WebAssembly as an Enabler for Next Generation Serverless Computing

Vojdan Kjorveziroski[1*] and Sonja Filiposka[1†]

[1*]Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Rudzer Boshkovikj 16, Skopje, 1000, North Macedonia.

*Corresponding author(s). E-mail(s): vojdan.kjorveziroski@finki.ukim.mk;
Contributing authors: sonja.filiposka@finki.ukim.mk;
†These authors contributed equally to this work.

Abstract

WebAssembly is a new binary instruction format and runtime environment capable of executing both client side and server side workloads. With its numerous advantages, including drastically reduced cold start times, efficiency, easy portability, and compatibility with the most popular programming languages today, it has the potential to revolutionize serverless computing. We evaluate the impact of WebAssembly in terms of serverless computing, building on top of existing research related to WebAssembmly in cloud and edge environments. To this end, we introduce a novel benchmarking suite comprised of 13 different functions, compatible with WebAssembly, and focusing on both microbenchmarking and real-world workloads. We also discuss possibilities of integrating WebAssembly runtimes with the application programming interfaces and command line interfaces of popular container runtimes, representing an initial step towards potential reuse of existing orchestration engines in the future, thus solving the open issue of WebAssembly workload scheduling. We evaluate the performance of such an integration by comparing the cold start delays and total execution times of three WebAssembly runtimes: WasmEdge, Wasmer, and Wasmtime to the performance of the containerd container runtime, using distroless and distro-oriented container images. Results show that WebAssembly runtimes show better results in 10 out of 13 tests, with Wasmtime being the fastest WebAssembly runtime among those evaluated. Container runtimes still offer better compute performance

Springer Nature 2021 LaTeX template

for complex workloads requiring larger execution times, in cases where cold start times are negligible compared to the total execution time.

Keywords: Serverless Computing, WebAssembly, Function as a Service, Internet of Things, Performance Evaluation, Benchmarks

1 Introduction

The introduction of cloud computing [1] and the emergence of the associated as a service products later offered by infrastructure providers have forever revolutionized the computing landscape. The elasticity, ease of use, pervasiveness, and cost efficiency of the cloud have given rise to three fundamental pillars, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), each offering various abstraction layers to the end-users, with the end-goal of simplifying deployments and management of complex infrastructures [2]. Recently, a number of new * as a service products have emerged [3], either trying to replace the already established ones, or gradually refining them, further improving the experience of end-users and developers alike. Function as a Service or more general, serverless computing [4] is one such offering whose aim is to build on top of the success of cloud computing and in some cases even extend its core concepts.

The term serverless computing encompasses both Function as a Service (FaaS) and Backend as a Service (BaaS) [5], allowing developers to focus exclusively on the business logic of their code, while disregarding many aspects such as: scalability, server configuration, application deployment, or infrastructure maintenance. Of course, servers are still necessary for the workload execution, but all details in regards to configuration and management are abstracted away by the serverless platform. The benefits of this approach have proven so popular among developers that the majority of cloud providers today offer their own serverless computing products [6], in addition to many open-source and self-hostable solutions [7, 8]. Serverless computing is applicable to many different areas, ranging from web workers [9], to event-driven workloads [10, 11], to augmented reality and virtual reality [12].

Serverless computing together with its associated benefits also has a large potential to impact the Internet of Things (IoT) [13] landscape. The virtually unlimited scalability of serverless functions, coupled with efficient use of resources together with flexible and granular billing policies are well-suited for the event-driven nature of IoT workloads, and the enormous amount of existing and future IoT devices. However, before realizing the true potential of serverless computing in regards to IoT and adjacent fields, there are a number of open issues which need to be resolved [14, 15]. Many IoT workloads are time-sensitive which makes the deployment of serverless functions to the cloud unfeasible as a result of the increased network latency. A potential solution to this problem is to move the serverless infrastructure closer to the data source,

thus reducing the network latency, while allowing the response to be delivered faster due to the initial preprocessing of the data done at the edge of the network. Furthermore, execution performance both in the cloud and at the edge also plays an important role in the total response time, especially when taking into account additional security measures whose aim is to improve function isolation in multi-tenant environments [16]. The choice of the runtime environment where the serverless function is executed plays an important role in its execution performance. Today, the most popular execution environments for serverless functions are micro virtual machines and containers. While they offer favorable performance in terms of execution speed once started, they both suffer from large delays during their initial startup, a delay also popularly known as the cold start time [17].

Numerous approaches for dealing with the unfavorable cold start times have been proposed both from academia and industry, such as keeping a pool of pre-warmed function instances running and ready to serve incoming requests [18], or employing intelligent scheduling algorithms that can predict spikes in incoming requests and preemptively increase the number of function instances [19]. While these solutions alleviate the cold start problem to some extent, they disregard the main benefits of serverless computing while doing so – scalability to zero replicas and billing only for the active instead of idling resources.

One potential solution to the pervasive problem of large cold start times in serverless computing is the introduction of more lightweight runtime environments, compared to the existing ones being employed today [5]. WebAssembly (WASM) is one such novel runtime environment, which if employed for the execution of serverless functions, has the potential to drastically reduce the impact of cold start times [20]. The reduction of the start up delay would also enable each runtime environment to be used only once, for the duration of a single function execution, further improving security and isolation, aspects of paramount importance in multi-tenant cloud and edge environments [21].

The goal of this paper is to explore ways in which WebAssembly can be integrated with existing and more mature runtime technologies for the execution of serverless functions, and to compare the cold start performance results of various WebAssembly runtimes between themselves, as well as with existing container runtimes.

The main contributions of this work are:

- Describe a strategy for integrating WebAssembly runtimes with container runtimes, allowing serverless functions to be executed in the most appropriate runtime environment depending on their requirements.
- Create and open-source a suite of WebAssembly serverless benchmarking functions comprised of micro benchmarks and real-world workloads with the aim of evaluating the execution and cold start performance of existing and future WebAssembly runtimes.
- Compare how cold start times impact execution performance of identical workloads when they are run in a WebAssembly environment integrated with the command line interface of an existing container runtime versus

natively executed in a container without employing WebAssembly. To the best of our knowledge, this is the first time that the execution performance of WebAssembly runtimes has been evaluated in regards to their integration with existing container runtimes which act as wrappers around their application programming interfaces (APIs).
- Make all results publicly available, facilitating future reuse or extension.

The rest of this paper is organized as follows: in Section 2, we present background information on WebAssembly and its possibilities for server side execution, as well as discuss existing related work to the topic of serverless functions execution in WebAssembly runtimes. In Section 3, we describe in details the employed testing methodology, including possibilities for integrating WebAssembly with existing container runtimes, and the process used for deriving the WebAssembly benchmarking suite. We then proceed with Section 4, where we present the obtained performance results for each of the 3 tested WebAssembly runtimes, as well as the two different strategies for execution in a traditional containerized environment. Section 5 offers a comparative analysis of the results and the different execution strategies, reflecting on the statistical significance of the associated differences in execution performance. We conclude the paper with Section 6, summarizing the work, outlining remaining issues and future plans.

2 Background Information and Related Work

WebAssembly is an emerging runtime environment which only recently started being applied to the area of server side workload execution. To get a better understanding of the underlying technology and its benefits, we provide a brief explanation of the most important concepts in subsection 2.1, before continuing with the discussion of the state of the art literature related to the problem of WebAssembly performance evaluation and its applicability to serverless computing in subsection 2.2.

2.1 WebAssembly as a Viable Server Side Technology

WebAssembly, at its core, is a binary instruction format which can be executed in a stack-based virtual machine. Since the completion of the WASM minimum viable product (MVP) in 2017 it has attracted a noticeable interest from a number of popular companies which contribute to its code-base to this day. At present, the standard is governed by the World Wide Web Consortium (W3C) [22].

Many high-level programming languages already support WebAssembly as their compilation target, including: C, C++, Rust, Go, and others [23]. This ability to compile arbitrary code to WASM and run it in an isolated environment is a major factor behind the popularity of WebAssembly. The most widely used internet browsers today also support WebAssembly, allowing them to execute legacy software and software that was not initially written

for the web in a sandbox, at the client side, providing integration options with other modern client side technologies [24, 25].

Standalone WebAssembly runtimes, decoupled from web browsers, have also been introduced by the wider community, transforming WASM from a client side technology, to a server side technology as well [26]. The main benefits of using this new instruction format for independent workload execution in multi-tenant environments are numerous, including: near native executing speed while providing sandboxing and isolation, support for multiple high-level programming languages, wide ranging cross platform hardware compatibility, modest binary file sizes [27].

Even though, theoretically, it should be possible to compile any software written in a programming language whose toolchain supports WebAssembly, in reality that is not always the case. Since the WebAssembly stack based virtual machine adds an additional abstraction layer between the operating system and the code being executed, many operations which are taken for granted in high-level programming languages need explicit support from the underlying runtime, such as file input/output (I/O) operations [28], access to the network [29], and threading [30]. To alleviate these problems and to provide a framework for implementing future platform specific constructs, the WebAssembly System Interface (WASI) was introduced [31], which provides a POSIX like interface for interaction with the resources of the underlying operating system. As a result of the work done on WASI so far, support for the previously mentioned features is already standardized across all runtimes which support the system interface, or is in the process of standardization.

Taking into consideration all of the current benefits of WebAssembly, especially those centered around code portability and isolation, is what makes WASM a viable choice for execution of serverless functions at different locations in the network, both in the cloud and at the edge [32, 33]. The continued development and introduction of new features which are of direct benefit to serverless computing is also encouraging. One such feature is the WebAssembly component model [34], allowing commonly used components to be shared across different WASM modules, instead of reimplementing and duplicating the common functionality across all of them. This is especially beneficial to serverless functions, since it allows the functions to be further slimmed down both in terms of code complexity and file size, externalizing common boilerplate code and providing standardized modules for interacting with other systems, such as handling inbound requests or communicating with a third-party backend.

WebAssembly today can certainly be used for server side execution of workloads, but a crucial piece of the puzzle is missing, workload orchestration. The question of workload orchestration for WebAssembly modules is an open research question today. In our vision for serverless computing, WebAssembly plays an important role, but it should not completely replace the existing serverless runtime environments widely used today. Instead, effort should be made to provide a seamless integration between both WebAssembly, containerization, and virtualization, leveraging the best of each solution depending on

the nature of the function being executed. By providing a common interface for deployment of WebAssembly modules, containers, and micro virtual machines, it would also be possible to reuse the existing orchestration tools which are already tried and tested, instead of implementing new ones from scratch, specific only to WebAssembly.

2.2 Related Work

The cold start problem has been identified as a potential issue for serverless workloads early in the development of this new paradigm [4]. To overcome it, a number of solutions have been proposed and evaluated, such as prematurely starting a serverless function shortly before a request is even received, keeping a pool of serverless function instances ready at all times anticipating any new incoming requests, temporarily pausing the execution environment instead of completely tearing it down after each execution, or reusing the same runtime environment for execution of unrelated subsequent function executions.

The first strategy of premature function scheduling is practiced by both [18, 19], leveraging machine learning to implement intelligent scheduling algorithms, with the aim of eliminating the cold start latencies in two popular open-source serverless frameworks. The resulting schedulers are capable of predicting the invocation rate of serverless functions in the cluster, which allows them to prematurely launch a corresponding number of containers in anticipation of the incoming requests. The authors of the Pigeon framework [35] take a similar approach, but in their case the resulting container scheduler is also capable of pre-warming container pools with varying hardware resources such as the amount of allocated memory and processor cycles, offering them flexibility in terms of the potential set of functions that they can execute.

In order to strike a balance between the advantages of pre-warmed containers and their disadvantages such as increased resource consumption, the authors of [36] describe a mechanism for function pausing and checkpointing, allowing running instances to be temporarily suspended, but quickly resumed if the need for doing so arises, such as in the case of a new incoming request. This approach is also more cost effective, since less resources are utilized while the environment is paused in respect to when it is left running. Pelle et al. [37] also identified the balance of execution cost versus execution latency as an interesting problem, proposing a method of optimizing the used packages by a given function through artifact merging and grouping of related functions in common groups. Elgamal et al. [38] take the concept of function grouping even further through the introduction of function merging, allowing the reuse of the same execution environment for multiple functions. Their implemented prototype shows that this approach can save up to 57% of the associated runtime costs, at the expense of execution latency which is increased by 5 to 15%.

Perhaps the approach with the highest potential of mitigating the cold start problem for function instances which need to be rapidly scaled up and down is the introduction of a new runtime environment altogether, whose design would be based on efficient start up times from the beginning, instead of trying to

optimize this aspect after the fact. The authors of [39] describe one such early WebAssembly serverless platform which is capable of offering better cold start performance. Comparative analysis of performance differences in terms of the initial start up delay between traditional Docker containers and WebAssembly modules has also been done by Long et al. [40], with WASM being 10 times faster to start up than the same workload executed in a containerized environment. However, further work is required when it comes to the execution of more complicated functions which require higher processing performance, since WebAssembly currently offers reduced execution speed when compared to a native one. The details of this performance gap are discussed in more details in [27, 41–43], which report that measured slowdowns of WASM modules range between 1.5 to 10 times when compared to native execution.

Even though there are existing papers which compare the execution performance between different server side WASM runtime environments and native execution, in all cases the WASM modules are executed directly, via the API interface of the specific runtime. To the best of our knowledge, there is no work available which compares the cold start latency and execution speed of WebAssembly modules to that of containers in cases where the WebAssembly runtime is integrated with the interface of a container runtime. Through the use of a software shim, it is possible to reuse the same API of the container runtime to execute WebAssembly workloads, while avoiding the drawbacks of using containerization as the runtime environment. Our aim is to fill this gap by looking at the performance implications when different WebAssembly runtimes are integrated with a container runtime through such a software shim, and whether this added abstraction can provide a uniform interface for the deployment of serverless functions, no matter their underlying runtime environment.

3 Methodology

To realize the stated goal of assessing strategies for integration of WebAssembly runtimes with container runtimes, and evaluating the resulting performance of such an integration, we have selected 3 existing software solutions which allow server side execution of WASM modules. More details about the selection process of WASM runtimes as well as a description of their characteristics is provided in subsection 3.1.

In order to evaluate the performance of the selected WASM runtimes and to compare it to container based execution, we have created a benchmarking suite consisting of 13 different workloads, mixing real-world application scenarios and microbenchmarks focusing on a specific hardware aspect, such as I/O speed or CPU performance. Details about the selection of the benchmarking programs to be included in the suite, as well as the strategy applied for their execution are presented in subsections 3.2 and 3.3, respectively.

Finally, subsection 3.4 outlines possibilities in which the previously described methodology along with all associated results can be reused by other

researchers in the future, through the use of the accompanying material to this paper, which has been open-sourced and is publicly available[1].

3.1 Choosing Runtimes

There are more than a dozen WebAssembly runtimes currently undergoing active development and maintenance [44]. Some of them are aimed solely at the client side and provide integration options with existing web browsers so that WASM modules can be integrated in interactive web pages, developed using a mix of different technologies. Others are optimized for standalone, server side execution of WebAssembly workloads, and in some cases even implement features which are still not part of the core WebAssembly specification. During our selection process for WebAssembly runtimes to be evaluated, we were driven by the following criteria:

- The runtime can execute WebAssembly workloads in a standalone fashion, without requiring a web browser or other client side technologies such as invocation through JavaScript.
- Offers full support for the WebAssembly System Interface, WASI.
- The code base is open-sourced under a permissive license, allowing reuse and modifications.
- It is actively maintained, with at least a single new version release during the course of 2022.
- Can be integrated with containerd container runtime through the use of software shims, allowing WebAssembly workloads to be spawned through the containerd application programming interface (API) or command line interface (CLI).

After applying the above criteria to the currently available WebAssembly runtimes, we selected 3 different candidates: WasmEdge[2], Wasmtime[3], and Wasmer[4]. Additional general details about them is provided in Table 1, where we also compare their features to other popular WebAssembly runtimes that only partially satisfy the prerequisites.

It is important to note that all selected runtimes also satisfy the last requirement in the criteria and they can be integrated with containerd[5]. Such an integration is facilitated through the use of crun [45], an open container initiative (OCI) runtime and programming library which is supported by containerd. Crun has the capability of executing both OCI containers using containerization, or WebAssembly modules through the use of one of the 3 different WASM runtimes, depending on the labels which are passed during the instantiation of the workload using the containerd API or CLI.

To better analyze any potential performance differences in terms of the cold start times, we have also included two more execution strategies, in addition

[1] https://github.com/korvoj/wasm-serverless-benchmarks
[2] https://wasmedge.org/
[3] https://wasmtime.dev/
[4] https://wasmer.io/
[5] https://containerd.io/

Table 1 Comparison of the evaluated WebAssembly runtimes

Runtime	Lang.	Contr[1]	Stars[2]	Standalone[3]	WASI	R.[4]	Int.[5]	Start[6]
WasmEdge	C++	124	5.2k	✓	✓	6	✓	2019
Wasmtime	Rust	341	11.2k	✓	✓	29	✓	2016
Wasmer	Rust	131	14.2k	✓	✓	7	✓	2018
Wasm3	C	54	5.8k	✓	✓	0	✗	2019
Awsm	C	6	210	✓	✗	0	✗	2018
Wamr	C	106	3.4k	✓	✓	6	✗	2019
Wasmi	Rust	40	1k	✓	~[7]	13	✗	2018
Spiderm.	C++	N/A[9]	N/A[9]	✗	✗	7	✗	N/A[8]
V8	C++	461	20.7k	✗	✗	N/A[9]	✗	N/A[8]

Source: Official data from the respective repositories on the popular code hosting platform GitHub.

[1]Number of users who have contributed code to the project on GitHub.

[2]Number of users who have expressed interest in the project and have bookmarked it on GitHub.

[3]Capable of executing WASM modules independently (e.g., outside of a web browser).

[4]Number of new version releases in 2022.
[5]Can be integrated with containerd in its current form.
[6]Date of project start, date of first commit.
[7]Partial support.
[8]Not relevant since WASM support added after first release of the browser engine.
[9]Not available due to the project not being hosted on GitHub, or GitHub is not the primary hosting location.

to the three existing ones with the previously discussed and chosen WASM runtimes. Both of these additional strategies are using an unmodified version of containerd to execute the same application as in the case of the WASM runtimes, but in an OCI container, instead of first compiling it to a WASM module and executing with one of the WebAssembly runtimes. For these two containerized executions, we use the native binary of the given benchmark obtained through the use of the default compiler for the programming language and create two different container images, based on a distroless base image [46] and based on the commonly used debian:bullseye base image. Distroless based container images are smaller in size compared to their counterparts, since only the absolutely necessary libraries required for the execution of the given workload are shipped. On the other hand, we have also selected a more common base image, represented by debian:bullseye in our case, since such images based on an existing GNU/Linux distribution are very common among existing serverless platforms.

3.2 Benchmark Functions Selection

Evaluating the performance of the WebAssembly runtimes and the two different types of container images requires multiple serverless functions, either corresponding to a real-world scenario, or focusing explicitly on the underlying

Table 2 Information about the functions included in the benchmarking suite

Name	P. Lang.[1]	Description	Type[2]
audio-sine-wave	Rust	Render a 440Hz sine wave and store it as a mono .wav file	M
fuzzysearch	Rust	Search for the occurrence of a phrase in a text file	R
n-body	Rust	Model the orbits of Jovian planets	M
prime-numbers	Rust	Search for prime numbers among the first n numbers	M
whatlang	Rust	Determine the natural language of a given string	R
zip-compression	Rust	Compress multiple files in a single zip archive	R
aes	Go	Encrypt a given text using AES symmetrical encryption multiple times	R
checksum	Go	Calculate the MD5, SHA256, and SHA512 checksums of a given file	R
diskio	Go	Write and then read back a file containing 100000 lines of text	M
float-operation	Go	Calculate sin, cos and square root of an arbitrary number	M
imageprocessing	Go	Rotate an image and change its color palette	R
linear-equations	Go	Solve a system of linear equations	M
matmul	Go	Square matrix multiplication	M

[1]Source programming language of the given benchmarking function

[2]M – Microbenchmark; R – Real-world workload

hardware characteristics of the platform, as is the case when using microbenchmarks. As discussed previously, even though theoretically it should be possible to compile any existing application written in one of the supported high-level programming languages to WebAssembly, in practice a number of issues are encountered, especially in cases where the program is leveraging functionality not yet natively available in WASI, such as threading. These limitations make it impossible to reuse existing and well-known benchmarking suites in a WebAssembly context. To overcome this problem, we have created and verified a new benchmarking suite comprised of 13 different sequential applications, representing both the functionality of common serverless functions as well as more theoretical microbenchmarks. Five of the benchmarks included in the suite are written in the Rust programming language, while the remaining 6 in Go, both supporting WebAssembly as a compilation target. To the best of our knowledge, at the time of writing this paper, there is no comprehensive serverless benchmarking suite comprised both of real-world workload scenarios and microbenchmarks aimed at WebAssembly or one that can easily be compiled to WebAssembly, so that the performance of server side WebAssembly runtimes can more easily be evaluated. More details about each of the 13 benchmarks

is available above, in Table 2. All tests were adapted for execution in a WebAssembly environment from public code repositories [47, 48] or documentation of open-source programming libraries [49–55]. In cases where external libraries were used, care was taken to ensure that they did not make use of multi-threading features, or that such features were explicitly disabled during compile-time, thus ensuring compatibility with the current WebAssembly proposals.

3.3 Execution Strategy

The execution strategy with which the performance of the 13 different benchmarking functions was evaluated consisted of the following steps:

1. For each benchmarking function create 8 different OCI images, testing different aspects specific to each of the selected runtimes:
   - just in time (JIT) compilation during execution or interpretation with each of the three different WebAssembly runtimes (WasmEdge, Wasmtime, Wasmer).
   - ahead of time (AOT) optimized compilation using the recommended WASM compiler for each of the three different WebAssembly runtimes (WasmEdge, Wasmtime, Wasmer).
   - a container image using a distroless base image containing only the binary of the given benchmarking function obtained using the default compiler options of the Rust and Go compilers, depending on the source programming language.
   - a container image using the common debian:bullseye base image containing only the binary of the given benchmarking function obtained using the default compiler options of the Rust and Go compilers, depending on the source programming language.
2. Upload the resulting image to a public container registry.
3. Pre-download the relevant container images on the required hosts which will be executing them.
4. Execute each OCI image 100 times on a dedicated bare-metal host configured for the specific workload.
5. Measure the execution time using the time command line utility available on UNIX based operating systems.

It should be noted that OCI images were used as the distribution medium for transferring and executing the WebAssembly modules by each of the three WASM runtimes. These OCI images were made using the scratch base image, meaning that they contain only a single layer with a single file in it – the WASM binary. Once the workload is instantiated, the WASM binary is extracted from the OCI image and it is passed on to the WASM runtime for execution. As customary, the AOT compilation was done on the exact same system architecture as the one where the workloads were later executed, to ensure the efficacy of the optimizations.

Crun, the necessary component integrated with containerd so that execution of WASM workloads is possible, supports both interpretation and AOT execution for WasmEdge, while only JIT is supported for the remaining two runtimes, Wasmtime and Wasmer by default. By modifying the source code of crun, we were able to add support for AOT execution of Wasmtime and Wasmer as well, allowing us to get more representative results.

Table 3 Details about the execution environment

Component	Description
CPU	Intel Xeon x5647, 2.93GHz, 4 cores
Storage	320GB mechanical hard drive
Memory	8GB DDR3
Operating System	Ubuntu 22.04.1 LTS
Crun Version	v1.6
Containerd Version	v1.5.9
WasmEdge Version	v0.11.0
Wasmtime Version	v1.0.1
Wasmer Version	v2.3.0
Rust Version	v1.64.0
Go Version	v1.19.1
TinyGo[1] Version	v0.26.0

[1]TinyGo is used as the compiler to compile the functions written in Go to WebAssembly.

To ensure reproducibility of all generated OCI images, multi-staged builds were utilized, ensuring that the size of the resulting image will not be increased by the inclusion of unnecessary tooling required only during the compilation process. A fleet of five completely identical (in terms of hardware configuration) workstations was used for performing all experiments. More details about the hardware configuration are available in Table 3. Three of the five utilized machines were dedicated to executing WebAssembly workloads with the different WASM runtimes, one was dedicated to executing standard OCI containers using an unmodified version of containerd, and the last one was dedicated to building all of the OCI images discussed previously, including those that contained AOT WASM modules. During the execution of all OCI images, even though they were predownloaded on the hosts, we ensured that in each run they would be executed against a clean installation of containerd, as to avoid any image layer caching, thus skewing the results when evaluating the cold start delays.

It should be noted that the same execution strategy described above can be applied to any virtual cloud environment. In our case, we opted for bare-metal execution on physical hardware as to ensure a uniform hardware configuration among all compute nodes and to make sure that there would be no impact on the obtained results of third-party compute workload which would be executed concurrently, in a multi-tenant virtual environment. Moreover, the proposed changes can also be applied to popular open-source serverless platforms which

are currently using a container runtime for executing functions, since a number of them already support containerd [56–58]. Evaluation using a commercial serverless platform is currently not possible since the majority of them do not natively support WebAssembly at present, and offer no options of manually customizing the runtime environment.

3.4 Reproducibility

Reproducibility is an important topic for every study that involves performance measurements and produces results which later need to be analyzed and interpreted. In order to allow others to extend the presented benchmarking suite, reuse the results discussed in the rest of this paper, or compare them with results obtained using other execution strategies in the future, we have decided to publish all of the accompanying material under a permissive open-source license. This includes the source code of the serverless functions in the benchmarking suite, source code for the build scripts for the OCI images, the OCI images themselves, the execution scripts used for batch execution of the workloads in a controlled environment, as well as the obtained results[6]. Step-by-step instructions are provided for each step in the repository, along with a description of the changes that need to be made to the source code of the different software tools which are being used. A Jupyter notebook has also been made available, which can be used for further exploratory analysis of the obtained results or extended with additional data, while reusing the same visualizations.

4 Results

The obtained results from the 100 execution runs for each of the 13 benchmarking functions executed across the 8 different environments are discussed in more detail in the subsections that follow. 6 of these environments are WebAssembly environments, testing the cold start times and execution performance of interpreted, JIT, and AOT compiled WASM modules in WasmEdge, Wasmtime and Wasmer. The remaining 2 are traditional containerized environments, using different base images. We focus on evaluating the performance characteristics of each individual runtime, its benefits, and any current shortcomings which we have detected during the testing, before moving forward with a cross platform comparison in section 5.

4.1 WasmEdge

WasmEdge is a versatile cloud native WebAssembly runtime which also supports edge deployments for IoT scenarios. It is currently a Cloud Native Computing Foundation (CNCF) sandboxing project. It offers support for WASI as well as additional WebAssembly feature proposals which are not yet part of the core specification [59].

[6] See footnote 1.

WasmEdge can either interpret WebAssembly modules on the fly or execute them directly in case where AOT compilation is used. Using the wasmedgec compiler it is possible to precompile existing WASM modules ahead of time, significantly improving their execution performance. This is in stark contrast to the default behavior of WasmEdge, which is to execute the WASM workloads in interpreter mode, providing easier debugging, but forgoing any performance optimizations. When using the wasmedgec tool for AOT compilation, we have opted to use its default configuration [60], to better judge the performance level that would be available to the widest range of users. It is worth mentioning that in this case, the default optimization level for compiled code is set to 2. Wasmedgec uses the same notation for optimization levels as clang, which defines 2 as a moderate level, enabling most optimizations [61]. Table 4 offers a comparison between the execution times of the 13 different benchmarking functions in cases where they were executed using the interpretation mode versus when they were precompiled.

Table 4 WasmEdge execution time comparison between interpretation and AOT compilation

Function	AOT time	Interp. time	Speedup[1]
audio-sine-wave	0.8011s	0.9675s	x1.21
fuzzysearch	0.8162s	3.3380s	x4.09
n-body	0.7696s	0.7155s	x0.93
prime-numbers	0.7637s	0.8491s	x1.11
whatlang	0.7929s	0.7369s	x0.93
zip-compression	3.0367s	36.9394s	x12.16
aes	0.8178s	3.1143s	x3.81
checksum	2.8217s	420.7572s	x149.11
diskio	0.9131s	13.2456s	x14.51
float-operation	0.8006s	0.6979s	x0.87
imageprocessing	5.5801s	957.1059s	x171.52
linear-equations	0.9700s	31.2551s	x32.22
matmul	3.6132s	709.4923s	x196.36

[1]Represents how many times AOT is faster compared to interpretation. Speedup of less than 1.x means that interpretation is faster than AOT execution.

Analyzing the results, we notice that the mean AOT execution times are lower than the ones obtained using interpretation in 10 out of 13 tests. As expected, the speedup coefficient largely corresponds to the complexity of the task, with more compute intensive functions such as checksum, imageprocessing and matmul being more than 100 times faster during AOT compilation in comparison to interpretation. Interestingly, in 3 tests (n-body, whatlang, float-operation) interpretation shows marginally better results compared to AOT. All of these three tests share the fact that they execute rather simple workloads which do not require large compute resources. Additionally, as can be seen from Figure 1 which visualizes the OCI image sizes for each of the execution strategies, the images hosting WASM modules which are to be

interpreted are smaller in all evaluated cases. This is also a contributing factor which explains the lower execution time of interpreted modules in the three edge cases – the lower image sizes require less time to be loaded into main memory, which coupled with the reduced complexity of the workload lead to marginal performance gain in what are already very short execution times.

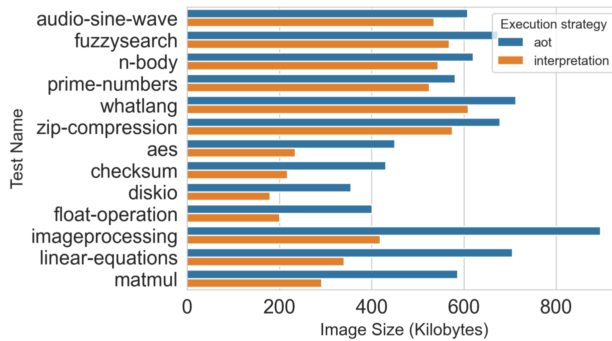

Fig. 1 OCI image size comparison between AOT and interpretation for WasmEdge

Even though Table 4 shows drastically different results for many of the evaluated functions, there are also cases which have similar mean execution times between AOT and interpretation (i.e. audio-sine-wave, prime-numbers, n-body, whatlang, float-operation). To test the statistical significance of these differences in the mean execution times, we have performed non-parametric Mann-Whitney U-tests for the results of each function, comparing the AOT and interpretation times. An alpha value of 0.05 and the following two hypotheses were used in all cases: H0) the two populations are equal; H1) the two populations are not equal. The obtained p-values are lower than 0.05 in all cases, leading to the rejection of the null hypothesis and concluding that there is a statistically significant difference between the obtained execution times of AOT and intepretation.

4.2 Wasmtime

Wasmtime is the oldest server side WebAssembly runtime among those selected for evaluation in this paper. In September 2022 it reached the important milestone of releasing version 1.0. It supports two modes of execution, leveraging just in time compilation or ahead of time compilation. Just in time compilation is expected to be faster than interpretation, while being slower than execution of already precompiled modules, since the relevant optimizations are done once the module is invoked, but before the relevant code starts to execute. It should be noted that modifications were required to the source code of the crun library so that it can support AOT, since only JIT is supported by default in the case of Wasmtime. The Wasmtime compiler was used with its default set of preferences, same as in the WasmEdge case. The code optimization level is

configurable as well, and the default is again 2, reflecting the default configuration used by WasmEdge. Table 5 offers a comparison between the AOT and JIT execution times of the evaluated WebAssembly benchmarking functions when executed with the Wasmtime runtime.

Table 5 Wasmtime execution time comparison between JIT and AOT compilation

Function	AOT time	JIT time	Speedup[1]
audio-sine-wave	0.6232s	0.8178s	x1.31
fuzzysearch	0.7046s	0.8600s	x1.22
n-body	0.7210s	0.8373s	x1.16
prime-numbers	0.8438s	0.8021s	x0.95
whatlang	0.6722s	0.8547s	x1.27
zip-compression	3.0558s	3.2597s	x1.07
aes	0.7370s	1.2215s	x1.66
checksum	3.6143s	4.1259s	x1.14
diskio	0.8756s	1.3369s	x1.53
float-operation	0.7048s	1.1013s	x1.56
imageprocessing	8.6605s	9.8001s	x1.13
linear-equations	0.9241s	1.5558s	x1.68
matmul	6.3049s	6.9273s	x1.10

[1] Represents how many times AOT is faster compared to JIT execution. Speedup of less than 1.x means that JIT is faster than AOT execution.

The results from Table 5 show that AOT is faster than JIT in 12 out of 13 benchmarks, with speedups ranging from from 1.07 (zip-compression) to 1.68 times (linear-equations). Prime-numbers is the only benchmark where JIT is slightly faster than AOT, clocking an average execution time of 0.8021s versus 0.8438s for AOT. Prime-numbers, in this case, is not a particularly resource intensive benchmark since it is searching for prime numbers only in the range of 1-100, spending the majority of the time on instantiating the function instead of executing it.

Wasmtime paints an interesting picture in the case of image sizes as well, as depicted in Figure 2. In 7 out of 13 tests (aes, checksum, diskio, float-operation, imageprocessing, linear-equations, matmul) the OCI image containing the AOT WASM module is larger than its JIT counterpart. In the remaining 6 cases (audio-sine-wave, fuzzysearch, n-body, prime-numbers, whatlang, zip-compression) the situation is reversed, with the JIT images having larger size than AOT.

Since in this case the similarity between JIT and AOT mean execution times is even more pronounced, with the highest difference being x1.68 slowdown for JIT versus AOT in the case of the linear-equations benchmark, further analysis in terms of the statistical significance of these differences is warranted. As previously, we have conducted the same Mann-Whitney U-tests, this time comparing the JIT and AOT execution times of the Wasmtime runtime, keeping the same alpha value of 0.05 and the two hypotheses: H0) the

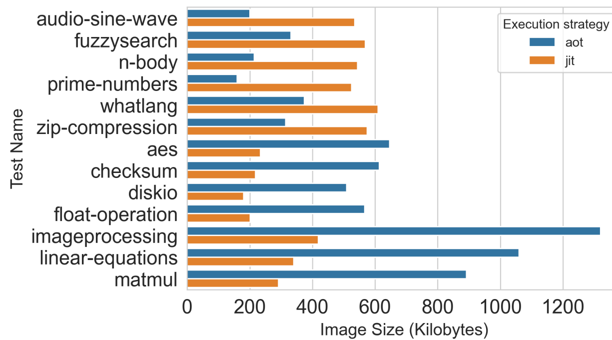

Fig. 2 OCI image size comparison between AOT and JIT for Wasmtime

two populations are equal; H1) the two populations are not equal. In all cases the obtained p-value is lower than the set alpha value of 0.05, thus allowing us to reject the null hypothesis and concluding that the difference in execution times is statistically significant.

4.3 Wasmer

The Wasmer WebAssembly runtime has the most GitHub stars among the tested runtimes and has also released its milestone 1.0.0 version earliest, in January 2021. It supports both JIT and AOT modes of execution, but similarly to the case of Wasmtime, Crun in version 1.6 does not support AOT, so manual source code changes were required as well. The AOT compilation strategy was the same as for both WasmEdge, and Wasmtime, leveraging Wasmer's AOT compiler default settings. Table 6 shows the obtained results for 6 out of the 13 benchmark runs of the evaluated functions. Measurements are missing for all tests which require I/O access to the underlying filesystem, since even though Wasmer does support WASI, it was not possible to execute these functions with crun's WebAssembly integration.

Table 6 Wasmer execution time comparison between JIT and AOT compilation

Function	AOT time	JIT time	Speedup[1]
n-body	0.7248s	0.8320s	x1.15
prime-numbers	0.7017s	0.8298s	x1.18
whatlang	0.7346s	0.8404s	x1.14
float-operation	0.7508s	1.2034s	x1.60
linear-equations	1.0072s	1.6638s	x1.65
matmul	6.5564s	7.0870s	x1.08

[1]Represents how many times AOT is faster compared to JIT execution.

Of the 6 out of 13 functions that did complete the 100 execution runs successfully, all show better AOT execution time compared to JIT. Taking into account that JIT instead of interpretation is used, the speedup coefficients of AOT are in the range of minimal 1.08 (matmul) to 1.65 (linear-equations) times.

We have decided to include all OCI images in the image size comparison, including those that did not complete their runs successfully, to get a better picture of the optimization levels of the Wasmer AOT compiler. Figure 3 shows that in all cases the WASM modules using AOT compilation have a larger OCI image size when compared to their JIT counterparts. This difference is most pronounced in the case of imageprocessing where additional third party libraries need to be included in the optimized WASM module, resulting in an image size of 896.29KB compared to 418.92KB in the case of JIT.

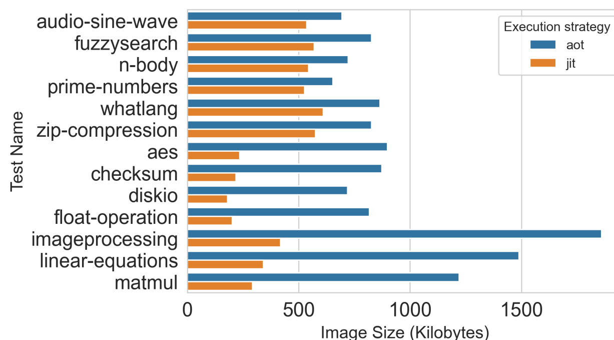

Fig. 3 OCI image size comparison between AOT and JIT for Wasmer

Conducting the same Mann-Whitney U-tests as previously for WasmEdge and Wasmtime, reusing the alpha value of 0.05 and the two hypotheses: H0) the two populations are equal; H1) the two populations are not equal;, we obtain that there is a statistically significant difference in the results obtained for all successfully executed tests, leading to the rejection of the null hypothesis.

4.4 Regular and Distroless Images

Concluding the individual analysis of the different execution environments, we have also evaluated the execution performance of the same 13 functions when they are run in standard OCI containers, without first being compiled as WebAssembly modules. This would allow us to directly compare the execution performance of the various WebAssembly runtimes to a more traditional containerized environment which is still the most popular choice today among many commercial and open-source serverless platforms.

Table 7 shows the difference between the execution times of the functions when they are run in a container instantiated from a distroless OCI image versus one that includes additional dependencies, in this case debian:bullseye.

Table 7 Containerized execution time comparison between distroless and debian:bullseye base images

Function	Distroless time	Debian time	Speedup[1]
audio-sine-wave	1.5910s	3.5241s	x2.22
fuzzysearch	1.4540s	3.5214s	x2.42
n-body	1.7005s	3.5800s	x2.11
prime-numbers	1.7005s	3.5919s	x2.11
whatlang	1.4946s	3.5440s	x2.37
zip-compression	3.9159s	5.9046s	x1.51
aes	1.8785s	4.1197s	x2.19
checksum	2.7728s	4.8078s	x1.73
diskio	1.6408s	3.7154s	x2.26
float-operation	1.5234s	3.5300s	x2.32
imageprocessing	2.4391s	4.5162s	x1.85
linear-equations	1.5887s	3.5529s	x2.24
matmul	2.6960s	4.7505s	x1.76

[1]Represents how many times distroless is faster compared to debian:bullseye base image execution.

In all 13 cases the distroless base image shows better execution time, with speedups ranging from 1.51 (zip-compression) to 2.42 (fuzzysearch) times. These results are expected when taking into account the fact that the debian:bullseye images are around 5 times larger compared to the distroless, adding significant execution delay as a result of the time it takes to instantiate the container and load it into memory. Differences in compute performance are not expected between the two execution methods, since the only difference is the base image, while the binary file being executed and the runtime technology is the same. Figure 4 visualizes the differences between the image sizes of distroless and debian:bullseye.

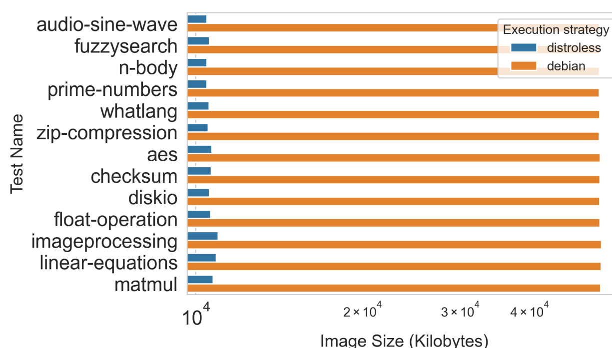

Fig. 4 OCI image size comparison between distroless and debian:bullseye base images

Even though the differences are evident from the results themselves, we repeated the same non-parametric Mann-Whitney U-test as in the previous

cases, with the same alpha value of 0.05 and hypotheses: H0) the two populations are equal; H1) the two populations are not equal. The differences in execution speed between the distroless and debian:bullseye base images are statistically significant in all cases, as expected, thus leading to the dismissal of the null hypothesis.

5 Discussion

The individual results obtained thus far from the different execution methods clearly show that AOT compiled WASM binaries have a noticeable performance advantage over both interpretation and JIT in the majority of cases, no matter the runtime. Furthermore, OCI images created from scratch and containing only a single WASM binary are an order of magnitude smaller than lightweight distroless OCI images and more popular GNU/Linux distribution based OCI images.

To offer an in-depth comparison of the obtained results across all 5 different execution strategies (WasmEdge, Wasmtime, Wasmer, distroless, and debian:bullseye), Figure 5 shows the mean execution times of AOT compilation of the 13 benchmark functions. As previously, results are missing for Wasmer since it was not possible to successfully execute functions requiring access to the underlying filesystem. In 10 out of 13 benchmarks, all three WebAssembly runtimes show better mean execution times than execution with a container runtime (aes, audio-sine-wave, diskio, float-operation, fuzzysearch, linear-equations, n-body, prime-numbers, whatlang, zip-compression). In the case of checksum, the native execution of the workload in a distroless base image is faster than both WebAssembly runtimes that successfuly executed the function (WasmEdge and Wasmtime), while execution in the more resource heavy debian:bullseye based image is the slowest. Image manipulation, a computationally heavy process, is faster in traditional containerized environments, while the results for the matrix multiplication (matmul) scenario are mixed. WasmEdge, the fastest WASM runtime in this case, is slower than distroless, but faster than debian:bullseye, while the remaining two WASM runtimes are both lagging behind debian:bullseye.

Comparing the AOT execution results only among the WASM runtimes, Wasmtime has the best results in 8 out of 13 benchmarks (aes, audio-sine-wave, diskio, float-operation, fuzzysearch, linear-equations, n-body, whatlang). WasmEdge is in the lead in 4 cases (checksum, imageprocessing, matmul, zip-compression), while Wasmer in only one instance comes out on top, in the case of whatlang.

Figure 6 compares the results only among the three WebAssembly runtimes. In the case of WasmEdge, the interpretation execution time is shown, since JIT is not supported. For both Wasmtime and Wasmer, JIT execution time is shown. Distroless and debian:bullseye have been omitted from the picture since neither interpretation nor JIT is supported when executing the benchmarks using a container runtime.

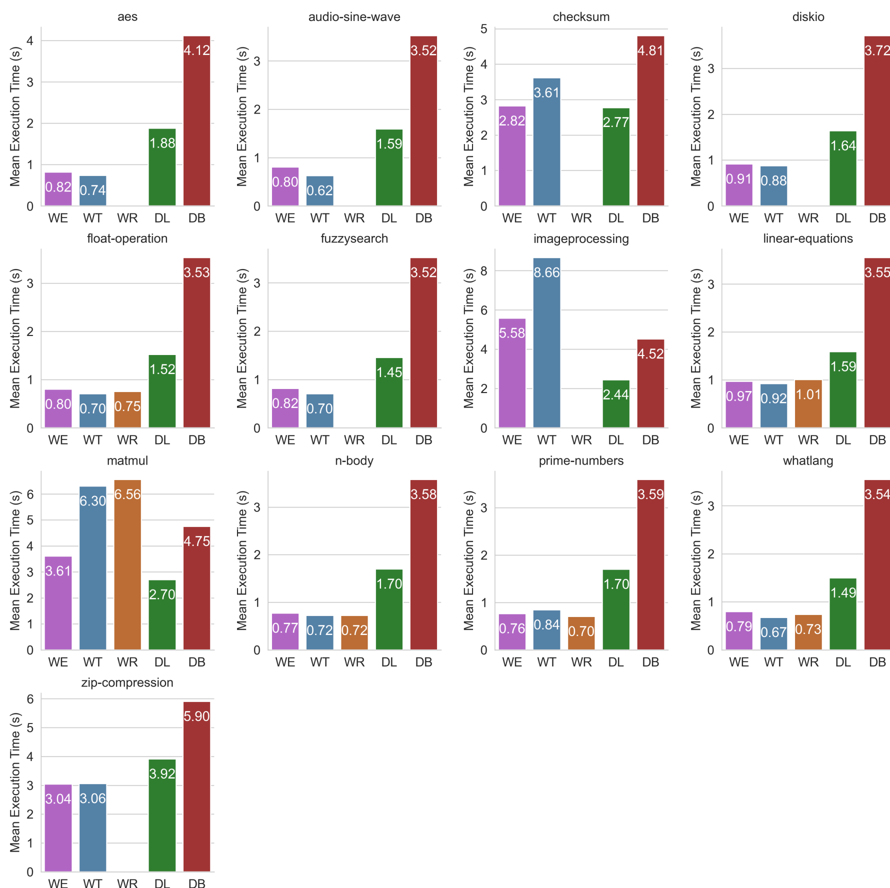

Fig. 5 Comparison of AOT mean execution times across all 5 execution environments (WE-WasmEdge; WT-Wasmtime; WR-Wasmer; DL-distroless; DB-debian:bullseye)

Intepretation, represented by WasmEdge, as expected, is the slowest method in all 13 cases except for 3 (float-operation, n-body, whatlang). Interestingly, in these 3 cases interpretation is faster than the JIT execution of both Wasmtime and Wasmer. All of these three benchmarks have in common the fact that they are not as compute intensive as the remaining ones, justifying the obtained results, and leading to the conclusion that the JIT compilation and relevant optimizations to the execution process are more time intensive than simply interpreting the code when it comes to such simple workloads. Furthermore, a non-trivial overhead of JIT can also be noticed when comparing, for example, the execution times of the float-operation function between WasmEdge (interpretation) on one hand and Wasmtime and Wasmer (JIT) on the other. In the case of WasmEdge's interpretation is faster than AOT, with also a smaller image size, while in the remaining case, AOT compilation is faster, despite the increased image size, corroborating the conclusion that

JIT compilation is a resource intensive process. However, in more resource intensive functions, interpretation results are worse than their JIT counterparts by at least an order of magnitude (checksum, diskio, imageprocessing, lilnear-equations, matmul, zip-compression), thus justifying the JIT overhead in such cases. Wasmtime is the fastest runtime in the case of JIT execution in 10 out of 13 benchmarks (aes, audio-sine-wave, checksum, diskio, fuzzysearch, imageprocessing, linear-equations, matmul, prime-numbers, zip-compression), but it must be noted, again, that Wasmer was excluded from the functions requiring filesystem access, due to incompatibility.

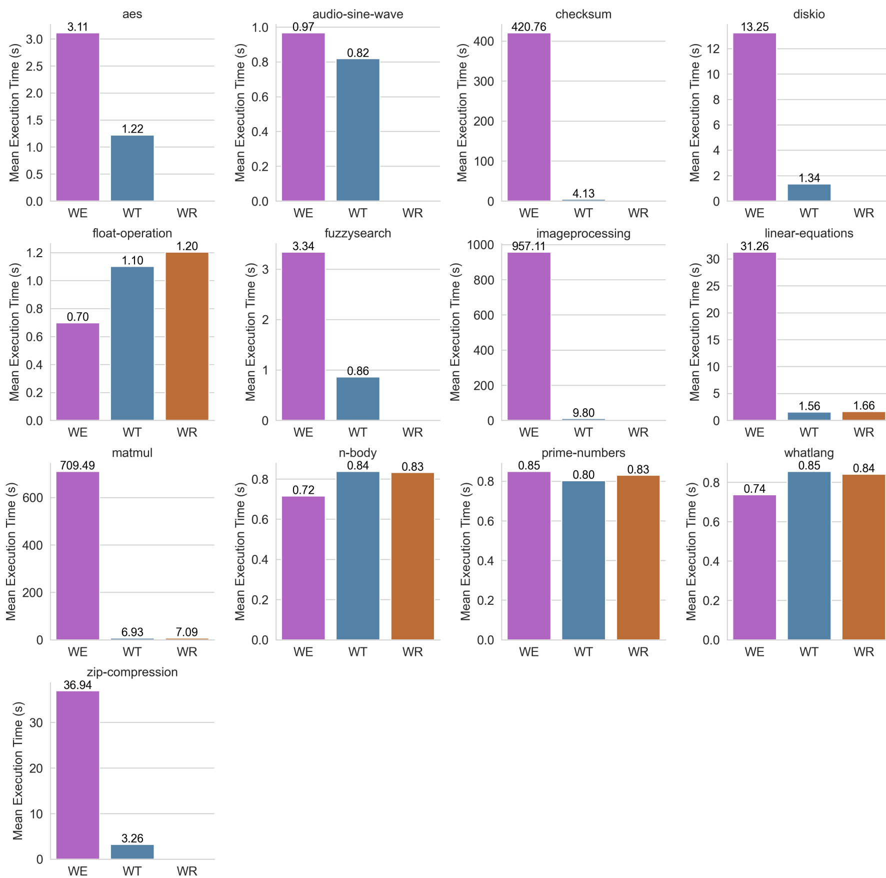

Fig. 6 Comparison of interpretation (WasmEdge) and JIT (Wasmtime, Wasmer) mean execution times across the 3 WebAssembly runtimes (WE-WasmEdge; WT-Wasmtime; WR-Wasmer)

In a number of cases, when discussing both AOT results on one hand, and JIT and interpretation results on another, we see close similarities between the

obtained mean execution times. To verify whether there is a statistically significant difference between the obtained results we conducted the non-parametric Mann-Whitney U-test between all possible test pairs for AOT execution and all possible test pairs for JIT and interpretation. The alpha value in all cases was 0.05 and the hypotheses were the same as the ones used in the previous section, namely: H0) the two populations are equal; H1) the two populations are not equal. Interestingly, the obtained results do not offer enough evidence to reject the null hypothesis in two cases when analyzing the AOT execution times. The first case are the AOT execution times of WasmEdge and Wasmtime for the n-body benchmark, and the second case is again related to the n-body function, but in this case the runtimes in question are Wasmtime and Wasmer. The obtained p-values in these cases are 0.4475 and 0.9766 respectively. In all remaining cases concerning AOT, the p-values were lower than alpha, leading to the rejection of the null hypothesis. Looking at U-test results for JIT and intepretation, a similar situation emerges, there is not enough evidence to reject the null hypothesis only in the case of the Wasmtime and Wsamer runtimes, for the execution of the n-body function. In all other cases, the differences in the obtained results are statistically significant, thus rejecting the null hypothesis.

Taking into account the bigger picture, it is evident that WebAssembly offers clear advantages in terms of the reduction of the cold start times when compared to execution in a traditional container runtime, achieving submillisecond execution times across multiple benchmarking functions. However, in the case of more compute intensive workloads, such as imageprocessing, primenumbers or zip-compression, the time savings in terms of the cold start time are offset by the slower compute performance of WASM, thus forfeiting the lead to containerized execution which shows better results when directly comparing the compute performance of the two different kinds of serverless runtime environments.

The obtained results further corroborate the fact that the next generation serverless platforms should take a more diverse, but balanced approach to the runtime environments which they support. WebAssembly has clear advantages when executing more lightweight functions where the cold start delay has greater impact on the total execution time. Contrary to this, container runtimes currently offer better sustained compute performance, advantageous to resource intensive functions, but suffer by cold start times which in many cases are an order of magnitude larger than those of their WebAssembly counterparts.

6 Conclusion

WebAssembly, with its increasing popularity and support for new and modern programming languages, brings a number of advantages both to client side and server side computing. These advantages have the potential to directly impact the future of serverless computing, as showcased in this paper. With the aim

of evaluating possibilities of integrating WebAssembly runtimes with the APIs and CLIs of existing container based runtimes, we have created and open-sourced a benchmarking suite consisting of 13 different serverless functions, written in the Go and Rust programming languages, while focusing both on microbenchmarking or real-world scenarios. We have then used these benchmarks to evaluate the cold start performance and total execution time of three different WebAssembly runtimes, comparing the results between themselves, as well as to those of a traditional container runtime. To the best of our knowledge, this is the first paper that compares the performance of WASM runtimes which are invoked not directly through their native APIs, but via an integration with a versatile and extensible container runtime, containerd in the case of this study.

Results show that the WebAssembly runtimes achieve better execution times in the majority of the tests (10 out of 13), when comparing WASM AOT execution performance to the execution performance of distroless and debian:bullseye container images. When it comes to sustained compute performance in complex workloads, such as image processing or matrix multiplication, further work is required in the case of WebAssembly runtimes, since container runtimes offer better results due to their increased compute performance, despite the associated cold start drawbacks. It remains to be seen what impact multi-threading support might have in this context, once it has been introduced as a stable feature to WebAssembly.

Comparing the performance of the three included WebAssembly runtimes among themselves, Wasmtime's integration with containerd offers the best results in 8 out of the 13 cases, with WasmEdge being second, coming out on top in 4 cases, and Wasmer achieving the best results in a single test.

In conclusion, we have shown that WebAssembly has clear advantages in terms of improved cold start times compared to container runtimes, as well as artifact sizes which are at least an order of magnitude smaller than the their OCI container image counterparts. However, WebAssembly cannot be seen as a complete replacement of container runtimes in serverless platforms, since containers still offer improved sustained compute performance, which is of great importance for more resource intensive computations. By integrating WebAssembly runtimes with the APIs and CLIs of existing container runtimes it is possible to achieve the best of both worlds, while reusing the existing mature tooling and orchestration tools developed primarily for container based workloads. In this manner, the problem of orchestration, one of the last remaining issues for server side execution of WebAssembly can be solved, not by designing new purpose built solutions, but instead of reusing existing components.

7 Threats to Validity

We attempted to eliminate as many potential threats to validity as possible during the design and execution of the described testing methodology. To

ensure relevant results, all tests were executed on identical hardware, within a single local area network which was dedicated to this research. We decided to preload all software artifacts before executing any tests, as to eliminate outside influences which are not under our control, such as network latency to remote destinations, or remote server overloading. The 13 serverless functions included in the benchmarking suite used for conducting the tests were sourced from the Internet and might not use the most efficient algorithms for solving a given problem. However, this should not be seen as a threat to the validity of the research, since the same functions were executed in all execution scenarios, so the results are relevant when compared between the different runtimes. Additionally, all of the benchmarks included in the presented benchmarking suite are written as sequential applications. These sequential versions were used both for evaluating WebAssembly (which currently does not have stable multi-threading support), and standard containerized execution (which does feature multi-threading support). It is possible that some of the benchmarking functions can have a more efficient multi-threaded implementation, but we deem that such evaluations are out of scope of this paper, since it has already been shown by work referenced in the Related Work section that native execution shows better execution performance when compared to WebAssembly. The focus of this paper is instead to evaluate the cold start performance of WebAssembly runtimes while their APIs are integrated with a container runtime. In these cases, such implementation difference details should not have an effect on the final results. Finally, the obtained results depict the performance of the runtime versions available at the time of writing the paper, and it is expected that future versions will offer improved performance.

Declarations

7.1 Ethics approval and consent to participate

Not applicable.

7.2 Consent for publication

Not applicable.

7.3 Availability of data and material

The generated raw data, software, and outputs from the data analysis are publicly available under a permissive license on https://github.com/korvoj/wasm-serverless-benchmarks.

7.4 Competing interests

The authors have no relevant financial or non-financial interests to disclose.

7.5 Funding

This study was funded by the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Skopje, North Macedonia under the "NSA" project.

7.6 Authors' contributions

Conceptualization: S.F. and V.K.; Investigation: V.K.; Methodology: S.F. and V.K.; Software: V.K.; Validation: S.F. and V.K.; Formal analysis: S.F. and V.K. Writing (original draft preparation): V.K.; Writing (review and editing): S.F and V.K. All authors have reviewed the manuscript.

7.7 Acknowledgements

Not applicable.

References

[1] Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing (2009)

[2] Mell, P., Grance, T.: The NIST Definition of Cloud Computing. Technical Report NIST Special Publication (SP) 800-145, National Institute of Standards and Technology (September 2011). https://doi.org/10.6028/NIST.SP.800-145

[3] Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N.C., Hu, B.: Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. In: 2015 IEEE 8th International Conference on Cloud Computing, pp. 621–628 (2015). https://doi.org/10.1109/CLOUD.2015.88

[4] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs] (2019) arXiv:1902.03383 [cs]

[5] Kratzke, N.: A Brief History of Cloud Application Architectures. Applied Sciences **8**(8), 1368 (2018). https://doi.org/10.3390/app8081368

[6] Wen, J., Liu, Y., Chen, Z., Chen, J., Ma, Y.: Characterizing commodity serverless computing platforms. Journal of Software: Evolution and Process **n/a**(n/a), 2394. https://doi.org/10.1002/smr.2394

[7] El Ioini, N., Hästbacka, D., Pahl, C., Taibi, D.: Platforms for Serverless at the Edge: A Review. In: Zirpins, C., Paraskakis, I., Andrikopoulos,

V., Kratzke, N., Pahl, C., El Ioini, N., Andreou, A.S., Feuerlicht, G., Lamersdorf, W., Ortiz, G., Van den Heuvel, W.-J., Soldani, J., Villari, M., Casale, G., Plebani, P. (eds.) Advances in Service-Oriented and Cloud Computing vol. 1360, pp. 29–40. Springer International Publishing, Cham (2021)

[8] Li, J., Kulkarni, S.G., Ramakrishnan, K.K., Li, D.: Analyzing Open-Source Serverless Platforms: Characteristics and Performance. arXiv:2106.03601 [cs], 15–20 (2021) arXiv:2106.03601 [cs]. https://doi.org/10.18293/SEKE2021-129

[9] Cloudflare Workers. https://workers.cloudflare.com/ Accessed 2022-11-09

[10] Pfandzelter, T., Bermbach, D.: IoT Data Processing in the Fog: Functions, Streams, or Batch Processing? In: 2019 IEEE International Conference on Fog Computing (ICFC), pp. 201–206. IEEE, Prague, Czech Republic (2019). https://doi.org/10.1109/ICFC.2019.00033

[11] Varghese, B., Buyya, R.: Next generation cloud computing: New trends and research directions. Future Generation Computer Systems **79**, 849–861 (2018). https://doi.org/10.1016/j.future.2017.09.020

[12] Salehe, M., Hu, Z., Mortazavi, S.H., Mohomed, I., Capes, T.: VideoPipe: Building Video Stream Processing Pipelines at the Edge. In: Proceedings of the 20th International Middleware Conference Industrial Track, pp. 43–49. ACM, Davis CA USA (2019). https://doi.org/10.1145/3366626.3368131

[13] Kjorveziroski, V., Filiposka, S., Trajkovik, V.: IoT Serverless Computing at the Edge: A Systematic Mapping Review. Computers **10**(10), 130 (2021). https://doi.org/10.3390/computers10100130

[14] Hellerstein, J.M., Faleiro, J., Gonzalez, J.E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C.: Serverless Computing: One Step Forward, Two Steps Back. In: CIDR 2019, Monterey, CA (2018)

[15] Kjorveziroski, V., Canto, C.B., Roig, P.J., Gilly, K., Mishev, A., Trajkovik, V., Filiposka, S.: IoT Serverless Computing at the Edge: Open Issues and Research Direction. Transactions on Networks and Communications **9**(4), 1–33 (2021). https://doi.org/10.14738/tnc.94.11231

[16] Bocci, A., Forti, S., Ferrari, G.-L., Brogi, A.: Secure FaaS orchestration in the fog: How far are we? Computing **103**(5), 1025–1056 (2021). https://doi.org/10.1007/s00607-021-00924-y

[17] Kjorveziroski, V., Filiposka, S.: Kubernetes distributions for the

edge: Serverless performance evaluation. The Journal of Supercomputing **78**(11), 13728–13755 (2022). https://doi.org/10.1007/s11227-022-04430-6

[18] Wang, B., Ali-Eldin, A., Shenoy, P.: LaSS: Running Latency Sensitive Serverless Computations at the Edge. In: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, pp. 239–251. Association for Computing Machinery, New York, NY, USA (2020)

[19] Agarwal, S., Rodriguez, M.A., Buyya, R.: A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 797–803 (2021). https://doi.org/10.1109/CCGrid51090.2021.00097

[20] Murphy, S., Persaud, L., Martini, W., Bosshard, B.: On the Use of Web Assembly in a Serverless Context. In: Paasivaara, M., Kruchten, P. (eds.) Agile Processes in Software Engineering and Extreme Programming – Workshops. Lecture Notes in Business Information Processing, pp. 141–145. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58858-8_15

[21] Marin, E., Perino, D., Di Pietro, R.: Serverless computing: A security perspective. Journal of Cloud Computing **11**(1), 69 (2022). https://doi.org/10.1186/s13677-022-00347-w

[22] W3C WebAssembly Working Group. https://www.w3.org/wasm/ Accessed 2022-11-09

[23] WebAssembly Language Support Matrix. https://www.fermyon.com Accessed 2022-10-29

[24] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 185–200. ACM, Barcelona Spain (2017). https://doi.org/10.1145/3062341.3062363

[25] Wang, W.: Empowering Web Applications with WebAssembly: Are We There Yet? In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1301–1305 (2021). https://doi.org/10.1109/ASE51524.2021.9678831

[26] Wang, Z., Wang, J., Wang, Z., Hu, Y.: Characterization and Implication of Edge WebAssembly Runtimes. In: 2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on

Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), pp. 71–80 (2021). https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00037

[27] Wang, W.: How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. In: IISWC 2022, Austin, TX, USA (2022)

[28] WASI Filesystem. WebAssembly. https://github.com/WebAssembly/wasi-filesystem Accessed 2022-11-09

[29] WASI Sockets. WebAssembly. https://github.com/WebAssembly/wasi-sockets Accessed 2022-11-09

[30] Wasi-Threads. WebAssembly (2022). https://github.com/WebAssembly/wasi-threads Accessed 2022-11-09

[31] WebAssembly System Interface – Proposals. https://github.com/WebAssembly/WASI/blob/bac366c8aeb69cacfea6c4c04a503191bf1cede1/Proposals.md Accessed 2022-11-08

[32] Gackstatter, P., Frangoudis, P.A., Dustdar, S.: Pushing Serverless to the Edge with WebAssembly Runtimes. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 140–149 (2022). https://doi.org/10.1109/CCGrid54584.2022.00023

[33] Gadepalli, P.K., Peach, G., Cherkasova, L., Aitken, R., Parmer, G.: Challenges and Opportunities for Efficient Serverless Computing at the Edge. In: 2019 38th Symposium on Reliable Distributed Systems (SRDS), pp. 261–2615 (2019). https://doi.org/10.1109/SRDS47363.2019.00036

[34] Component Model Design and Specification. WebAssembly (2022). https://github.com/WebAssembly/component-model Accessed 2022-11-09

[35] Ling, W., Ma, L., Tian, C., Hu, Z.: Pigeon: A Dynamic and Efficient Serverless and FaaS Framework for Private Cloud. In: 2019 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 1416–1421. IEEE, Las Vegas, NV, USA (2019). https://doi.org/10.1109/CSCI49370.2019.00265

[36] Karhula, P., Janak, J., Schulzrinne, H.: Checkpointing and Migration of IoT Edge Functions. In: Proceedings of the 2nd International Workshop on Edge Systems, Analytics And Networking. EdgeSys '19, pp. 60–65. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3301418.3313947

[37] Pelle, I., Czentye, J., Doka, J., Kern, A., Gero, B.P., Sonkoly, B.: Operating Latency Sensitive Applications on Public Serverless Edge Cloud Platforms. IEEE Internet of Things Journal, 1–1 (2020). https://doi.org/10.1109/JIOT.2020.3042428

[38] Elgamal, T.: Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In: 2018 IEEE/ACM Symposium on Edge Computing (SEC), pp. 300–312. IEEE, Seattle, WA, USA (2018). https://doi.org/10.1109/SEC.2018.00029

[39] Gadepalli, P.K., McBride, S., Peach, G., Cherkasova, L., Parmer, G.: Sledge: A Serverless-first, Light-weight Wasm Runtime for the Edge. In: Proceedings of the 21st International Middleware Conference. Middleware '20, pp. 265–279. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3423211.3425680

[40] Long, J., Tai, H.-Y., Hsieh, S.-T., Yuan, M.J.: A lightweight design for serverless Function-as-a-Service. IEEE Software **38**(1), 75–80 (2021) arXiv:2010.07115 [cs]. https://doi.org/10.1109/MS.2020.3028991

[41] Hockley, D., Williamson, C.: Benchmarking Runtime Scripting Performance in Wasmer. In: Companion of the 2022 ACM/SPEC International Conference on Performance Engineering. ICPE '22, pp. 97–104. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3491204.3527477

[42] Jangda, A., Powers, B., Berger, E., Guha, A.: Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code (2019). https://doi.org/10.5555/3358807.3358817

[43] Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: WebAssembly as a Common Layer for the Cloud-edge Continuum. In: Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge, pp. 3–8 (2022). https://doi.org/10.1145/3526059.3533618

[44] Stephen: Awesome WebAssembly Runtimes (2022). https://github.com/appcypher/awesome-wasm-runtimes Accessed 2022-11-09

[45] Containers/Crun. https://github.com/containers/crun Accessed 2022-11-09

[46] "Distroless" Container Images. GoogleContainerTools (2022). https://github.com/GoogleContainerTools/distroless Accessed 2022-11-09

[47] Kmu-Bigdata/Serverless-Faas-Workbench (2021). https://github.com/kmu-bigdata/serverless-faas-workbench Accessed 2023-01-15

[48] Kim, J., Lee, K.: FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 502–504. IEEE, Milan, Italy (2019). https://doi.org/10.1109/CLOUD.2019.00091

[49] Hound - Crates.Io: Rust Package Registry. https://crates.io/crates/hound Accessed 2023-01-12

[50] Simon, A.N.: anthonynsimon/bild (2023). https://github.com/anthonynsimon/bild Accessed 2023-01-15

[51] N-Body (Benchmarks Game). https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/nbody.html Accessed 2023-01-15

[52] Prime Numbers - The Algorithms. https://the-algorithms.com Accessed 2023-01-15

[53] Lok, A.: andylokandy/simsearch-rs (2023). https://github.com/andylokandy/simsearch-rs Accessed 2023-01-15

[54] Potapov, S.: greyblake/whatlang-rs (2023). https://github.com/greyblake/whatlang-rs Accessed 2023-01-15

[55] zip-rs/zip (2023). https://github.com/zip-rs/zip Accessed 2023-01-15

[56] OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com/ Accessed 2023-01-17

[57] Knative. https://knative.dev/ Accessed 2023-01-17

[58] Kubeless. https://kubeless.io/ Accessed 2023-01-17

[59] Supported WASM And WASI Proposals - WasmEdge Runtime. https://wasmedge.org/book/en/features/proposals.html. https://wasmedge.org/book/en/features/proposals.html Accessed 2022-11-09

[60] Wasmedgec AOT Compiler - WasmEdge Runtime. https://wasmedge.org/book/en/cli/wasmedgec.html Accessed 2023-01-14

[61] FreeBSD Manual Pages – Clang - the Clang, C, C++ and Objective-C Compiler. https://www.freebsd.org/cgi/man.cgi?query=clang++&sektion=1&manpath=FreeBSD+9.0-RELEASE Accessed 2023-01-14