Evaluating WebAssembly for Orchestrated Deployment of Serverless Functions

Vojdan Kjorveziroski
Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University
Skopje, North Macedonia
vojdan.kjorveziroski@finki.ukim.mk

Sonja Filiposka
Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University
Skopje, North Macedonia
sonja.filiposka@finki.ukim.mk

Anastas Mishev
Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University
Skopje, North Macedonia
anastas.mishev@finki.ukim.mk

***Abstract*—Serverless computing has made a significant impact in the cloud computing landscape, and has even been extended beyond the cloud, up to the edge of the network. Existing serverless platforms which use containers and micro virtual machines as function runtimes incur a significant startup latency, hindering the performance and scalability of the executed functions. One potential solution to this problem is the use of WebAssembly. In this paper we discuss recent developments which allow WebAssembly to be used for server-side applications, as well as serverless functions, and evaluate potential orchestration options with the end goal of integrating WebAssembly with existing cloud and edge infrastructure. We conclude that while WebAssembly is a solution to the cold start problem, further work is needed in this area. To realize the end-goal of seamless and user-friendly serverless platforms that can be deployed across the edge-cloud continuum and can dynamically adapt to compute and latency requirements, WebAssembly should not be seen as the exclusive technology, and instead multiple runtime environments should be supported in addition to WebAssembly.**

Keywords—WebAssembly, serverless computing, function as a service, cloud computing, edge computing, internet of things

I. Introduction

The advent of cloud computing has completely changed the computing landscape for everyone involved, from academia to large business corporations to end users. The ability to on-demand request compute resources and granularly pay only for the duration for which they are provisioned has simplified the development of new applications and services. The introduction of various infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) products [1] has allowed both developers and system administrators alike to think in more abstract terms and to forgo the installation and management of physical infrastructure to dedicated cloud providers.

The next popular addition to the long line of *as a service* offerings by cloud providers is serverless computing [2], with its two main components, function as a service (FaaS) and backend as a service (BaaS). Using these two new concepts, developers need to only write the code containing the core business logic in as slimmed down version as possible, usually a single function, which will then be executed in ephemeral and lightweight execution environments. Persistence and integration with external systems is enabled by additional services, such as database as a service or message broker as a service, both representatives of BaaS.

Serverless computing, with its advantages over traditional serverful computing such as potentially unconstrained scalability, faster paced development, and flexible pricing [3], billing only the time for which the function is executed, has become commonplace in many areas, including web development. However, the event driven nature of serverless computing has a potential to impact other emerging topics as well, namely internet of things (IoT), introducing a new set of challenges. To accommodate the billions of new devices expected to be deployed in the coming years, and to be able to respond adequately to time-critical events, the compute infrastructure needs to be moved to the edge of the network, closer to the end devices [4]. The main goal of edge computing is to reduce latency by offloading time sensitive computation to nearby compute infrastructure.

The most common runtime environments for execution of serverless functions today, regardless of whether in the cloud or at the edge, are containers and micro virtual machines (VMs). One of the main challenges faced by serverless computing at present is how to reduce the initial startup time before the runtime environment is ready to serve the incoming request. This startup time is commonly referred to as cold start [5]. Despite the fact that both containers and micro virtual machines are more lightweight than full-fledged virtual machines usually offered by cloud providers, their cold start is still in the order of hundreds of milliseconds if not couple of seconds, making them unsuitable for dealing with time-critical data [6]. Many commercial and open-source platforms have devised ways in which to either eliminate or reduce the cold start times of serverless functions, for example by keeping a pre-warmed pool of function instances ready to accept incoming requests [7]. However, this negates one of the main benefits of serverless computing, namely the option to dynamically scale to zero replicas, forgoing the flexible pricing in the process. Another common strategy is to reuse the same execution environment for multiple requests, which does eliminate the cold start latency for subsequent executions, but sacrifices per-request runtime isolation, thus reducing the security of the platform and its users.

A possible solution to the problem of cold start faced by serverless computing might not be the optimization of existing runtime environments which were initially meant to be executed in powerful datacenters in the cloud, but the introduction of a new technology tailored to more resource restricted devices. WebAssembly (WASM) [8] is a new runtime environment whose primary focus is to simplify and transform client-side web programming. Nevertheless, recent developments in this area have made WebAssembly a feasible choice for developing server-side applications, and as a result serverless functions as well. WebAssembly has the potential of drastically reducing the cold start times, while still offering competitive performance to other runtime environments, as well as cross-platform portability.

The goal of this paper is to explore the recent developments in the WebAssembly ecosystem and how they relate to the topic of serverless computing, both in the cloud and at the edge. The main contributions of this work are: 1) overview of the state-of-the-art research related to moving WebAssembly out of the browser and using it for server-side

978-1-6654-7273-9/22/$31.00 ©2022 IEEE

applications, including serverless computing and function as a service; 2) Report on current efforts to integrate WebAssembly with existing and popular container runtimes, exploiting their modularity and allowing interoperability between containers and WebAssembly modules; 3) Evaluation of two WebAssembly frameworks capable of orchestrating WASM modules in the cloud and at the edge.

The rest of this paper is organized as follows: in section II we discuss necessary background information related to WebAssembly and report on related work dealing with the performance of WebAssembly and its transition as a runtime for serverless functions. We then continue with section III where we present how recent advancements in the WebAssembly landscape can act as enablers for the next generation of serverless platforms, focusing on integration with existing container runtimes and WebAssembly orchestrators. We conclude the paper with section IV, outlining remaining challenges and future plans.

II. Background and Related Work

A. The Rise of WebAssembly

WebAssembly is a World Wide Web Consortium (W3C) standard first released in 2017 with the aim of improving the client-side performance of web applications. It offers a stack based virtual machine and can be used as a compilation target for many popular programming languages today including C, C++, Rust, and Go. It is expected that even more languages will be supported in the future, as the tooling matures. Interpreted languages can also be ported to a WebAssembly environment, by compiling the language interpreter itself as a WebAssembly module [9].

In theory it should be possible to compile existing source code from a supported language directly to WebAssembly, but in practice this might require additional effort, especially for more complex code bases. As a result of the strong isolation requirements for executing remotely downloaded and untrusted code in the browser, combined with the novelty of the WebAssembly ecosystem, a number of features which are taken for granted in modern high level programming languages have only recently been introduced, or are still in development, such as network sockets, file-system access, and threading [10]. Nevertheless, with the introduction of the WebAssembly Systems Interface (WASI) it is possible for WebAssembly runtimes to interact with the underlying operating system using POSIX style capabilities, and thus gradually add support for the currently missing features. WASI has also enabled WebAssembly to move away from the browser and to be used for standalone server-side applications and serverless functions.

As with any new technology undergoing an intensive standardization process, a number of WebAssembly runtimes currently exist today, all of which implement the base specification, but also introduce additional specialized features which have not been standardized yet, extending its capabilities even-further, and exploring new use-cases [11]–[13]. Some of these features also relate to improvements directly associated with serverless functions and web assembly module orchestration. Section III focuses on these developments in more detail, while the next subsection describes the state-of-the-art literature in terms of developing new and evaluating existing WebAssembly runtimes.

B. Related Work

As the number of supported features by WebAssembly runtimes increases, so does the research interest in this topic. Recognizing the potential for WebAssembly to be used for server-side applications, Long et al. [14] have conducted a study into the performance differences between Docker containers and WebAssembly modules. They have compared the cold start times of WebAssembly modules on three different runtimes (V8, Lucet, SSVM) to those of Docker containers. Their results show that WebAssembly modules running on the tested runtimes have at least 10 times faster startup compared to the same code shipped in a Docker container, as well as faster execution time and faster file input/output operations (I/O). WebAssembly runtimes especially optimized for use with server-side workloads such as Lucet and SSVM show even better results. Continuing the trend of WebAssembly performance evaluation, Hockley et al. [15] benchmark WebAssembly workloads against native execution. In this case, the presented results show that WebAssembly modules incur a 5 to 10 times performance penalty compared to native execution. Other works comparing WebAssembly performance to native execution have also been published, and while they also report worse performance when using WebAssembly, the slowdowns are significantly lower, up to 2 times [16], [17]. It should be noted that slowdowns compared to native execution are of course expected because of the additional layers of abstraction introduced by the WebAssembly runtimes.

Serverless computing has also been identified as a possible area where server-side WebAssembly with the help of WASI could be used. Murphy et al. [18] suggest ways in which WebAssembly modules support could be added to existing and well-established serverless platforms such as OpenWhisk or AWS Lambda. By invoking the WebAssembly runtime from native Node.js code, large modifications to the underlying serverless platform are avoided. Hall et al. [19] also demonstrate the feasibility of WebAssembly for serverless computing, this time at the edge of the network.

WASM, together with its cross-platform compatibility, portability, and efficient performance, has been identified by Mäkitalo et al. [20] as a mean for lightweight containerization of code for use in IoT scenarios in the future. They argue that with the provided cross-platform portability, WebAssembly should allow seamless migration of workloads from device to device, as needed.

In conclusion, current research has identified several distinct areas where WebAssembly could be used to solve existing issues or enable new use-cases. To the best of our knowledge, the currently published work focuses exclusively on either developing or evaluating standalone WebAssembly runtimes, without reflecting on the possible integration options with existing technologies. As with any new and emerging trend, a careful balance must be stricken not to overuse the new technology in scenarios which are not a perfect fit. Our vision is that while WebAssembly is certainly part of the solution which would allow efficient and easy-to-use serverless platforms to exist and would undoubtedly contribute to the overcoming of the current limitations of existing serverless platforms, it is not the sole component. A number of different technologies will need to be integrated together to realize the end-goal of designing a performant infrastructure which can target both the cloud and the edge, thus establishing an edge-cloud continuum. The motivation

behind this paper is to fill this gap and to provide an overview of recent developments in the area of WebAssembly which would allow it to be used in an orchestrated manner at a large scale, as well as integrated with existing and well-known container and micro VM runtimes.

III. Web Assembly as an Enabler For Next Generation Serverless Platforms

WebAssembly has a number of advantages that make it a suitable fit for use in serverless computing scenarios, including support from popular programming languages. Existing WASM runtimes can run on a wide variety of computing architectures ranging from cloud based x86 servers to ARM single-board computers (SBC) deployed at the edge, allowing function portability. Different WASM runtimes are tailored to different scenarios, favoring cross-compatibility, speed, or new features still not part of the official specification, further extending the possibilities of this new ecosystem.

A. Integrating Container and WebAssembly Runtimes

Containers have become the most popular runtime environment for FaaS today, used by both commercial and open source serverless platforms. Since their initial popularization with the release of Docker, a number of both high-level and low-level container runtimes have emerged, modularizing the original architecture and introducing a layered approach which enables easier extension in the future. This modularized architecture allows multiple different lower-level runtimes to be used by a single higher-level one using shims. The same approach can also be used to integrate one of the most popular high-level container orchestrators today, containerd, with third-party WASM runtimes.

To ease the creation of shims for popular WASM runtimes, the Runwasi Rust library has been published as open-source software by Deislabs [21]. The majority of WASM runtimes have first class support for the Rust language, which simplifies the usage of the Runwasi library, making it feasible to create a shim for a currently unsupported runtime. Once all containerd shims have been compiled and specified in its configuration, it is possible to instantiate workloads with the desired WASM runtime by simply specifying it on the command line interface of containerd.

Another approach taken by some container runtimes is to avoid shims and instead bundle support for WASM directly, during compilation time. Such an example is crun [22]. During compilation, the user can pass additional feature flags which would enable support for the chosen WASM runtime from the list of currently supported ones: Wasmtime [11], Wasmer [12], or WasmEdge [13]. The decision whether to instantiate a given workload either using the native container runtime or the WASM runtime is based on the usage of specific labels.

One major question that arises when integrating WASM runtime shims with container runtimes is how to overcome the problem of artifact distribution. Container images use the Open Container Initiative (OCI) image specification, while WASM modules are distributed as .wasm binaries by default. An elegant solution to this problem is to simply reuse the OCI image specification for distributing the .wasm binaries using a container image with only one layer that contains a single file – the .wasm binary. With this approach, there is no difference between a WASM workload and a standard container, at least from the perspective of image fetching and unpacking. Once the OCI image is downloaded, the specified WebAssembly runtime which is connected either using a shim, or directly integrated with the OCI runtime, can instantiate the workload. An added benefit is the fact that existing OCI image registries can be reused for distribution of WebAssembly modules, avoiding the need for setting up specialized distribution mediums.

It should be noted that integrating WASM with container runtimes in this manner ensures the execution performance of the WebAssembly modules, since they are in fact executed using the chosen WebAssembly runtime directly on the host operating system, and not bundled into an OCI container. Furthermore, the cold start latencies are severely reduced, while providing per runtime isolation for WASM workloads.

B. WebAssembly Frameworks, Tooling, and Orchestration

Before WebAssembly could be widely adopted as a target runtime for serverless functions, a number of open issues remain which are active research topics today. Many conventional serverless platforms have published dedicated libraries for popular programming languages, which do simplify the development of new functions, but also lead to vendor lock-in. The majority of these libraries bundle either an HTTP server which can be used to receive the original request which triggers the function or provide means to interact with an external message broker to realize the same functionality. This invocation pattern is overly complicated for WebAssembly, since support for network sockets is still in development for WASI (even though there are runtimes which already support them, such as WasmEdge). Furthermore, statically bundling such complex libraries in each module adds unnecessary computational and storage overhead which also impacts cold start latencies.

Another open issue which has also been identified [14] is the lack of orchestration tools for WebAssembly modules. A WebAssembly capable orchestrator is required to be able to deploy WASM modules at scale, across different hardware and even across different locations, spanning both cloud and edge infrastructures.

Finally, even though compilers for a number of popular programming languages do support WASM as a compilation target, debugging is still a largely manual process, and there is a need to improve this aspect of the developers' experience.

In the two subsections that follow we analyze how two WebAssembly frameworks, WasmCloud [23] and Spin [24], have dealt with these open problems.

1) WasmCloud – WasmCloud is a WebAssembly framework and platform which makes use of actors and capability providers. This approach allows it to solve the problem of bundling complex libraries with each .wasm binary. WasmCloud actors are the smallest deployable unit and they handle messages delived to them by the host runtime via capability providers. Actors can also invoke exposed functions of capability providers which have been explicitly assigned to them. Capability providers on the other hand represent code which is not part of the core business logic, such as an HTTP server, or a message broker. While actors are compiled to WebAssembly, this is not required for capability providers. WasmCloud supports orchestration of the deployed actors across multiple hosts and management through either a command line interface (CLI) tool or a web

portal. Both actors and capability providers are bundled and distributed via OCI images. Actors can currently be written in either Rust or Go, and hot-reloaded during development, thus easing debugging.

2) Spin – Spin is a new WASM platform and framework which is based on the Wasmtime runtime. It solves the problem of bundling complex dependencies not part of the business logic in one of two ways, depending on the source programming language used. For programming languages whose toolchains supports the WebAssembly Component Model, user provided functions can be dynamically linked to existing WASM modules, fulfilling the requirements for the supported invocation mechanisms: HTTP and Redis triggers. For all the other programming languages, an intermediary component called Wagi can be utilized which works similarly to common gateway interface (CGI) scripts. During an invocation, the provided parameters by the user are intercepted and are passed to the WASM module as standard input (stdin) parameters. All of the returned data on standard output (stdout) is part of the returned response which Wagi sends back to the invoking user. Orchestration is possible through the Fermyon Platform, which leverages the Hashicorp Nomad orchestrator to distribute WASM modules across different hosts part of the cluster. Instead of using OCI images as the distribution medium for the .wasm binaries, a new tool is introduced, Bindle, which allows every WASM module to specify its dependencies, and for these dependencies to be fetched without any user interaction.

IV. Conclussion

WebAssembly has come a long way since its initial release in 2017. The introduction of both the WASI and the WASM component model make it an enticing technology for developing server-side applications, including serverless functions. We have outlined recent developments in this area related to serverless functions based on WASM, as well as presented two frameworks, WasmCloud and Spin, which further extend the possibilities of WASM. Both solutions have clear practical advantages in terms of serverless computing, mitigating the cold-start problem while offering per runtime isolation of the executed workloads.

To fully realize a cross-platform serverless solution that can span both the cloud and the edge, while supporting multiple runtime environments which adapt to the performance requirements of the executed workloads, integration with existing technologies is required. In our vision for serverless computing, WASM should be an important piece of a wider ecosystem, where the current know-how and tooling related to containers and micro VMs as runtimes for serverless function should not simply be thrown away. Instead, serverless platforms should be flexible to choose the appropriate runtime environment depending on the use-case at hand and its latency requirements. Our future work will be centered around the design and development of such a platform, capable of orchestrating WASM workloads.

Acknowledgment

The work presented in this paper has received funding from the Faculty of Computer Science and Engineering under the “NSA” project.

References

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

[2] V. Kjorveziroski, S. Filiposka, and V. Trajkovik, ‘IoT Serverless Computing at the Edge: A Systematic Mapping Review’, *Computers*, vol. 10, no. 10, Art. no. 10, Oct. 2021, doi: 10.3390/computers10100130.

[3] V. Kjorveziroski *et al.*, ‘IoT Serverless Computing at the Edge: Open Issues and Research Direction’, *Transactions on Networks and Communications*, vol. 9, no. 4, Art. no. 4, Dec. 2021, doi: 10.14738/tnc.94.11231.

[4] L. Bittencourt *et al.*, ‘The Internet of Things, Fog and Cloud continuum: Integration and challenges’, *Internet of Things*, vol. 3–4, pp. 134–155, Oct. 2018, doi: 10.1016/j.iot.2018.09.005.

[5] M. S. Aslanpour *et al.*, ‘Serverless Edge Computing: Vision and Challenges’, in *2021 Australasian Computer Science Week Multiconference*, New York, NY, USA, Feb. 2021, pp. 1–10. doi: 10.1145/3437378.3444367.

[6] V. Kjorveziroski and S. Filiposka, ‘Kubernetes distributions for the edge: serverless performance evaluation’, *J Supercomput*, vol. 78, no. 11, pp. 13728–13755, Jul. 2022, doi: 10.1007/s11227-022-04430-6.

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

[8] A. Haas *et al.*, ‘Bringing the web up to speed with WebAssembly’, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelona Spain, Jun. 2017, pp. 185–200. doi: 10.1145/3062341.3062363.

[9] ‘WebAssembly Language Support Matrix’, *Fermyon Technologies (@FermyonTech)*. https://www.fermyon.com (accessed Aug. 29, 2022).

[10] ‘WebAssembly System Interface – Proposals’. WebAssembly. Accessed: Aug. 29, 2022. [Online]. Available: https://github.com/WebAssembly/WASI/blob/bac366c8aeb69cacfea6c4c04a503191bf1cede1/Proposals.md

[11] ‘Wasmtime’. https://wasmtime.dev/ (accessed Aug. 29, 2022).

[12] ‘Wasmer - The Universal WebAssembly Runtime’. https://wasmer.io/ (accessed Aug. 29, 2022).

[13] ‘WasmEdge’. https://wasmedge.org/ (accessed Aug. 29, 2022).

[14] J. Long, H.-Y. Tai, S.-T. Hsieh, and M. J. Yuan, ‘A lightweight design for serverless Function-as-a-Service’, *IEEE Softw.*, vol. 38, no. 1, pp. 75–80, Jan. 2021, doi: 10.1109/MS.2020.3028991.

[15] D. Hockley and C. Williamson, ‘Benchmarking Runtime Scripting Performance in Wasmer’, in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, New York, NY, USA, Jul. 2022, pp. 97–104. doi: 10.1145/3491204.3527477.

[16] A. Jangda, B. Powers, E. Berger, and A. Guha, ‘Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code’. May 31, 2019. doi: 10.5555/3358807.3358817.

[17] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, ‘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*, Jul. 2022, pp. 3–8. doi: 10.1145/3526059.3533618.

[18] S. Murphy, L. Persaud, W. Martini, and B. Bosshard, ‘On the Use of Web Assembly in a Serverless Context’, in *Agile Processes in Software Engineering and Extreme Programming – Workshops*, Cham, 2020, pp. 141–145. doi: 10.1007/978-3-030-58858-8_15.

[19] A. Hall and U. Ramachandran, ‘An execution model for serverless functions at the edge’, in *Proceedings of the International Conference on Internet of Things Design and Implementation*, New York, NY, USA, Apr. 2019, pp. 225–236. doi: 10.1145/3302505.3310084.

[20] N. Mäkitalo *et al.*, ‘WebAssembly Modules as Lightweight Containers for Liquid IoT Applications’, in *Web Engineering*, Cham, 2021, pp. 328–336. doi: 10.1007/978-3-030-74296-6_25.

[21] ‘deislabs/runwasi’. Accessed: Aug. 29, 2022. [Online]. Available: https://github.com/deislabs/runwasi

[22] ‘containers/crun’. Accessed: Aug. 29, 2022. [Online]. Available: https://github.com/containers/crun

[23] ‘wasmCloud Documentation’. https://wasmcloud.dev/ (accessed Aug. 29, 2022).

[24] ‘Introducing Spin’, *Spin Documentation*. https://spin.fermyon.dev (accessed Aug. 29, 2022).