iteration of processors grounded in the principle of the reduced instruction set computer (RISC). Its popularity has surged in recent times due to its open nature and technical advantages. The standard is now managed by RISC-V International [46], which has more than 3,000 members. RISC-V enables efficient task execution and allows designers to create numerous custom processors for faster market deployment while the shared processor IP reduces software development time. RISC-V International reported that more than 10 billion chips containing RISC-V cores had been shipped by the end of 2022 [48].

Key benefits of the RISC-V ISA include its open-standard architecture fostering industry-wide collaboration and innovation. The common ISA simplifies software development across a range of devices from embedded systems to supercomputers. Besides the base instruction set that allows for the implementation of a simplified general-purpose computer, the RISC-V ISA also provides support for ISA customization through extensions with unique features tailored to specific requirements and needs. Finally, the RISC-V ISA provides enhanced security through open-source designs and tools that allow thorough public scrutiny and prevent from back doors and hidden vulnerabilities.

2.3 QEMU

QEMU [43] serves as a versatile, open-source tool capable of emulating machines and functioning as a virtualization layer. Its primary application lies in system emulation, offering a simulated environment comprising a CPU, memory, and emulated peripherals to support the execution of guest operating systems (OS). This emulation can either fully replicate the CPU behavior or collaborate with hypervisors, like KVM, to enable direct execution of the guest OS on the host CPU. Additionally, QEMU facilitates user-mode emulation, enabling the execution of applications designed for one processor architecture on another through CPU emulation. Beyond these core functionalities, QEMU includes various standalone command-line tools, notably the `qemu-img` utility for managing disk images, including creation, conversion, and modification operations.

2.4 gem5

gem5 [32, 19] is a modular open source computer architecture simulator that is widely used in academia and industry for computer-system architecture and microarchitecture research. Its development started roughly 15 years ago at the University of Michigan as the m5 project, and at the University of Wisconsin as the GEMS project. The two projects merged in 2011 and, since then, gem5 has been cited by over 2900 publications.

gem5 supports multiple ISAs (Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86) and provides four CPU types. A simple one-CPI CPU, a detailed model of an in-order CPU, and a detailed model of an out-of-order CPU. These CPU models use a common high-level ISA description. In addition, gem5 features a KVM-based CPU that uses virtu-

alization to accelerate simulation. Furthermore gem5 comes with rich memory simulation support. In particular, the memory system is event driven and includes caches, crossbars, snoop filters, and a fast and accurate DRAM controller model, for capturing the impact of current and emerging memories, e.g. LPDDR3/4/5, DDR3/4, GDDR5, HBM1/2/3, HMC, WideIO1/2. The components can be arranged flexibly, e.g., to model complex multi-level non-uniform cache hierarchies with heterogeneous memories.

2.4.1 gem5 System Modes

gem5 can run in two different modes called “full system” (FS) and “syscall emulation” (SE) [21]. In full system mode, gem5 emulates the entire hardware system and runs an unmodified kernel. Full system mode is similar to running a virtual machine.

Syscall emulation mode, on the other hand, does not emulate all of the devices in a system and focuses on simulating the CPU and memory system. Syscall emulation is much easier to configure since it is not necessary to instantiate all of the hardware devices required in a real system. However, syscall emulation only emulates Linux system calls, and thus only models user-mode code.

2.4.2 gem5 CPU Models

gem5 supports various CPU models that provide different trade-offs between accuracy and simulation speed. Next we breifly describe the CPU models that we use in this thesis.

KVM CPU Model. gem5’s KVM model [4] is a handful way to speed up parts of the simulation that are not taken into consideration during the collection and analysis of statistics, such as the booting stage of a machine. When using KVM, the simulator executes commands fast directly to the host’s CPU. However, as it will be mentioned later, since the simulation lacks accuracy in favor of execution speed, the state of the core (i.e., the simulation) can be inconsistent.

Atomic CPU Model. When the Atomic CPU model (AtomicSimpleCPU) [4] is used, memory accesses happen instantaneously and there is no CPU pipeline. This is the fastest simulation CPU model after KVM, but it is not realistic at all. It is mostly useful for booting Linux fast and then checkpointing and switching to a more detailed CPU.

Out-of-Order CPU Model. The Out-of-Order CPU model (DeriveO3CPU) [39] is a detailed core model that is loosely based on the microarchitecture of the Alpha 21264 microprocessor. The model uses five pipeline stages (i.e., fetch, decode, rename, issue / execute / writeback, and commit). It provides cycle-level accuracy, as it actually executes the instructions at the execute stage of the pipeline. In general, we use the Out-of-Order CPU model for measuring the performance of the application’s region of interest by simulating the CPU in the highest possible detail.

2.4.3 gem5 Utilities

Checkpoints. In this thesis the term checkpoint is used in several occasions since it is one of gem5’s helpful mechanisms. Checkpoints are essentially snapshops of a simulation. A typical Ubuntu disk image boot can take up to 6 hours using an atomic core, given that so many services need to start. So checkpointing after getting terminal access in a

successful boot is one of the most common practices when working with gem5. Doing that allows the user to run different kinds of experiments without waiting each time for executing the common initialization instructions. The user restores the system state from the checkpoint and continues the execution of the workload with a more detailed core model exactly from the next instruction.

gem5 Standard Library. Configuring the simulator to perform experiments can be tricky, especially if one uses the fs.py and se.py configuration scripts. Despite being provided as examples of how to configure a simulation with gem5, these scripts have been extended and used as a de-facto configuration approach to run experiements with gem5, while they were never intended to be. These configuration scripts are an inefficient, poorly documented way to configure a system and leads to many problems [12]. To resolve this issue, a gem5-stdlib was created. With the gem5 stdlib users can configure simulations in a few lines of Python. In addition, stdlib is a part of the project that is continually tested, and significant engineering resources are targeted towards maintaining its stability and extending it with new features.

M5 Magic Instructions. M5 magic instructions are used in full system (FS) mode to issue special instructions to trigger simulation specific functionality, e.g., stat resetting, stat dumping, checkpoint, end of simulation, and others.

3. PORTING SERVERLESS BENCHMARKING TO RISC-V

In this section, we provide information about our approach and experience with enabling benchmarking support for serverless computing in RISC-V CPUs. The selection of representative serverless workloads is critical in the realization of our goal. In addition, the porting process itself is not straightforward. The major challenges arise from the fact that the RISC-V software ecosystem is less mature compared to that of other well-established ISAs, e.g., x86 and Arm. This current status of the RISC-V software ecosystem introduces additional software dependencies that, in turn, affect the selection and utilization of the software tools and components. Additionally, we faced difficulties in generating a Linux Kernel to run our workloads in gem5 for both RISC-V and x86.

In summary, in our porting effort, we had to address the following challenges: (i) selection of an existing benchmarking framework, (ii) creation of a RISC-V development platform, (iii) installation of system dependencies, (iv) porting of the serverless workloads, and (v) enabling the execution of the workloads in gem5.

3.1 Selecting the Serverless Benchmark Suite

Table 3.1 summarizes the currently available serverless benchmark suites. vSwarm [62, 66] is a serverless benchmark suite that is part of the vHive ecosystem [64]. vSwarm includes multi-function benchmarks and standalone function benchmarks, and provides support for multiple languages and runtimes, ISAs, and allows experimentation on both real and simulated platforms. FunctionBench [30, 31, 17] is a suite of various function workloads that are implemented in Python and provides support for executing them on multiple cloud service providers. ServerlessBench [71, 56] is a benchmark suite for characterizing serverless platforms that allows the exploration of several metrics of serverless platforms and provides four real-world serverless workloads. FaaSdom [33, 15] is another benchmark suite for serverless computing platforms. It comes with a variety of benchmark tests written in multiple implementation languages and runtimes. BeFaaS [24, 6] is a serverless benchmarking framework that also supports federated benchmark runs, in which the benchmark application is distributed over multiple providers. SeBS [11, 55] is a comprehensive benchmark suite for public cloud providers that consists of the specification of representative workloads, their implementation, and the evaluation infrastructure and methodology.

In this work we decided to use the vSwarm benchmark suite for the following reasons: (i) it comes with a variety of representative workloads for serverless computing, ranging from simple functions to real-world applications, (ii) it supports multiple languages and runtimes,

Benchmark suite	Languages & Runtimes	Infrastructure	ISAs	gem5 support
FunctionBench [17]	Python	Public & Private	x86	No
ServerlessBench [56]	C, Java, Python, NodeJs, Ruby	Public & Private	x86	No
FaaSdom [15]	Node.js, Python, Go, .NET	Public	x86	No
BeFaaS [6]	Node.js	Public & Private	x86	No
SeBS [55]	Python, Node.js	Public	x86	No
vSwarm [66]	Python, Go, Node.js	Private	x86/Arm	Yes

Table 3.1: Summary of the currently available serverless benchmark suites.

(iii) it supports both the x86 and Arm ISAs, and (iv) it comes with additional support for executing serverless functions in the gem5 simulator.

3.1.1 The vSwarm Benchmark Suite

vSwarm [62, 66, 65] presents a curated selection of serverless benchmarks, tailored for easy deployment and designed to reflect real-world scenarios involving intensive data processing. These benchmarks are composed of interconnected serverless functions, aiming to simulate practical workload conditions. This suite encompasses not only complex, multi-function benchmarks but also simpler, standalone functions compatible with both x86 and Arm ISAs. vSwarm offers two types of microbenchmarks: those involving the combination of synchronous and asynchronous functions, known as multi-function benchmarks; and standalone function benchmarks, which consist of individual functions without any composite structures (e.g., avoiding producer-consumer setups).

The benchmarks that we test ın our thesis can be narrowed into three large categories:

- The standalone functions (i.e., AES, Auth, Fibonacci) that are written in all three tested languages, i.e., Go, NodeJs, Python (Table 3.2).
- The online-shop collection is derived from Google's Online Boutique. It consists of several functions that are written in Go, NodeJs, or Python (Table 3.3).
- The hotel-app collection that is composed of Go microfunctions that simulate the backend of a hotel booking site. The hotel-app collection is based on DeathStarBench's Hotel Reservation Application [18]. Each function communicates with a database instance and some of them also with a lightweight caching instance (Table 3.4).

Function	Go	Python	NodeJs
Fibonacci	Yes	Yes	Yes
Auth	Yes	Yes	Yes
Aes	Yes	Yes	Yes

Table 3.2: Overview of the vSwarm standalone functions and the supported runtimes.

Function	Runtime
Product Catalog Service	Go
Shipping Service	Go
Recommendation Service	Python (Used with Product Catalog)
Email Service	Python
Currency Service	NodeJs
Payment Service	NodeJs

Table 3.3: Overview of the functions of the vSwarm Online Shop application and their corresponding runtimes.

Function	Runtime	MongoDB	Memcached
Geo	Go	Yes	No
Recommendation	Go	Yes	No
User	Go	Yes	No
Reservation	Go	Yes	Yes
Rate	Go	Yes	Yes
Profile	Go	Yes	Yes

Table 3.4: Overview of the functions of the vSwarm Hotel application, their corresponding runtimes, and their dependencies on the MongoDB database and the Memcached caching system.

3.1.2 The vSwarm-u Framework

vSwarm-u [67] provides additional support for running most of the standalone functions in the gem5 [32] microarchitectural simulator. The standalone functions are particularly useful for analyzing the performance of serverless computing at the microarchitecture level using the gem5 cycle-accurate full-system CPU simulator. They serve as valuable microbenchmarks, initially identifying potential bottlenecks in serverless workload execution on physical hardware, followed by detailed analysis and optimization using the gem5 simulator. However, the faithful execution of serverless workloads in simulation platforms is difficult due to the complex software stack of serverless frameworks. vSwarm-u provides a set of tools, configurations, and documentation for gem5 to facilitate the client-server setup, load generation, function deployment, and results analysis.

3.2 Creating a RISC-V Development Platform

For the porting and development process of the serverless workloads, we needed a RISC-V-based development platform. Since we do not have access to a real hardware platform, we decided to use an emulated RISC-V virtual machine (VM) based on QEMU [43].

There are many guides available for setting up a RISC-V VM on several linux distributions, but we chose to use this guide [50] that targets Ubuntu for several reasons. First, the host x86 machine was also running Ubuntu, so we could test something fast on the host machine, and be more optimistic when later following the same steps on the slower emulated RISC-V VM. Second, Ubuntu is one of the most widely used Linux distributions, has a vast community of users, extensive documentation, and numerous forums (such as Ask Ubuntu). Third, the choice of Ubuntu helped us to not differentiate our approach from that supported in the vSwarm-u framework.

3.2.1 Selecting Linux Distribution

We used an x86-based host system that uses Ubuntu 20.04.6 (Focal Fossa). After installing QEMU and booting the RISC-V VM with the Ubuntu image, we enlarged the disk image and setup an ssh connection. We initially used the Ubuntu Focal 20.04 distribution (Ubuntu-focal-preinstalled-server image) in order to use the same software infrastructure with the one used in the vSwarm-u framework for the x86 ISA. However, after working with Ubuntu Focal 20.04 for about a month we were faced constantly with challenges. In particular, it became clear that the RISC-V software ecosystem is less mature that the

x86 and Arm ecosystems. It was necessary in several occasions to compile individual packages from source. Considering the fact that Ubuntu 22.04 (Jammy Jellyfish), as the latest LTS at that time that includes support for the RISC-V ISA, might have packages that were not available in Focal, we decided to switch to Ubuntu 22.04 for the RISC-V VM with QEMU. We opted for the preinstalled server image for RISC-V for SiFive's HiFive Unmatched platforms.

3.2.2 Installing Docker

The setup procedure of Docker is a classic example of the immaturity of the RISC-V software support. Despite being a popular tool for software development and deployment, the configuration of Docker is not feasible through the packet manager of Ubuntu as of June 2024. To overcome this, we manually compiled and built docker [13]. This was a time consuming process, because it was performed within the RISC-V VM. Building Docker and other necessary packages, such as containerd and rootlesskit among others, took almost 3 hours in our setup. Then we ensured that the RISC-V VM's kernel was capable of running docker containers [9]. Finally, we installed an SSH server in the RISC-V VM.

3.3 Porting Serverless Benchmarks to the RISC-V ISA

We now describe our effort on porting the serverless benchamrks. We first focus on the vSwarm standalone functions, and then we describe the porting process for the Online Shop and the Hotel applications, respectively.

3.3.1 Standalone Functions

3.3.1.1 Go and NodeJs

We started the porting process with the Fibonacci-Go benchmark. In theory the steps are relatively simple; we had to find a Go base image for the RISC-V ISA and replace the corresponding part on the function's Dockerfile. The rest of the Dockerfile's instructions are architecture independent. Indeed, finding a compatible image was relatively easy. We searched Docker Hub for a GO image and set a filter for RISC-V [54]. Afterwards we changed the **FROM** line at Dockerfile and built the container. Finally, we ran the function. After compiling the client that performs the requests for RISC-V, we managed to successfully execute the functions in the RISC-V VM. We followed the same steps for the Fibonacci-NodeJs, as well as for the rest of the standalone functions, i.e., AES and Auth.

3.3.1.2 Python

The primary issue that we faced with porting the Python standalone functions to RISC-V was importing the gRPC module [26]. We searched Docker Hub to find a compatible image. The whole trial and error process looked like this: (i) find a candidate image, (ii) install requirements, (iii) success or failure to finish docker build, (iv) run docker, and (v) import error for the grcp module (undefined symbol: atomic-compare-exchange-1).

We started our building process with a vanilla Ubuntu RISC-V image and our first guess was that it is a python version related error. We experimented with various python versions, but did not make any progress. The procedure's major setback was that installing grpcio and grpcio-tools with pip [41] lasted around 4 hours when done inside the RISC-V VM. We tried installing those modules via package manager but the error was not resolved. Additionally we made an effort to manually compile the grpc source code and pass it into the container as a binary or executable, but failed because the bazel tool was needed. When trying to compile bazel we could not do it in our RISC-V VM and also, we did not succeed in cross compiling it for RISC-V in our x86 host image.

To solve this issue, we changed our way of thinking and started from a different basis. We tried to run the python function directly inside the RISC-V VM without using containers. We came across the same error due to undefined symbol to atomic-compare-exchange-1. While searching GitHub issues we stumbled upon this one [25], which says that the atomic library needs to be preloaded. We applied the same technique on similar python modules and we managed to run the python function. Afterwards, we found a similar python base image to our Jammy RISC-V VM, repeated the steps and eventually fibonacci-python successfully ran on the RISC-V VM. We followed the same steps for the AES and Auth functions.

3.3.2 Online Shop Application

Porting the Online Shop application did not impose any substantial problem. In order to have shorter python builds, we created a Python 3.10 image with grpc and grpcio prebuilt.

3.3.3 Hotel Application

The functions of the Hotel Application are written in Go, a language that did not impose any important obstacles when porting the corresponding standalone functions. However, all the functions communicate with a database container, which is initialized in the beginning of the execution of the application, and then is used for the execution of the application's functions. The application uses MongoDB as database.

MongoDB [36] is a source-available database management program. NoSQL (Not only SQL) is used as an alternative to traditional relational databases. NoSQL databases are quite useful for working with large sets of distributed data. MongoDB is a tool that can manage document-oriented information, and store or retrieve data. MongoDB is used for high-volume data storage, helping organizations store large amounts of data while still performing rapidly. Organizations also use MongoDB for its ad-hoc queries, indexing, load-balancing, aggregation, server-side JavaScript execution and other features. Instead of using tables and rows as in relational databases, as a NoSQL database, the MongoDB architecture is made up of collections and documents.

However, porting the Hotel Application to RISC-V was not straightforward. The reason is that MongoDB has not been ported yet to the RISC-V ISA. We tried to port it ourselves but we did not succeed. We could not resolve the issues that stopped this attempt [35], nor found relevant information regarding other efforts towards RISC-V porting. As mentioned here [49], ”MongoDB is not a RISC-V friendly database”. Hence, we decided to replace MongoDB with another NoSQL database. That database should have similar characteristics with MongoDB and should be available in the RISC-V ISA.

3.3.3.1 Alternatives to MongoDB

Apache Cassandra. Apache Cassandra [8] is a distributed NoSQL database created at Facebook and later released as an open source project in July 2008. Cassandra delivers the continuous availability (zero downtime), high performance, and linear scalability that modern applications require, while also offering operational simplicity and effortless replication across multiple data centers and geographies. It can handle petabytes of information and thousands of concurrent operations per second, enabling organizations to manage large amounts of structured data across hybrid and multi-cloud environments. In addition, one positive aspect of Cassandra is that, there are many RISC-V containers already uploaded in Docker Hub, that made us optimistic about the porting process.

MariaDB. Another database that we considered as a MongoDB alternative was MariaDb. MariaDB [34] is a community-driven, commercially supported variant of the MySQL relational database management system (RDBMS), designed to stay free and open-source under the GNU General Public License. It is developed by some of the original MySQL creators who forked it following Oracle Corporation's acquisition of MySQL in 2009. An RDBMS is a common type of database that manages predefined relationships between data, in which data is organized as a set of tables, columns, and rows. The columns in the table store data attributes, and each row is a record with values for each attribute. A unique ID or primary key makes it possible to create relationships between the data. The relational database model is widely used in organizations of all sizes. MariaDB aims to maintain high compatibility with MySQL, including exact alignment with MySQL APIs and commands, which often allows it to serve as a direct replacement for MySQL. However, MariaDB is evolving with new features and diverging from MySQL, incorporating additional storage engines such as Aria, ColumnStore, and MyRocks.

Redis. Redis [44] is a source-available, in-memory storage solution designed as a distributed, in-memory key-value database, cache, and message broker with optional durability. By storing all data in memory, Redis provides low-latency read and write operations, making it particularly effective for caching use cases. As the most popular NoSQL database and one of the leading databases overall, Redis is utilized by major companies including Twitter, Amazon, and OpenAI. Redis supports a variety of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams, and spatial indices. Redis is in fact RISC-V friendly, boots rapidly and is NoSql. Nonetheless we turned down this option. The reason is that Redis is rarely used for main database. Its most common usage is as a caching instance in order to reduce the more time consuming requests towards the main database.

3.3.3.2 Introducing alternative databases in the Hotel Application

Cassandra. We modified the hotel application to use Cassandra, instead of MongoDB. When running these new containers in our RISC-V VM we noticed that booting the container with the database was taking a significant amount of time (greater than 10 minutes). This was a common observation among all pre-built docker containers for Cassandra. We were able to perform requests only after 17-18 minutes. To tackle this we built our own container with Cassandra and modified settings like heap-size , num-of-tokens, num-of-nodes, etc. Despite our efforts, we did not manage to lower the 17 minutes threshold. It is worth noting that in the native x86 environment the corresponding Cassandra boot time is 30-40 seconds, i.e., five times slower compared to the MongoDB boot time.

MariaDB. We also modified the hotel application to use MariaDB, instead of MongoDB. MariaDb features similar times to MongoDB in x86 and takes around 3-4 minutes to boot in RISC-V VM with QEMU. Additionally, the porting process was pretty straightforward due to the fact that MariaDB is a ”RISC-V friendly” database (in contrast to Cassandra that we had to manually compile it because we could not install it via package manager). Nonetheless, since MariaDB is a relational database we abandoned the porting process with that database.

Summary. Taking into consideration all the aforementioned trade-offs, we concluded in Cassandra. In our experiments we study the interval between a function request and its reply. The boot time of the database is not a factor that we take into account. Furthermore, it is safe to suppose that a database instance might have different criteria for deactivation, and will probably be active for a longer period with respect to a simple/regular function container.

3.4 Enabling the Execution of the Benchmarks in gem5

We now describe our approach on porting the vSwarm-u framework to RISC-V and enabling the execution of the aforementioned serverless workloads on simulated RISC-V CPUs with the gem5 simulator.

3.4.1 The vSwarm-u Framework

We initially got familiar with the existing vSwarm-u framework that allows the execution of the vSwarm serverless workloads on simulated x86 CPUs using the gem5 simulator. More specifically, the important steps for using the vSwarm-u framework are:

1. Download prebuilt resources (Ubuntu disk image and Linux Kernel)
2. Install Qemu on host machine
3. Using Qemu:
   (a) Install Docker, Go, and the rest of the dependencies for running Docker containers
   (b) Download the image and run simple simulations for testing purposes
4. Download gem5, apply the gem5 patch, and compile gem5.opt for x86
5. Using gem5:
   (a) Boot with atomic kvm core for every function
   (b) Perform functional warming (*setup mode*)
   (c) Take a checkpoint
   (d) Boot again from checkpoint with O3 detailed core (*evaluation mode*)
   (e) Collect statistics
6. Analyze results

While getting familiar with the existing vSwarm-u framework, we came across a bug quite often. The setup mode of the framework with the gem5's KVM core model was not stable. A lot of times, the gem5 simulator was freezing when a magic M5 instruction was executed. The most common case was when a checkpoint was taken. This behavior resulted in failures to complete the setup mode for every function. Other times, despite the setup mode was being completed successfully, we were unable to boot with the O3 detailed core model using the corresponding checkpoint. The instability of using the KVM core model has been acknowledged by the authors of the vSwarm-u framework [68]. To overcome these issues, we tried disabling the KVM core and using gem5 with the atomic core, but the host OS terminated the simulation process because it consumed all the memory resources. This behavior is probably due to some memory leakage in the implementation of the simulator. We made several other attempts to solve this problem and make the simulations more deterministic without success. In our opinion, a major reason for the overall lack of stability is that the gem5 system configuration file is based on earlier gem5 configuration files, and not on the later gem5 standard library.

We tackled this issue in our setup by using a more reliable in terms of memory leakage script that enabled us to boot using AtomicCore and take a deterministic and reliable checkpoint, as explained next.

3.4.2 gem5 & RISC-V

After having experimented with vSwarm-u and successfully ported the standalone functions to RISC-V, it was time for also porting the vSwarm-u framework to RISC-V.

3.4.2.1 Building gem5

Initially we were working in an x86 environment without root privileges, and failed to install all the required dependencies for compiling the gem5.opt executable for RISC-V. We worked around this issue by creating an x86 Ubuntu Jammy VM that run with KVM on the host. After setting up this VM, we managed to build the gem5.opt file.

3.4.2.2 RISC-V Linux Kernel for gem5 simulations

Our first thought was booting the simulator with a prebuilt kernel found in the gem5 resources site [20]. In spite of getting terminal access with such kernel, we were only able to use it with a vanilla Ubuntu disk image. In other words, when running simulations with a disk image that is capable of executing the serverless functions, the system was entering in emergency mode. In emergency mode the root file system was mounted as read-only and almost nothing was set up. This problem was emerging both with the Ubuntu Jammy and Focal distributions.

Next we tried using the same linux kernel image that we used for the RISC-V VM with QEMU. This approach would resolve the problem regarding missing dependencies, that we faced with the prebuilt linux kernel image. However, we also dismissed this alternative because gem5 is incapable of loading modules dynamically. Even after building all the modules into the kernel image, we still could not proceed as the resulting linux kernel image had a size of around 1 GB. This increased size was due to including numerous modules that were needed by the applications that we wanted to execute.

Compiling a custom linux kernel was the option that we had to pursue. The search for a linux configuration file that could handle our demands was ineffective. It is a quite difficult procedure for x86, let alone for a lesser utilized architecture like RISC-V. We tried using some kernel configuration files that we came across, but the emergency mode boot remained. Thus, generating our own config file was our next challenge. We were working with linux versions that vSwuarm-u and gem5.org were providing for the x86 ISA, aiming not to deviate from the original vSwarm-u framework. We successfully managed to build custom kernel images based on the 6.5.5 and 5.15.59 versions that are compatible with the gem5 simulator. The successful attempt consisted of the RISC-V-default-config file, manual addition of flags provided by this script [9] and mod2yes config so that all modules are statically built into the kernel.

3.4.2.3 RISC-V Bootloader

During our kernel/disk image trial and error approach there was a third component that we also had to resolve. This was the bootloader, a necessary file for a full system RISC-V simulation to run in gem5. This component was not a point of concern in x86 simulations as it is built into the kernel. Nevertheless, this is not the case for RISC-V. We also had to pass the OpenSBI [40] executable that is used by QEMU, as an argument in the gem5 configuration file.

3.4.2.4 gem5 RISC-V configuration file

As mentioned earlier, our starting configuration script was fs-riscv.py. It was the script that we made our first simulations and got familiar with plenty aspects of gem5. On the other hand, it constitutes a quite complex setup that is challenging to comprehend, much less debug in case something goes wrong. Additionally it would be also tricky to port this into x86. After some research we found out the gem5 stdlib and rewrote our workflow process based on that effortlessly.

3.5 Enhancing the existing infrastructure for x86

Based on our experience with porting vSwarm and vSwarm-u to RISC-V, we decided to enhance also the existing infrastructure for the x86 ISA, in order to enable a more direct and fair comparison between the RISC-V and x86 ISAs.

3.5.1 Serverless Functions for x86 CPUs

When it was time for developing the x86 side of the experiment, we rebuilt the vSwarm images using the most close to their RISC-V counterpart base images. This means, that it is possible for a x86 python container to get smaller, however, that is not the case for the RISC-V correspondent. For instance, in spite of available alpine python base images for x86, we could not find a RISC-V alpine candidate, so both images have an ubuntu jammy base. In general our goal is to compare the two ISAs with the same variables, so our development choices reflect our purpose.

3.5.2 gem5 & x86

3.5.2.1 Configuration file and disk image

Due to vSwuarm-u's known problems that were mentioned earlier, i.e., the instability of using the KVM core model and the fact that the original configuration script is not using the gem5 stdlib, we concluded that creating a new script would be the optimal alternative. Indeed, implementing our goal was straightforward, given that there is an x86 Ubuntu configuration file, to which we added components from the RISC-V counterpart. It was also easy to setup the disk image setup.

3.5.2.2 x86 Linux Kernel for gem5 Simulations

In contrast, building a custom x86 Linux kernel for Ubuntu Jammy was more complex than expected. Our initial strategy was to produce the kernel image in the same way we did for RISC-V. Despite trying with various kernel versions, we did not manage to get a running kernel using the defconfig option with the necessary script flags and mod2yes. When using the defconfig option as base, we could not get the init service to start due to a missing IDE driver.

Using the Linux kernel images and the configuration files from the vSwarm-u framework was our next step. However, those kernel images, whether they were locally compiled or downloaded, failed to completely fulfill their purpose. More specifically, we managed to run the functions that are written in Go and Python, but we did not manage to run functions in NodeJs. The reason is that those kernel images do not support the NodeJs framework for Ubuntu Jammy. We performed a lot of research on this issue and we were still getting segmentation fault, despite having made countless attempts with various modifications (different compilers, NodeJs versions, configuration flags, fresh installation of all componennts, etc).

Our last option was to use a gem5 resources script with the script flags mentioned earlier. Unfortunately, we did not overcome the obstacle of running NodeJs functions. We either got the first problem, of Ubuntu not booting, or a segmentation fault when typing node on the terminal.

3.5.2.3 Limitations

After spending a significant effort on this, we decided to not proceed with the NodeJs benchmarks for Jammy x86. In addition, we later discovered that we were also unable to boot MongoDb in gem5. This was a major setback because a MongoDB vs Cassandra comparison in gem5 was not feasible, despite being possible in QEMU. Due to time pressure we are unable to port our entire workload for both x86 and RISC-V in the Ubuntu Focal environment. One key factor is the RISC-V Cassandra gem5 boot which took about one week for Jammy. This is an issue that we will revisit again in the future. It is worth noting that a full system simulation boot, using atomic core, takes around 7 hours to complete for RISC-V and around 5 hours for x86. Furthermore, the simulator consumes all available resources, which forbids us from engaging with another task. Additionally, KVM usage was not an option neither for x86, as it is not stable, nor for RISC-V, as we did not possessed a RISC-V board. Finally, gem5's inability to load modules dynamically obliged

us to perform countless trial and error attempts to come up with an appropriate linux kernel configuration.

4. EVALUATION

In this section we describe in detail the experimental methodology that we followed and then we present the results.

4.1 Experimental Methodology

4.1.1 Software and Hardware Configuration

Table 4.1 lists the common configuration parameters that were used for the simulation of the x86 and RISC-V multicore processors with gem5. Tables 4.2 and 4.3 show additional configuration paramaters that are specific to the RISC-V and x86 processors, respectively.

L1 I Cache	2 Cores x 32KB, 8-way set associative
L1 D Cache	2 Cores x 32KB, 8-way set associative
L2 Cache	2 Cores x 512KB,4-way set associative
RAM	2GB,DDR3 1600, 800MHz, Single Channel
ITLB Page walk caches	2 Cores x 8KB
DTLB Page walk caches	2 Cores x 8KB
ROB	192 entries
LSQs	32 Load entries + 32 Store entries
Registers	256 Int + 256 Float
Number Of Cores	2
Clock Frequency	1GHz
Linux Kernel	5.15.59
Docker Version	25.0.0

Table 4.1: Common configuration parameters that were used for the simulation of the x86 and RISC-V processors with gem5.

Os	Ubuntu Jammy 22.04.3 Preinstalled Server
kernel compiled with gcc	riscv64-unknown-linux-gnu-gcc 13.2.0

Table 4.2: RISC-V specific configuration parameters.

Os	Ubuntu Jammy 22.04.4 Live Server
kernel compiled with gcc	gcc 11.4.0

Table 4.3: x86 specific configuration parameters.

4.1.2 Step-by-step Experimentation Process

4.1.2.1 Image Preparation

We start our experiment by creating the Ubuntu disk image via QEMU. That image will be later used with gem5 to perform detailed simulation experiments. We download a server image and install any dependencies needed for running it with QEMU. Afterwards we integrate everything required for Docker Container Engine to operate. After having a running Docker process, we can pull our containers and test them. We now have our benchmarks saved on the disk. We deactivate some unnecessary services to speed up the gem5 booting process and we shutdown the VM.

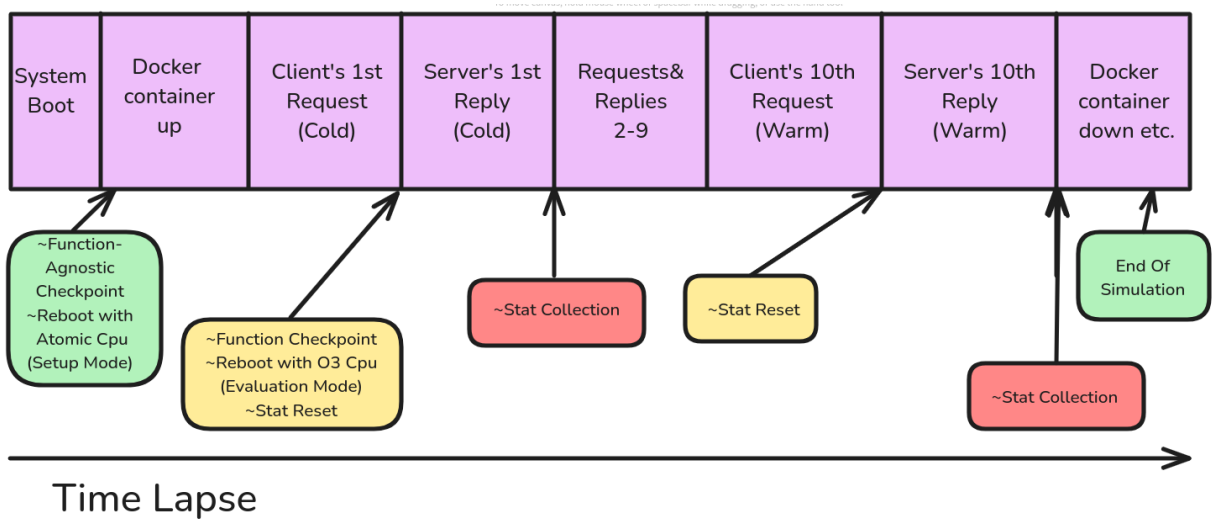

Figure 4.1: Experiment Process.

4.1.2.2 Setup Mode

We boot our image with gem5 using AtomicCPU. This takes several hours to complete. After that we perform a checkpoint. This allows us to begin each experiment from that state. For each function, we launch the container, pin it into a specific core, and perform 10 requests. We take a checkpoint right before the first request.

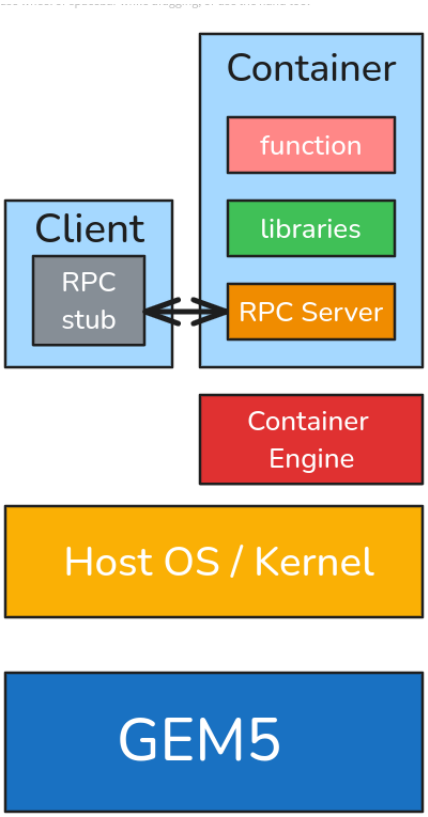

Figure 4.2: System Stack.

Figure 4.2 presents the system stack that is simulated using gem5. The figure is based on the vSwarm-u Original Picture.

4.1.2.3 Evaluation Mode and Stat Collection

In evaluation mode we initiate the simulator with an out-of-order detailed CPU. This is a substantially slower CPU model than Atomic, but it is very accurate. With this CPU we reset the simulator stats , then make the first request and finally dump the stats after getting a reply from the server. This also takes for the 10th request. In other words we measure the response time of a serverless function. To differentiate the two cases we refer to the first execution as *cold execution* and to the tenth execution as *warm execution*.

For each function we collect the following stats from the Cold and Warm execution: Number of Cycles, Number of Instructions Issued, Cycles Per Instruction, and Cache misses.

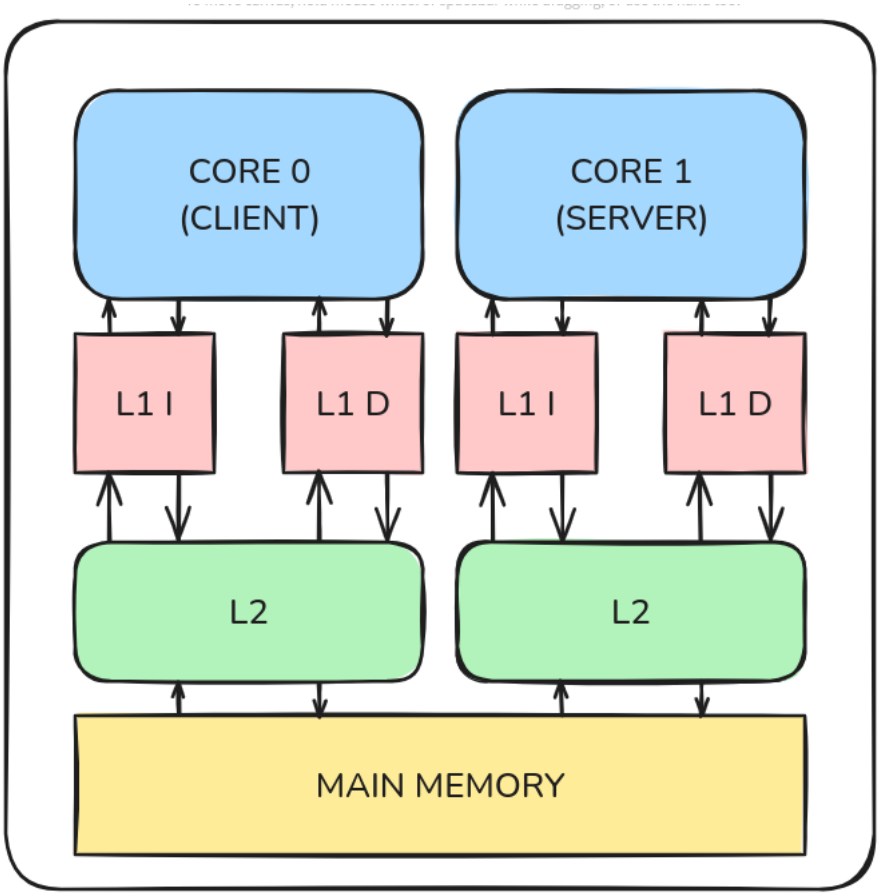

Figure 4.3: Overview of the multicore system that is used in the experiments with gem5.

Figure 4.3 depicts how the CPU of our setup is organized. The server containers are pinned to a specific core and any data we collect come from that core. The client is pinned to the other core. The figure is based on the vSwarm-u Original Picture.

4.2 Results

4.2.1 RISC-V Results

In this section we present the results and discuss our findings regarding the RISC-V ISA.

4.2.1.1 Standalone Functions and Online Shop

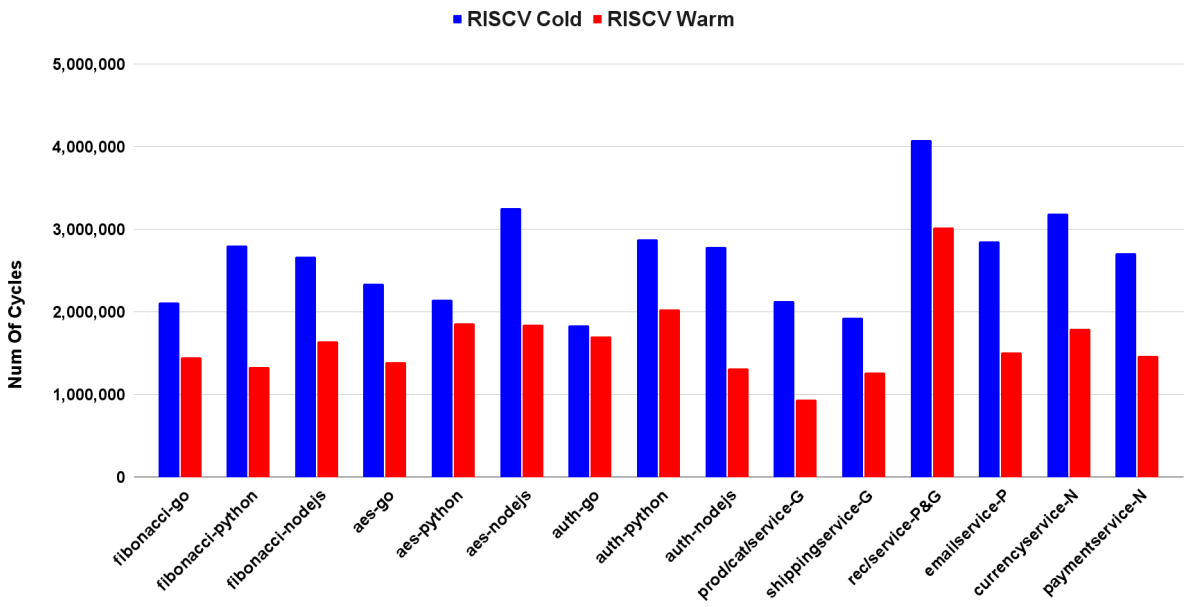

Figure 4.4: Number of cycles for the standalone functions and the online shop application on the RISC-V simulated system.

Figure 4.4 shows the number of cycles for the standalone functions and the online shop application on the RISC-V simulated system. We observe that the Go benchmarks tend to have the fewest cold cycles. On the other hand, the NodeJs benchmarks feature a 50% speedup in warm executions. Regarding the Fibonacci set of functions, we notice that the Python version despite having the longest cold execution, takes the shortest amount of time to complete in the warm execution.

4.2.1.2 Hotel application

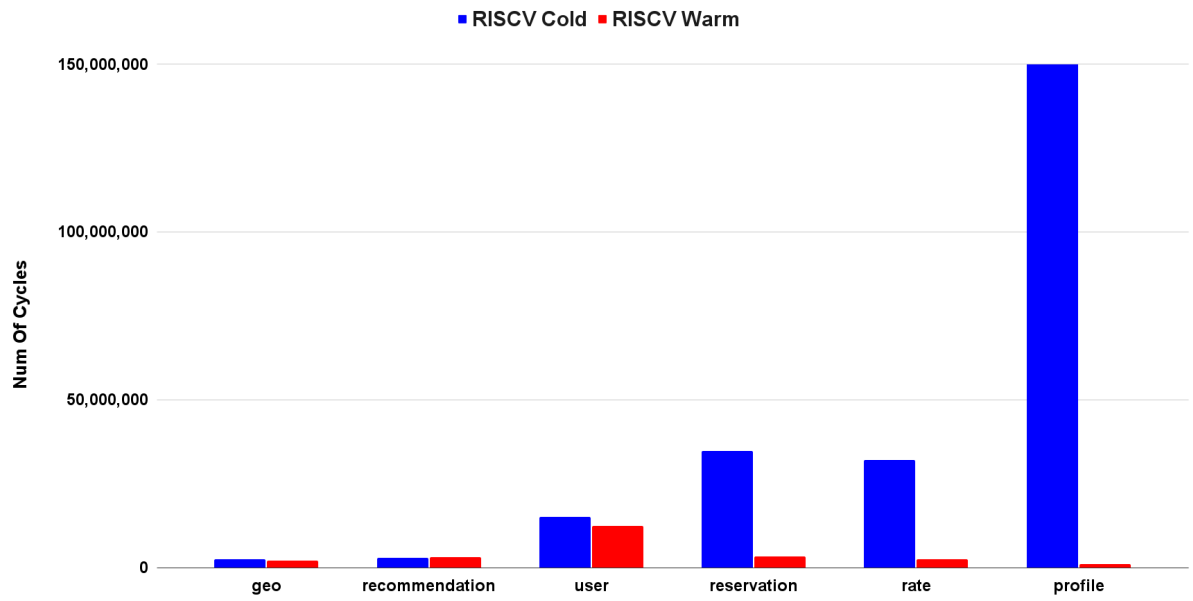

Figure 4.5: Number of cycles for the hotel application on the RISC-V simulated system.

Figure 4.5 shows the number of CPU cycles when running the Hotel Suite on the RISC-V simulated system. Regarding Hotel, that consists only of Go functions that also communicate with a Cassandra database instance, we notice that the cold execution times last significantly longer with respect to that of the standalone functions. In other words, the standalone functions had cold executions of around two and three million cycles, while in Hotel we observe sizes ten time greater. In particular for the profile function, cold boot lasts 351 million cycles, but we decided to not depict that in the graph. We opted for that because otherwise we would be unable to make any substantial comments. Furthermore, we detect smaller amount of cycles for the first three functions but not for the last three functions. This stems from the fact that, the last three functions also use a caching instance database. They firstly communicate with Memcached and afterwards perform requests to the Cassandra database. After getting a reply from Cassandra, they are obliged to populate the middle base, for later usage. This back and forth is also visible in the cache statistics.

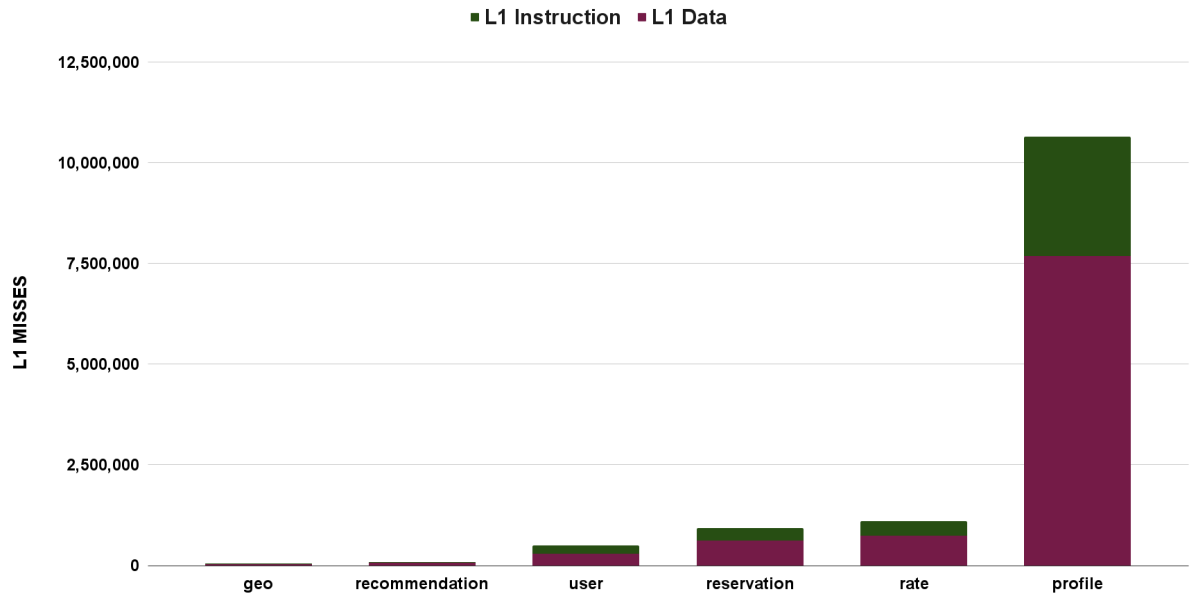

Figure 4.6: Number of L1 cache misses for the hotel application on the RISC-V simulated system after cold execution.

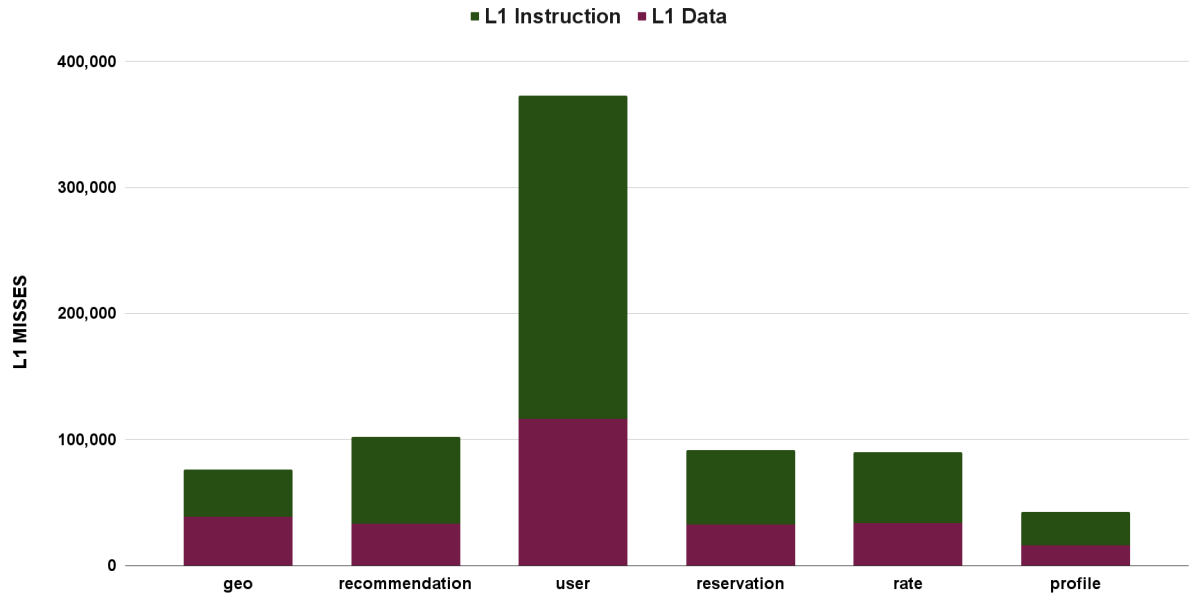

Figure 4.7: Number of L1 cache misses for the hotel application on the RISC-V simulated system after warm execution.

Figures 4.6 and 4.7 portray the combined data and instruction L1 cache misses during the cold and warm executions. We see that the functions that depend on Memcached undergo slowdown due to cache misses. On the other side, those functions take advantage of the middle base in the warm execution, with profile, the least fast function in Cold, having the least misses and therefore number of cycles.

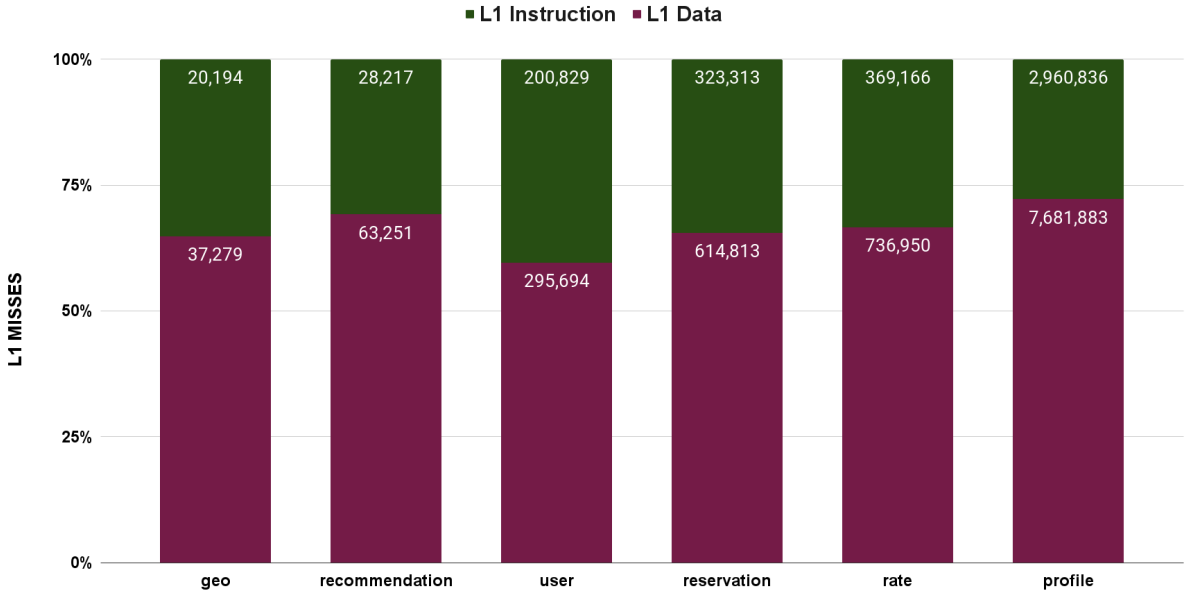

Figure 4.8: Percentage of L1 cache misses for the hotel application on the RISC-V simulated system after cold execution.

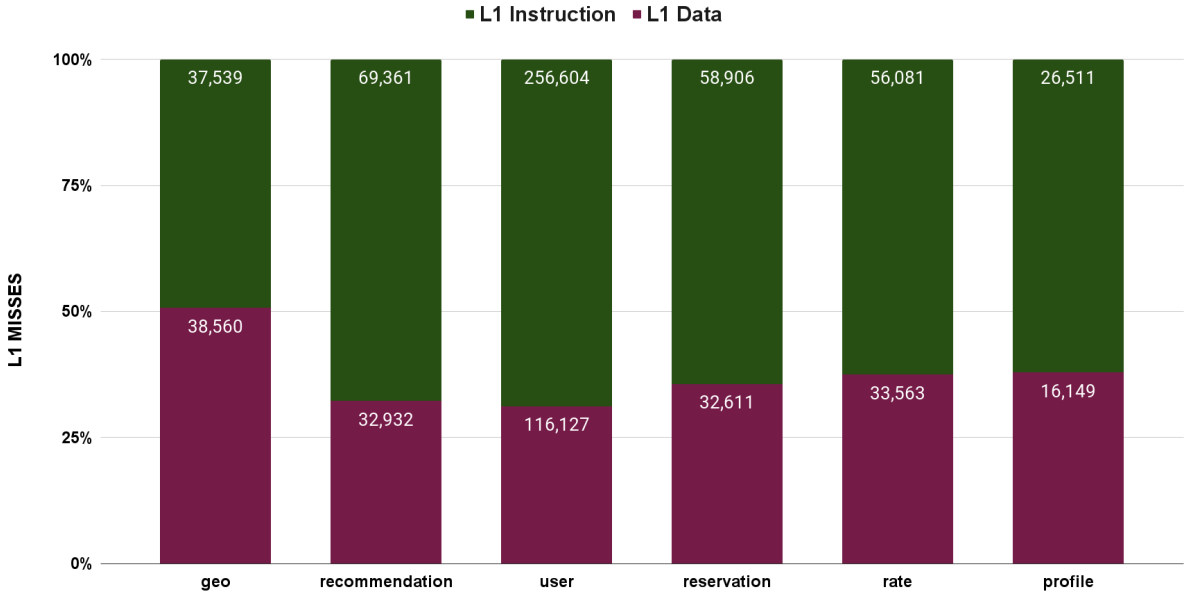

Figure 4.9: Percentage of L1 cache misses for the hotel application on the RISC-V simulated system after warm execution.

Figures 4.8 and 4.9 display the total percentage of Instruction and Data cache misses. We observe that in cold executions, the data cache misses are 60% of misses on average while in warm execution they are close to 30%. It is a behavior that we expect since in the first execution the functions request plenty of data for the first time. Subsequently, on the 10th run of the function, some of that data, are already present in cache hierarchy.

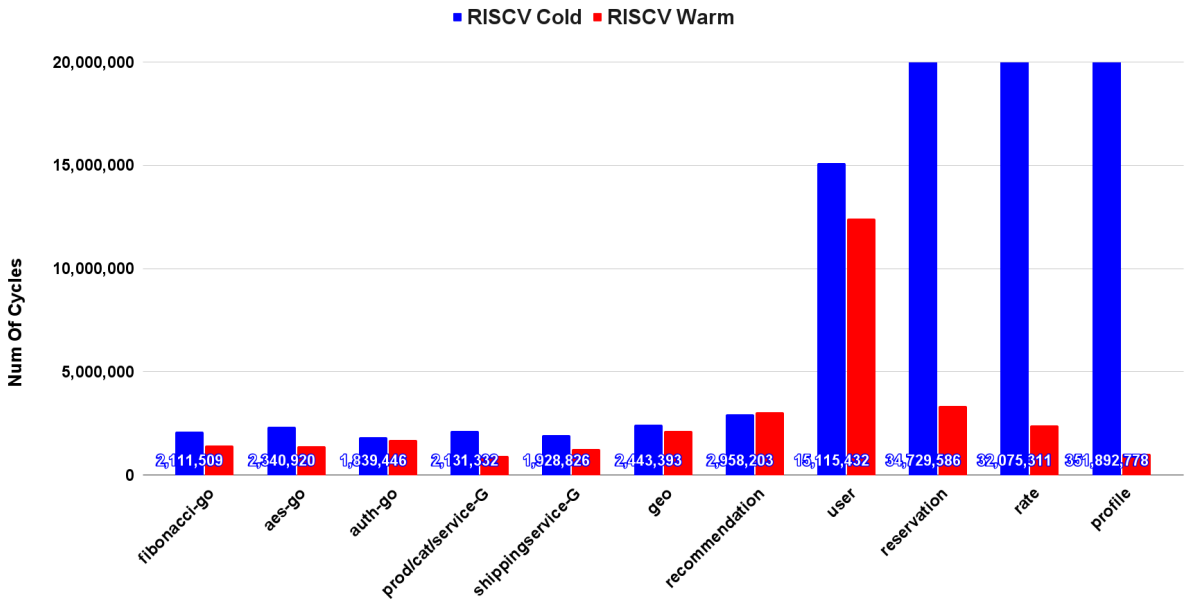

Figure 4.10: Number of cycles for the Go functions on the RISC-V simulated system.

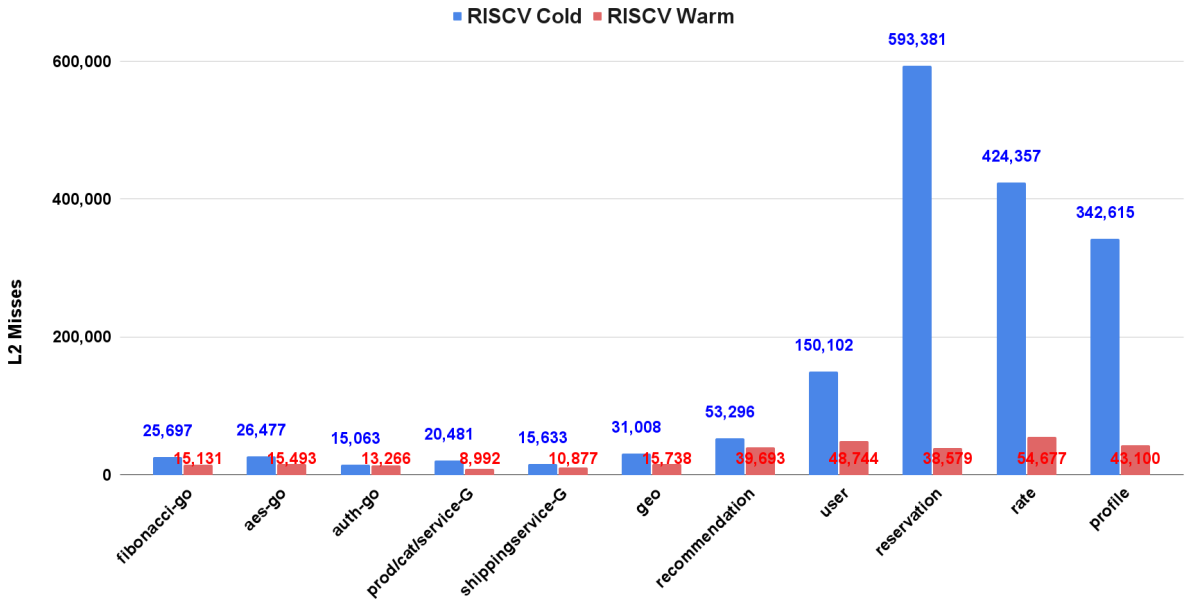

Figure 4.11: Number of L2 misses for the Go functions on the RISC-V simulated system.

Figures 4.10 and 4.11 illustrate the reason the Memcached subgroup from the hotel applications exhibits 10 times slowdown in relation to other Go Benchmarks. Those functions get plenty of L2 misses and hence they frequently experience the costly process of accessing the main memory.

4.2.2 x86 Results

4.2.2.1 Standalone Functions and Online Shop

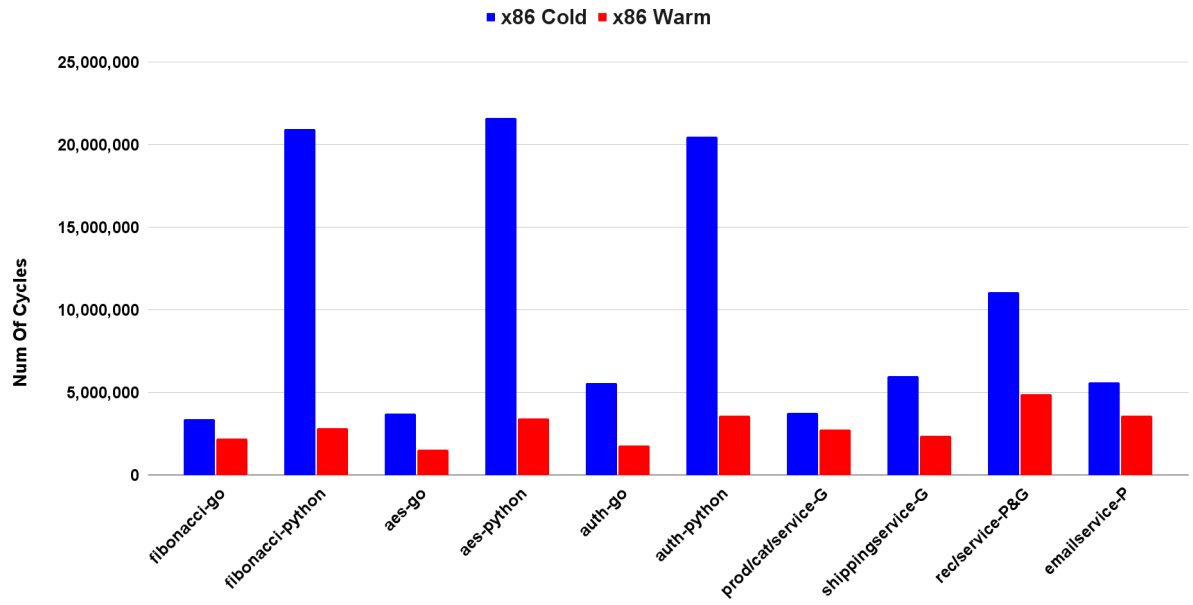

Figure 4.12: Number of cycles for the standalone functions and the online shop application on the x86 simulated system.

In the x86 experiments we come into different conclusions. Looking at Figure 4.12 we observe that the Python benchmarks perform poorly in cold executions. They are near 10 times slower compared to warm executions. Nonetheless, we see an exception to this phenomenon. This exception is the emailservice benchmark. Its better performance is thanks to its lower number of L2 cache misses as depicted below.

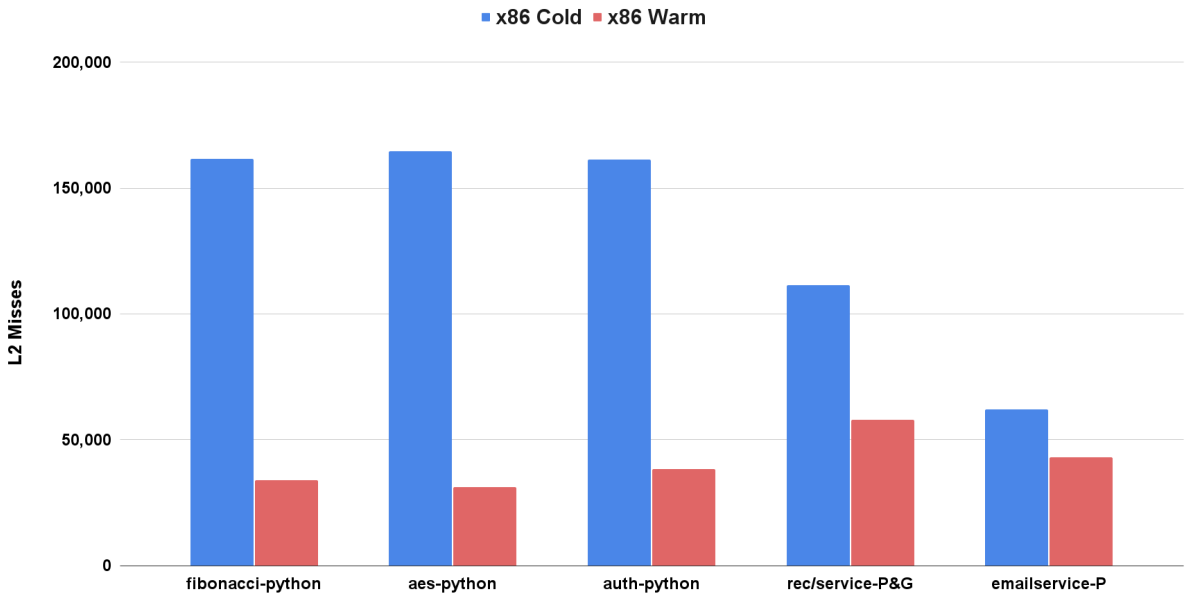

Figure 4.13: Number of L2 misses for the Python functions on the x86 simulated system.

4.2.2.2 Hotel application

For the Hotel collection we see similar results to its RISC-V counterpart. The only difference is the absence of the extreme execution of the profile benchmark.

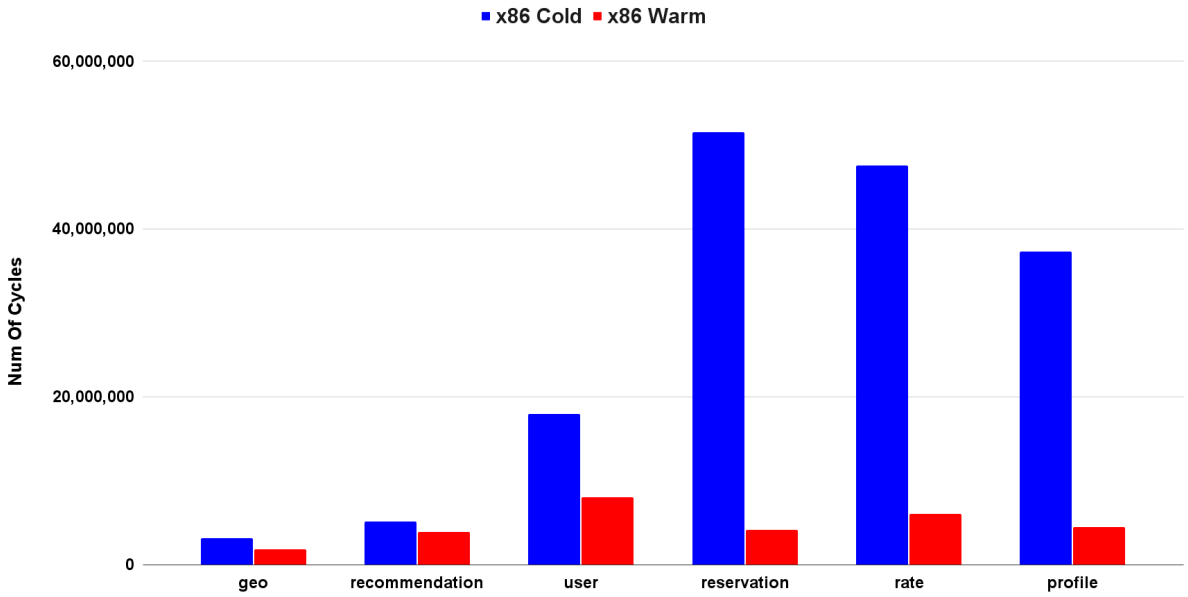

Figure 4.14: Number of cycles for the hotel application on the x86 simulated system.

4.2.3 RISC-V vs x86 Results

4.2.3.1 Standalone Functions and Online Shop

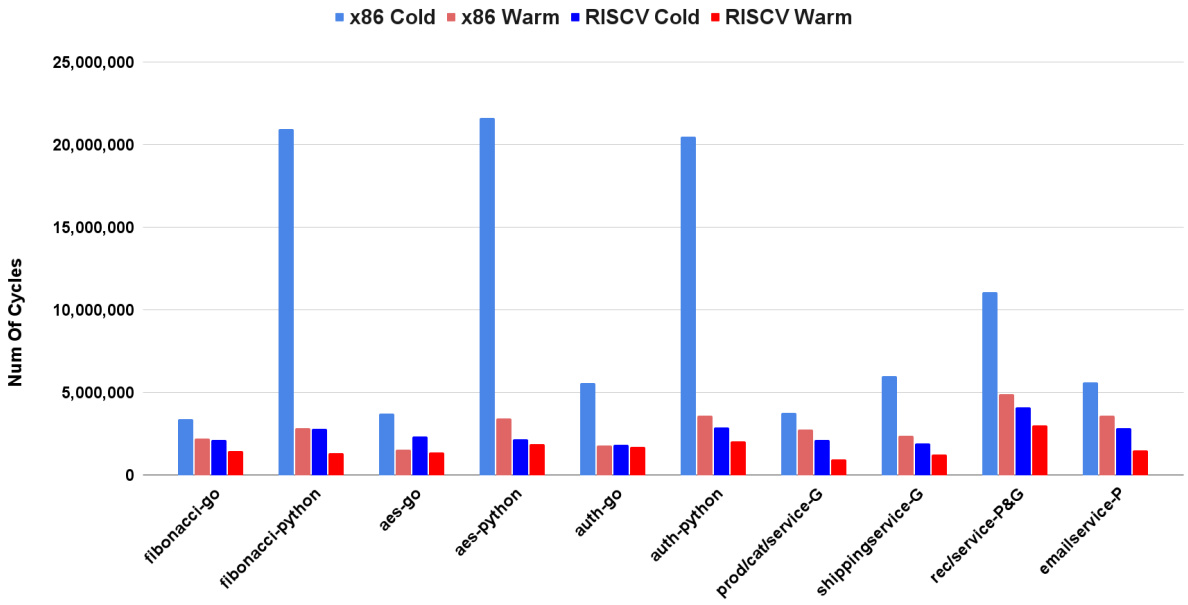

Figure 4.15: Number of cycles for the standalone functions and the online shop application on the RISC-V and the x86 simulated systems.

Figure 4.15 shows the number of cycles for the standalone functions and the online shop application on the RISC-V and the x86 simulated systems. Our first observation is that the RISC-V containers seem to be doing better than their x86 counterparts. In fact, most of the times, the cold execution time in the RISC-V simulated system is even shorter than the warm execution time in the x86 simulated system. The RISCV containers run for approximately 2,5 million and less cycles. We cannot say the same for the cold executions of the x86 ones. The main reason for this performance difference is the fact that the execution of the functions in the RISC-V simulated platform resulted in significantly fewer executed instructions than the execution of the functions in the x86 simulated platform.

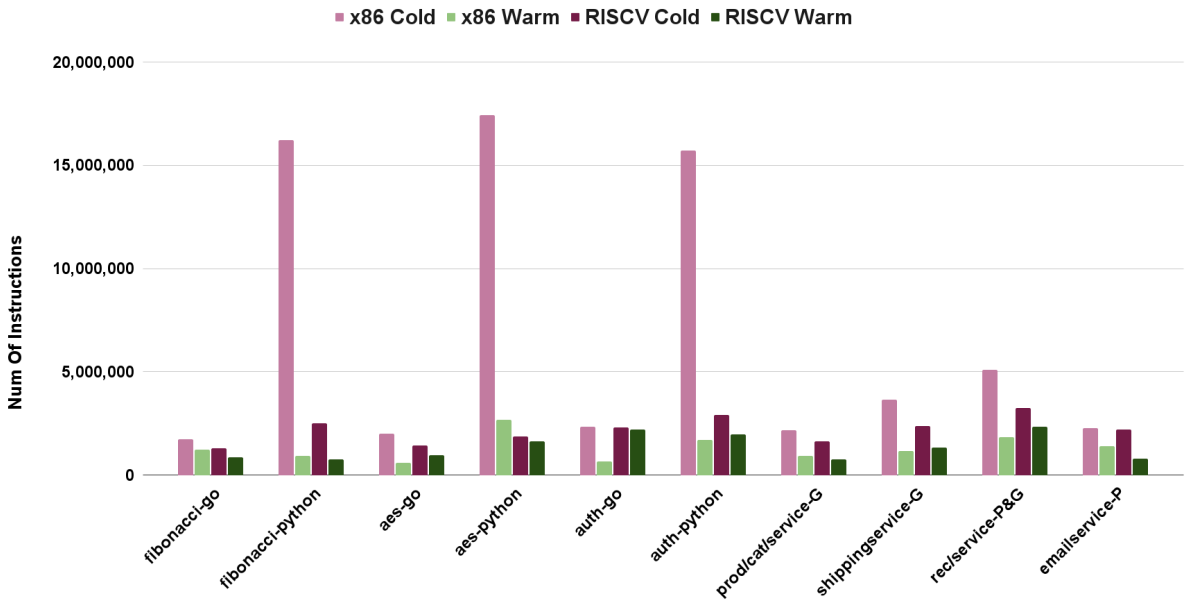

Figure 4.16: Number of executed instructions for the standalone functions and the online shop application on the RISC-V and the x86 simulated systems.

Figure 4.16 depicts the number of instructions executed. Looking at this we observe that x86 containers execute more instructions than the RISC-V containers in the cold execution, but that is not the case in the warm phase. Here, we can point some cases where x86 is more effective (aes-go, auth-go, auth-python)

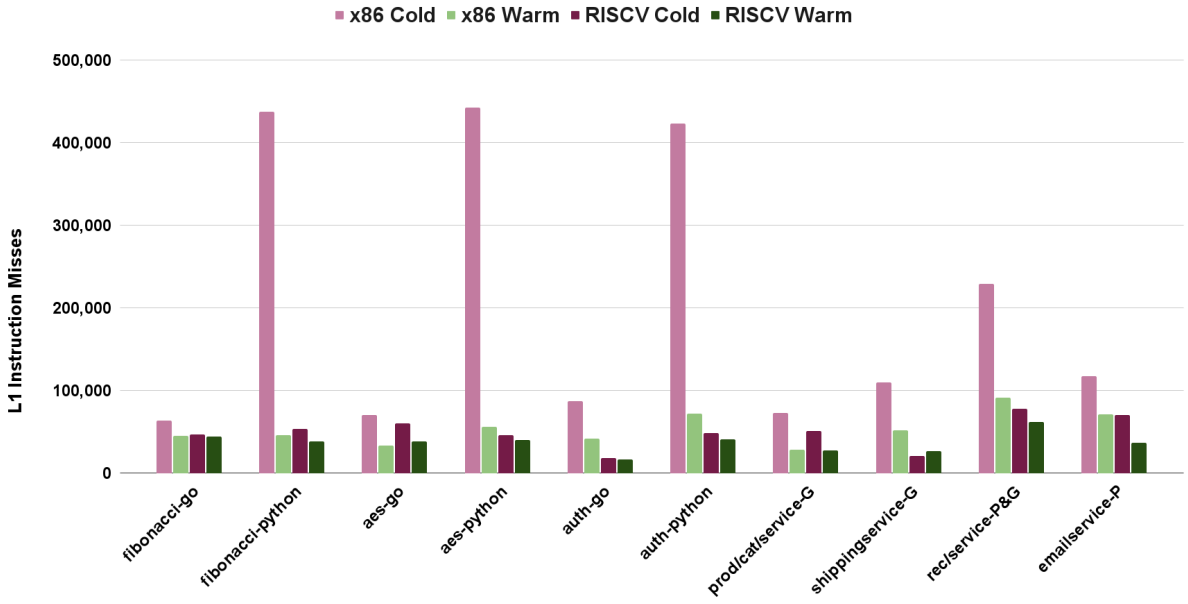

Figure 4.17: Number of L1 instruction misses for the standalone functions and the online shop application on the RISC-V and the x86 simulated systems.

Figure 4.17 shows the misses in the L1 Instruction Cache. It is very clear that for the majority of the comparisons RISC-V comes victorious either with slight margins (Warm

Go benchmarks) or with more visible ones (Python Benchmarks).

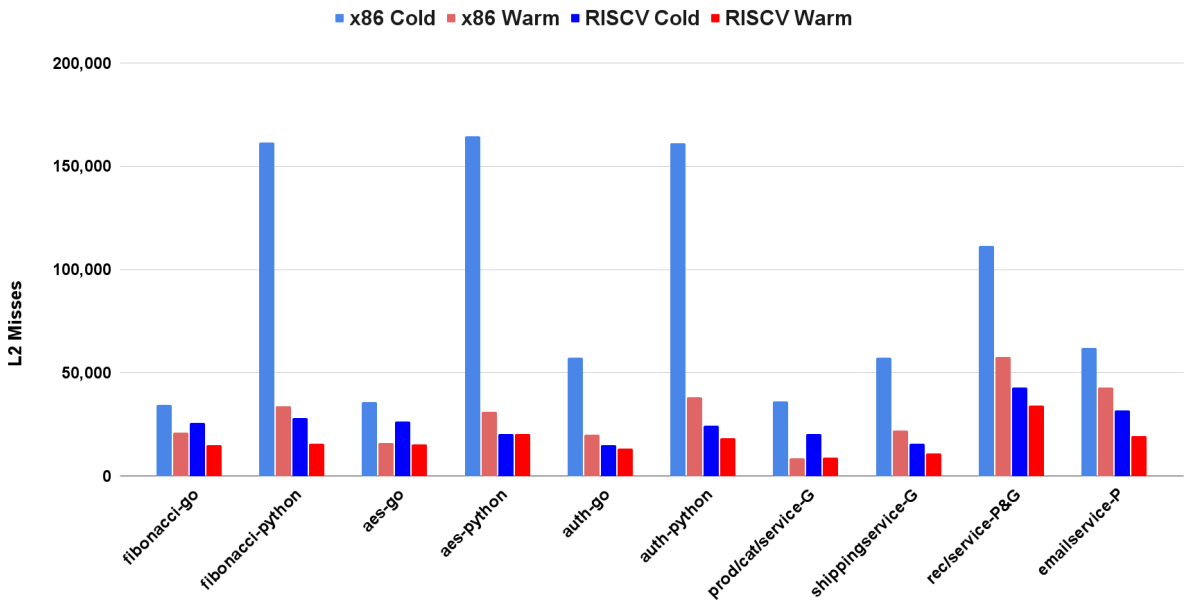

Figure 4.18: Number of L2 misses for the standalone functions and the online shop application on the RISC-V and the x86 simulated systems.

This figure is very similar to 4.15. It provides us with confidence to claim that the L2 cache is possibly responsible for the fact that we see better performance in RISCV. Note that a L2 miss costs at least 600 cycles which shutters the CPU'S pipeline. Additionally, as the size of function is encouraged to get smaller, its going to become ever more difficult for the CPU to mask the time lost with another segment of code.

4.2.3.2 Hotel application

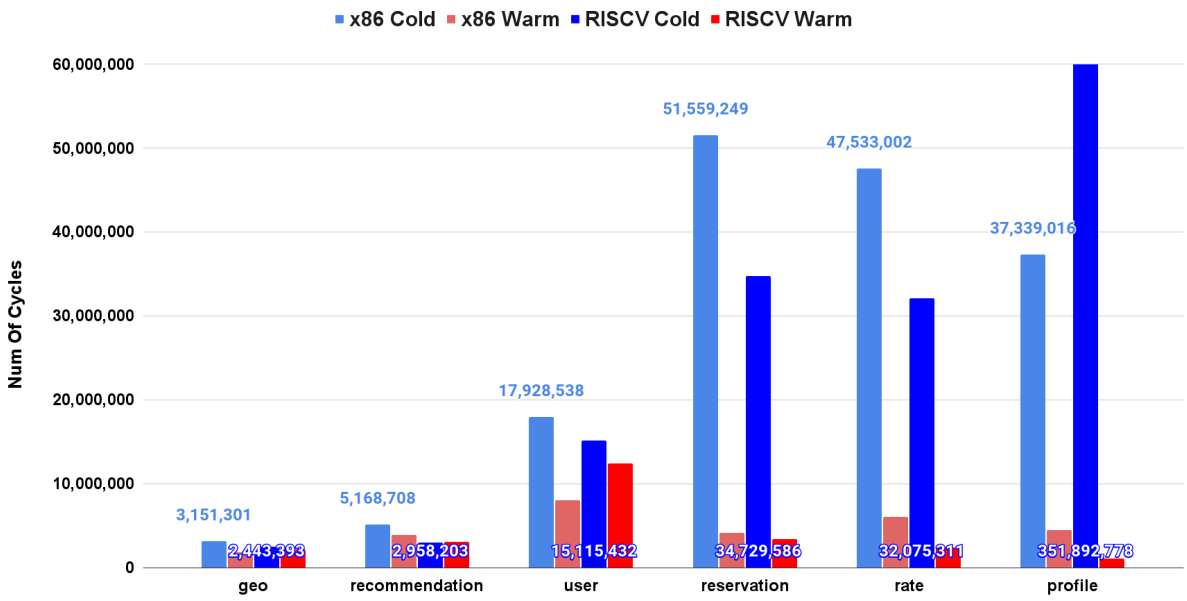

Figure 4.19: Number of cycles for the hotel application on the RISC-V and the x86 simulated systems.

In Hotel we continue to see RISCV performing better on most occasions. In Figure 4.19 we see neither architecture can perform well in the cold execution. An interesting fact is that the cold RISCV profile benchmark that has the worst performance of all the workloads, is the quickest in warm executions.

4.2.4 MongoDb vs Cassandra

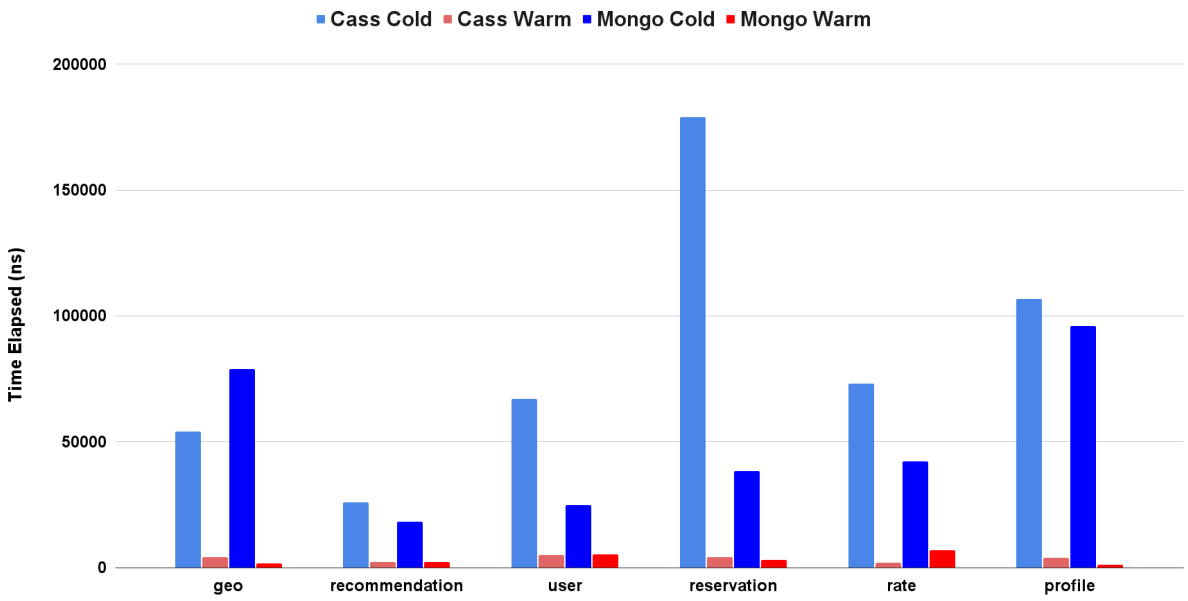

Figure 4.20: Execution time comparison between MongoDB and Cassandra using QEMU for the x86 ISA.

As mentioned earlier, we were left with no choice but comparing those two databases in QEMU. Figure 4.20 shows the performance of Cassandra and X86 in cold and warm executions. MongoDB appears to have shorter times in cold executions. However, we cannot say that this also happens to a substantial extend in the warm execution of the experiment. In general, we could argue that MongoDB performs better, but the nature of the experiment does not grand us with confidence to claim that this can also happen in the gem5 simulation.

4.2.5 RISC-V vs x86 Container Sizes

Table 4.4 depicts the compressed size for x86 and RISC-V containers. It is quite clear that the Go runtime containers are the lightest. NodeJs come second and the Python ones come last. Looking at this graph along with the results in Section 4.2.3 we have reasonable basis to claim that the container size plays an important role in cold execution time. Aside from the hotel benchmarks that have dependency on the database container, we observe that the shortest cold boots are from the Go containers and the longest are from those having a Python base image.

4.2.6 Similar Work Size Comparison

During the porting process, we used to compare our RISC-V compressed sizes to their vSwarm counterparts in order to see how close we are. During such searches, we stumbled upon this profile in Docker Hub [37]. To our surprise all the workloads where successfully ported, even the hotel suite. We tested these containers and validated using QEMU that

Function	x86	RISC-V
Fibonacci-Go	8.39	7.76
Fibonacci-Python	99.4	132.62
Fibonacci-NodeJs	58.43	35.16
Aes-Go	8.67	8.04
Aes-Python	99.45	132.67
Aes-NodeJs	57.11	35.42
Auth-Go	8.67	8.04
Auth-Python	99.4	132.62
Auth-NodeJs	70.5	48.81
Product-Catalog-service-Go	10.81	10.33
Shipping-service-Go	10.8	10.3
Recommendation-service-Python	108.09	114.68
Email-service-Python	107.7	114.46
Currency-service-NodeJs	60.12	38.44
Payment-service-NodeJs	59.04	80.64
Geo-Go	8.17	7.76
Recommendation-Go	8.14	7.74
User-Go	8.12	7.73
Reservation-Go	8.18	7.79
Rate-Go	8.18	7.79
Profile-Go	8.19	7.79

Table 4.4: Docker Container Compressed Size in MB.

Function	Natheesan	Gpour
Fibonacci-Go	6.72	7.76
Fibonacci-Python	299.56	132.62
Fibonacci-NodeJs	107.74	35.16
Aes-Go	6.95	8.04
Aes-Python	299.62	132.67
Aes-NodeJs	107.81	35.42
Auth-Go	6.95	8.04
Auth-Python	299.57	132.62
Auth-NodeJs	121.21	48.81
Product-Catalog-service-Go	26.15	10.33
Shipping-service-Go	26.14	10.3
Recommendation-service-Python	401.46	114.68
Email-service-Python	313.06	114.46
Currency-service-NodeJs	58.16	38.44
Payment-service-NodeJs	57.07	80.64

Table 4.5: GPour/Natheesan RISC-V Docker Container Compressed Size in MB.

they provide the same results as we and vSwarm do. However, we were unable to run the hotel application because it seemed to try to connect to a MongoDB. As mentioned earlier, there is no available port of MongoDB to RISC-V. That is the reason why we do not report those containers.

5. RELATED WORK

In this section we review prior approaches that have targeted serverless on RISC-V platforms. We also describe other prior works that have focused on benchmarking serverless systems (besides those that are covered in Section 3.1). Finally, we discuss prior works that have focused on the implications of the microarchitectural components in the execution of serverless workloads, as well as works that have proposed microarchitectual optimizations for improving their performance.

Serverless Computing & RISC-V. Feng et al. [16] used an image processing serverless application as a case study to evaluate the performance of a hardware-software enclave co-design for RISC-V processors. Starc et al. [61, 60] evaluated the execution of serverless workloads on several RISC-V soft-cores on FPGA. However, they did not port or use any existing benchmark suite. Instead, they used standalone functions without considering the containerization/virtualization layers that play a critical performance role in the serverless computing stack. Moreover, they do not provide support for running serverless workloads in microarchitectural simulators, e.g., gem5, for the RISC-V ISA. Finally, the developed infrastructure is not publicly available.

Other Benchmarking & Profiling Approaches. Gan et al. [18] proposed DeathStarBench, a benchmark suite that targets microservice-based applications and enables studying the architectural characteristics of microservices and their implications on the computing stack. Shahrad et al. [57] developed a testing and profiler platform for serverless computing based on OpenWhisk targeting real platforms. Finally, Xu et al. [70] recently proposed MindPalace, a simulation framework that combines QEMU and ChampSim and allows studying the architectural behavior of serverless systems for x86 processors.

Impact of Microarchitecture on Serverless Computing. Zhu et al. [72] studied the microarchitectural implications of server-side JavaScript applications using the NodeJs framework. Kanev et al. [29] presented a microarchitectural analysis of the Google's warehouse-scale computer workloads that typically consist of multiple microservices. Shahrad et al. [57] focused on the architectural implications of serverless computing and showed that the interleaved execution of different functions affects negatively the microarchitectural processor components that rely on locality. Later, Shahrad et al [58] characterized the entire production FaaS workload of Azure Functions; however that study did not analyze results at the microarchitectural level. Finally, Asheim et al. [3] analyzed further analyze the performance sensitivity of serverless workloads to microarchitectural state thrashing due to interleaved execution.

Microarchitectural Optimizations for Serverless Computing. Schall et al. [52] highlighted the positive impact of executing serverless functions with warm microarchitectural state and proposed an instruction prefetcher that is specifically designed for reducing the start-up latency of warm function instances. Later, the authors [53] expanded the scope of their approach by proposing a restoration mechanism for the entire front-end of the processor pipeline that includes instructions, the branch target buffer (BTB) and the conditional branch predictor (CBP). Wang et al. [69] showed that the memory management subsystem plays also a critical role in the performance of serverless workloads, and proposed architectural support for memory management that is specifically tailored to the needs of serverless computing. Finally, Antoniou et al. [2] proposed a new microarchitectural power management technique for optimizing the cold-start latency for latency-critical applications based on microservices.

6. CONCLUSIONS & FUTURE WORK

In this thesis we focused on bridging the gap between serverless computing, that is an emerging cloud computing paradigm, and RISC-V, that is an open-source RISC-V ISA that has received a lot of interest recently. To bridge that gap, we ported the vSwarm and vSwarm-u serverless benchmarking infrastructure to the RISC-V ISA. It was a quite demanding procedure as the path that we had to follow was not always paved. After a lot of effort, we managed to: (i) create our RISC-V development environment based on QEMU, (ii) port several workloads from vSwarm to RISC-V addressing various challenges due to the immaturity of the RISC-V software ecosystem, and (iii) port the vSwarm-u framework to enable the execution of the workloads in the gem5 simulator. In addition, we enhanced the existing x86 infrastructure to enable fair comparison between RISC-V and x86 CPUs through the gem5 simulator with minimal divergence in the configuration of the simulated platforms.

Our evaluation using a RISC-V platform with the gem5 simulator shows the important performance trade-off between cold and warm execution of function instances. In addition, our preliminary results that compare the RISC-V and x86 software stacks in a near identical setup on CPUs with similar microarchitectural characteristics show that the execution of functions in the RISC-V simulated platform are faster than that in the x86 simulated platform. The main reason for this performance difference is the fact that the execution of the functions in the RISC-V platform resulted in significantly fewer executed instructions than the execution of the functions in the x86 platform. Taking into consideration all the above, we conclude that our work might provide confidence for more research to be devoted at RISCV ISA as well as Serverless Computing. We hope that our contributions are a solid foundation for this research to grow.

Our work can be extended in several ways. First of all, we plan to port the rest of the vSwarm application to RISC-V and enable their execution in the gem5 simulator. We also plan to port MongoDB to RISC-V to enable the seamless porting of the Hotel application avoiding any code modification to the application itself. Regarding the software stack, we plan to update our infrastructure by using newer components, such as Ubuntu v24.04 and Linux v6. In terms of experimentation, we plan to run the ported serverless workloads and measure their performance on real RISC-V platforms. Another interesting direction that we plan to follow is to perform a detailed design space exploration with respect to various microarchitectural characteristics, such as caches, branch predictors, and prefetchers, using the gem5 simulator. Finally, improvements in the gem5 simulator will foster further research on the topic of serverless computing. These improvements could target on making stable the execution with the KVM CPU model and extending the current infrastructure to support the KVM CPU model for the RISC-V as well.

ABBREVIATIONS - ACRONYMS

RISC-V	Fifth version of Reduced Instruction Set Computing
CISC	Complex Instruction Set Computer
ISA	Instruction Set Architecture
SQL	Structured Query Language
NoSQL	Not Only Structured Query Language
gRPC	generic Remote Procedure Calls
KVM	Kernel-based Virtual Machine
VM	Virtual Machine
O3	Out Of Order
CPU	Central processing unit
DB	DataBase
OS	Operating System
L1I	Level 1 Instruction cache
L1D	Level 1 Data cache
L2	L2 cache

BIBLIOGRAPHY

[1] Amazon AWS Lambda. https://aws.amazon.com/lambda/.

[2] Georgia Antoniou, Davide Bartolini, Haris Volos, Marios Kleanthous, Zhe Wang, Kleovoulos Kalaitzidis, Tom Rollet, Ziwei Li, Onur Mutlu, Yiannakis Sazeides, and Jawad Haj Yahya. Agile c-states: A core c-state architecture for latency critical applications optimizing both transition and cold-start latency. *ACM Trans. Archit. Code Optim.*, jul 2024. Just Accepted.

[3] Truls Asheim, Tanvir Ahmed Khan, Baris Kasicki, and Rakesh Kumar. Impact of microarchitectural state reuse on serverless functions. In *Proceedings of the Eighth International Workshop on Serverless Computing*, WoSC '22, page 7–12, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Atomic and KVM core info. https://cirosantilli.com/linux-kernel-module-cheat/#gem5-cpu-types.

[5] Microsoft Azure Functions. https://azure.microsoft.com/engb/services/functions.

[6] BeFaaS. https://github.com/Be-FaaS/BeFaaS-framework.

[7] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. Isa wars: Understanding the relevance of isa being risc or cisc to performance, power, and energy on modern architectures. *ACM Trans. Comput. Syst.*, 33(1), March 2015.

[8] What is Apache Cassandra. https://cassandra.apache.org/_/index.html.

[9] Kernel compatibility for running Docker verification script.

[10] Google Cloud. Configuring Warmup Requests to Improve Performance. https://cloud.google.com/appengine/docs/standard/python/configuring-warmup-requests.

[11] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: a serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Why the fs.py got depracated. https://www.mail-archive.com/gem5-users@gem5.org/msg21888.html.

[13] Install Docker Engine and Client in Riscv Ubuntu. https://forum.rvspace.org/t/docker-engine-and-docker-cli-on-riscv64/267.

[14] What is docker. https://www.docker.com/.

[15] The FaaSdom benchmark suite. https://github.com/faas-benchmarking/faasdom.

[16] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 275–294. USENIX Association, July 2021.

[17] Function Bench: A Suite of Workloads for Serverless Cloud Function Service. https://github.com/ddps-lab/serverless-faas-workbench.

[18] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 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, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[19] What is gem5. https://www.gem5.org/about/.

[20] Linux kernel configurations provided by gem5 resources. https://github.com/gem5/gem5-resources/tree/develop/src/linux-kernel/linux-configs.

[21] gem5 System Modes. https://www.gem5.org/documentation/learning_gem5/part1/simple_config/.

[22] https://cloud.google.com/blog/products/gcp/google-cloud-platform-your-next-home-in-the-cloud/.

[23] Google Cloud Functions. https://cloud.google.com/functions/.

[24] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach. Befaas: An application-centric benchmarking framework for faas platforms, 2021.

[25] Solution to grpc module loading. https://github.com/grpc/grpc/issues/24249#issuecomment-972555615.

[26] What is grpc. https://grpc.io/#:~:text=gRPC%20is%20a%20modern%20open,tracing%2C%20health%20checking%20and%20authentication.

[27] IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[28] What is serverless. https://www.ibm.com/topics/serverless.

[29] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, jun 2015.

[30] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.

[31] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 477, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020.

[33] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery.

[34] What is MariaDb. https://mariadb.com/.

[35] Mongo to riscv attempt. https://forum.sophgo.com/t/risc-v-public-beta-platform-released-database-ada273.

[36] What is MongoDB. https://www.mongodb.com/company/what-is-mongodb.

[37] Natheesan Profile in Docker Hub. https://hub.docker.com/u/natheesan.

[38] Goncalo Neves. Keeping Functions Warm – How To Fix AWS Lambda Cold Start Issues. https://serverless.com/blog/keep-your-lambdas-warm.

[39] O3 core info. https://www.gem5.org/documentation/general_docs/cpu_models/O3CPU.

[40] Open SBI, the Runtime Component of the RISC-V bootflow. https://github.com/riscv-software-src/opensbi.

[41] What Is pip in python. https://pypi.org/project/pip/.

[42] Timothy Prickett Morgan. AWS Adopts Arm V2 Cores For Expansive Graviton4 Server CPU. https://www.nextplatform.com/2023/11/28/aws-adopts-arm-v2-cores-for-expansive-graviton4-server-cpu/.

[43] What is qemu. https://www.qemu.org/docs/master/about/index.html.

[44] What is Redis. https://redis.io/.

[45] Aws release announcement. https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/.

[46] Risc-v international standard management. https://riscv.org/technical/specifications/.

[47] What is Risc-v. https://www.synopsys.com/glossary/what-is-risc-v.html.

[48] Risc-v 10 billion cores. https://riscv.org/announcements/2022/12/risc-v-sees-significant-growth-and-technical-progress-in-2022-with-billions-of-risc-v-cores-in-

[49] RISCV Database Compliance. https://riscv.org/blog/2023/08/risc-v-public-beta-platform-release-%C2%B7-database-adaptation-evaluation-on-risc-v-server/.

[50] Boot Riscv ubuntu with Qemu. https://wiki.ubuntu.com/RISC-V/QEMU.

[51] Scaleway launches its RISC-V servers in the cloud. https://www.scaleway.com/en/news/scaleway-launches-its-risc-v-servers-in-the-cloud-a-world-first-and-a-firm-commitment-to-techno

[52] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 757–770, New York, NY, USA, 2022. Association for Computing Machinery.

[53] David Schall, Andreas Sandberg, and Boris Grot. Warming up a cold front-end with ignite. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 254–267, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Docker search for RISC-V images. https://hub.docker.com/search?q=go&architecture=riscv64.

[55] SeBS: Serverless Benchmark Suite. https://github.com/spcl/serverless-benchmarks.

[56] ServerlessBench: A benchmark suite with serverless workloads. https://github.com/SJTU-IPADS/ServerlessBench.

[57] 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, page 1063–1075, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[59] SiFive Processors. https://www.sifive.com/risc-v-core-ip.

[60] Roberto Starc, Tom Kuchler, Michael Giardino, and Ana Klimovic. Serverless? risc more! In *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies*, SESAME '24, page 15–24, New York, NY, USA, 2024. Association for Computing Machinery.

[61] Roberto Patrick Starc. Exploring the microarchitectural implications of serverless workloads using risc-v. Master thesis, ETH Zurich, Zurich, 2023-04. D-INFK Master's Thesis Nr. 439.

[62] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 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, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[63] Ventan High Performance RISC-V CPUs and System IP. https://www.ventanamicro.com/technology/.

[64] vHive: Open Source Framework for Serverless Experimentation. https://github.com/vhive-serverless/vHive.

[65] What is vSwarm. https://github.com/vhive-serverless/vSwarm/blob/main/README.md.

[66] vSwarm: Serverless Benchmarking Suite. https://github.com/vhive-serverless/vSwarm.

[67] vSwarm-u: Microarchitectural Research for Serverless. https://github.com/vhive-serverless/vSwarm-u.

[68] vSwarm-u: Simulation Methodology. https://vhive-serverless.github.io/vSwarm-u/simulation/basics/.

[69] Ziqi Wang, Kaiyang Zhao, Pei Li, Andrew Jacob, Michael Kozuch, Todd Mowry, and Dimitrios Skarlatos. Memento: Architectural support for ephemeral memory management in serverless environments. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 122–136, New York, NY, USA, 2023. Association for Computing Machinery.

[70] Kaifeng Xu, Georgios Tziantzioulis, and David Wentzlaff. Mindpalace: A framework for studying microarchitecture design of function-as-a-service. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 313–315, 2024.

[71] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[72] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural implications of event-driven server-side web applications. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 762–774, 2015.