Benchmarking Support for RISC-V CPUs in Serverless Computing

Georgios Pournaras
University of Athens, Greece
gpournaras@di.uoa.gr

Vasileios Karakostas
University of Athens, Greece
vkarakos@di.uoa.gr

George Papadimitriou*
University of Patras, Greece
gpapad@upatras.gr

Dimitris Gizopoulos
University of Athens, Greece
dgizop@di.uoa.gr

***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 of interest and the first RISC-V systems have already started to appear in the server market for datacenters. The combination of these computing trends necessitates the performance assessment of the impact of the RISC-V ISA and relevant processor implementations when executing serverless workloads. However, currently there is no benchmarking support for systematically evaluating serverless workloads on RISC-V systems. In this paper we bridge this gap in benchmarking support across the layers of the computing stack, from the application to the microarchitecture. We port the vSwarm serverless benchmark suite to the RISC-V ISA and enable the execution of the workloads in both real and simulated RISC-V platforms using the gem5 microarchitectural simulator. To demonstrate the usefulness of the infrastructure, we quantify the performance trade-off of cold vs. warm execution on a simulated RISC-V system and validate the results against a real platform. Overall, our enhanced benchmarking support creates new opportunities for further experimentation with serverless workloads on RISC-V systems, enabling the analysis and optimization of their performance across the computing stack.**

I. Introduction

Serverless computing has gained a lot of interest recently, as it provides important benefits for both users and cloud service providers. The 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 [16], [20]. However, the performance challenges of serverless computing require careful consideration of various parameters, including the ISA and the processor microarchitecture. In parallel, the interest in RISC-V has increased significantly, thanks to its open-source nature, that fosters processing sovereignty, and the growing support from system vendors. While CISC-based (primarily x86) systems have dominated the server market during the past two decades, cloud service providers have started to adopt RISC-based (Arm) systems and more recently to introduce RISC-V systems.

Despite the growing interest in serverless computing and RISC-V, there is currently no benchmarking support that combines these two technologies together and that allows systematic analysis and evaluation of the architectural and microarchitectural parameters of RISC-V systems in serverless computing. Having such benchmarking support would be crucial to quantify the potential of RISC-V processors for serverless computing – particularly with respect to well-established ISAs and processor designs – towards the successful adaptation of RISC-V. In addition, 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 [4], [15] and have proposed novel mechanisms for improving performance [13], [14], [19]. Such serverless benchmarking support would enable the evaluation of the impact of various microarchitectural designs for RISC-V systems.

While there are multiple serverless benchmark suites [2], [5], [9], [11], [12], [21], they lack support for RISC-V. Only a few prior works [6], [10], [17], [18] have focused on serverless workloads on RISC-V systems. However, those works [6], [17], [18] have only considered a few individual serverless workloads without porting any existing benchmark suite, or they [10], [17], [18] lack support for running serverless workloads in microarchitectural simulators for RISC-V.

Our goal is to bridge the gap in benchmarking support between serverless computing and the RISC-V ISA. In this paper we enhance the vSwarm infrastructure [20] that supports serverless benchmarking along two axes[1]. First, we port the vSwarm [2] serverless benchmarks to the RISC-V ISA. This enables experimentation on real RISC-V platforms. Second, we enhance the vSwarm-u [3] benchmarking methodology to support the evaluation of the serverless benchmarks on RISC-V CPUs using the gem5 simulator [1]. This enables detailed evaluation of the impact of various microarchitectural components on the execution of serverless workloads, as well as direct comparison of different ISAs for this emerging cloud computing paradigm.

To demonstrate the usefulness of our enhanced simulation benchmarking infrastructure, we quantify the performance trade-off of cold vs. warm execution on a simulated RISC-V out-of-order multicore system using gem5. Our results highlight the impact of the function and the execution environment (i.e., language/runtime) on performance and cache behavior. To validate our porting effort, we execute the ported serverless workloads on a real multicore RISC-V hardware platform (HiFive Premier P550). Our results show that the performance trade-off of cold vs. warm execution for the simulated system is similar to that observed in the real system across functions.

*This work was done while the author was at University of Athens.

[1]Code available at https://github.com/vitamin-v-software/vSwarm-RISCV.

TABLE I
THE VSWARM FUNCTIONS THAT ARE USED IN THIS WORK.

Application	Functions	Runtime
	AES, Auth, Fibonacci	Go/Python/NodeJS
Online Shop	Product Catalog, Shipping Recommendation, Email Currency, Payment	Go Python NodeJS
Hotel	Geo, Profile Rate, Recommendation Reservation, User	Go

II. RISC-V SERVERLESS BENCHMARKING SUPPORT

A. Selecting the Serverless Benchmark Suite

Several representative serverless benchmark suites have been proposed [5], [9], [11], [12], [20], [21]. In this work we use the vSwarm serverless benchmark suite [2], [20] because it comes with a variety of workloads, ranging from simple functions to multi-function applications, it supports multiple languages and runtimes, and it supports both the x86 and Arm ISAs. In addition, vSwarm allows experimentation not only on real systems but also on simulated systems through vSwarm-u [3], i.e., a set of tools, configurations, and scripts to run serverless functions in gem5 [1].

In this paper we focus on the functions that the original vSwarm infrastructure supports running in gem5 with the x86 and Arm ISAs with Docker containers (Table I). These functions can be classified into three categories: (i) The AES, Auth, and Fibonacci functions that are all implemented in Go, NodeJS, and Python. (ii) Functions from the Online shop application that is based on Google's Online Boutique [8]. The Online shop functions are implemented in Go, NodeJS, or Python. Most functions are standalone except for the recommendation function that also uses the product catalog function. (iii) Functions from the Hotel application that is based on DeathStarBench's Hotel Reservation Application [7]. The Hotel functions are implemented in Go and communicate with a database container (i.e., MongoDB). The profile, rate, and reservation functions also communicate with a lightweight caching container (i.e., Memcached).

B. Porting the vSwarm Functions to the RISC-V ISA

1) AES, Auth, and Fibonacci: Regarding the Go and NodeJS functions, we identified the corresponding base images for RISC-V, replaced the corresponding part in the function's Dockerfile, and built the containers. Then we compiled the client that performs the requests, and managed to successfully execute the functions. Regarding the Python functions, we faced problems with the gRPC module that is necessary for their execution. We solved this issue by specifying missing libraries via `LD_PRELOAD`. We applied the same technique on similar modules.

2) Online Shop application: Porting most of the Online shop functions was straightforward, as we followed practically the same steps as for AES, Auth, and Fibonacci.

3) Hotel application: The functions of the Hotel application are written in Go, so porting them to RISC-V should be straightforward. However, all Hotel functions also communicate with a MongoDB database container, which is initialized at the beginning of the application, and then is used during the execution of the functions. Hence, porting the Hotel application functions required porting the database container as well. Unfortunately, MongoDB has not been ported yet to the RISC-V ISA. We decided to replace MongoDB with another NoSQL database that should be available in the RISC-V ISA. After considering Apache Cassandra and Redis that both come with RISC-V support, we decided to use Cassandra because Redis is rarely used as main database and its most common usage is caching. We modified the Hotel application to use Cassandra in the database container and managed to successfully run the Hotel functions. We quantify the impact of this change in Section III.

C. Running the Functions

1) Simulated system: The original vSwarm-u framework uses QEMU to create the disk image and install all the dependencies, and then it uses gem5 to perform simulations. We follow the same approach and use a RISC-V virtual machine with Ubuntu 22.04.5 LTS server for SiFive HiFive Unmatched platforms to create the disk image. We manually install Docker and other necessary packages, such as containerd and rootlesskit, and deactivate unnecessary services to speed up booting Linux in gem5. We create a custom Linux kernel configuration file that allows the execution of Docker containers in gem5 and ensure that all the necessary kernel modules are statically built because gem5 does not allow the dynamic loading of kernel modules. We generate gem5 checkpoints for each function after initiating the container and right before the first client request using the AtomicCPU model. Then, we use the O3CPU model and collect statistics regarding the 1st and the 10th request, i.e., *cold* and *warm execution* of each function. Finally, we create our own simulation configuration script using the gem5 standard library.

2) Real system: The functions can also be executed on real RISC-V platforms. We create scripts that automate the execution of the functions and collection of various statistics through the Linux perf utility for both cold and warm executions, as well as scripts for parsing and plotting the results.

III. EVALUATION

A. Experimental Methodology

We use gem5 v24.1.0.3 in full-system mode to model a multicore system with the configuration parameters listed in Table II. The simulated system runs Linux kernel v5.15.59, Docker v25.0.0, Go 1.21.6, Python 3.10.16, and NodeJS v18.16.1. Furthermore, we use a SiFive HiFive Premier P550 hardware platform that consists of four out-of-order cores. The core count and frequency, and the organization and sizes of the cache hierarchy, ROB, LSQs, and physical register files are the same as those used in the simulated system. Regarding the software stack, the HiFive P550 system runs Ubuntu 24.04.1

TABLE II
CONFIGURATION PARAMETERS FOR THE RISC-V SIMULATED SYSTEM.

Cores	4 out-of-order cores @ 1.4GHz
L1 I Cache	private, 32KB, 4-way set assoc.
L1 D Cache	private, 32KB, 4-way set assoc.
L2 Cache	private, 256KB, 8-way set assoc.
L3 Cache	shared, 4MB, 16-way set assoc.
DRAM	2GB, DIMM_DDR5_6400
I TLB	private, 64 entries
D TLB	private, 64 entries
ROB	96 entries
LSQs	20 Load entries + 16 Store entries
Registers	128 Int + 119 Float

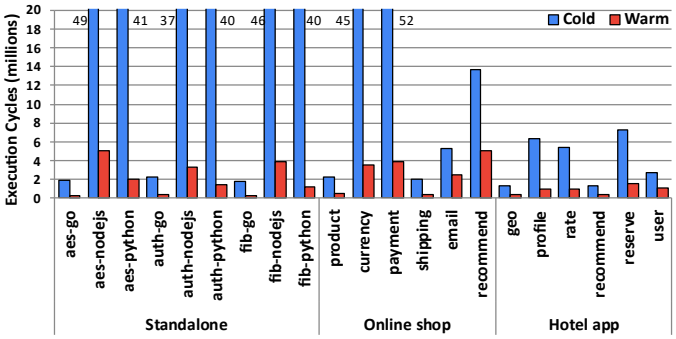

Fig. 1. Number of execution cycles on the RISC-V simulated system.

LTS, Linux kernel 6.6.21-9-premier, and Docker 27.5.1. We run the same function containers as in the simulated system. To accurately measure function performance, we pin containers to cores. Finally, to reduce the performance variability in the HiFive platform, we isolate the core on which the function runs, perform each experiment 10 times, and report the median.

B. Results

1) RISC-V Simulated System: Figure 1 shows the number of execution cycles for the cold and warm execution of all functions in the RISC-V simulated system. In general, we observe that the execution time for the cold and warm executions differs across functions and depends on the language/runtime. We also observe that the performance trade-off between cold and warm execution is significant, ranging from 2.1× to 32.7×. Next we analyze the results for each class of functions.

AES, Auth, and Fibonacci. We observe that the Go versions of the functions are the fastest for both cold and warm executions. The Python versions for AES and Fibonacci follow next in terms of performance, while the NodeJS versions of the functions are the slowest ones. These results show the clear performance benefit that Go offers as a compiled language, compared to NodeJS that uses JIT compilation and Python that uses interpretation.

Online Shop. Similarly, we observe that the Go functions (i.e., the product catalog and the shipping services) are the fastest in both cold and warm executions, then the Python functions (i.e., the email and recommendation services) follow, and the NodeJS functions (i.e., the currency and payment services) are the slowest. However, the warm execution of the recommendation function is slower than that of the NodeJS functions because it also invokes the product catalog function.

Hotel App. We observe that the reservation, rate, and profile functions are slower than the geo, recommendation, and user functions for the cold execution. These functions also communicate with the Memcached container and then, if needed, they perform requests to the Cassandra container. After receiving a reply from the Cassandra container, the Memcached container is populated so that it can be used later. On the other hand, these functions in warm execution take advantage of the Memcached container and their execution time is comparable to that of the rest of the Hotel functions.

Cache Analysis. Figure 2 breaks down the cache misses across the cache hierarchy for the cold and warm executions of all functions. The results for the warm execution are normalized to those for the cold execution. We observe that the contribution of L1 instruction cache misses is higher than that of L1 data cache misses for most functions. Only the Python functions exhibit more L1 data cache misses than L1 instruction cache misses. We also observe that the contribution of L2 misses is significant, while the contribution of L3 misses with respect to the total number of cache misses is rather low for most of the functions except for the cold execution of the Go functions. Finally, we observe that the L1 data cache, L1 instruction cache, L2 cache, and L3 cache misses are reduced by 73.9%, 70.8%, 68.9%, and 95.6% on average for the warm execution with respect to the cold execution.

2) Validation of simulation results: We run the functions on the HiFive P550 platform using perf. We observe that gem5 underestimates the execution time compared to the real platform by 58% and 183% on average for the cold and warm execution. Although we use in gem5 the same parameters for various microarchitectural components as in the HiFive P550 platform, there are differences in the core simulation that affect performance. Still, we observe that the two systems exhibit identical trends in the performance trade-off between cold and warm invocations across functions (Figure 3). We also observe that the trade-off is more pronounced in the simulated system (5.9× on average) than in the HiFive P550 platform (3.3× on average). These results show that the enhanced benchmarking support can be used in both real and simulated systems and strengthen our confidence in using it for evaluating RISC-V systems with gem5.

3) Impact of the database change in Hotel: We quantify the impact of changing the database container from MongoDB to Cassandra on a real x86 system with an Intel Xeon E5-1620 v4 processor, Ubuntu 24.04.2 LTS with Linux 6.8.0-62-generic, and Docker 27.5.1. We observe that Cassandra performs faster than MongoDB for the cold execution by 19% on average, while it performs slower than MongoDB for the warm execution by 1% on average. However, we observe that booting the Cassandra container takes 5× more time than booting the

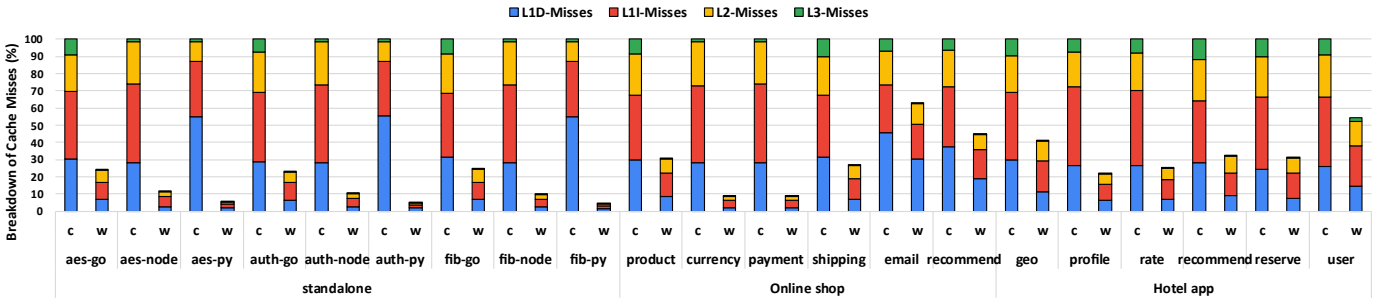

Fig. 2. Breakdown of cache misses for the cold and warm execution of the functions on the RISC-V simulated system.

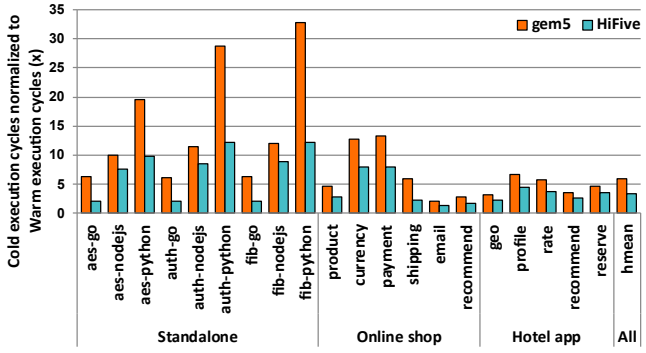

Fig. 3. Cold vs warm execution performance trade-off (speedup) for the RISC-V simulated system with gem5 and for the HiFive P550 platform.

MongoDB container. Note that in this paper we focus on the latency of the function request, without considering the boot time of the database or the caching containers for the Hotel functions, assuming that such containers have different deactivation criteria, and will be active for a longer period than regular function containers. We conclude that our porting approach enables the execution of the Hotel application in the RISC-V ISA without affecting significantly the performance of the application functions.

IV. Conclusions

In this paper we bridge the gap between serverless computing and RISC-V. We believe that our contributions will motivate and help the research community explore and evaluate optimizations in serverless computing and RISC-V across the computing stack, from the application and the runtime layers, to the operating system, architecture, and microarchitecture levels, as well as compare the performance and energy-efficiency of the RISC-V ISA with respect to well-established ISAs, such as x86 and Arm.

Acknowledgments

This work was supported by the European Union's Horizon Europe research and innovation programme under grant agreement No 101093062 (Vitamin-V). Views and opinions expressed are, however, those of the authors only and do not necessarily reflect those of the European Union or the HaDEA. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] "The gem5 simulator," https://www.gem5.org/about/.
[2] "vSwarm: Serverless Benchmarking Suite," https://github.com/vhive-serverless/vSwarm.
[3] "vSwarm-u: Microarchitectural Research for Serverless," https://github.com/vhive-serverless/vSwarm-u.
[4] T. Asheim, T. A. Khan, B. Kasicki, and R. Kumar, "Impact of microarchitectural state reuse on serverless functions," in *WoSC*, 2022, p. 7–12.
[5] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: a serverless benchmark suite for function-as-a-service computing," in *Middleware*, 2021, p. 64–78.
[6] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the PENGLAI enclave," in *USENIX OSDI*, 2021, pp. 275–294.
[7] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno *et al.*, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *ACM ASPLOS*, 2019, p. 3–18.
[8] "Microservices Demo: Google Online Boutique," https://github.com/GoogleCloudPlatform/microservices-demo.
[9] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, "BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms," in *IEEE IC2E*, 2021, pp. 1–8.
[10] T. Iliadis, N. C. Papadopoulos, K. Nikas, and D. Pnevmatikatos, "Vitamin-V: Serverless Cloud Computing Porting on RISC-V," in *SAMOS*, 2024, pp. 119–126.
[11] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *IEEE CLOUD*, 2019, pp. 502–504.
[12] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "FaaSdom: a benchmark suite for serverless computing," in *ACM DEBS*, 2020, p. 73–84.
[13] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: characterization and optimization," in *ISCA*, 2022, p. 757–770.
[14] D. Schall, A. Sandberg, and B. Grot, "Warming Up a Cold Front-End with Ignite," in *IEEE/ACM MICRO*, 2023, p. 254–267.
[15] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *IEEE/ACM MICRO*, 2019, p. 1063–1075.
[16] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC*, 2020, pp. 205–218.
[17] R. Starc, T. Kuchler, M. Giardino, and A. Klimovic, "Serverless? RISC more!" in *SESAME*, 2024, p. 15–24.
[18] R. P. Starc, "Exploring the Microarchitectural Implications of Serverless Workloads Using RISC-V," Master Thesis, ETH Zurich, 2023-04.
[19] J. Stojkovic, E. Choukse, E. Saurez, I. Goiri, and J. Torrellas, "Mosaic: Harnessing the Micro-Architectural Resources of Servers in Serverless Environments," in *IEEE/ACM MICRO*, 2024, pp. 1397–1412.
[20] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *ACM ASPLOS*, 2021, p. 559–572.
[21] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in *ACM SoCC*, 2020, p. 30–44.