TU WIEN Informatics

Eine WebAssembly Container Runtime für Serverless Edge Computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Philipp Gackstatter, BSc.
Matrikelnummer 11846040

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Univ.Ass. Pantelis Frangoudis, PhD

Wien, 18. Mai 2021

Philipp Gackstatter	Schahram Dustdar

TU WIEN Informatics

A WebAssembly Container Runtime for Serverless Edge Computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Philipp Gackstatter, BSc.
Registration Number 11846040

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Univ.Ass. Pantelis Frangoudis, PhD

Vienna, 18th May, 2021

Philipp Gackstatter | Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Philipp Gackstatter, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschlieÿlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Mai 2021

Philipp Gackstatter

Acknowledgements

I want to thank Prof. Schahram Dustdar and, in particular, Pantelis Frangoudis, PhD. for being open to the idea of this work and enabling me to write my thesis in a topic I am excited about. Thank you for your guidance and valuable feedback over the course of this work!

I am also grateful to the Rust language community as a whole, whose talks, books and blogs have gotten me into systems programming, which turned out to be a passion of mine. Without this initial spark, this thesis would not have happened. Finally, thank you to Markus Raab who facilitated my Rust journey at TU Wien and beyond, thereby laying a cornerstone of this work, too.

Kurzfassung

Serverless Computing ist durch die Abstraktion des Infrastrukturmanagements ein beliebter Teil des Cloud Computings geworden. Es ermöglicht automatisch skalierende Funktionen zu schreiben, wobei nur die verbrauchte Rechenzeit in Rechnung gestellt wird. Dieses Modell ist daher ideal um Fluten von Anfragen zu verarbeiten, wie sie etwa aus dem Internet der Dinge kommen. Allerdings können selbst die räumlich nahen Edge Computer latenzsensible Anwendungsfälle aufgrund eines signifikanten Performanceproblems nicht bedienen.

Beim ersten Aufruf einer Funktion muss dessen Ausführungsumgebung gestartet werden – üblicherweise ein Docker Container. Dieser *Kaltstart* benötigt mehrere hundert Millisekunden bis hin zu mehreren Sekunden bei parallelen Anfragen, was Serverless in Kombination mit Docker für diese Anwendungsfälle ungeeignet macht. Zudem erhöht der Betrieb dieser Plattformen auf Edge Computern mit begrenzten Ressourcen die Kaltstartzeit noch weiter.

Um dieses Problem zu beheben verfolgen wir einen radikalen Ansatz, bei dem Docker, die de-facto Standard Serverless Runtime, gänzlich durch eine leichtgewichtigere Technologie ersetzt wird. Jede Alternative muss eine ähnliche Sicherheit, Sprachunabhängigkeit und Performance bieten. WebAssembly ist eine aufkommende Technologie die diese Eigenschaften bereitzuhalten scheint. Daher untersuchen wir in dieser Arbeit die Eignung von WebAssembly in einer Serverless Container Runtime mit einem Fokus auf Edge Computing. Dazu modifizieren wir das Serverless Framework Apache OpenWhisk und wählen drei WebAssembly Runtimes aus, um unsere Container Runtime darauf aufzubauen. Insbesondere wird dabei dessen Design diskutiert, mit dem schnelle Kaltstarts bei hoher Performance erzielt werden können.

Wir vergleichen OpenWhisk in der Dockervariante mit den drei WebAssembly Container Runtimes im Hinblick auf Kaltstartzeit, Performance und Speicherverbrauch in umfangreichen Experimenten. Auf einem Raspberry Pi, einem typischen low-end Edge Computer, erreichen unsere WebAssembly Container Runtimes einen 99.5% schnelleren Kaltstart als Docker und einen 94% schnelleren Start auf Serverhardware. Docker's Performance wird stark von der Kaltstartzeit beeinträchtigt, während der Durchsatz von WebAssembly Containern nicht spürbar davon beeinflusst wird. In einem Lasttest mit CPU- und I/O-gebundenen Funktionen erreichen unsere Container Runtimes das 2,4- bis 4,2-fache des Durchsatzes von Docker. Insgesamt zeigt sich WebAssembly als geeignete Alternative für Serverless Plattformen durch den leichtgewichtigen Isolationsmechanismus, der schnelle Starts ermöglicht, sowie Unterstützung für vorzeitige Kompilierung zu schnellem nativen Maschinencode. Unser Ansatz ist daher ein aussichtsreicher Schritt hin zu latenzarmem Serverless Edge Computing.

Abstract

Serverless computing has become a popular part of the cloud computing model, thanks to abstracting away infrastructure management. It enables any developer to write functions that auto-scale, while only paying for the used compute time. This model is ideal for handling the unpredictable and bursty workloads in the Internet of Things. However, even the physically close edge computers cannot facilitate low latency use cases due to a significant performance issue.

The first invocation of a function requires creating its execution environment – typically a Docker container. This *cold start* can take hundreds of milliseconds or even multiple seconds under concurrent requests, making serverless with Docker unsuitable for these use cases. Running these platforms on edge devices with limited CPU power and memory available, increases cold start latencies even further.

To alleviate this issue, we follow a radical approach and replace Docker, the de-facto standard runtime for serverless computing, with a lighter-weight technology. However, any alternative must offer similar levels of security, language-agnosticism and high performance. A nascent technology, which claims to have these properties, is a target format called WebAssembly. In this work, we examine WebAssembly's suitability for use in a serverless container runtime for Apache OpenWhisk, a serverless framework, with a focus on edge computing. We choose three WebAssembly runtimes to build our container runtime on top of and present its design to achieve fast cold starts while retaining a high performance.

We compare the Docker-based OpenWhisk with the three WebAssembly container runtimes in terms of cold start time, execution performance and memory usage via extensive experiments in a testbed. On a Raspberry Pi, a typical low-end edge computing device, we find our WebAssembly container runtimes to achieve 99.5% faster cold starts than Docker, and 94% faster starts on server-grade hardware. Docker's execution performance is strongly linked to the cold start time, while the throughput of WebAssembly containers is not noticeably susceptible to cold starts. In a mixed CPU- and I/O-bound workload, our container runtimes achieve at least 2.4 and up to 4.2 times the throughput of Docker. Overall, we find WebAssembly to be very suitable for use in serverless platforms due to its lightweight isolation mechanism, enabling fast startups, as well as support for ahead-of-time compilation to fast, native code. Our approach is thus a promising step towards low-latency serverless edge computing.

Contents

CHAPTER 1

Introduction

1.1 · Motivation

With the growing number of devices at the network edge, the limitations of today's cloud computing model have become apparent. Low latency and data-intensive use cases cannot be facilitated by the cloud, due to its physical distance and the sheer amount of data that these edge devices would have to transmit – over a very finite amount of bandwidth [Asl+21]. Hence, application scenarios at the edge involving data analytics [Nas+17], machine learning [Rau+19] or augmented reality [BF19], require moving away from the cloud-centric model. The edge computing paradigm has emerged as a solution to address these issues. Applying the traditional cloud model to the edge, however, does not promise to be an effective solution. Due to the limited resources available, allocating resources like virtual machines or even containers in a fixed manner may prove to be too inefficient [BF19]. Instead, a model with a higher resource elasticity is required; one that can scale up and down fast, thereby efficiently using the available resources. One approach that implements this is serverless computing – most commonly in the form of Function-as-a-Service (FaaS). In this computing model, developers can execute functions without having to specify how the necessary infrastructure is set up. Instead, the allocation of computing resources is automated, which enables a higher elasticity. This property is well-suited to the resource- and energy-constrained environments of edge devices, given that overprovisioning is even less desirable than in the cloud model [Asl+21]. Because resources can be utilized whenever required, this model is ideal for event-driven architectures, such as those typically found in Internet of Things scenarios, where the time at which devices require computing resources is unpredictable, e.g. when a sensor receives input or a user presses a button. Therefore, it may happen that at times, a serverless framework receives only a few requests, while at others, it needs to process a large amount of requests, simultaneously. To enable this elasticity, one user's function might run on the same host as that of another user. To isolate these instances, serverless frameworks make use of OS-level virtualization through containers – usually of the Docker flavor. Next to security, it also enables packaging language

runtimes, so developers can write functions in a language-agnostic manner, and statelessness due to cheap create and destroy operations. However, cheap is relative. Historically, containers have been compared to the heavier-weight virtual machines, where the creation is indeed much more time intensive. However, on the time scale of serverless, where an incoming request ought to be finished as soon as possible, but the container needs to be created first, it turns out to be not so cheap after all. Not pre-allocating resources to avoid overprovisioning is the point of serverless, but that presupposes that the allocation itself is timely.

When a function is first executed, its container must be started – referred to as the cold start. This can introduce a latency of a few hundred milliseconds to multiple seconds [Man+18; Wan+18]. Consider that on average, 50% of serverless functions execute for less than one second [Sha+20]. Thus, the latency observed by the end user can often be double of what the function itself is responsible for. This infrastructure-induced latency is unacceptable for time-sensitive applications, and thereby most user-facing ones. To make matters worse, these cold start latencies increase under concurrent workloads – precisely those supposed to be handled well by serverless platforms [Moh+19]. Containers were not designed for this use case and the startup time they require is inherent to how they achieve isolation. This problem has been partially addressed by keeping containers warm in-between requests; that is, only the first of potentially many invocations incur this cost. However, this is a form of overprovisioning – though a less severe one – and therefore opposed to the serverless promise of being able to scale to zero. Evidently, this is already a significant problem for the cloud, so applying this approach at the resource-constrained edge, like Amazon Web Services' (AWS) Greengrass[1] would only exacerbate the issue. As Nastic and Dustdar point out, the »architectural and design assumptions behind such approaches need to be reevaluated« [ND18]. Consequently, to rectify this problem, a new approach is needed.

At the heart of the problem lies the container runtime, due to its expensive startup procedure. We believe OS-level virtualization to be unsuitable for serverless edge computing and thus follow a radical approach by replacing the container runtime entirely. This new runtime needs to provide similar functionality and guarantees, while offering a more efficient cold start. One nascent technology that could facilitate this is called WebAssembly (Wasm). WebAssembly is a portable, binary instruction format for memory-safe, sandboxed execution in a virtual machine [W3C20]. Its portability means that the compiled WebAssembly modules can be executed wherever a runtime exists. Thus, a function can easily be moved between architecturally heterogeneous devices. Its sandboxing features enable Wasm to be used for the execution of arbitrary, user-supplied code without compromising the security of a multi-tenant system. WebAssembly can be compiled with different strategies, some of them reaching near-native speeds. This is important since a running Docker container can execute at native speeds or as fast as the language runtime allows for. In the process of alleviating the cold start, we have to avoid suffering a loss in execution performance due to an expensive compilation strategy or runtime checks. Finally, these modules can be created and destroyed in a matter of microseconds. The cloud provider fastly has begun offering execution of Wasm code through the use of their custom `lucet` WebAssembly runtime.

[1] `https://aws.amazon.com/greengrass/`

In its announcement, they write:

> »Lucet can instantiate WebAssembly modules in under 50 microseconds, with just a few kilobytes of memory overhead. By comparison, Chromium's V8 engine takes about 5 milliseconds, and tens of megabytes of memory overhead, to instantiate JavaScript or WebAssembly programs« [fas19].

This gives us a first indication that WebAssembly runtimes does not suffer from a cold start on the same order of magnitude as Docker's. In summary, Wasm allows for secure, portable and fast execution. With these properties it may be a good replacement for containers on edge computers. Solomon Hykes, one of the founders of Docker, even commented:

> »If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing« [Hyk19].

Hence, WebAssembly's potential for serverless computing shall be examined more closely in this work.

1.2 · Aim of the Work

The goal of this work is to evaluate the viability of WebAssembly in serverless computing in the context of edge computing, while also attempting to accommodate the cloud. We assess that potential by gathering requirements from the current approaches to serverless, as well as those from edge computing, which supersede the current ones. Based on those, we study several WebAssembly runtimes for their suitability towards our goals. We then discuss two approaches, which would enable WebAssembly to be used in serverless, and implement one of them. Given that WebAssembly runtimes themselves are geared towards different use cases, we leverage three of them and aim to find the one most suitable for our needs. Ultimately, the work is guided by and seeks to answer the following research questions.

RQ1: By how much can WebAssembly runtimes alleviate the cold start latency?

The first research question examines whether our core hypothesis holds. That is, can our serverless container runtime provision WebAssembly containers more quickly than the traditional Docker runtime can? If so, how much faster, and with what optimizations in place? We evaluate that by measuring the cold start latency for both approaches on an edge as well as a server-grade device.

RQ2: How do the performance characteristics of the WebAssembly and Docker container runtimes differ?

The goal of the second question is to measure the performance of both container technologies at runtime. More specifically, at what speed can certain workload types be executed? We run CPU- and I/O-bound functions in isolation, to test performance in these specific aspects, but also mixed workloads to test scenarios closer to reality. Moreover, we take the cold start into account in some test runs while isolating it in others to give a holistic picture and examine the effect the cold start has on the overall performance.

RQ3: What are the resource costs for keeping containers warm in both container runtimes?

The third question explores, what the cost of not scaling to zero is, i.e. when containers are kept warm. This is reflected in the memory usage of such containers, which are being kept ready for potential incoming requests. That ties in with the cost of cold starts. If cold starts are cheap, we can reduce or eliminate the time that containers need to be kept warm. Orthogonally, if keeping containers warm is cheaper with WebAssembly than Docker, keeping more of them around is feasible to improve performance overall. Both measurements let us explore the space of configurations this keep-alive optimization has and compare it to that of Docker.

RQ4: Which WebAssembly runtime is the most suitable for use on edge devices?

The fourth question is directed at finding the WebAssembly runtime best suited for the execution of modules on low-energy and resource-constrained edge devices. We use a Raspberry Pi single-board computer as our example edge device to run our serverless environment on and evaluate the different runtimes from there.

RQ5: How suitable is WebAssembly for serverless execution?

The final question aims to make a conclusion about the suitability of WebAssembly itself, based on our evaluation of multiple runtimes, but not necessarily limited to it. All of the preceding measurements as well as our experiences in the design phase are amalgamated to produce an answer to this question.

1.2.1 Terminology

Since *runtime* is a very overloaded term in our context, we may also refer to container runtimes as executors, i.e. the programs responsible for managing containers. Runtime also appears as the word for language runtimes, for example, `node.js` for JavaScript. WebAssembly runtimes are virtual machines executing WebAssembly code, which is called a WebAssembly module. A

module can be instantiated to produce an executable instance. A WebAssembly container is a more generic term encompassing both a WebAssembly module and an instance.

1.2.2 Contributions

The first contribution is a review of the literature, to provide a background for serverless computing in the cloud and at the edge as well as a comprehensive overview of WebAssembly and its salient properties, which make it a viable alternative to Docker for our use case. That includes an assessment of the current research space as to what constitutes serverless use cases and typical workloads, in order to facilitate a realistic evaluation of our system.

The major contribution revolves around modifying an open-source serverless framework, Apache OpenWhisk, to enable the use of such a new container runtime. We explore two auspicious options for bringing WebAssembly support to OpenWhisk and settle for one. At the heart of the work is the implementation of the container runtime using three different WebAssembly runtimes. We discuss the properties the executor needs to fulfill and the design that aims to implement them. Subsequently, we evaluate the three resulting executors against the baseline OpenWhisk vanilla for the cold start latency, the performance under different workload types as well as memory usage for warm containers. Our system shows reductions in cold start latencies of 99.5% on a Raspberry Pi and around 94% on a server machine. The throughput in a realistic workload is up to 4.2× more on a server machine, and 3× more on a Raspberry Pi. Finally, we explore alternative approaches to alleviate the cold start problem and conclude with answers to the research questions we posed. The source code for our contributions can be found on GitHub[2].

1.3 · Structure of the Work

In Chapter 2 we expand on the introduced topics, to establish a common background for the remainder of the work, particularly the serverless workload and WebAssembly.

In Chapter 3 we discuss approaches for potential solutions to our problem and discuss the implementation of our proposed solution. We settle on one approach and implement it, work out the differences between the WebAssembly runtimes we leverage, and discuss trade-offs along the way.

In Chapter 4 we evaluate the system for the cold start latency, different workload types and memory usage in warm containers.

In Chapter 5 we examine the current research for other approaches that alleviate the cold start problem and discuss the differences to our approach.

In Chapter 6 we summarize the results of our work by answering our research questions and give an overview of open challenges and potential future work.

[2] https://github.com/PhilippGackstatter/wow

CHAPTER 2

Background

In this chapter we establish a common background for serverless computing and WebAssembly.

2.1 · Serverless Computing

2.1.1 Function-as-a-Service

> »Serverless computing is an emerging cloud-based execution model in which user-defined functions are seamlessly and transparently hosted and managed by a distributed platform« [Nas+17].

Most developers know serverless computing in the form of Function-as-a-Service (FaaS). The major cloud providers have FaaS offerings, such as AWS Lambdas[1], Google Cloud Functions[2], Microsoft Azure Functions[3] or IBM Cloud Functions[4]. They seamlessly and transparently host the functions for the users without them having to actively manage scaling or administer infrastructure. Fox et al. point out, that these offerings are more appropriately described as »server-hidden«, since these platforms hide how functions are run or how scaling is done, and since there are, in fact, servers, but hidden ones [Fox+17]. They see serverless as a natural evolution from running applications on bare metal, then in virtual machines, later in containers and now as functions. They point to the convenience of the application developer not having to worry about provisioning infrastructure and the associated scalability concerns, as the major advantage of serverless. Castro et al. observe a similar perspective: serverless is the

[1] https://aws.amazon.com/lambda/
[2] https://cloud.google.com/functions/
[3] https://azure.microsoft.com/en-us/services/functions/
[4] https://www.ibm.com/cloud/functions

successor to Platform-as-a-Service (PaaS) such as Google's App Engine, but it improves on PaaS by charging users only for what they use – the »pay-as-you-go« model [Cas+19]. As much as serverless is an evolution of this area of cloud computing, it is an addition in others. Functions are essentially stateless; and they have to be programmed with that context in mind, due to their nature of potentially being destroyed and re-created at any given time. Since pure functions by themselves are not very useful, they are often combined with other offerings of the Backend-as-a-Service model, such as databases or event buses [Eis+21a]. One concrete example is the AWS API Gateway[5], a managed service for creating APIs. The serverless AWS Lambdas can handle requests to endpoints of such an API, i.e. each endpoint could be mapped to a different Lambda. This can act as a replacement for the traditional monolithic web server to implement backend APIs following the concept of microservices.

McGrath and Brenner describe serverless as the manifestation of the idea, that applications are defined by events and the actions they trigger [MB17]. Events can be a rule firing, a file being uploaded to a storage system or an edge device measuring sensor data and posting it via HTTP. If many events occur at once, the serverless platform is able to scale-up to meet the demand. Serverless is also ideal if no events occur, because no cost for the user is incurred. Serverless platforms are able to offer that, because they follow the scale-to-zero principle: functions that are not used do not take up resources. In turn, these resources can then be used for other function executions, exemplifying the principle of elasticity.

2.1.2 Serverless Architecture

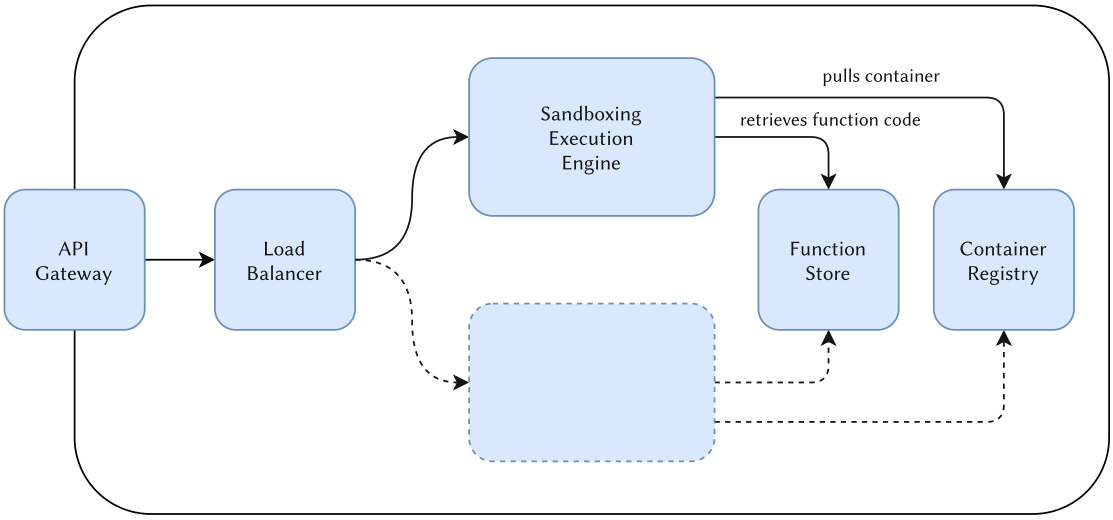


Figure 2.1: A generalized serverless architecture based on OpenWhisk, OpenFaaS and OpenLambda.

[5] https://aws.amazon.com/api-gateway/

Based on popular open-source serverless frameworks we can infer what a typical serverless architecture looks like. We use OpenFaaS[6], Apache OpenWhisk[7] and OpenLambda[8] as our basis and sketch this generalized architecture in Figure 2.1. The system is exposed through an API Gateway. A user can interact with the system through the gateway and manage (i.e. Create, Read, Update and Delete) functions, set up scheduled triggers or retrieve the results of previous invocations. Once a function is created and an execution request comes in, the load balancer will pick an appropriate host for execution. Each of these runs a sandboxing execution engine, such as Docker or a similar container runtime, which runs user functions in an isolated manner. Serverless frameworks allow developers to write their functions in a variety of languages; they are almost language-agnostic. Since many popular languages need a runtime, such as `node.js`, a Python interpreter or a Java `JRE`, these runtimes are baked into container images. In order to execute a function, the container image with the appropriate runtime is pulled from a container registry. The function's code is retrieved from an internal database. The container is started and the code is injected and prepared for execution. Finally, the function is invoked with the user's parameters and the result returned to the framework, which in turn returns it to the user. Typically, frameworks implement optimizations such as pre-pulling images onto the hosts or keeping started containers around for some time, to avoid the costly first startup on subsequent invocations. Here, we see a discrepancy between the aforementioned ideal of scaling to zero and real world implementations. If post-invocation, containers are still running but users are no longer billed for them, the incurred cost falls to the platform operator. An ideal framework would be truly scale-to-zero, but there is one issue that prevents them from being so.

2.1.3 Cold Start

In the context of serverless computing, a cold start refers to the invocation of a function, when all of the necessary resources for its execution must be provisioned from scratch. A cold start thus delays the execution of the function itself, by the amount of time it takes for the resources to become ready. This is what we call the cold start latency or cold start time. Wang et al. have measured the cold start latency at major cloud providers such as AWS and Google Cloud Platform [Wan+18]. The cold start time depends on the allocated memory, from which the CPU allowance is determined. The median cold start latency for AWS was between 250 ms (1536 MB) and 265 ms (128 MB) and between 110 ms (2048 MB) and 493 ms (128 MB) for Google, both depending on the amount of memory assigned to the instance.

Cold starts prevent serverless platforms from rapidly scaling to zero in practice. If every function would be executed in a newly provisioned container, subsequent invocations of the same function would all incur the cold start debt and would not be performant enough. Thus, typical optimizations are to keep these resources provisioned, such that repeated requests are faster to execute. Apache OpenWhisk, for instance, keeps a function's container paused and

[6] https://www.openfaas.com/
[7] https://openwhisk.apache.org/
[8] https://github.com/open-lambda/open-lambda

ready for reuse for 10 minutes, before removing it entirely. AWS Lambda's cold start policy has been reverse-engineered and the findings indicate that an instance stays alive for a duration of 5 to 7 minutes, during which no cold start will take place [Shi21].

Cold starts are an issue for latency-sensitive applications, where additional hundreds of milliseconds are unacceptable. Cui gives examples such as food ordering services, e-commerce sites or social networks, which all have unpredictable bursts of requests at certain points in time. In serverless versions of these applications, these bursts would translate into multiple concurrent cold starts, of which each individual one is slower than a singular, non-concurrent cold start. Thus, the numbers cited above increase significantly with higher concurrency [Moh+19; Cui18] – something that auto-scaling platforms ought to handle particularly well.

2.1.4 Serverless Edge Computing

Judging by the increasing adoption of the IoT, we are »entering the post-cloud era«, as Shi and Dustdar write [SD16]. The edge computing paradigm promises to be a solution to the massive amounts of data that will be generated at the edge. A wearable body sensor measuring health data for example, can use the wearer's smartphone – here, the edge device – for data processing rather than sending the workload all the way to the cloud. In general, edge devices are machines, that sit between the data source and the cloud. This processing close to the data source has a number of advantages, such as enabling very short response times as well as working with privacy-sensitive data. This is necessary, since sending all data to the cloud has comparably higher latency and puts more strain on the wide-area network.

One piece of the puzzle towards realizing this paradigm is serverless edge computing. Aslanpour et al. lay out the vision for serverless edge computing, of which we highlight two points [Asl+21].

- Due to IoT devices' unpredictable workload, an under- or overprovisioning of static resources such as VMs or containers would be likely. Serverless with its ability to scale to zero, even with aboves caveat, provides more precise provisioning of resources and only incurs cost when used, bringing advantages for operator and user, respectively. Executing serverless workloads in an energy-constrained setting, such as on single-board computers (SBC), this becomes doubly important.
- Serverless computing tends to be used for unpredictable, bursty workloads that require immediate up-scaling. Application architectures for IoT devices are usually event-driven and often unpredictable, making them a perfect fit for serverless.

Serverless computing can be the enabler for edge computing. But some of its original design decisions need to be re-examined, as Nastic and Dustdar point out, due to the »[...] inherently different nature of Edge infrastructure, for example, in terms of available resources, network, geographical hyper-distribution, very large scale, etc« [ND18].

IBM has reacted to these changed requirements and introduced Edge Functions on the IBM Cloud Internet Services [Far+19]. This offering allows developers to run code closer to the user

by deploying functions to their edge network. As a consequence of the resource constraints of edge devices, their offering uses the JavaScript V8 engine and its *isolates* feature. This is a similar step as moving to a WebAssembly executor, i.e. using a less costly sandboxing mechanism than containers to account for the resource constraints on the edge. Some of their serverless edge computing use cases include »personalized user experiences through conditional routing and originless responses.« The latter means returning responses directly from the edge without going all the way to the cloud [IBM21]. This leads to another use case: increased API responsiveness through aggregation and caching from multiple cloud-based API endpoints.

2.1.5 Serverless Workload

Since we aim to redesign the fundamental building block of a serverless framework, namely the underlying container runtime, it is important to examine what the serverless workload looks like. Here, we provide an overview of the research in this area.

Shahrad et al. have characterized the serverless workload for a large cloud platform: Microsoft Azure [Sha+20]. Because of their large number of users, this study can be considered a very good representation of the average serverless workload in the cloud. They find that, on average, 81% of functions are invoked less than once per minute. However, those accessed more frequently make up 99.6% of all invocations. Those frequently accessed functions should thus be kept in memory, to avoid the cold start entirely. The less frequently accessed functions should not be kept in memory, but created, executed and destroyed immediately, in order to save resources. For this to be viable, the cold start needs to be a cheap operation. In general, the cheaper the cold start, the smaller the amount of time that functions need to be kept in memory. Customers are only billed for the execution time of the function in FaaS. Operators bear the cost of keeping functions in memory – or resources warm, more generally. Thus, shorter cold starts reduce costs.

Shahrad et al. also find that on average, 50% of functions execute for less than one second.

> »The main implication is that the function execution times are at the same order of magnitude as the cold start times reported for major providers. *This makes avoiding and/or optimizing cold starts extremely important for the overall performance of a FaaS offering*« [Sha+20].

This is corroborated by the cold start times of Wang et al. cited above. Because of these findings, the primary goal in this work is that of reducing the latency of cold starts and make them a more efficient process, in turn enabling reduced keep-alive times. Even in light of a cheap cold start, we still need to make sure that we have a resource-efficient way to keep the function in memory. Consider that the most frequently accessed functions make up an overwhelming share of invocations. It is unlikely that we achieve a reduction of cold start times to such a degree, that always cold starting these functions will be more efficient than a short keep-alive

period allowing a reuse of an already initialized function. Thus, unless we can make calling a cold function as efficient as calling a warm one, we will need some amount of keep-alive. This is again a violation of the scale-to-zero principle, although a less severe one, since we can get away with a smaller period. As Shahrad et al. point out, there is a fundamental trade-off between using memory to keep functions warm, and forgoing it, but incurring the additional latency of a cold start. Because of that, we also examine the memory cost of our approach since it has implications for the keep-alive policy.

Eismann et al. conducted a study on why companies adopt serverless, for which applications it is suited best and how they are implemented [Eis+21a]. They analyzed 89 open-source serverless applications, of which 84% have bursty workloads. Thus, these spikes in demand – manifested in concurrent requests, in turn causing up-scaling – ought to be handled well by serverless platforms. They find 69% of the applications have a data volume of less than 10 MB. Thus, memory-bound applications are likely not a primary use case. Furthermore, 39% of the applications have a high traffic intensity, 47% have a low traffic intensity and 17% utilize scheduled functions. Thus, the latter combined 64% represent on-demand scenarios where the platform will experience cold starts, since it is less likely that a warm function exists in those cases. According to the authors, 88% of the applications make use of Backend-as-a-Service (BaaS) offerings. Storage, databases and messaging services are the most popular ones. Leitner et al. carried out a mixed-method study based on grey literature, interviews with serverless practitioners and web-based surveys, to provide an overview of the state of serverless [Lei+19]. They find that developers who successfully adopted serverless have a different mental model than traditional web-application developers. The former design the application as a composition of standalone, external components, which they have not written themselves. Similar to Eismann et al., they also find developers use databases (78%), API gateways (69%) or logging services (66%) in conjunction with their serverless function. They argue that the real power of serverless comes from its ability to glue together various existing cloud services. Typical use cases, according to Leitner et al., include processing application data (76%), like transforming images; performing scheduled jobs (64%), like notifications or backups; or processing monitoring and telemetry data (39%). We can infer application data processing to be potentially CPU-bound, while the other examples are more likely I/O-bound, i.e. glueing external services together.

From the large number of applications using BaaS, we can infer that these functions will make requests to external services and thus spend the majority of time waiting for network I/O to complete while not actively utilizing the CPU. More generally, network I/O-bound workloads are a primary type that serverless platforms should be able to handle well. This is further corroborated by the languages typically used to write serverless functions. Eismann et al. report that the overwhelming share of serverless functions is implemented in JIT-compiled or interpreted languages such as JavaScript (42%) and Python (42%). Data from dashbird.io[9], a serverless monitoring tool, underpins the assumption that these languages are dominant. Of their 3000+ users, 76% use JavaScript, 13% use Python and 8% use Java [Reh19]. We can see that `node.js` in particular is a highly used engine for serverless. It is best known for its

[9] `https://dashbird.io/`

event-driven I/O; less so for making the most efficient use of the CPU. This has two implications. One is, that CPU-bound tasks are not the primary serverless use case. If this were the case, we would likely see a much higher share of compiled languages. Although a language like Python can be extended with performant modules written in C, the dashbird data in particular indicate that `node.js`-level performance is almost always good enough. The other is, if a new container runtime can be as fast as these interpreted or JIT-compiled languages, the new platform does not suffer from a loss in performance and, if it can be even faster than those, it even has benefits for CPU-bound applications to move to the new platform.

From this research, we might conclude that the current serverless platforms already fulfill these use cases. However, selection bias tells us that the current representation plays towards the strengths of the current approach and would not include use cases which are not working in this environment in the first place. Indeed, interviewees from the Leitner et al. study explicitly and repeatedly voiced concerns about building user-facing applications because of the high response times in case of cold starts. The authors concluded that applications requiring high performance or real-time requirements are not suitable to be built on serverless platforms – at least, not yet. This leads back to the cold start being a real problem that not only incurs resource costs, as explored previously, but, more importantly, completely inhibits certain use cases. However, it also shows that the lack of high performance in serverless may be an inhibiting factor as well. With that in mind, we can neither exclude CPU-bound applications from our focus, nor make them the primary one.

2.2 · WebAssembly

For the longest time, JavaScript was the sole client-side language in web browsers. With the rising popularity of the web platform and its growing number of APIs, more and more complex web apps were written, and increasingly in other programming languages. Necessarily, these languages had to treat JavaScript as a target format, like Java bytecode or machine-level assembly, in order to run on the web. Of course, JavaScript was not designed for this and so performance was lacking.

In 2013, a solution to this problem was introduced by engineers at Mozilla, aptly named `asm.js`. It restricts itself to the parts of JavaScript that can be optimized ahead-of-time [HWZ14]. It could be used to compile a C/C++ program to the `asm.js` target format to execute it with a JavaScript runtime faster than the equivalent JavaScript program would be. Benchmarks even showed it to run no more than 1.5× slower than native code [ZN13].

Finally, WebAssembly (Wasm) was born out of `asm.js` in 2015, with more layers of optimizations. It is a portable and universal binary instruction format for memory-safe, sandboxed execution in a virtual machine. It is possible to write programs in a variety of languages like C, C++, Rust, AssemblyScript, C#, Go, Swift and many more, and compile them to Wasm, which finally relieved JavaScript of its role as the universal target format of the web [W3C20].

2.2.1 Performance

Compared to `asm.js`, the Wasm binary format is smaller in size (10-20%) and faster to parse, by an order of magnitude [Cla19]. It is 33% faster than `asm.js` on average [Haa+17]. Jangda et al. ran the PolyBenchC benchmark suite to compare Wasm to native code and found that 13 of the 24 benchmarks performed within 10% of native code [Jan+19]. However, they argue that the roughly 100 lines of code long benchmarks are not fully representative of typical use cases. They run the SPEC CPU benchmark suite, which are significantly larger POSIX applications. On average, they find Wasm to be slower than native by 1.55× in Firefox and 1.45× in Chrome. Wasm modules tend to be smaller than native `x86_64` binaries. On average, Wasm modules have 85% of the size [Haa+17].

Serverless functions, by design, are microservices and thus limited in scope. Developers are compelled to write short and efficient functions, since they are billed for the execution time. The PolyBenchC benchmark results are thus likely more representative than the SPEC suite. However, this assumption is good enough and will be an aspect of our evaluation, regardless. To increase performance and potentially startup time, Wasm modules can be compiled to native code, either by JIT engines at the time of execution, or ahead-of-time by the same JIT engines or AoT compilers. This process produces an artifact specific to the WebAssembly runtime that compiled it. At this point, the universality of the Wasm format is lost. Doing the translation to native code ahead-of-time means the compilation process does not take up time when we need to execute. Consequently, this is an interesting technique to reduce the time it takes to get a WebAssembly module ready for execution.

2.2.2 Security

Wasm defines two important goals for security.

> »(1) protect users from buggy or malicious modules, and (2) provide developers with useful primitives and mitigations for developing safe applications, within the constraints of (1)« [W3C20].

To that end, Wasm uses fault isolation techniques to sandbox the executing module. Interaction with the host environment is only possible through imported functions [W3C20]. This mechanism represents a »safe foreign function interface« as it can communicate with the outside environment, but not escape the sandbox [Haa+17]. This is an important security feature of Wasm. A module without imports is side-effect free from the host perspective. Even printing `Hello World!` requires importing a `print` function. Similarly, writing to files, network sockets or reading the clock requires imports, which will be discussed more in the next section.

In Wasm, all memory access is confined to a module's *linear memory*. This memory is separate from the code space, which prevents programs from overwriting instructions. Programs can

only operate in their own execution environment, but cannot escape. This means Wasm runtimes can safely execute multiple untrusted modules, with their own linear memories, in the same process memory space and without requiring additional isolation [Haa+17]. In particular, »Reading and writing to arbitrary memory locations cannot be expressed in WebAssembly«, because Wasm's memory instructions work with offsets rather than addresses. Furthermore, bounds checking at runtime ensures the instructions write only to the linear memory [Den19]. This is a key aspect that enables Wasm as a lightweight container technology. Only a single instance of a runtime is needed to execute many (e.g. serverless) functions.

In the assembly produced by compiling a C program, the call to a function is expressed as jumping to the address of the function's first instruction. In exploits of such programs, these addresses are often crafted or changed by malicious actors to take control of the program's control flow. In WebAssembly on the other hand, a function is represented as an index in a table, i.e. it introduces an additional level of indirection to express the address. That adds to security, since an arbitrary address to a Wasm function cannot be crafted with Wasm instructions [Den19]. And since the program's memory is separate from the function instructions, an attacker also cannot overwrite them. Wasm thus features control flow integrity, because it enforces structured control flow. Jump targets and other properties are validated in a single pass before a module is instantiated. In particular, validation prevents jumps from targeting arbitrary locations. This validation must happen, according to the Wasm specification, before a module is executed by a runtime [Haa+17].

Lehmann, Kinder, and Pradel provide a critique of WebAssembly's binary security [LKP20]. Many mitigations that have made `x86` binaries safer, are not present in WebAssembly. According to them, this »re-enables several formerly defeated attacks«. Some of the important points include the following:

- Because all data such as constants, the call stack and the heap are in Wasm's linear memory, and any offset from zero up to the current memory's size is valid, overflows can corrupt any of these. In native binaries on the other hand, these are separate sections protected through unmapped pages, such that an overflow would result in program termination.
- Buffer overflows on the stack or heap, as well as stack overflows themselves can be used to obtain a write primitive. Native binaries prevent stack overflows from corrupting data with stack canaries [Cow+98].
- There is no address space layout randomization in Wasm, which would make exploits harder to execute, since a pointer to the stack or heap needs to be obtained first.
- Wasm programs ship with their own memory allocator, whose size is important since Wasm is often transmitted over the network. Smaller implementations of allocators such as `emmalloc` re-enable attacks against allocator metadata.

Wingo points out that writing software in memory-safe languages, such as Rust or garbage-collected ones, is »a comprehensive fix to this class of bug« [Win20]. C and C++, on the other

hand, are not memory-safe languages. Thus, while the criticism above is valid for programs written in memory-unsafe languages, it does not apply to memory-safe programs. But there is more subtlety to it, since, for example, Rust programs may also link to C programs, which are always considered memory unsafe by the compiler. They can also include blocks of `unsafe` code, where developers can generally use operations that the compiler cannot prove to be safe like dereferencing raw pointers. In both cases the compiler cannot guarantee the program to be memory-safe. Whether WebAssembly should include mitigations for issues, that certain classes of programs do not need, or whether they have to be added to the appropriate compiler toolchains, remains to be discussed.

The presented vulnerabilities apply to a Wasm program itself, but do not enable compromising the host system. Thus, to the best of our knowledge, the current research does not show breakage of WebAssembly's isolation. With both performance and host security under its belt, as well as being a portable and universal target format, Wasm turns out to be useful not just in the browser, but also outside of it.

2.2.3 System Interface

Wasm targets an abstract machine, so it needs an interface to the system. In the web browser, that works by calling into the browser through JavaScript glue code. Outside of it, a new solution was needed: The WebAssembly System Interface (WASI). This interface was designed with Wasm's goals – portability and security – in mind. WASI is binary-compatible, which means that Wasm binaries are portable between different concrete systems like Linux and Windows, but also browsers. The Wasm runtimes effectively implement that interface and translate it to the underlying concrete system. WASI also innovates in security, over the classic coarse-grained access control. It specifies a fine-grained capability-based security model, that consists of two parts. First, WASI makes use of the aforementioned import mechanism of Wasm, to adhere to its security model. The Wasm module cannot directly call an OS system call due to sandboxing, but imports equivalent WASI functions instead [Cla19]. For instance, a module that wants to open a file needs import statements like these, given in the WebAssembly text format:

```
(import "wasi_snapshot_preview1" "fd_read" (func (;5;) (type 8)))
(import "wasi_snapshot_preview1" "path_open" (func (;7;) (type 22)))
```

The runtime needs to supply those imports, or the module cannot run. In effect, the runtime and module need to be in consensus about which functions can be accessed. In a serverless context for example, the cloud provider could grant or refuse access to certain functions, enabling security policies on a per-function basis. However, even that is not fine-grained enough. Since WebAssembly does not use virtualization, all modules share the same resources, like the file system. Therefore, the cloud provider might only want to allow serverless functions to cache some data temporarily in a function-specific `/tmp/<function_id>/` directory, but not allow opening `/etc/passwd`. This is where WASI takes ideas from capability-based security.

In order to open a file, the module needs to call functions accessing the file, like `fd_read`, with a preopened file descriptor [Cla19]. The Wasm module then has the *capability* to access *that* file, but not others. The file can be opened by the Wasm runtime itself, or the program that embeds the runtime. That file descriptor is stored in a special WASI module, which the runtime – similar to a dynamic linker – links to the Wasm module from where it imports the functions. Again, this gives the embedder, e.g. a cloud provider, the chance to customize the exact capabilities on a fine-grained level.

2.2.4 Execution

Just like Java Bytecode needs a JVM, Wasm needs a runtime as well. Since WebAssembly itself is just a specification, we need a concrete implementation to execute a module. There are many implementations available, the most used ones likely being those in the major browser JavaScript engines. However, since we do not need support for JavaScript in this work, we choose to use standalone runtimes. The absence of JavaScript support may also provide a smaller runtime footprint. Those come optimized for different use cases and different compilation strategies. Wasm can be interpreted, just-in-time compiled or compiled to native code, ahead-of-time. All of these have different properties in terms of their cold start latency as well as their runtime performance. We discuss and select concrete runtimes in Chapter 3.

CHAPTER 3

Design

In this chapter we lay out the requirements that an implementation of a WebAssembly container runtime should satisfy in general. We introduce several WebAssembly runtimes and pick three of them for our implementation. By looking at OpenWhisk specifically, we expand on the requirements and the parts that need to be modified in order to enable our new container runtime. Other approaches that could be used to run WebAssembly workloads are also detailed. Finally, we describe and discuss our concrete implementation in Apache OpenWhisk, how to achieve fast cold starts and high performance, while also examining the trade-offs involved.

3.1 · Core Approach

As we have seen in the previous chapter, there are a number of issues with serverless in general, but also serverless on the edge. In this section we summarize the core problem and describe our idea for rectifying these issues to make serverless (edge) computing more viable.

A substantial cold start latency has emerged as the paramount issue of the serverless platform. It prevents them from implementing true scale-to-zero, since functions are kept warm after invocations, wasting energy and resources. While we may not be able to get rid of the keep-alive completely, a significant reduction in cold start latency would allow for a significant reduction in keep-alive time. Since energy and resources are even more precious on typical edge devices, this is also very important for their quality of service. Average cold start times are on the same order of magnitude as average function execution times, thereby significantly impacting the quality of service a serverless platform can offer. Moreover, serverless is employed by developers for bursty, unpredictable workloads, a pattern that is also common at the edge. In this scenario, we see concurrent requests, which translate to concurrent cold starts. The more concurrent cold starts in Docker, the longer each one takes [Cui18]. Operators are forced to choose between saving cost and resources or enabling a high quality of service – in particular shorter response times. While this trade-off is inevitable to some degree, the cold start exacerbates it. A new

container runtime must significantly alleviate that initial latency, to provide mutual benefits for the serverless user and the operator.

Serverless enables the execution of »polyglot tenant-provided code« [ND18]. A new runtime must keep up these principles of being programming language agnostic to continue enabling all developers to use the serverless platform. To allow for multi-tenancy on the same physical host, containers need to be sandboxed securely. As described in the previous chapter, WebAssembly provides sandboxing and support for many languages with support for more likely to come. At the same time, it is a lighter-weight solution than virtualization with Docker containers. In contrast to Docker, its isolation does not depend on operating system features, which is where the bottleneck for Docker originates [Moh+19]. Thus, it seems like a good candidate to replace Docker containers in serverless platforms while alleviating the cold start latency. This is also why WebAssembly may fare better on low-power edge devices.

Mendki compared WebAssembly and Docker container startup times and found WebAssembly being 91% faster than containers and used 75% less memory [Men20]. Hall and Ramachandran found their WebAssembly-based serverless platform fared 60% better, on average, than the Docker container solution, for their concurrent »Multiple Client, Multiple Access« workload [HR19]. These promising results validate the core idea of the approach.

3.2 · Requirements

Some of the requirements for a WebAssembly container runtime are a direct consequence of the issues we have assessed. We expand on others in the following sections. The summary of concerns and requirements are:

1. Be easily integratable with an existing serverless framework.
2. Be programming-language agnostic.
3. Provide a cheap sandboxing mechanism to ensure multi-tenant capability and no adverse effects on the cold start latency.
4. Be as close to native speed as possible to be a viable alternative to existing container runtimes.
5. Run on devices with potentially different instruction set architectures, at least `x86_64` and `arm`.
6. Be able to efficiently handle a large amount of I/O concurrently.
7. Be thread-safe.

Finding a suitable serverless framework that we can modify and integrate with, is the topic of the following section. In Chapter 2 and the preceding section, we have seen how WebAssembly

provides some of these requirements, such as sandboxing, potential for high execution speed and language-agnosticism. However, for our implementation we need concrete WebAssembly runtimes that provide these properties, so we will explore the options for such runtimes and work out the differences. The choice for which language and libraries to implement our executor with will be driven by how the Wasm runtimes we choose can be embedded. Other concerns include having access to highly performant, concurrent I/O and support for correctness guarantees, including thread-safety.

3.2.1 Apache OpenWhisk

We use Apache OpenWhisk as the framework to implement WebAssembly support for. OpenWhisk is a production-ready framework, used in the commercial IBM Cloud Functions. Thus, implementing Wasm support in OpenWhisk would show the viability of the approach in the real world. The framework is also frequently used by serverless researchers to implement new ideas to reduce cold start times [Moh+19] or Wasm support via `node.js` [HR19], making it a well-respected and frequently used framework. OpenWhisk also has *lean* components, which are lighter-weight replacements for its heavier ones, such as Kafka. These enable OpenWhisk to be deployed on edge devices where the available memory is limited, such as Raspberry Pis, on which we plan to evaluate our implementation. Furthermore, because OpenWhisk is designed to be easily extensible with new languages, the language runtime protocol is well-refined and straightforward to implement.

3.3 · WebAssembly Runtimes

WebAssembly is a specification[1] and implementations adhering to it are WebAssembly runtimes. Since Wasm was originally designed for the web, the virtual machines of browsers, such as Google's V8 or Mozilla's SpiderMonkey, can also execute WebAssembly in addition to JavaScript. With the rise of Wasm outside the web, a large number of standalone runtimes have emerged and many are actively being developed. This section introduces a number of runtimes with different compilation strategies and detail the differences. Each Wasm runtime we choose ultimately results in a separately usable container runtime, while sharing a common wrapper which implements the OpenWhisk communication layer.

3.3.1 Wasmtime

Wasmtime[2] is a runtime that has support for just-in-time compilation, available for the `x86_64` and `aarch64` instruction set architectures (ISA). It is written in Rust and thus easily embeddable from there. Support for the latest WASI standards and future proposals for Wasm are

[1] `https://webassembly.org/specs/`
[2] `https://github.com/bytecodealliance/wasmtime`

implemented. It is developed as part of the Bytecode Alliance[3], with founding members like Mozilla, fastly, Intel and RedHat. The `cranelift` code generator is developed as part of the alliance and `wasmtime` uses it for just-in-time compilation. The code generator's documentation points out that the backend for `x86_64` is »the most complete and stable; other architectures are in various stages of development [All21].« Thus, it will most likely run on an `aarch64` Raspberry Pi, but performance may not be on-par with an `LLVM`-based runtime.

3.3.2 Lucet

Lucet[4] is the compiler and runtime used for cloud provider fastly's Terrarium, a platform for running WebAssembly on edge devices [fas19]. It consists of a compiler and a runtime to execute WebAssembly. The compiler, `lucetc`, compiles Wasm modules to native object or shared object files. The complementary `lucet-runtime` can then load the native code and call the contained Wasm functions. Because of this compilation model, execution can reach near-native speeds. However, the execution is not sandboxed by default, requiring mechanisms like `seccomp-bpf`[5], and the security model is actively being developed. It is currently only available on the `x86_64` ISA. Since we require support for smaller edge devices, often built on the `arm` ISA, we will not be using `lucet`. Furthermore, the lack of mature sandboxing makes other runtimes a better choice. However, the idea of separating the compilation and execution is appealing to enable short startups and fast execution, and is applied by us, but using other runtimes.

3.3.3 Wasmer

Wasmer[6] is a WebAssembly runtime with support for the three major platforms – Linux, Windows and macOS – and `x86_64` and `aarch64` ISAs. Notably, `wasmer` has already reached version `1.0`, indicating a certain level of maturity and stability. The runtime offers three different compilers, `singlepass`, `cranelift` and `LLVM`, whose compilation times get slower and execution times get faster, in that order, respectively. The runtime has a JIT and a native engine, which differ in the way the module is loaded. In particular, the native engine produces a shared object, which can be loaded with a fast `dlopen` call. The support for high-quality AoT compilation with `LLVM` and fast startup times makes `wasmer` a promising choice for our purpose.

[3] `https://bytecodealliance.org/`
[4] `https://github.com/bytecodealliance/lucet`
[5] `https://wiki.mozilla.org/Security/Sandbox/Seccomp`
[6] `https://github.com/wasmerio/wasmer`

3.3.4 WASM3

Due to Wasm's simplicity in the binary format, there is wide-spread interest for running it on embedded devices. A result of that interest is `wasm3` [7], a Wasm interpreter that claims to be the fastest. They argue in favor of interpretation over JIT compilation, since startup latency is more important in many situations. Furthermore, portability and security are easier to achieve with this approach and development with the virtual machine is simpler. Still, a performance penalty of 4 – 5× compared to JIT or 12× compared to native execution, leads us to believe that no amount of gain in startup latency can make up for the loss in performance. Even though our primary goal is that of reducing the cold start latency, the execution that follows cannot massively lag behind Docker containers, if adoption is to succeed. The relationship between startup and execution performance is too far out of balance for this runtime.

3.3.5 WebAssembly Micro Runtime

However, there is still value in runtimes written specifically for embedded devices. The WebAssembly Micro Runtime[8] (`wamr`) has support for all three of the mentioned compilation strategies. It has low memory usage and binary size. The claimed binary size of the AoT runtime is just 50 kB. Furthermore, the AoT runtime claims near-native speed. For execution on edge devices, where memory is scarce, this runtime may allow more modules to be kept in-memory compared to heavier-weight runtimes.

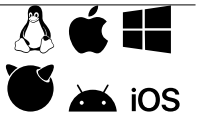
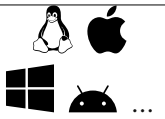
Runtime	ISA support	Compilation	Platforms
`wasmtime`	`x86_64` , `aarch64`	AoT, JIT	
`lucet`	`x86_64`	AoT	
`wasmer`	`x86` , `x86_64` , `aarch64`	AoT, JIT	
`wasm3`	`x86` , `x86_64` , `arm` , `RISC-V` , ...	Interpreted	
`wamr`	`x86` , `x86_64` , `arm` , `aarch64` , ...	AoT, Interpreted	

Table 3.1: WebAssembly runtime feature overview.

A summary of the runtimes' features can be found in Table 3.1. The diversity of compilation strategies sticks out – clearly no single one is the best for all scenarios. Except for `lucet` , all runtimes have fairly good platform support for our use. However, particularly for edge devices,

[7] https://github.com/wasm3/wasm3
[8] https://github.com/bytecodealliance/wasm-micro-runtime/

the lack of support for the 32-bit `arm` architecture in the runtimes written in Rust, `wasmtime`, `wasmer` and `lucet`, may be an issue for future widespread adoption. The support in the other ones, written in C, is there.

3.4 · WebAssembly in OpenWhisk

In this section, we first examine the possibilities for bringing WebAssembly support to OpenWhisk before exploring the available options and discussing them in the context of our requirements.

3.4.1 OpenWhisk's Anatomy

In order to understand the requirements for our Wasm-flavored OpenWhisk, we first need to understand the design of OpenWhisk itself.

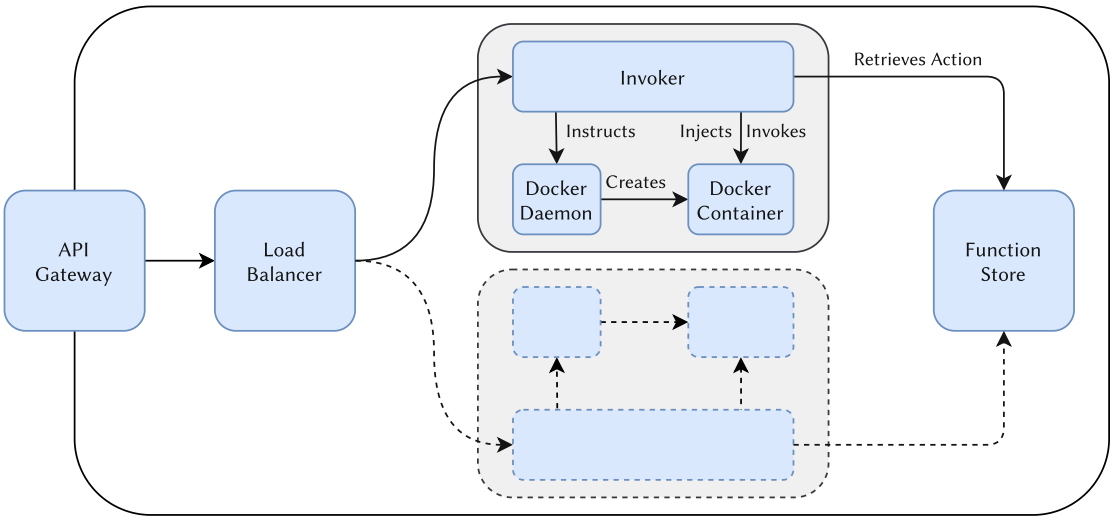


Figure 3.1: The invocation flow of an action in Apache OpenWhisk, based on the source code and documentation [Dev19]. Potentially many Invokers can exist on different hosts in this setup.

Figure 3.1 sketches the important parts of OpenWhisk's design. Users interact with it through its API Gateway, which consists of `nginx`, mostly for TLS termination, and the `Controller`, which implements the main logic. It handles the creation of actions – OpenWhisk's name for serverless functions – authentication and authorization as well as invoking the action. In the latter case, the controller uses its load balancer to determine one of the potentially many `Invoker`s, that should handle the request. The `Invoker`s are subscribed to an Apache Kafka queue, to which the load balancer publishes the request. The controller and its load balancing apparatus take a central role in OpenWhisk, but in our figure they are only represented by the

API gateway and load balancer, because we will not be modifying these parts and can treat them as blackboxes accordingly. Our focus is on the `Invoker`. It is the part of OpenWhisk responsible for *invoking* the action. A number of steps are involved in that process [Dev19].

1. It retrieves the action's code and execution permissions from an Apache CouchDB database, where the Controller stored it during action creation.
2. It instructs the Docker daemon to start a new container, based on the runtime that the action needs to execute. This could be an `openwhisk/action-nodejs-v10` image for JavaScript actions, but could also be a generic `openwhisk/dockerskeleton` blackbox image that can execute any binary, such as one written in Rust or C.
3. Finally, it injects the action's code into the container, invokes it with the given parameters and returns the result [Dev19].

Because OpenWhisk supports many runtimes and even the mentioned blackbox image, it needs a common protocol to communicate with all of them. The just-described step 3 is the gist of that protocol. Each runtime needs to implement a number of functions. The `start` function creates a new container and returns its address. The following requests to the `/init` endpoint are then sent to this address, where the container receives the code and takes the necessary steps to make the action ready for execution. Thus, over a container's lifecycle, this endpoint is called exactly once. Once initialized, the `/run` endpoint can be called with different parameters many times to execute the action [Dev19].

This implies that the container is not destroyed immediately after it has been invoked. Indeed, OpenWhisk, just like other known serverless platforms, uses various optimizations of the described flow, in order to improve the system's performance. Among those are pre-warming containers or keeping the container running for some time after it has been invoked – the previously introduced keep-alive time. Both have the same effect: When the user wants to invoke the action, no initialization is necessary and `/run` can be called immediately. These optimizations exist precisely because the cold start has been recognized as an expensive operation. In the absence of invocations and some threshold time elapsed, the container is removed entirely. In that case, the container runtime's `destroy` function is called, which removes the container and frees up the memory it used [Dev19]. These optimizations are part of the OpenWhisk logic and independent of the underlying container runtime. Our container runtime will profit from them too, but we have to take them into account in our design.

3.4.2 OpenWhisk on Kubernetes

OpenWhisk can be deployed on a Kubernetes cluster, a container orchestration tool for automating the deployment and scaling of containers [Aut21]. OpenWhisk has a service provider interface (SPI) called `ContainerFactoryProvider`, which represents the underlying container management platform. It provides two implementations of that SPI [Dev20].

1. The `DockerContainerFactory` implements the architecture shown above, except that most components of OpenWhisk run in separate Kubernetes pods. In non-Kubernetes deployments, they usually run in separate Docker containers. The crucial part of this deployment is that the `Invoker` runs in a single pod and also spawns all of the Docker containers inside that pod. Multiple `Invoker`s can run on different worker nodes. Because OpenWhisk communicates directly with the local Docker daemon, the container management latency is rather small, but this SPI implementation does not take advantage of Kubernetes' scaling mechanisms.

2. The `KubernetesContainerFactory` is the counterpart to these properties. It interfaces with the Kubernetes API to create and schedule actions in containers – in turn contained in pods – to run on different Kubernetes worker nodes. This trades higher container management latency for access to Kubernetes' scaling and scheduling mechanisms.

3.4.3 Executing WebAssembly with OpenWhisk

To execute Wasm modules in OpenWhisk, we have two similar options with similar trade-offs.

1. The first is using one of a number of WebAssembly runtimes that are currently being developed and are in various stages of maturity. We would need to embed one of these runtimes in a program of our own, which provides a similar interface as Docker. This WebAssembly container runtime would be able to replace the Docker daemon in the architecture presented above, and run Wasm modules instead of Docker images. Since this program would sit in-between OpenWhisk and the Wasm runtime, we would have precise control over the management of the modules, i.e. whether the Wasm module is just-in-time compiled, interpreted or compiled to native code ahead-of-time; the caching layer that keeps initialized modules ready for fast execution; or the configuration of the system interface with which Wasm modules can interact with the underlying operating system.

2. The second option is to use Kubernetes as the management system for our Wasm modules. Kubernetes is generic over the underlying container runtimes, and as such it runs on `containerd`, `CRI-O` and Docker [Aut21]. However, it is also possible to let Kubernetes pods run on other runtimes by implementing the `kubelet` API. The `krustlet` project provides a `kubelet` implementation that can execute Wasm workloads [Dei21]. With that as the container runtime set up, we can instruct Kubernetes to run Wasm modules instead of the usual container image.

We will explore those two options in more detail and discuss which is the better fit for our requirements.

3.4.4 Krustlet: The WebAssembly Kubelet

As a `kubelet`, `krustlet` runs on Kubernetes worker nodes and accepts workloads of the `wasm32-wasi` architecture. Just like a regular `kubelet`, `krustlet` will then pull the Wasm module from an Open Container Initiative (OCI) compatible registry, like a Docker image can be pulled from Docker Hub. This process allows us to schedule our Wasm workloads from an `Invoker`, similar to how the `KubernetesContainerFactory` implementation schedules actions in Docker containers. However, this execution model is also different from the OpenWhisk model in some ways.

- In OpenWhisk action runtimes, there is a separation between the runtime environment, provided through the Docker image, and the action code, that is stored in a database, retrieved from there and injected into the container. `krustlet` expects the Wasm module to be the equivalent of the container, but with the code baked in. Thus, no injection is necessary and there is no separation. This is an incompatibility with the current OpenWhisk model, since the Wasm module does not need to be stored in the database, but rather in the OCI registry. To implement this in OpenWhisk, we would have to modify where the module is stored, as well as remove the retrieval from the database and the subsequent injection, which are then useless operations. The OCI registry would become part of the OpenWhisk deployment, for latency reasons alone. We would not expect the cold start to be significantly affected by these changes. Not having to retrieve the code from the database, but pulling the image from the registry instead, should roughly be time-equivalent operations.

- Since there is no separation anymore, every Wasm module needs to implement the OpenWhisk runtime protocol itself, instead of being able to rely on the runtime implementing it once. In practice, this means implementing the `/init` and `/run` endpoints, where the former would no longer do anything useful, since there is no code to inject. It follows that every Wasm module needs to include an HTTP server; increasing the size of its binary and runtime cost above what would be needed only for its unique functionality. Of course, the current OpenWhisk version implements it in the same way: Each container must come with an implementation of the OpenWhisk protocol. Thus, we can assume that the extra cost is not substantial. An advantage of this approach is that the system is more resilient, since one container failure does not affect others.

- Evidently, Docker container operations are already barely fast enough to facilitate serverless platforms. As the OpenWhisk developers pointed out in the documentation for their `KubernetesContainerFactory`, Kubernetes pod management operations come with an even higher latency. More generally, Kubernetes is a tool that has not primarily been designed for serverless platforms. While some platforms like OpenFaaS or Knative[9] do depend on it, they also suffer from significant cold starts. It is therefore reasonable to re-examine this choice. Because of these aspects, we have concerns about *not* alleviating

[9] `https://knative.dev/`

the cold start latency with this approach. What we may win by starting Wasm rather than Docker containers, we may lose through higher management latency.

- Because network support in WASI is still under active development, the current alternative is to implement a `wasmcloud-actor`, which essentially provides capabilities such as networking to Wasm modules [Tea21]. This would be necessary due to the aforementioned HTTP server requirement. The `wasmcloud` uses either `wasm3` or `wasmtime` as the underlying WebAssembly runtime. In the preceding section, we have excluded `wasm3` for its lack of performance. `wasmtime` uses a JIT compiler, which compiles the module when it first receives the code. Recall that our fourth requirement is near-native execution speed. Compared to embedding a runtime, the `wasmcloud` approach does not give us the same level of control over the compilation strategy and execution. However, having control over these aspects is a priority, since these can substantially affect startup and execution time. A JIT compiler, in its typical usage, may impose a substantial startup tax.

While this approach has its merits, we believe that there would be higher friction for implementing it in OpenWhisk than writing our own purpose-built Wasm executor.

3.4.5 An OpenWhisk-WebAssembly Executor

Writing our own executor allows us to integrate WebAssembly into OpenWhisk with fewer changes to it. Furthermore, the more precise control over container management as well as the compilation strategy and execution are the main arguments in favor of this approach. These were some of the main requirements we laid out. We therefore implement our own layer between OpenWhisk and different WebAssembly runtimes which enable the execution of Wasm modules. It will implement the OpenWhisk protocol and manage WebAssembly containers. We refer to them as the Wasm executors, to distinguish them from Wasm runtimes. Our layer implements the parts independent of WebAssembly runtimes such as OpenWhisk communication and will be generic over the underlying runtime. Thus, each WebAssembly runtime will result in a separate executor binary and can be used independently of the others. We aim to replace the Docker daemon in the architecture shown in Figure 3.1, leveraging the existing OpenWhisk interface for container management.

In Figure 3.2, we have sketched the high-level OpenWhisk modifications necessary for the integration. In this invocation flow, we need to modify OpenWhisk's `Invoker` such that it communicates with the Wasm executor instead of the Docker daemon. Fortunately, the `Invoker` is already well-separated from the concrete containerization technology through another Service Provider Interface. As previously mentioned, OpenWhisk supports Docker as well as Kubernetes through the `ContainerFactory` trait. We add a `WasmContainerFactory` that implements this trait. Via a configuration property, we can instruct OpenWhisk to use this factory. It uses a `WasmClient`, that can start and destroy a container by invoking the endpoints of the same name on the Wasm executor. The start function returns its result

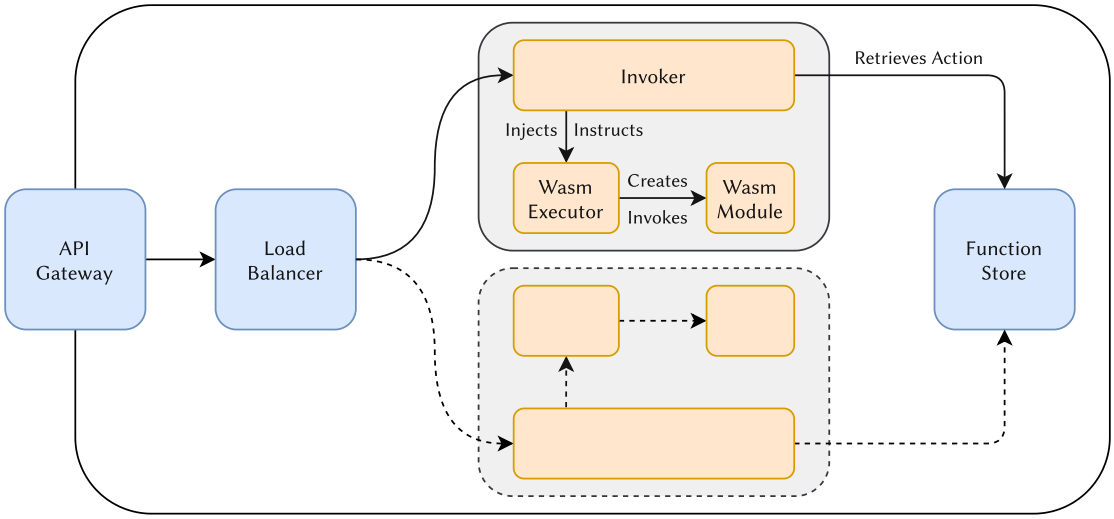


Figure 3.2: Invocation flow of a Wasm action in our modified Apache OpenWhisk. The `Invoker` *injects* the code into the executor which *creates* a Wasm module ready for execution. It then *instructs* the executor to *invoke* the module with the parameters it passes. The result is passed back to the Invoker. Again, a deployer can define whether one or more Invokers exist.

as a `WasmContainer` that extends OpenWhisk's `Container` abstraction. This trait handles container communication, such as calling the `/init` and `/run` endpoints. It also implements `suspend` and `resume` functions, which we do not implement. Docker containers potentially use CPU cycles and memory while they are up, so OpenWhisk pauses them between invocations, which suspends all of its processes by preventing them from using the CPU. Containers in our executor, on the other hand, consume only memory but no CPU in-between requests, so pausing is not a meaningful operation. The `Container` also allows collecting the logs that an action produced. This functionality is not strictly necessary for correct operation, so we do not implement it. Since we want to implement three runtimes in our executor, keeping the requirements for each of them minimal is important. However, implementing a log mechanism with WASI is entirely possible.

In OpenWhisk's Docker container implementation, OpenWhisk only uses the Docker daemon to create a container, but then communicates with the container directly. In our implementation, every request is proxied through the Wasm executor to the container itself. This design presupposes that the Wasm executor can handle a large amount of concurrent requests, which we noted in our requirements. The advantage is, that not every Wasm module needs to implement the OpenWhisk protocol, as discussed under the `krustlet` topic before. The disadvantage is the single point of failure. However, that is mitigated by the bigger picture of OpenWhisk. In a truly resilient deployment, we would have multiple `Invoker`s. A failing Wasm executor would let the entire `Invoker` unit fail, but other `Invoker`s could take over requests. Furthermore, the executor itself could be scheduled as, e.g. a Kubernetes pod with a restart policy of `Always`. Finally, because WebAssembly runtimes isolate faults of their

containers, it is impossible for a faulty Wasm module to affect the entire executor – barring bugs in the runtimes. The absence of bugs in the executor and the runtimes – or, at least a small likelihood of their existence – is therefore also a basic requirement. The mentioned handling of concurrent requests further implies that thread-safety is another one. We noted both of these requirements under the umbrella term of program correctness. The executor also needs to be generally fast, in order to aid in alleviating the cold start problem. That includes efficient handling of receiving the module's code from OpenWhisk, instantiating it and setting up the underlying Wasm executor. In order to enable execution on all kinds of platforms, including edge devices, another requirement is the executor's ability to run on different instruction set architectures.

3.5 · Executor Implementation

To implement these requirements in Apache OpenWhisk, we take the following steps. The current Docker-based `Invoker` is replaced to forward all the incoming requests to our new Wasm executor.

The Wasm executor is written from scratch in the Rust[10] programming language. There are a number of reasons for this choice.

- It is a compiled language with performance similar to that of C or C++.
- The Rust compiler uses `LLVM` as its backend so Rust programs can be compiled for any architecture we care about, in particular also `aarch64`. Moreover, we can also compile to `wasm32-wasi`, the WebAssembly WASI target. This allows us to write highly performant WebAssembly modules in Rust, such that we can test our executor *and* the Docker actions under the best possible conditions.
- It has good support for asynchronous I/O, so Rust web frameworks can efficiently handle a large amount of concurrent requests.
- Program correctness is one of Rust's primary goals, which is enabled through a large amount of guarantees that can be checked at compile time. In particular, writing memory-safe programs is much easier than in C or C++, while providing a similar level of performance.
  - A subcategory of these checks is particularly worth pointing out: Rust allows for »fearless concurrency«, that is, having concurrent programs checked for thread-safety at compile time. In particular, the compiler can detect subtle bugs like data races.

[10] `https://www.rust-lang.org/`

- Two of the three used Wasm runtimes are written in Rust, so embedding them is well documented, straightforward and their APIs tend to be the most idiomatic, powerful and up-to-date. The other Wasm runtime, `wamr`, is written in C, which can be efficiently embedded from Rust through bindings.

```rust
#[async_std::main]
async fn main() -> anyhow::Result<()> {
    #[cfg(feature = "wasmtime_rt")]
    let runtime = ow_wasmtime::Wasmtime::default();

    #[cfg(feature = "wasmer_rt")]
    let runtime = ow_wasmer::Wasmer::default();

    #[cfg(feature = "wamr_rt")]
    let runtime = ow_wamr::Wamr::default();

    let mut executor = tide::with_state(runtime);

    executor.at("/:container_id/init").post(core::init);
    executor.at("/:container_id/run").post(core::run);
    executor.at("/:container_id/destroy").post(core::destroy);

    executor.listen("127.0.0.1:9000").await.unwrap();

    Ok(())
}
```

Listing 1: The main function of our WebAssembly executor. The `#[cfg(feature = "...")]` macros allow us to define multiple runtimes while deciding at compile-time which of them to enable. Thus, we can build three separate binaries, with each only containing the code necessary for its runtime.

The main function in Listing 1 suits itself well to describe the high-level architecture of the executor. The entry point into our executor is through HTTP endpoints. We do not make use of any advanced HTTP features, so the choice of the web framework does not matter too much. We use `tide` [TCW21], an asynchronous by-default web framework ideal for prototyping. We discuss asynchronous Rust in greater detail in Section 3.5.4. Suffice it to say, that the `#[async_std::main]` macro starts this program in an asynchronous runtime, which enables us to use `async` functions and `await` them. The async runtime allows multiple concurrent requests to any of the endpoints, without us having to spawn threads explicitly.

We instantiate one of our three implemented Wasm runtimes, only one of which is compiled into the executable, based on the feature flag we pass to the compiler. The runtime is used

as *state*, in `tide` terminology, which is to say that it is injected into each request and thus shared among requests. Sharing objects requires them to be thread-safe. This is where Rust's compile-time checks help us ensure this property. The `tide::with_state` function can take any argument that implements three traits: `Clone`, `Send` and `Sync`. The first simply gives objects implementing it the ability to be duplicated. This is necessary, such that each request can receive a copy of the runtime. However, a deep copy would be too expensive for each request. Our runtime wrappers therefore implement `Clone` as a shallow copy, that is, they create a new reference to the same inner object that all other wrapper objects also point to. For this to work safely, we need the other traits. These are only *marker traits*[11], indicating that an object satisfies a property. `Send` means a type can be transferred across thread boundaries, while `Sync` means that references to that type can be shared between threads. These properties are satisfied, if the types that we use in the definition of our runtimes and those used transitively, have these properties. That is all to say that the types of the runtimes we use need to be thread-safe, i.e. implement those marker traits. Here is where the differences between the runtimes start to emerge, but we discuss those in a later section.

To start a container, the `WasmContainer` object in OpenWhisk generates a new universally unique identifier (uuid), without having to communicate with the Wasm executor at all. This is possible because there is no setup for a Wasm container, that needs to happen ahead-of-time, like network namespace creation for Docker containers. That shows an important aspect of what constitues a WebAssembly container. In-memory, it is ultimately only represented by the action's code. No external operating system resource is used. It follows, that actual work only happens in the endpoints, which are parameterized by the container id. `/init` is called once by OpenWhisk per container to initialize the container with the module's code. Although different for each runtime, in general, this endpoint will do any work that can be frontloaded. `/run` is invoked potentially many times, so any work that is not frontloaded would multiply there. Once OpenWhisk decides to destroy the container, it does so through `/destroy`. Of course, `/init` is the potential culprit for the cold start latency, so our focus is on reducing even the amount of work this endpoint needs to handle.

3.5.1 Fast Cold Starts

In serverless platforms we can identify three important steps, in order for code to be executed. Uploading the code to the platform, initializing the execution environment, and running it. The latter two steps are already part of the execution path; they occur when an execution request is actively waiting for the result to be returned. Thus, during initialization, the action should already be well-optimized for execution such that as little preparation as possible is needed. WebAssembly is a binary target format, however, not one that is ready for *fast* execution. It can be interpreted right away – thus a small cold start time – but execution performance will lag behind massively compared to native code, which is unacceptable. So compilation to native code is essential for good runtime performance, but it also takes time for a module to be

[11] `https://doc.rust-lang.org/std/marker/index.html`

translated. Given this trade-off, it would seem that just-in-time compilation should be the best model. In `wasmtime`, our JIT-runtime of choice, compilation of a simple, WASI-enabled module (1,6 MB) still takes 35 milliseconds on our test machine (or 40 milliseconds when optimizing for speed). This time represents 98% of the entire setup time, including every other part of the runtime. Thus, unsurprisingly, the cold start is defined almost entirely by the compilation phase. For comparison, the cold start time of the `hello-world` Docker image on the same machine is 452 milliseconds. This number is roughly in line with the cold start times of AWS Lambda or Google Cloud Function, that we have cited in Chapter 2. While the JIT cold start is an order of magnitude faster than the startup times of Docker containers, it is still on the same order of magnitude as the execution time of some actions. Given that the compilation phase takes up such a big chunk of the overall time, it is the most obvious starting point for further optimizations. And there is room for improvement.

In contrast to browsers, which receive JavaScript or WebAssembly just-in-time, and thus need to compile it in the same manner, a serverless platform takes ownership of a module earlier. When a user uploads a module, the platform has much more time to apply optimizations to the module than on the execution path, thereby making it ready for a fast startup *and* execution, solving the previously described dilemma. Thus, the key is to run the expensive module creation ahead of the time of execution. It does not matter in this case, whether the actual compilation strategy of the underlying runtime is JIT or AoT. It is obvious for AoT – the code needs to be compiled ahead-of-time in its entirety, by definition. As we have seen, JIT also has a cold start penalty and we can complete that step ahead-of-time, too. We refer to this generically as *precompilation*. Once the module is precompiled, we can serialize it and store it in the database. During initialization, all that is left to do is to deserialize the precompiled bytes into an in-memory `Module`, which is a very fast operation. At the time of execution, the module is ready to produce an instance and execution can start immediately.

Both `wasmer` and `wamr` support AoT compilation. `wasmer` in particular, allows us to leverage `LLVM` for highly optimized native code to produce a shared object. The runtime loads this via a fast `dlopen` call. `wamr` produces its own AoT format which it can deserialize at runtime and achieve a fast startup as well. For `wasmtime` it is the same in all but the name. We can run the relatively expensive module creation ahead-of-time, and serialize and store the result. Thus, all the runtimes can support some form of precompilation.

3.5.2 Warm Starts

Even though the cold start of our executor is well-optimized with this technique, cold starting for every single invocation would still be too expensive for sequential invocations. After all, the `init` procedure still requires retrieving the module from the database and copying it to the executor. We can save on that too, by simply taking advantage of OpenWhisk's current design. In order to adhere to it, we need to store the module between `init` and `run`. The advantage of that is, of course, that after the initialization, `run` can be called again and again, without having to go through any of the setup. This is how OpenWhisk works with Docker containers already. After the initial cold start and first invocation, OpenWhisk pauses the container. If

another subsequent request to the same action comes in, it unpauses the container and the action invocation experiences a so-called warm start. This scenario skips the expensive cold start and is much faster. After a threshold without any invocations – which defaults to ten minutes – OpenWhisk calls the `destroy` function on the container, removing it entirely. Thus, by adhering to OpenWhisk's design, we have support for fast warm starts as well.

The advantage of keeping a module in the executor's memory, is that it is already deserialized and new instances can be instantiated from it immediately. An instance is the runtime's representation of a callable Wasm module. If a module is the equivalent of a Docker image, then the instance is the running Docker container. Note that, unlike Docker-based OpenWhisk, we do not keep instances (containers) in-memory, but only the modules (images). The cost of instantiating an instance is negligible. On our test machine it took, on average, 340 microseconds to set up a `wasmtime` instance and the parts needed for execution. This represents the overhead on every warm invocation. Moreover, two instances can execute concurrently, even if the module itself is not thread-safe, for instance when re-entrant execution would be unsafe. Finally, implemented this way, it would also be safe to execute two requests from different users, since instances are isolated from each other. They operate in their own memory space and have their own WASI environment.

3.5.3 The WasmRuntime Abstraction

The Wasm executor is generic over the underlying WebAssembly runtime, such that we can easily implement different ones and facilitate our comparison. To that end, we abstract the common runtime tasks into a trait – essentially an interface in Rust's terminology. It is shown in Listing 2. The state of our executor can be any object that implements this trait. For each Wasm runtime we have selected, we implement a wrapper, each of which implements the `WasmRuntime` trait. All our wrapper implementations work in similar ways and differ only in details. We describe the implementation for each trait method in the wrappers, while pointing out the important differences between runtimes. Afterwards, we discuss alternatives.

Initialization

The executor decodes the `base64` string given by OpenWhisk and unzips it, before calling `initialize` with the result. The function initializes the container identified by the given id. All wrappers store the code they are given in a thread-safe HashMap – provided by the `dashmap` [12] library.

A note on WebAssembly runtime terminology first, based on `wasmtime`'s[13] and `wasmer`'s[14] documentation and the WebAssembly specification[15]. An engine is used for compiling and

[12] `https://crates.io/crates/dashmap`
[13] `https://docs.rs/wasmtime`
[14] `https://docs.wasmer.io/integrations/examples/instance`
[15] `https://webassembly.github.io/spec/core/exec/runtime.html`

```rust
pub trait WasmRuntime: Clone {
    fn initialize(
        &self,
        container_id: String,
        capabilities: ActionCapabilities,
        module: Vec<u8>,
    ) -> anyhow::Result<()>;

    fn run(
        &self,
        container_id: &str,
        parameters: serde_json::Value,
    ) -> Result<Result<serde_json::Value, serde_json::Value>, anyhow::Error>;

    fn destroy(&self, container_id: &str);
}
```

Listing 2: The `WasmRuntime` abstraction that each of our Wasm runtime wrappers implements. Note the previously mentioned `Clone` trait that is required for using a `WasmRuntime` object as `tide` state. This is required here through the *trait bound*, which means the type implementing this trait needs to also implement `Clone`.

managing modules. Recall that a module represents compiled WebAssembly code, similar to a container image, and an instance is the callable instantiation of that code, i.e. the running container. A store is necessary to hold on to memories, tables, functions and other auxiliary data structures.

Wasmtime In `wasmtime`, we instantiate a single, global `Engine` when the executor starts, which is then used to instantiate all `Module`s. The bytes passed through `initialize` are deserialized into a `Module` – a fast operation – and then stored in the internal HashMap. This is only possible because a `Module` is thread-safe so the concurrent HashMap it is contained in can also be safely shared across threads.

Wasmer In `wasmer` we also store an `Engine` globally, and instantiate a `Store` per `Module`. In contrast to `wasmtime`, the `Store` is thread-safe and could be stored globally as well. However, it holds on to the memory-heavy parts of the `Module` and these would not be freed even if the `Module` was. Thus, until the executor terminates, the `Store` would only grow in size. Like in `wasmtime`, the `Module` is thread-safe and thus enables our pre-creation optimization of them in the first place.

Wamr

`wamr` is originally written in C, and we use `wamr_sys` [16] bindings to embed it in Rust. Contrary to the other runtimes, a module in `wamr` is not thread-safe. Thus, only the serialized bytes representing the module are stored in the HashMap. It follows that the deserialization happens in every `run` call instead, where the cost multiplies. Additionally, two functions need to be called globally for the runtime to be instantiated and torn down, `wasm_runtime_init` and `wasm_runtime_destroy`, respectively. Our implementation requires each runtime state to implement `Clone`. A common pattern in Rust (also used by `wasmtime` and `wasmer`) is to implement the clone operation to produce a shallow copy. Internally, these objects typically use Rust's atomically reference-counted smart pointer (`Arc`) to store a pointer to the actual object, in order to safely share this object across threads. Then only a cheap copy of the `Arc` is necessary to share the wrapper object across different threads, while having access to the same underlying data. On each request, a clone of the runtime is created and dropped at the end, i.e. its memory freed. Because of that, we cannot call `wasm_runtime_destroy` in the wrapper's `Drop` implementation, since that would destroy the runtime too early. Instead, we use another internal `Arc` in the wrapper, which is instantiated with an object that calls the initialization function when it is created, and destroys the context when it is being dropped. Because the `Arc` will live as long as the runtime and only calls the contained object's `drop` function when all other references to the runtime object itself have gone away, this implements the desired behavior. This exemplifies that `wamr`'s API is harder to use safely and correctly than the idiomatic Rust APIs of the other runtimes, where memory- and thread-safey properties are encoded in the type system and checked by the compiler.

Both `wasmtime` and `wasmer` underline the importance of the validity of the bytes given to the respective `deserialize` methods. Unlike a Wasm module, the compilation artifacts can not be validated in the same way since they already represent a compiled module. So they have to be trusted even when the Wasm module they were produced from would not have to be trusted. In our prototype, the users actually supply the compilation artifacts themselves, for development convenience reasons. For an actual serverless framework making use of Wasm, it would have to take ownership of the Wasm module and do the precompilation itself, in order to trust the code.

Execution

All used runtimes support the WebAssembly System Interface. Recall that WASI follows a capability-based security model, which allows for fine-grained control over what the module has access to. Hence, building a so-called WASI context – named differently in each runtime

[16] `https://crates.io/crates/wamr_sys`

– means setting up those capabilities. That might include writing parameters to `stdin`, receiving logs from `stdout` or setting environment variables and `argv` arguments. By default, the module also has access to WASI APIs like `random_get` for high-quality randomness or `clock_time_get` to access various clocks. WASI also controls on a per-file basis what directories the module has access to. Specifically, the module needs a preopened file descriptor in order to access files. Our wrappers therefore have to open the files and pass them to the module. However, what files a module should have access to is not definable in the OpenWhisk protocol, since it does not make use of Docker's host-to-guest mappings, so we need a way for users to specify these capabilities. OpenWhisk actions can have optional annotations attached to them, which are simple key-value pairs. We can use those to let users specify capabilities. For instance, using the `wsk` cli:

```
wsk action create cache cache.zip --annotation dir "/tmp/cache"
```

Here a new action named `cache` is created from a file called `cache.zip`. A key-value pair `dir = /tmp/cache` is attached, letting the executor know that the module should have access to `/tmp/cache`. With some minimal code changes to OpenWhisk's container abstraction, these annotations are then passed to the executor during `initialize`. On the executor side, it is passed to each runtime wrapper which can then act on that data. It would be easily possible for a service provider to implement policies on top of it. Access could only be granted to the `/tmp` directory in general; or a default cache directory `/tmp/<container_id>` could be created for each container, where information can be cached and shared among instances of the container, but only *that* container has access to it. As WASI is currently under heavy development and does not, for example, specify any networking interfaces, the amount of capabilities we can grant is limited. Our prototype implements directory access and a function that simulates an HTTP GET request. The latter will be relevant for testing I/O-bound performance in Chapter 4.

The `run` method executes the module associated with the given container id and with the given parameters as input. As required by OpenWhisk, the parameter is an object in JavaScript Object Notation (JSON). The WebAssembly minimum-viable product (MVP), only supports integer data types natively. Higher-level data types, like strings, often differ in their implementation between languages, so a universal target format like Wasm cannot support a specific language's implementation, without shutting certain languages out or making it harder for them to be used. Making Wasm modules interoperable – even those compiled from different source languages – is an ongoing process with WebAssembly interface types[17]. The `wasm-bindgen` Rust library already makes this process easy for JavaScript-WebAssembly interoperability. Unfortunately, it does not work with any of the runtimes we use. Even if both our executor and the module are written in Rust originally, passing a raw pointer to the JSON object to the module is not possible, because WebAssembly is sandboxed and could not access the pointee. To work around this, we require that the actions written for our executor use our library `ow-wasm-action`. Internally, the library allocates a global buffer of bytes. It exports three functions to work with this buffer,

[17] https://hacks.mozilla.org/2019/08/webassembly-interface-types/

```rust
ow_wasm_action::pass_json!(handler);

pub fn handler(json: serde_json::Value)
        -> Result<serde_json::Value, anyhow::Error> {
    let param1 = json
        .get("param1")
        .ok_or_else(|| anyhow::anyhow!("Expected param1 to be present"))?
        .as_i64()
        .ok_or_else(|| anyhow::anyhow!("Expected param1 to be an i64"))?;

    let param2 = json
        .get("param2")
        .ok_or_else(|| anyhow::anyhow!("Expected param2 to be present"))?
        .as_i64()
        .ok_or_else(|| anyhow::anyhow!("Expected param2 to be an i64"))?;

    Ok(serde_json::json!({ "result": param1 + param2 }))
}
```

Listing 3: A simple add action in Rust that uses a macro from `ow-wasm-action` to pass the received json to the `handler` and return the `Result` back to the runtime.

most notably one that returns a pointer to it. When the module is run, the runtime wrapper serializes the JSON object as bytes and – via one of the exported functions – allocates enough space in the buffer to store them. It then acquires the pointer to the global buffer and writes the JSON bytes into it. The wrapper effectively writes into the memory region that the Wasm module can read from; its linear memory. Since the module is unaware of the number of bytes that have been written, the runtime passes that as its `argv` argument. Once the write process is done, the instance's `main` function – implemented in our library – is called. It reads the bytes from the buffer and deserializes them into a JSON object. Then it calls the user-provided handler function with that object. The return value from the handler is serialized back into the buffer, from where the wrapper can read it, using a similar process. Conveniently, a developer only has to implement a function that takes a JSON object and returns one in the success case; an error otherwise. Thus, this crude method of passing parameters can be completely hidden from action developers and only requires a single line of code to be included. A simple action can be seen in Listing 3.

The action development process is visualized in Figure 3.3, which makes use of the `binaryen`[18] toolchain to optimize the Rust-produced WebAssembly. The `ow-wasmtime-precompiler` is a simple wrapper around `wasmtime`'s API, that creates a module from Wasm bytes, in turn kicking off the internal compilation process. Once the module is created, it is ready for execution.

[18] https://github.com/WebAssembly/binaryen

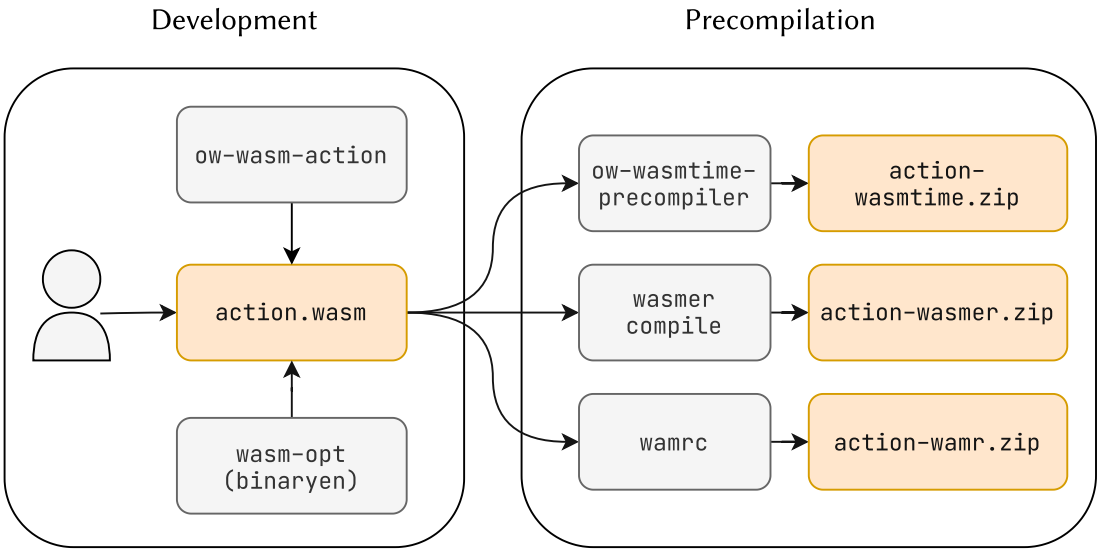


Figure 3.3: There are two phases in the action development. Our `ow-wasm-action` library provides the abstraction to read the parameters the runtime has passed and to write the return value. To that end, it uses `serde_json`, a (de)serialization library to read and write the JSON objects. Afterwards, `wasm-opt` runs an optimization pass on the produced WebAssembly, since not all runtimes do so themselves. Depending on what runtime is compiled into the executor, the corresponding precompiler needs to be used. For `wasmtime`, we have written the small `ow-wasmtime-precompiler` program that precompiles the Wasm code into a format ready for execution, while `wamrc` and `wasmer compile` can be used for the others. Finally, the resulting precompiled binary is zipped which reduces the transmission size and instructs the OpenWhisk cli to treat it as binary data. The zip file is then ready to be uploaded to OpenWhisk.

At this point, we serialize it to a file and zip it. This process precompiles the module for the current host architecture. `wamrc` and `wasmer compile` can also take a target argument to precompile for a different host, e.g. a Raspberry Pi with the `aarch64` ISA. However, as of version `0.26`, the `wasmtime` command line interface (cli) has also gained similar capabilities.

The creation of the WASI context as well as the actual execution is slightly different for each runtime.

Wasmtime The `wasmtime` runtime, as of version `0.23`, uses `cap-std`, a capability based implementation with a similar API as the Rust standard library. We can construct its `Dir` type and use it while building a `WasiCtxBuilder`, ultimately yielding a `WasiCtx`. The context is *linked* with the module using the runtime's `Linker`, which is responsible for resolving the imports the module needs, WASI or otherwise. Hence, we can also use it to provide functions as imports to the modules. To execute, we need to create a `Store`, which requires a reference to an `Engine`. Since one was instantiated globally, we can simply refer to it.

The `Store` in turn is needed to instantiate an `Instance` from the `Module` we deserialized during initialization. The `Store` is neither thread-safe nor meant to be long-lived, according to the documentation. Thus, the `Store` and the `Instance` are instantiated per-request, in each `/run` invocation, and discarded afterwards. It ensures that the memory allocated in the `Store` does not stick around for the entire lifecycle of the `Module`. Evidently, this runtime's API is a very good fit for our use case.

Wasmer In `wasmer`, we build up a `WasiEnv` with a similar API as `wasmtime`'s, including directory access. To provide functions to the module, the resulting `ImportObject` can be combined with another one, by implementing a custom `Resolver`. It simply maps the name of an import to a function, in our case. The module combined with the `Resolver` then produces an `Instance`. In contrast to `wasmtime`, the necessary `Store` was already created in the `initialize` method, so the memory allocated in it does stick around until the entire container is removed.

Wamr `wamr`'s API is lower-level and less ergonomic compared to the other runtimes, due to it being a C rather than an idiomatic Rust API. On every `run` invocation, we need to load the runtime, instantiate the module from the precompiled bytes and create an instance from it as well as create an execution environment. Only then can we start execution. Due to API limitations, we cannot frontload meaningful work to the `init` method, so all of it needs to happen for every invocation. While this costs performance, it still implements what is required. However, other parts of `wamr`'s API are real restrictions. One is that function imports for WebAssembly modules need to be specified on – what `wamr` calls – the runtime, which is the equivalent of the `Engine` type in the other Wasm runtimes. Since it is global and instantiated only once, we can not define different imports for different modules, only one set of imports for all modules. While we do not need this flexibility for our examples, it may be desireable for a full-featured Wasm executor, e.g. only modules where the developer has set the permissions accordingly can work with networking functions. The other Wasm runtimes allow us to specify imports per instance. The API of `wamr` is the least appropriate for our use case.

In summary, `run` looks up the previously stored module or its bytes by the container id, creates a WASI context with the associated capabilities and potential imports, passes the parameters and calls the module.

Cleanup

Finally, the `destroy` function removes the container, which simply means freeing the memory taken up by the module. This is fast and thread-safe, since each container manages its own

Wasm module. Once OpenWhisk has decided to destroy the container, it will synchronize internally to not call `run` on it again. An overview of the entire executor workflow can be seen in Figure 3.4.

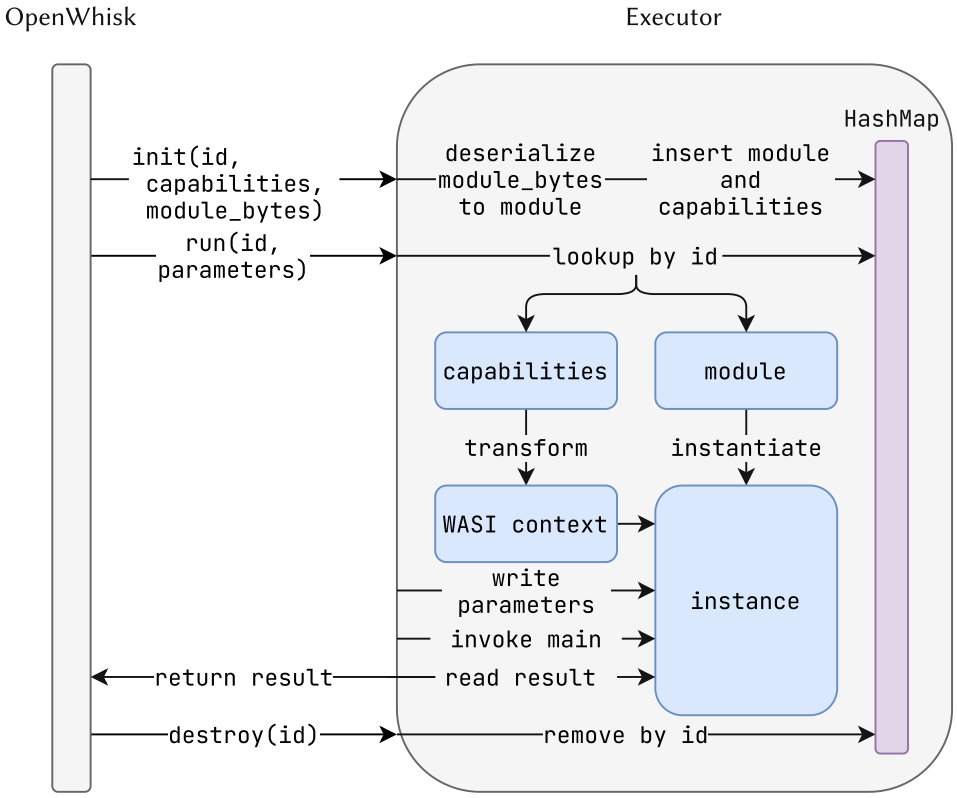


Figure 3.4: Overview of the executor architecture. `init` decodes the received bytes using `base64` and unzips the result. From there, they are deserialized back into a `Module` instance, in the case of `wasmtime` and `wasmer`. The module and capabilities are then stored in the HashMap. When a `run` request comes in, the module and capabilities are looked up. The capabilities are transformed to a WASI context, which is used to instantiate an instance, together with the module. The parameters for this `run` request are serialized to JSON and written into the instance's memory. Calling the `main` function in turn invokes the actual developer-provided handler function. The read result is returned to OpenWhisk. A `destroy` request simply translates to a remove operation on the HashMap. Note that this figure is accurate for `wasmtime` and `wasmer`, but not entirely for `wamr`. The latter runtime's module creation is slightly different, as explained above.

3.5.4 Enabling Concurrency

In order for our executor to handle requests to these endpoints concurrently, we use Rust's `async` features. It is a programming model that »lets you run a large number of concurrent tasks on a small number of OS threads« [Dev21b]. With this model, a large number of I/O-bound tasks can be handled with less resources than a thread-based model could. Rust's async model works with `Future`s – similar to `Promise`s in JavaScript – which represent values that have not yet been computed. Contrary to `Promise`s however, Rust's `Future`s are lazy: They do nothing unless actively polled. This task falls to `async` runtimes, of which `async-std` provides one. In `async-std`, the unit of execution is a `Task`, which is similar to a `Thread`, except it is driven to completion by the user space runtime instead of the OS. Many of those tasks are executed on the same thread, and by default, a `Thread` per logical CPU core is spawned. Thus we have $M \cdot N$ tasks in total, where M is the number of tasks per thread and N the number of logical CPU cores. Because multiple `Task`s are multiplexed onto one thread, it should not do a blocking or long-running operation, like reading from a file via the standard library's synchronous I/O functions. Instead, it should use the `async` version of that method, from `async-std`, which does not block the thread. Otherwise, all $M - 1$ tasks, which are executed on the same thread will be blocked, even though they may be able to make progress. Note that we could still process N requests concurrently, even if tasks block, but that does not scale as well for I/O-bound tasks.

The question then becomes, what the execution of WebAssembly is bound by. That mostly depends on the WebAssembly module. If the module calculates a large prime number, executing it is CPU-bound; but if it makes an I/O-bound HTTP request, then so is the task running the module. We have explored the typical serverless workload in detail in Chapter 2. Most definitely, I/O-bound functions are a primary type of workload for serverless due to their frequent usage of external services. CPU-intensive functions on the other hand, are less common, but still need to be accommodated, if only to expand the range of applications that can leverage serverless.

Using this programming model enables us to potentially handle many concurent requests from OpenWhisk. However, since we embed Wasm runtimes and – at the time of implementation – calling WebAssembly functions is not `async`, they block the thread they are running on, until the module has finished execution. If the module's execution is CPU-bound, it essentially blocks the other `Task`s on the thread from making progress until it is finished. This is less of an issue if we only have CPU-bound modules, given that no more work can be done concurrently than the CPU has logical cores. In a more realistic, mixed workload scenario, however, modules hogging the CPU could prevent I/O-bound tasks from concurrently making progress, even though they hardly need CPU time. In both cases, only N Wasm modules can be executed concurrently, directly corresponding to the number of available threads. That means, we cannot execute modules on the same thread concurrently and that, essentially, both workload types are not well suited for this execution model. Since one of the appealing offerings of serverless frameworks is their scalability, we clearly need a way to handle many more requests, even if they block.

To let other `async` functions on the same thread make progress, while one executes the module,

we can push the entire execution onto a thread pool. The function then has the simple job to asynchronously wait for that execution to finish and collect the result. That allows the other functions to process more requests in the meantime. Unfortunately, this effectively corresponds to a thread-based model where fewer I/O-bound tasks can be handled than possible with an asynchronous task model.

Recall that WebAssembly itself does not interact with the system, so it could not do blocking I/O without its (WASI) imports. If we could provide these imports as asynchronous functions, the async runtime that drives our executor could also drive them to completion. In fact, the Rust library implementing WASI capabilities `wasi-cap-std-sync`[19] notes that an `async` version is being worked on, as of April 2021. But until these APIs become available, we have to work around that with the thread pool.

To test the whether the thread pool model is able to handle more requests than blocking in the threads directly, we run two short experiments.

1. CPU-bound: Calculating a large prime number.
2. Network I/O-bound: Making a long-running (1 second) HTTP request.

Our test machine has 8 logical CPU cores, resulting in 8 threads being started for the async-std runtime. We thus send 16 concurrent requests to be able to see the potential blocking in the result. The first experiment showed no difference in the total execution time between the default and thread pool model. As we would expect, in the default model the CPU-bound task finishes the first 8 requests, before starting the second set. The execution in a thread pool executes all 16 requests concurrently, but each took roughly twice as long, resulting in no significant difference between the total execution time. The default model operates more along the first-come, first-serve principle, where an early request will also get a response more quickly. In this simplified example, where all workloads are CPU-bound, this is preferable, since the first 8 requests get their responses sooner.

The result of the second experiment is shown in Figure 3.5. In the default model, the first 8 requests block the other 8, while they wait for the HTTP request to finish, resulting in roughly 2 seconds of total execution time. Unsurprisingly, the thread pool model finishes in half the total execution time, since it can handle all 16 requests simultaneously, due to using a new thread from the pool for each blocking operation.

Both models seem to have a workload better suited to them. However, in a real-world, mixed-workload scenario, a long-running, CPU-bound function may prevent another I/O-bound one from initializing its computationally cheap request. A mixed scenario may therefore also be better served by the thread pool model.

In Figure 3.6, a mixed scenario is run, where 16 CPU-bound and 16 I/O-bound requests are sent concurrently. In the default model, the CPU-bound requests finish in about one third of the

[19] `https://docs.rs/wasi-cap-std-sync`

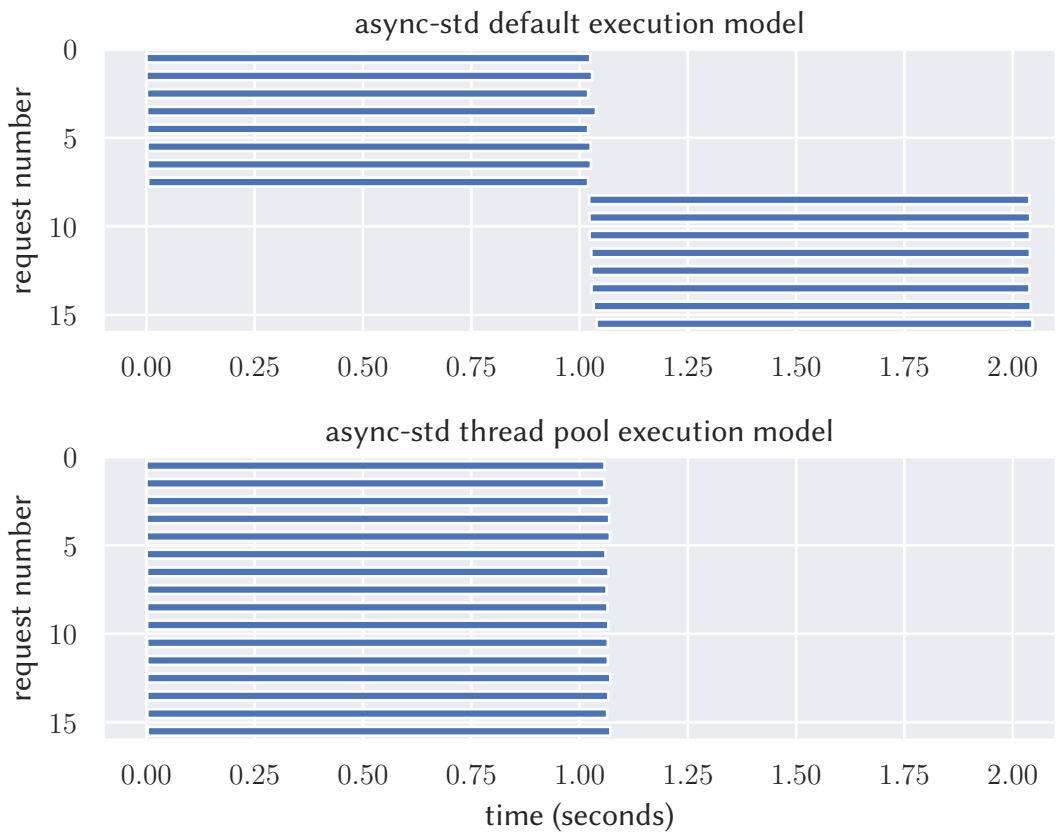


Figure 3.5: Executing blocking Wasm workloads under the `async-std` default model with 8 threads, compared with running each in a thread from a pool.

time they take under the thread pool model. This is again due to more requests being processed concurrently in the latter model, and is also why its standard deviation is slightly higher. For I/O-bound workloads, there is a slight advantage for the default model. However, in the context of the plot on the right side, the thread pool model is able to handle more requests per second in general, but specifically excels at the I/O-bound ones. There is an inherent trade-off between serving a request as soon as possible, and the time it takes to finish serving it. It depends on the expected workload, which model should be used. Based on our research in Chapter 2, we would expect I/O-bound modules to be prevalent, so we use the thread pool model going forward.

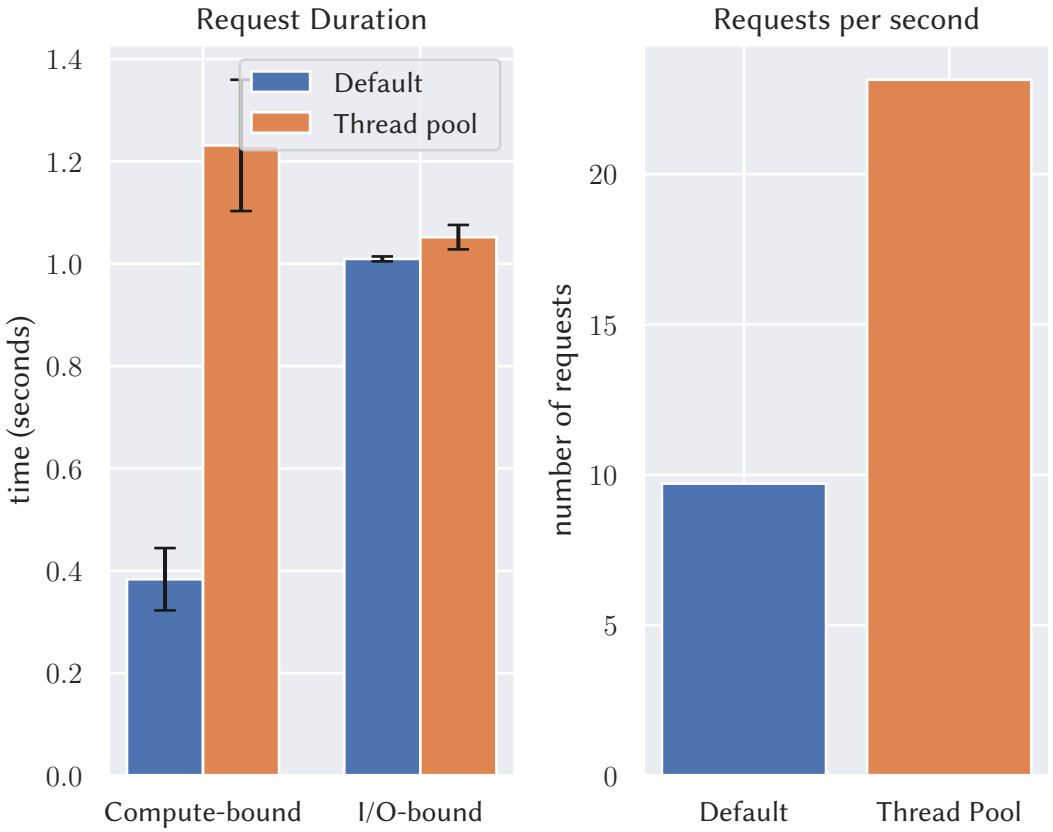


Figure 3.6: The average request duration and the average amount of requests finished per second under the default and thread pool execution model, sampled over 50 runs.

3.5.5 Implementation Alternatives

Instead of caching modules, we may wish to cache instances. The case for reusing instances is to avoid the costly setup in the action itself, like connection pools to external resources or to avoid rebuilding a cache on every request. We need to manage instances in that case, i.e. we need a pool of instances and a management system around it. When a new request comes in, we would ideally like to reuse an existing instance. That requires synchronizing with other threads to decide which thread gets to reuse an instance and which needs to create a new one. We would need per-instance synchronization to check whether an instance is currently in use or usable. It is questionable whether the synchronization overhead would be less costly than always creating a new instance, in particular because creating instances is a relatively cheap operation. Furthermore, with a new instance per request, we have per request isolation and fully stateless execution – something that the action developer can rely upon. For these

reasons we decided not to implement it.

As seen in Figure 3.4, a WASI context is created per `run` invocation. For example, if a module needs access to a file, creating the context requires opening file descriptors. Similarly, with a future networking API in WASI, this might require invoking more system calls. Frontloading this resource acquisition as much as possible can have net performance benefits. In particular, we should pre-create the WASI context, if possible. In `wasmtime` this is not possible, because the `WasiCtx` is not thread-safe. In `wasmer` on other hand, the `WasiEnv` is thread-safe and can be cheaply cloned to send across threads. Our current implementation needs one context per instance. That is due to the way we pass parameters to instances. This is not a fundamental limitation and could be done differently, which in turn would allow the `wasmer` wrapper to, for example, lazily create one `WasiEnv` per action the first time it is needed, and then reuse it for all future invocations of that action.

One potential inefficiency of our design is that our executor stores every container's module once. If OpenWhisk decides to create two containers `A` and `B`, both containing the same module `M`, then `M` will take up memory twice. One reason for that design is that in OpenWhisk vanilla, containers are completely separated from each other and no such optimization would be possible. Our executor is a more centralized solution and would allow for this optimization in the first place. Currently, we trade memory for speed and store the module per container. Thus, adding or removing a container is a fast insert or remove operation on the HashMap. Internally, the executor uses a thread-safe HashMap to store the modules and let many tasks or threads work with it simultaneously. A store-modules-once approach would require a global lock on the HashMap. During initialization, the executor needs to check whether the module already exists in the HashMap and if it does, increase its atomic reference count. This is already a critical section, since without synchronization, a concurrent destroy operation may have removed the module between the existence check and the increment. Consequently, this requires a global lock on the HashMap to avoid race conditions. This is probably not a performance issue, since lock contention is unlikely given that the average time between container starts or removals is much longer than the amount of time the lock needs to be held.

The question then is, whether it would help alleviate resource waste, that is, how often are multiple containers for the same action started? This primarily happens under bursty or batch workloads. Outside of such workloads, multiple containers with the same module are less likely to be started. But as we have seen in Chapter 2, an analysis of open-source serverless applications showed 84% of them having bursty workloads [Eis+21a]. The likelihood of receiving concurrent requests for the same action and in turn having to create multiple containers is therefore quite high. Hence, it seems implementing this optimization makes sense. However, OpenWhisk itself already has the potential to eliminate the waste with its per-action concurrency limit. This limit states how many concurrent invocations a container can handle at most. If we increase this value beyond its default of one, then OpenWhisk will send invocation requests for the same action to the same container, up to that limit, and without having to start a new container thereby duplicating the module. In particular, Figure 3.7 shows how a module can be reused in the same container – skipping the creation of a separate container entirely. Making use of this limit can thus be very important for performance and to avoid

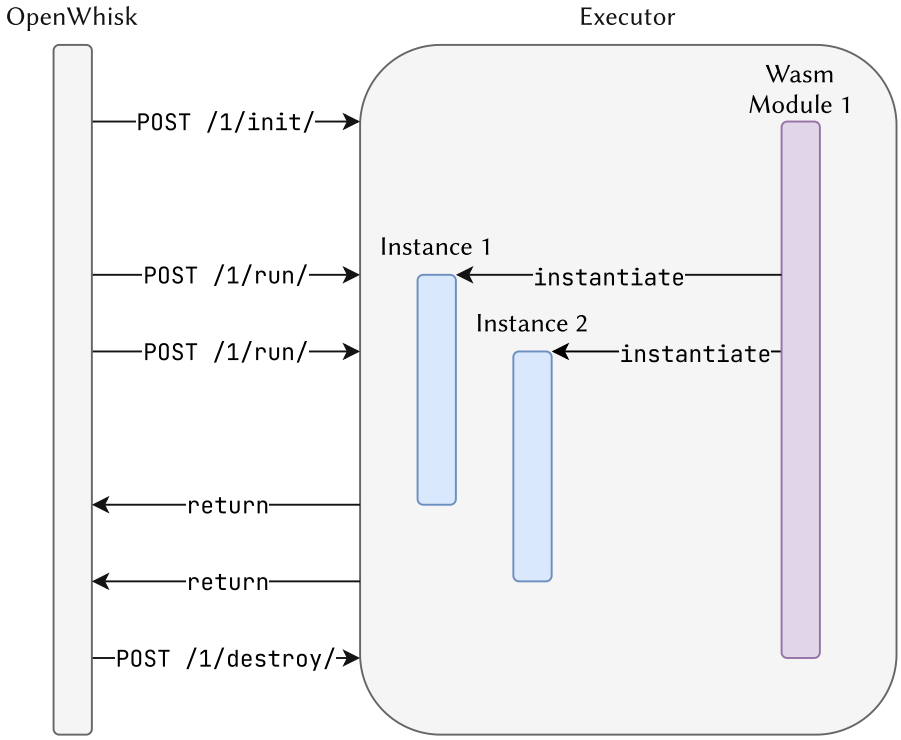


Figure 3.7: An example of a container's lifecycle when OpenWhisk's concurrency limit is set to a value greater one for that action. Note that a container is only represented in-memory by its module and instances, which is why there is no explicit »container« label in the figure. The container is initialized with a module by OpenWhisk's call to `/init`. Every `/run` request to the same container id spawns a new instance of a WebAssembly module. In particular, two concurrent requests to `/run` can be executed concurrently by the same container. In that case, the instances are instantiated from the same module. The instances are the actual »containers« here, since they are the ones who are isolated from each other and the host. They are short-lived and never reused to guarantee that isolation, in other words, there is a one-to-one relationship between `/run` requests and instances. The Wasm module will stay in memory until OpenWhisk's explicit call to `/destroy`, which occurs after a container has been idle for some threshold time.

resource waste. With a concurrency limit of c, up to c requests could be handled by a single container. If set appropriately for the underlying hardware and workload, no two containers need to be started on the same host and thus no resource waste would be necessary. Now, whether developers can be expected to set this limit properly is debatable. After all, their lack of knowledge about the hardware is rather the point, so this limit would perhaps better be set by the serverless operator. Overall, because of these considerations we decided adding this executor-internal optimization was not necessary.

CHAPTER 4

Evaluation

In this chapter, we evaluate our WebAssembly executor by comparing it to OpenWhisk vanilla. First, we discuss and define the types of workloads we use in our experiments. Then we go into details of our methodology, the experimental setup, which metrics we measure and what hardware we deploy on. All that will lead to answers for our research questions.

4.1 · Methodology

4.1.1 Goals

Recall our research questions.

1. By how much can WebAssembly runtimes alleviate the cold start latency?
2. How do the performance characteristics of the WebAssembly and Docker container runtimes differ?
3. What are the resource costs for keeping containers warm in both container runtimes?
4. Which WebAssembly runtime is the most suitable for use on edge devices?
5. How suitable is WebAssembly for serverless execution?

4.1.2 Workload Types

In Chapter 2, we described the serverless workload in detail, which guided decisions in our implementation. Specifically, we already used that research to get an idea about what concurrency model may be a better fit for our executor.

This research workload represents the current *cloud* workload and may or may not be transferable to workloads at the edge. Because of that, we also take some examples of visionary and potential use cases into account and infer scenarios, i.e. the order of cold starts and warm starts, as well as workload types, i.e. whether functions are CPU- or I/O-bound. That includes both serverless edge workloads and latency-sensitive ones.

Vital Signs Analysis Nastic et al. introduce measuring a patient's vital signs with wearable sensors during a situation of crisis [Nas+17]. Since lives are at stake, decisions must be made quickly. The scenario is both a latency-critical and an edge computing application. It serves as an example, where a serverless platform on edge computers can provide lower latency than sending the data to the cloud. Thus, part of the latency is reduced by moving the computation physically closer to the sensor. However, the second important factor for latency is the platform's execution time. When a sensor comes online and sends its first request, the execution suffers from a cold start. Once sensors are online, they need to analyze data in real time and continuously. Therefore, when the cold start happens, the platform may already be under load from other requests, and there may be concurrent cold starts. This is a scenario where high performance and efficient usage of the limited resources is required, and one where the same functions are called repeatedly – warm starts. The workload itself is data analytics, but not described in more detail. We classify this as primarily CPU-bound, because we can assume it involves data preprocessing, statistical analysis and perhaps even machine learning inference.

Edge AI Rausch et al. describe a motivational use case for AI at the serverless edge. A personal bio sensor measures blood sugar levels and sends it to an edge device, which refines a pre-trained model with that data and serves the model for inferencing [Rau+19]. The workload for this type of serverless function needs a significant amount of CPU-time for the machine learning tasks but also writes and reads parts of the model to and from disk. We classify this as both CPU- and file I/O-bound. Considering that measuring blood sugar levels is sufficient in greater intervals, e.g. 10 minutes, or whenever the user does a manual measurement, each of those would spawn a function that does the analysis and training, but could be discarded once it is finished. That is, keeping the function warm would be unnecessary and wasting resources. If the user actively took the measurement and waits for the analysis results, a low latency is required for a good quality of experience. Thus, the cold start of the analysis function should be as short as possible.

Augmented Reality Huang, Lin, and Lee implemented a mobile augmented reality (AR) system that analyzes images of book spines locally, e.g. on a smartphone, and then sends the features to the cloud to find a match in a database [HLL12]. Baresi and Filgueira Mendonça propose to offload the image recognition task to a more powerful edge device, to save on battery on the hosting device [BF19]. AR

applications require a low latency to provide a good user experience. Offloading such recognition tasks to nearby and more powerful edge devices presupposes their capability to execute them at high speed. This is a scenario where a serverless platform would receive a high number of concurrent CPU-bound requests.

APIs According to Eismann et al. – who conducted a manual analysis of serverless applications – 28% of their analyzed applications were APIs [Eis+21b]. Even with today's cold start latency, developers are already using serverless for user-facing applications, where latency is rather important. However, as Leitner et al. noted in their interviews with practitioners, many of them voiced concerns about the high latency associated with cold starts [Lei+19]. If the latency could be reduced significantly, even the APIs of latency-sensitive applications can be built on serverless. A useful API is typically accessing or modifying state. Since serverless functions by themselves are stateless, they need to communicate with other services such as databases, event buses and logging services. This composition requires many network I/O-bound calls and is usually light on the CPU, since the function spends most of its time waiting for I/O to complete.

We infer two workloads from these, while also taking our previous research into account. Both CPU- and network I/O-bound workloads are a primary type that ought to be handled well by a serverless platform. Some limitations apply due to WASI's immaturity or serverless idiosyncrasies. In particular, to model a full machine learning example, we should access an already trained model and run predictions on it. However, due to the lack of proper networking in WASI, we cannot retrieve it from an external service. Due to OpenWhisk vanilla using Docker containers, storing the model on disk would require granting each container access to the model's path on the host, which is not officially supported by the platform. Since file and network I/O are perceived similarly by the waiting function, we can focus on simulating one of these, which is going to be network I/O due to its prevalence.

We write the test actions in Rust and compile them to `wasm32-wasi` and `aarch64` or `x86_64` respectively, depending on whether we test a WebAssembly or Docker executor and which hardware we test on. This ensures that our results are not distorted by side effects from two different implementations in separate languages. In order to execute the native binary in OpenWhisk we use its blackbox feature. It is implemented by the `dockerskeleton` image, which lets us execute any action that adheres to a JSON-in, JSON-out protocol. We could also use the official Rust support by OpenWhisk, however, that takes a Rust source file and compiles it during the initialization phase – massively increasing the cold start latency. We believe using the blackbox version makes for a fair comparison between both container runtime types, since it measures the startup time of the respective container implementations, making it much more comparable to other serverless platforms. Although interpreted languages, such as JavaScript or Python, make up the overwhelming share of serverless languages, we use natively compiled actions to compare the precompiled WebAssembly to this optimum in performance. To also give an idea how performance compares to these prevalent languages, we also write

an experimental action in a language, which compiles to both JavaScript and WebAssembly. OpenWhisk makes its Docker runtime images available for the `x86_64` architecture only, so we need to build this image for `aarch64` [1] first, in order to execute on a Raspberry Pi. The concrete implementations of our evaluation actions are the following programs.

CPU-bound For this workload, we repeatedly hash a bytestring in a loop using the `blake3` [2] algorithm. This is a convenient choice since its reference implementation is written in Rust. Hashing is CPU-bound and free from system calls, so it is a good candidate for this workload type. We choose the number of iterations such that the completion takes roughly 100 ms in a native binary on the respective hardware, in order to have a non-trivial amount of work to be done for each invocation – significantly more than what OpenWhisk needs for its internal scheduling.

I/O-bound For this workload, we want to measure the effect of a blocking operation, such as an HTTP request. However, due to the lack of networking in WASI, we instead use the `sleep` system call. In WebAssembly this is implemented through capabilities. The module declares an import called `http.get`, which the WebAssembly executor provides. It is a function compiled into the executor, which blocks for exactly 300 ms, so the perceived effect of this function is the same as waiting for I/O completion.

4.1.3 Hardware Classes

Based on the introduced scenarios, we can also infer what types of hardware to evaluate our system on. The Vital Signs and Edge Analysis scenarios both work with lower end devices, like single board computers (SBC) or smartphones. For timely image recognition in the Augmented Reality example on the other hand, a more powerful machine would be required. The API example is more independent of the hardware and could run anywhere from SBCs to the cloud. Overall, SBCs can be considered characteristic of these edge computing scenarios and further ones include urban sensing, industry 4.0 and vehicular networks [Rau+20]. However, other configurations exist, for instance in edge data centers, where more powerful hardware can be found. Accordingly, we use a Raspberry Pi as an exemplary SBC and a server-grade machine to represent more powerful devices possibly located at the edge or in cloud data centers.

4.1.4 Setup

We want to measure the cold start latency and action execution time across all our evaluations. To that end, we utilize OpenWhisk's activation record, which is a collection of data resulting from each action invocation. Recall that OpenWhisk either initializes and runs the action,

[1] `https://hub.docker.com/r/pgackst/dockerskeleton`
[2] `https://crates.io/crates/blake3`

in the case of a cold start, or skips the initialization and runs it in a warm container. Each activation record contains a `waitTime` annotation, which is the time between the arrival of the request and the provisioning of a container. For Docker this means creating and starting the container. In WebAssembly executors, no equivalent operation exists, such that the `waitTime` will only correspond to the OpenWhisk internal overhead. Rather, the WebAssembly container is created during function initialization, which is not included in `waitTime`, but recorded as a separate annotation: `initTime`. Thus, for a fair comparison between both container runtimes, we define the cold start time as `waitTime + initTime`. Measured this way, the cold start time is simply the time between OpenWhisk receiving the request and the container being ready for execution, independent of the underlying container runtime. Since both times are part of the latency the user perceives, we believe this to be a meaningful measurement. It is important to note that this allows for a fair comparison in relative terms, that is, the recorded cold start times are not made up entirely of the container runtimes' startup cost, but also of OpenWhisk internal operations, which may account for a significant portion. This may limit the comparability to cold start measurements from other research. To facilitate this comparison, we provide the pure startup time, i.e. only `initTime` of WebAssembly executors as well. In Figure 4.1, we show a visual representation of what constitutes cold start latency and the request latency perceived by the caller, which is the amount of time from which throughput will be calculated.

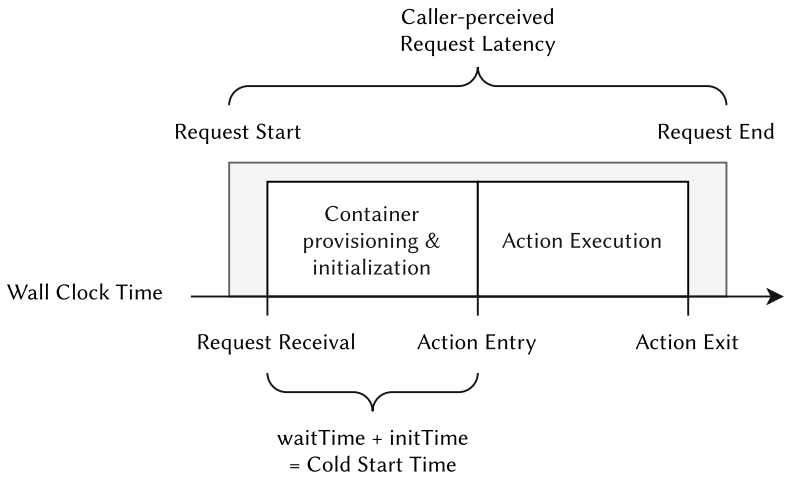


Figure 4.1: The two time measurements we use to evaluate the system's performance.

There are a two papers which describe their approach to measuring serverless performance in more detail, which we use as a basis for our evaluation.

- McGrath and Brenner use concurrency tests to evaluate a platform's ability to scale [MB17]. The concurrency test starts by issuing a request to the platform and reissues it as soon as it completes. At a fixed interval, it adds another request up to an upper bound.

In this manner, the load increases over time until a plateau is reached, where an increase in requests no longer increases the number of completed requests per time unit.

- Hall and Ramachandran characterize access patterns to serverless functions based on a real-world scenario. They describe three different patterns: Non-concurrent requests from a single client; multiple clients making a single request, possibly concurrent and multiple clients making multiple requests, also possibly concurrent [HR19].

Based on these, we devise the following experiments, building on the methodologies of McGrath and Brenner and Hall and Ramachandran to measure cold start times and throughput.

- To measure cold start times we first configure OpenWhisk's deallocation time as 10 seconds. We can then send requests at intervals exceeding 10 seconds to always trigger cold starts. In order to measure the effect of concurrency on cold starts, we send i concurrent requests, where i iterates through the set $\{1, ..., N\}$ and N is an upper bound we determine during the tests. After each iteration, we wait for OpenWhisk to destroy all containers before continuing. That results in exactly i concurrent cold starts on each iteration. This measurement will aid in answering RQ1.
- To evaluate the performance of the systems, we run concurrent requests under various workloads consisting of either CPU- or I/O-bound actions, or both. In the mixed scenario, we use the approach by McGrath and Brenner in their concurrency test, and pick workload types at random. This is to ensure that we evaluate the system for both types and close to a real-world scenario. It is also similar to the approach of Hall and Ramachandran, in that we have multiple concurrent requests but multiple clients are replaced by diverse workload types. The isolated experiments help us evaluate the pure performance of the respective workload types. Finally, we run a workload based on research data introduced in Chapter 2, which simulates that of a cloud provider. We go into more detail for each measurement in the respective sections. These experiments will help us answer RQ2.
- The third experiment aims to evaluate what the resource costs of actions are, specifically their code size and the amount of memory they consume while being held in-memory, ready for execution. We use the Docker API to measure each container's memory footprint and measure the allocated memory of the WebAssembly executor process to infer the costs for WebAssembly containers. This measurement supports the answer to RQ3.

Finally, all measurements and the experience of the design phase will lead us to a conclusion on RQ4 and RQ5. Our test procedure follows this pattern:

1. Based on the latest available stable release of OpenWhisk, which is `1.0.0`, compile the standalone version in our modified Wasm variant, as well as the unmodified vanilla version.

2. Start the OpenWhisk variant under test on the test machine.
3. Use the local `wsk` command line tool to create the test action(s).
4. From a separate machine, start the test procedure, which will send requests to the test machine.

4.1.5 Deployment

We run our evaluation on a resource-constrained edge device as well as an `x86_64` server-class machine. The edge device is a Raspberry Pi Model 3B, which has 1 GB of RAM and is running the latest version of the 64-bit Raspberry Pi OS (version `2020-08-20`). We are using the 64-bit version, because the `wasmtime` runtime cannot be compiled for the 32-bit `arm` architecture. The server has an Intel® Xeon® E3-1231 v3 (3.40 GHz) server-grade CPU with 4 physical cores and 8 logical threads. Mass storage is a Samsung SSD 850 EVO – which is where Docker images are stored and loaded from – and it has 8 GB of RAM and is running Ubuntu 20.04.2 LTS.

Due to difficulties in deploying OpenWhisk on the Raspberry Pi with Kubernetes and ansible, we use the standalone Java version for evaluation. The standalone version uses OpenWhisk's *lean* components for internal messaging and load balancing. These are components written specifically for deployment on resource-constrained devices. Since we are not benchmarking OpenWhisk itself, but rather the underlying container technology, we do not believe this to be a threat to validity.

4.2 · Cold Start

During testing, various issues arose that we want to discuss, in order to give context about the deployment process. The first version of the test created our CPU-bound test action, called `hash`, and then ran concurrent requests against OpenWhisk. In the result we noticed that for 5 and more concurrent requests, there were only 4 cold starts, independent of the underlying container runtime. It turned out that OpenWhisk would only create 4 containers and let other requests wait until one of the currently executing ones finished, so it could reuse the container. This is a fair optimization, but it does not let us test how 5 or more cold starts actually affect each other. So, instead of creating just one action, we create N actions with different names but the same code. OpenWhisk will treat these as different actions and execute them in separate containers accordingly. However, it will still only allocate 4 containers at a time, something that does not appear to be configurable, waiting for one of them to complete before removing it and starting the next. Thus, the performance of the container removal operation also becomes important, as it blocks the provisioning of the next. It is implicitly part of the cold start time as we define it, since a container that needs to wait due to a slow remove operation from another container will have a higher `waitTime`.

OpenWhisk vanilla showed further issues in its default configuration. On every container start, it executes a Docker pull operation even if the image was locally present, which meant checking

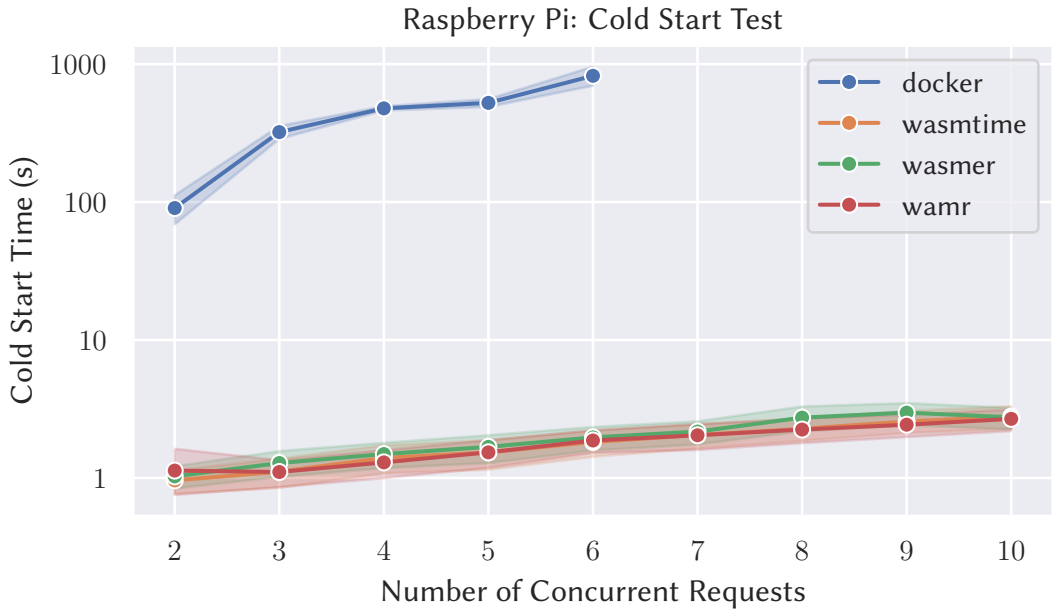


Figure 4.2: The cold start time with a CPU-bound workload on a Raspberry Pi with both Docker and Wasm-based container runtimes. Note that the scale is logarithmic rather than linear.

if the local image was up to date with the remote version, resulting in a higher `waitTime` than necessary. Therefore, we turned this behavior off, since it is not an inherent part of the container startup, which improved cold start times.

With two running Docker containers, OpenWhisk starts to approach the limits of the Raspberry Pi's memory. If OpenWhisk decides to start a third container, the system runs out of memory and freezes. Only after increasing the swap memory from the default 100 MiB to 1024 MiB were we able to test up to 6 concurrent cold starts. This is reflected in Figure 4.2 – where we plotted the result of the cold start test – with the Docker runtime having orders of magnitude higher cold start latencies than any of the Wasm executors. At 5 and 6 concurrent requests, the OpenWhisk logs showed Docker commands timing out and the testing time became unreasonably long – as reflected in the figure – which is why we stopped testing at that point.

Even at 2 concurrent requests, the Wasm executors have a cold start time of around one second, increasing with more requests. `wasmtime` has the most consistent behavior, followed by `wamr` and `wasmer`. Overall, the runtimes are similar in their startup performance, especially `wasmtime` and `wamr`, even though they are written in different languages and use different compilation strategies. The Wasm executors have, on average, less than 0.5% the cold start time of Docker.

For comparison, the cold start times on our `x86_64` server machine are plotted in Figure 4.3.

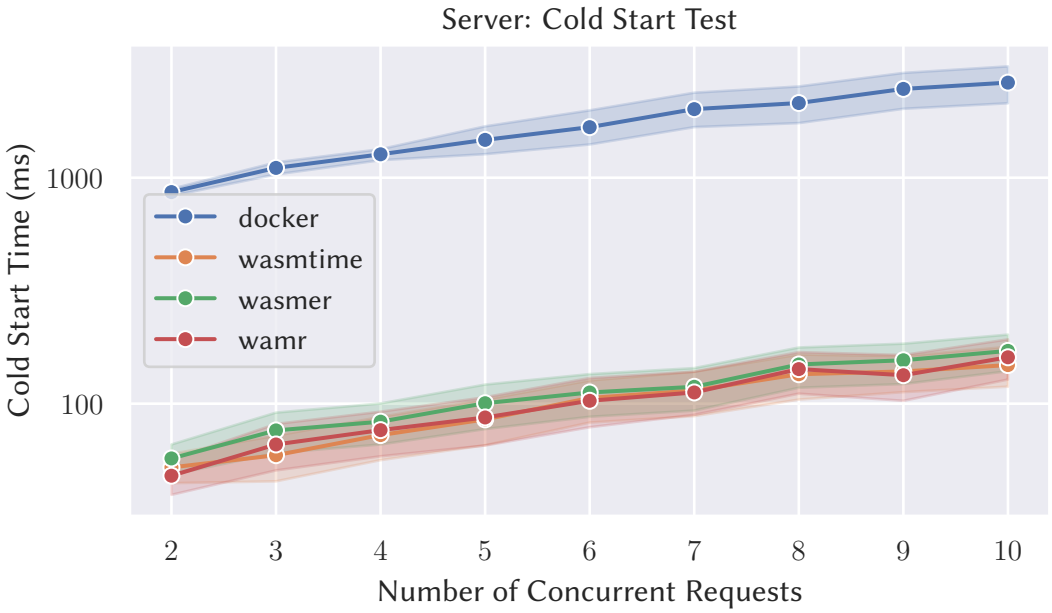


Figure 4.3: The cold start time with a CPU-bound workload on a server machine with both Docker and Wasm-based container runtimes. The Docker numbers fall between 869 ms and 2873 ms, while the numbers for WebAssembly executors range from 47 ms to 180 ms.

The relative differences are similar to the Raspberry Pi. In particular, the Wasm executors behave more predictably on both machines. Docker cold start latencies suffer under rising concurrency and vary much more. The average reduction in cold start time is 94% for `wasmtime` and `wamr` and 93% for `wasmer`.

These comparisons give a good holistic picture of the performance of both runtimes in the OpenWhisk context. However, because the measurements include other OpenWhisk specifics, they are less useful for a general container runtime comparison. Thus, we plotted `initTime` for the Wasm executors in Figure 4.4. This time represents the duration of the `/init` endpoint on the executor. It provides comparability to serverless platforms other than OpenWhisk since it isolates the overhead introduced *only* by the executor. It also shows the Wasm executors in more detail – providing a clearer picture for the evaluation among them, since the Docker runtime distorted the original graph to a large degree.

Independent of the hardware, `wasmer` is about 2× slower than `wasmtime` and 3× slower than `wamr`. At initialization time, the runtime receives the bytes and deserializes them into a `Module`, similar to `wasmtime`. However, since we use the `NativeEngine` in `wasmer` and it needs to open the shared object via `dlopen`, the bytes need to be written to a temporary file first. We do not expect this accounts for all of the differences seen, however. The runtime seems to have an inherently higher cost for deserialization and is slightly less consistent in its

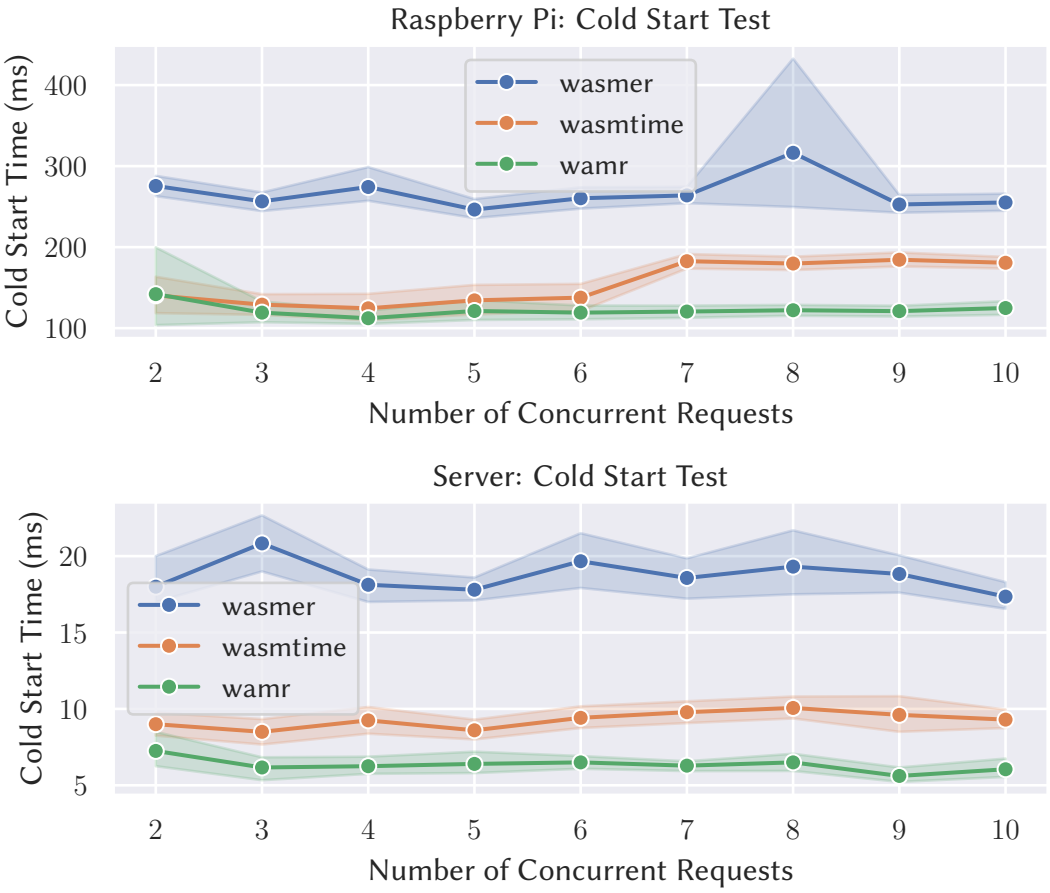


Figure 4.4: The cold start time with a CPU-bound workload on the Raspberry Pi only for the Wasm executors and with a confidence interval of 95%.

performance, based on the confidence intervals.

4.3 · Throughput

4.3.1 Mixed Workload

In the previous cold start test it became apparent that running OpenWhisk vanilla with Docker on the Raspberry Pi is challenging. Because OpenWhisk runs four containers under heavy load, it puts too much strain on our test edge device. Thus, we make use of the previously introduced concurrency limit that we can set for each action. We set the limit to 10, such that 10 actions can be executed concurrently in the same container. With this setup, we can run the

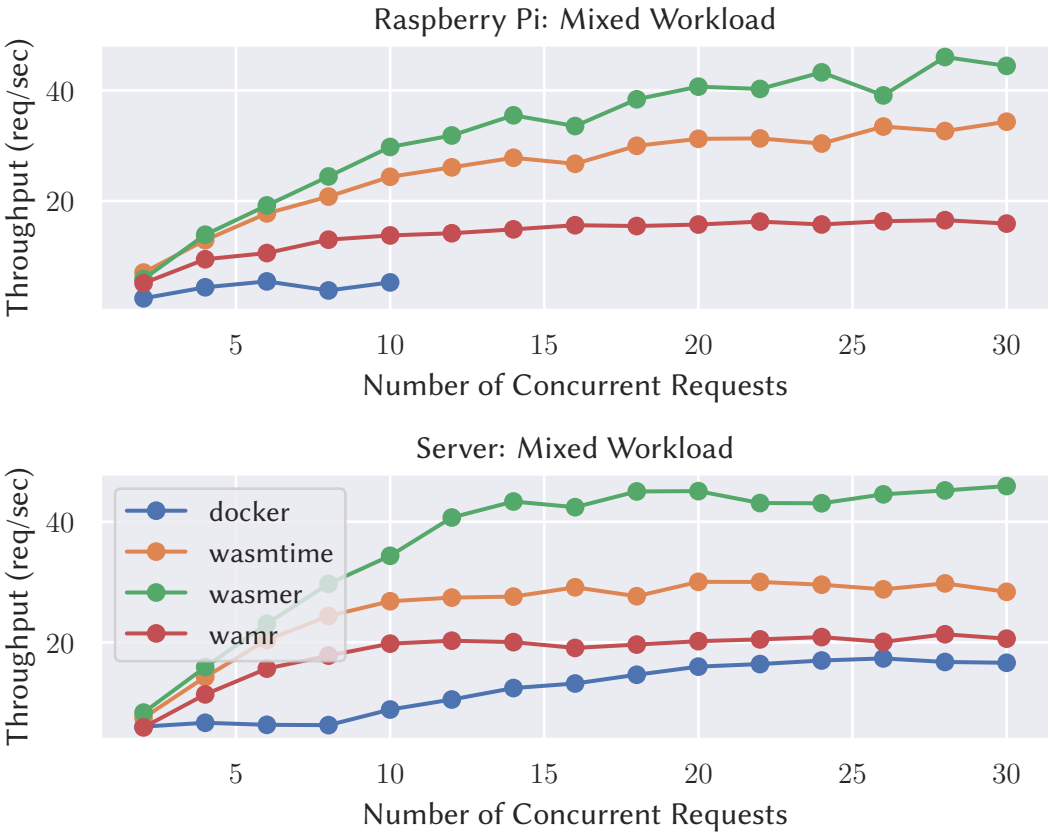


Figure 4.5: The throughput in requests per second with an equal amount of I/O- and CPU-bound workload on a Raspberry Pi and a server machine with both Docker and Wasm-based container runtimes.

load test for the Pi, because OpenWhisk only creates two containers – at least up to a certain threshold. Log collection is another process that tends to fail often, so we use `--logsize 0` when creating the action, to prevent OpenWhisk from attempting to collect them. We use the same parameters for creating the Wasm actions. We run one iteration of this test before starting the measurement, in order to pre-warm containers. This is to test the system in a warm state isolated from the effects of the first cold starts and get an unaffected measurement of the performance of WebAssembly containers compared to native binaries in Docker containers. At some points during the test run the system might decide to scale up if the load crosses a certain threshold and perform a cold start of another container, which would be included in the measurement. This would be the most realistic performance for the Vital Signs Analysis example we gave earlier, i.e. continuous load on a function with the occasional cold start. With this setup, we get the results in Figure 4.5.

OpenWhisk vanilla manages to handle up to 10 concurrent requests on the Pi, at which point it creates a third container, which forces the Pi to use its swap memory and become orders of magnitude slower. Again, at this point we stopped testing. The workload we run is mixed and made up of an equal amount of I/O- and CPU-bound actions. Whether one or the other is executed is determined at random, such that, on average, half will be I/O-bound and half will be CPU-bound. We execute two runs and take the average for each number of concurrent requests. Note that the CPU-bound workload is chosen such that the execution time is similar on the Raspberry and the server machine, since the blocking I/O-bound workload also takes the same amount of time on both devices. That makes the results more comparable. This test is executed with Apache JMeter[3], a load testing tool.

Runtime	Raspberry Pi	Server
`wasmtime`	3.8	2.2
`wasmer`	4.2	3.0
`wamr`	2.4	1.6

Table 4.1: The average throughput that the respective WebAssembly executors achieve over to Docker, in the mixed workload scenario.

We can immediately see a stronger difference in all container runtimes than in the cold start test. The performance improvements are listed in Table 4.1, where we can see, that on both devices, the WebAssembly executors are better able to handle a mixed workload. The executors perform similarly on both hardware classes when examining the order of fastest to slowest. However, on the server, `wasmer` has a greater lead on `wasmtime` than on the Pi. Both of these runtimes perform with higher fluctuations than `wamr`, which has a very consistent behavior. In general, the effect is more pronounced on the Raspberry Pi, indicating that Docker is less suitable for resource-constrained edge devices than for cloud hardware.

The lead is taken by `wasmer`, likely because we configured it to use the `LLVM` compiler toolchain, which produces high-quality native code. On the other hand, `wamrc` also uses `LLVM` for ahead-of-time compilation, but performs worse. In our later experiments we will see that `wamr` has a stronger tendency to optimize for size, perhaps at the expense of speed, which may be a contributing factor. Additionally, its API does not allow us to thread-safely initialize the module once and run it multiple times. Instead, the module needs to be instantiated from the raw bytes on every `run` call. Because of that, the implementation is less efficient compared to the other executors. `wasmtime` is configured to use the `cranelift` JIT compiler. Because of the inherent trade-off such a compiler has to make between compilation speed and code quality, it is perhaps unsurprising that it is less performant than `wasmer`. Interestingly, it still performs better than `wamr`.

[3] `https://jmeter.apache.org/`

4.3.2 I/O-bound Workload

To get a more isolated picture of the container runtimes performance, we test the workloads separately on our server machine. The I/O-bound test is supposed to test the container runtimes' ability to handle concurrent network requests. As we have explained in Chapter 2, serverless functions are frequently used with external services, such as databases. This test simulates this behavior by blocking with a thread `sleep` just like an I/O request would block. In the Wasm executors this is implemented with capabilities, which allows the Wasm runtimes to supply native host functions as imports to the module. In this instance, we create our action using `--annotation net_access true` with the `wsk` cli. The Wasm executor then provides the `http.get` function as an import to the module. Each executor implements this with the aforementionend `sleep`. The module can then use these functions as if they were part of the module. This way, we could easily implement an actual HTTP GET request. However, the lack of interface types means the module can only call native functions with integer parameters and return values, such that the practical use is still inhibited.

To isolate the performance from cold starts, a warmup phase is executed before gathering data, which lets OpenWhisk create the containers and keep them warm. The throughput is plotted in Figure 4.6. OpenWhisk imposes the same 4 container limit for all container runtimes, which puts a limit on the throughput for I/O-bound workloads. However, again we can use the concurrency limit to increase performance – here we set it to 3. The `dockerskeleton` we use in OpenWhisk vanilla performs very poorly for I/O-bound workloads. Two concurrent requests take twice the time that one request takes. This may be a limitation of the action proxy used in the `dockerskeleton`. It can only achieve more throughput by starting new containers. Thus, the concurrency limit essentially delays the peak I/O-bound throughput for the Docker version until OpenWhisk creates 4 containers.

The Wasm executors on the other hand, have no such issue and can scale up to the 4 container limit. Since we set the concurrency limit to 3, this allows 12 requests to execute concurrently and we reach the throughput peak at that point. A higher concurrency limit would consequently increase the possible throughput. Since blocking takes the same amount of time in every Wasm runtime, this performance is mainly determined by the executor itself and its threading model – something we have explored at the end of Chapter 3. Other than the OpenWhisk limitations, the performance rests on the number of threads that can be spawned on the system.

Because the concurrency limit has such a pronounced effect on the performance, we also plotted the same workload with a concurrency limit of one. Given what we just described, it is no surprise that at 4 concurrent requests the system's throughput is saturated. This test also implicitly shows the direct comparison of the container runtime's overhead, even when the execution time is predetermined. Evidently, Docker has a larger overhead than the Wasm executors, all else being equal.

Figure 4.6: The throughput with an I/O-bound workload on a server machine with both Docker and Wasm-based container runtimes. The y-axis shows the throughput in requests per second, while the x-axis shows the number of concurrent requests, each of which is repeated 50 times to generate a consistent load over a short period of time.

4.3.3 CPU-bound Workload

Figure 4.7 shows the result of the fully CPU-bound load test. This test shows a large performance gap among the Wasm executors, but also to Docker. On average, the Wasm executors process only 39% (`wamr`), 57% (`wasmtime`) and 88% (`wasmer`) of the throughput that Docker achieves.

Docker was particularly hard to evaluate for this workload. OpenWhisk – at least in the standalone version – is hardly able to saturate the CPU, because the system will not create enough containers to fully utilize it. This likely occurs because each container is run with the `--cpu-shares` Docker option, limiting the share of the CPU the container is allowed to use. Hence, if not enough containers are created, some invocations have to wait for others to complete, even though the CPU is not fully utilized. By increasing the concurrency limit,

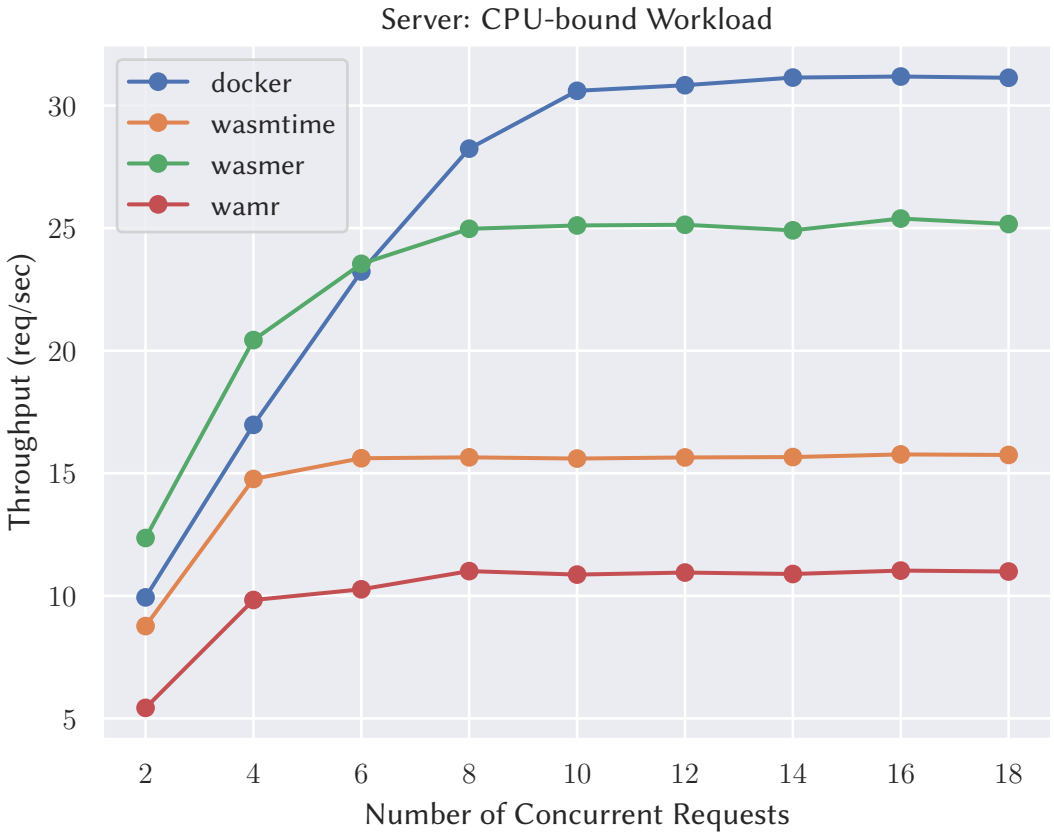


Figure 4.7: The throughput with a CPU-bound workload on a server machine with both Docker and Wasm-based container runtimes. The y-axis shows the throughput in requests per second, while the x-axis shows the number of concurrent requests, each of which is repeated 50 times to generate a consistent load over a short period of time.

we can work around that and let OpenWhisk execute actions in the same container to fully leverage the assigned CPU shares. Increasing this limit too much, forces more actions to share the same CPU shares, while not creating new containers. However, increasing the memory on the container, which in turn makes OpenWhisk increase the CPU shares, does not have a positive effect either. Rather, it delays OpenWhisk's decision to create new containers even longer, resulting in even poorer performance. After some trial and error, we arrived at values which seemed to leverage the CPU rather well, the default 256 MB of memory and the concurrency limit set to 3. The Wasm executors showed no such signs, simply because no memory or CPU limiting is implemented in the first place. This will have to be a feature of a fully production-ready Wasm executor, however.

Cold start latencies and resource costs for keeping containers warm are ignored in this test.

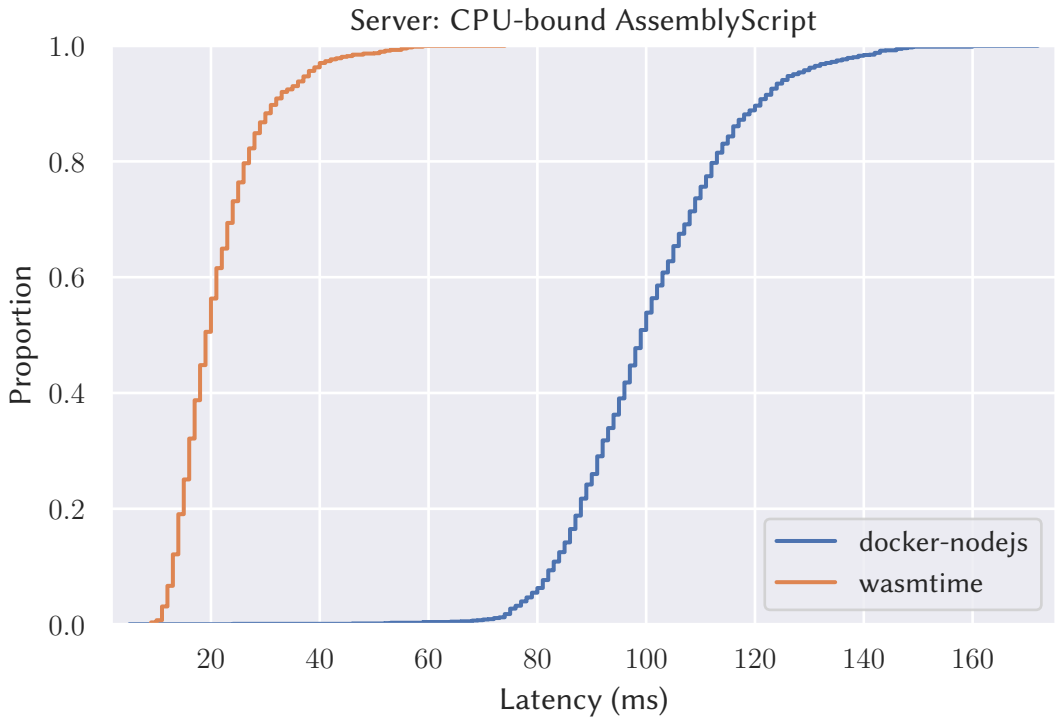


Figure 4.8: The ECDF for the execution of a WebAssembly module calculating prime numbers executed in the `wasmtime` executor and JavaScript run in the `openwhisk/action-nodejs-v14` image. The y-axis shows the proportion of total requests that were finished with the latency given on the x-axis, or less.

With that in mind, the test shows that for a steady stream of highly concurrent CPU-bound requests, using Docker is the best option; in particular if the container is likely to be reused many times. This comes perhaps not unsurprisingly, given that the executing code is Rust compiled to a native `x86_64` binary. There is hardly a way to generate faster code.

In fact, if we compare the pure execution time of our WebAssembly action outside the serverless context, we see the native binary being even faster. For that, we precompile our action for `wasmer` and then measure only the pure function execution time, without the setup of necessary contexts like `Engine` or even `Module`. In that case, `wasmer` achieves just 60% of the throughput compared to the native binary. Compared to the 88% from above, it indicates that OpenWhisk with Docker is unable to fully utilize the available hardware, either because of the discussed configuration or because of internal overhead, like queuing and container management.

Note that using native code in serverless functions is not a particularly popular approach. Recall the serverless workload types we described in Chapter 2. We noted that functions which

exhaust the CPU are more of a niche and that, if the executor can offer performance on the level of `node.js`, it would roughly be on the same level as most of today's functions. To put this into perspective, we run an experiment to demonstrate the potential improvement WebAssembly can give in comparison to today's prevalent serverless runtime: `node.js`. To that end, we write a function in AssemblyScript[4], a strict variant of TypeScript[5]. TypeScript itself is *typed JavaScript* and a popular way to write programs that compile to JavaScript. Given the widespread support for TypeScript in serverless SDKs such as the AWS SDK[6] or the *serverless*[7] framework, we expect most developers who run JavaScript to be actually programming in TypeScript. AssemblyScript can compile directly to WebAssembly with the `binaryen` toolchain, that we already use for optimizing our Wasm modules. Because AssemblyScript uses the types from WebAssembly itself as its basis, not all TypeScript code can simply be compiled to WebAssembly. Some minimal changes may be necessary to compile to both targets. We set up a sieve of Eratosthenes to calculate prime numbers; a CPU-bound workload. The produced WebAssembly module does not run in all our Wasm executors out of the box, so only `wasmtime` is shown. Calculating prime numbers is a use case where raw performance matters and `wasmtime` achieves more than 3× the throughput of `node.js`. The empirical cumulative distribution function (ECDF) of the results can be seen in Figure 4.8. It shows that `wasmtime` performs more consistently than the `node.js` execution, evident by the lesser spread on the former's curve. Given that the WebAssembly code is precompiled, while `node.js` must compile hot code paths just-in-time, this is not surprising. On the other hand, since the sieve is a loop and thus a very hot code path, the loss in performance is slightly surprising. Thus, for a computationally intensive serverless function written in TypeScript, it may be worth porting it to AssemblyScript, so it compiles to WebAssembly. However, AssemblyScript's goal is *not* to be a TypeScript to WebAssembly compiler but rather an extension to the web ecosystem, complementing TypeScript and JavaScript. As programming languages go, AssemblyScript is still fairly young. While it shows that the accessibility to write source code which compiles to WebAssembly is lower than using systems programming languages like C or Rust, whether it could eventually act as a general-purpose language to write serverless WebAssembly, is uncertain.

In light of this experiment and the preceding one, a fully CPU-bound WebAssembly module reaching 88% of the performance of a native binary in Docker, but with much reduced startup costs, would seem to be an acceptable compromise.

4.3.4 Realistic Workload

Given what the research about serverless workloads showed in Chapter 2, we run another experiment. Recall the analysis from Microsoft Azure that found 81% of functions are invoked less than once per minute, but those accessed more frequently make up 99.6% of all invocations

[4] `https://www.assemblyscript.org`
[5] `https://www.typescriptlang.org/`
[6] `https://github.com/aws/aws-sdk-js`
[7] `https://www.serverless.com/`

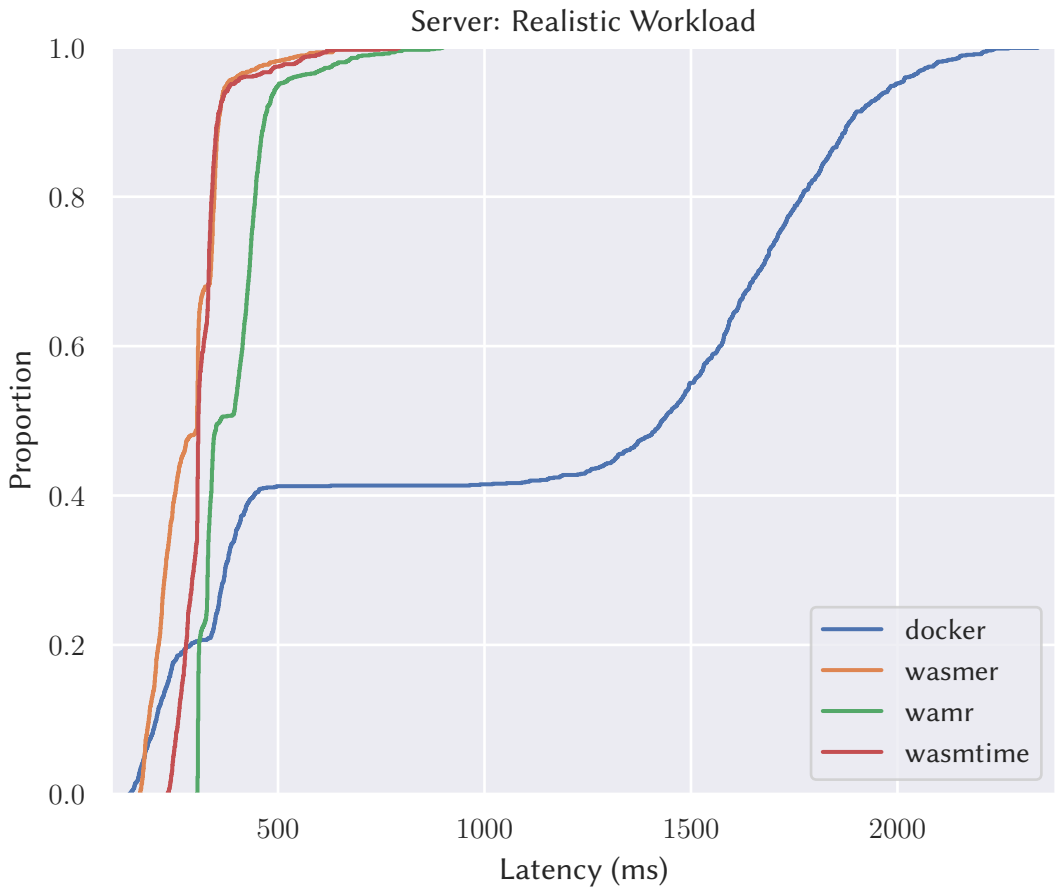


Figure 4.9: The ECDF of our experiment, designed based on serverless usage data from Microsoft Azure. The y-axis shows the proportion of total requests that were finished with the latency given on the x-axis, or less.

[Sha+20]. This shows that there is a small number of functions accessed frequently, and a larger amount of functions accessed sporadically. Since we cannot simulate a test on the scale of Azure, we map the workload to a smaller scope. The test sends a sufficient number of concurrent requests in 4 threads. The latter comes from the 4 container limit imposed by OpenWhisk. As we saw, blocking actions are not handled well by OpenWhisk vanilla. As a consequence, we set the concurrency limit of the individual actions to their default of one, in order to not distort the result due to this architectural issue. For each request sent, we use randomization such that 90% of requests are sent to the same two actions. These are equally likely to be either CPU- or I/O-bound. For the other 10% of requests, one of 60 actions is selected at random, where the likelihood is again 50% for them to be either CPU- or I/O-bound.

Thus, there is a high probability that a function is called, for which a warm container should be available. Occasionally, one of the rare functions is called which will inevitably cause a cold start. In summary, this test will show the performance of the system under load, where most functions are warm but the occasional cold start will need to be handled. We have plotted the ECDF of this test in Figure 4.9.

For the Docker runtime we can see that around 40% of requests were finished in less than 500 ms. However, around 60% of requests were only finished in less than 1500 ms. We can reasonably explain this with warm and cold starts. The pure function execution times are always less than 300 ms, because we configured them so. From Figure 4.3, we know that the average cold start time of Docker on the server at 4 concurrent requests is 1269 ms. Thus, once a cold start is incurred, it adds to the latency seen by the caller. Therefore, either functions run in a warm container for around 500 ms, or they cause a cold start and need the same amount of time in addition to the cold start, which is why we see the sudden increase at around 500 ms. Once cold starts are occurring, they have a detrimental effect on the overall system performance as indicated by the increasingly higher latencies beyond the plateau.

Up to 20% of invocations handled by the Docker runtime finish in the same amount of time or faster than WebAssembly executors. Given the bend at 40% and our experimental setup, this is because half of the actions executed warm are CPU-bound, while the other half is I/O-bound. This is consistent with the previous data. As we have seen in Figure 4.7, the Docker runtime is able to handle CPU-bound actions faster, thanks to directly executing native code. But it became clear in Figure 4.6 that the runtime is less well equipped to handle I/O-bound ones. Hence, why we see the bend at around 20%, where the Docker curve migrates from being faster than the Wasm executors to being slower.

The WebAssembly executors show latencies beyond 500 ms only for a very small proportion of the requests. Similar to Docker, there is a visible bend in some of the executor's curves, but not from cold starts. Since it is located at around the 50% mark, this has to be due to the different workload types. Data from the earlier experiment show that the cold starts for the Wasm executors are 72 ms (`wasmtime`), 76 ms (`wamr`) and 83 ms (`wasmer`) for 4 concurrent requests. Therefore, the cold starts are not significant enough to saliently show up in the ECDF. About half of `wasmer` 's requests are faster before it drops to `wasmtime` 's levels. Taking Figure 4.7 into account once more, where `wasmer` was the fastest Wasm executor, half of these requests correspond to CPU-bound actions. The curve for `wamr` is analogous. Notably, `wasmtime`

shows no salient points, but rather has the smoothest graph. Most likely, it so happens to execute both workload types at around the same speed.

One question that might arise while looking at this figure, is why 60% of the requests running on Docker seem to be suffering a cold start. If we send 90% of requests to the same two functions, then the same amount should be in the hot path. Due to being limited to 4 containers, OpenWhisk will remove a container to make room for another one under sufficient load and if it deems it necessary. If we look at the number of cold starts, indeed we see around 58% of requests requiring one. This is a fundamental limitation of OpenWhisk, perhaps due to our usage of the standalone version and would likely not apply to a, for example, Kubernetes deployment. However, it affects Wasm executors as much as Docker according to our numbers. But evidently, the Wasm executors are able to cope much better with repeated cold starts.

4.4 · Resource Usage

This test aims to evaluate what the resource costs are, for keeping actions in memory and ready for execution. To that end, we run our cold start test again and measure the amount of memory for each container once the actions have finished execution, but before OpenWhisk removes them. At this point, Docker containers have been paused by OpenWhisk. We use the memory usage output of `docker stats` as the basis for evaluation. For the Wasm executors we read the resident set size (rss) of the entire process. Hence, we also measure the overhead of the web framework we run on, but also the Wasm runtimes' objects which are instantiated globally, such as `Engine`s. Thus, this method is slightly imprecise in that it reads more than the pure memory occupied by the container. It would be difficult to define what constitutes the container, too, since the data structures, global or not, are all linked together to produce the running system. Since the Docker container also contains other data only more or less relevant to the execution, this is acceptable. It gives us an estimate for the average total cost of keeping a container in memory. Ultimately, the entire memory consumption is what matters. This method works slightly in favor of Docker, since we do not measure the memory allocated to the Docker daemon process. For the Wasm executors, we divide the measured amount of memory by the number of containers that are kept alive at that point, to get an estimated memory usage for each container. The amount of memory consumed when idling, i.e. after startup, are 7.07 MiB for `wasmtime`, 4.23 MiB for `wamr` and 5.28 MiB for `wasmer`. At first glance, it may seem that `wasmtime` and `wasmer` fare the worst. However, this may be due to the fact that their APIs are the most suitable to our needs, because they let us instantiate more of the necessary data structures ahead-of-time than `wamr`. We are happy to trade this memory for less setup costs later.

For the Wasm executors, we expect the amount of consumed memory to depend primarily on the size of the Wasm module. So we run this test thrice. For that we compile the `hash` module with different options to get different module sizes.

- Release mode with link-time optimization (LTO)

- Release mode without LTO
- Debug mode

LTO enables compilers to perform expensive, but very effective optimizations at compile time, when the different libraries of the program are linked together. Compiling our hash module with LTO brings the size down to 16.5% (312 KiB) of the non-LTO version. The resulting sizes of the Wasm modules and the native binaries are given in Table 4.2, to provide context for the following results.

Compilation mode	WebAssembly	Native Binary
Release + LTO	312 KiB	2.01 MiB
Release	1.51 MiB (×4.83)	3.80 MiB (×1.89)
Debug	4.76 MiB (×15.25)	6.46 MiB (×3.21)

Table 4.2: Size of the hash module when compiled to `wasm32-wasi` and `x86_64-unknown-linux-musl` targets respectively. To provide three different container sizes we use release mode with LTO and without as well as debug mode. The relative size increase over the release with LTO are also given.

The measurements can be seen in Figure 4.10. These confirm that for the Wasm executors, the in-memory container size is largely dependent on the module size, simply because the container *is* the module in memory plus the necessary supporting data structures. In particular, the modules are held in memory for the fastest possible access. For Docker on the other hand, we have to explore in some more detail how an action is executed to understand the results. In order to run native binaries, we use the `dockerskeleton` image. When a container is created from that image, the so-called action proxy starts. This is a proxy service that implements the familiar `/init` and `/run` endpoints. It is generic over the underlying executable, so it could run bash or perl scripts as well as any native binary. When `/run` is invoked and the initialized action is executed, the script or binary runs, its result is read by the proxy and returned. Consequently, after the binary finished execution, the memory of its process is freed as usual. Hence, when we measure the memory in-between action executions, we only measure the amount needed for the action proxy and the rest of the container. It does not matter whether the action itself is 1 MiB or 10 MiB; the memory cost of keeping the container warm is the same. This is confirmed by our measurement. We can expect Docker containers to be consistent in their memory usage independent of the concrete action. This particular execution model applies to the `dockerskeleton`, but other images, such as the one based on `node.js`, may behave differently.

Between different module sizes, the results show `wasmer`'s containers to double their memory usage from the LTO to the non-LTO version, with another increase by roughly 3× to the debug module. For `wasmtime`, the difference between LTO and non-LTO is negligible, and then the memory usage increases by 2.4×. For `wamr` the difference between the release modes is barely noticeable and then goes up by 36%. Docker's container sizes between release modes hardly differ and only increase by 2.5% to the debug version.

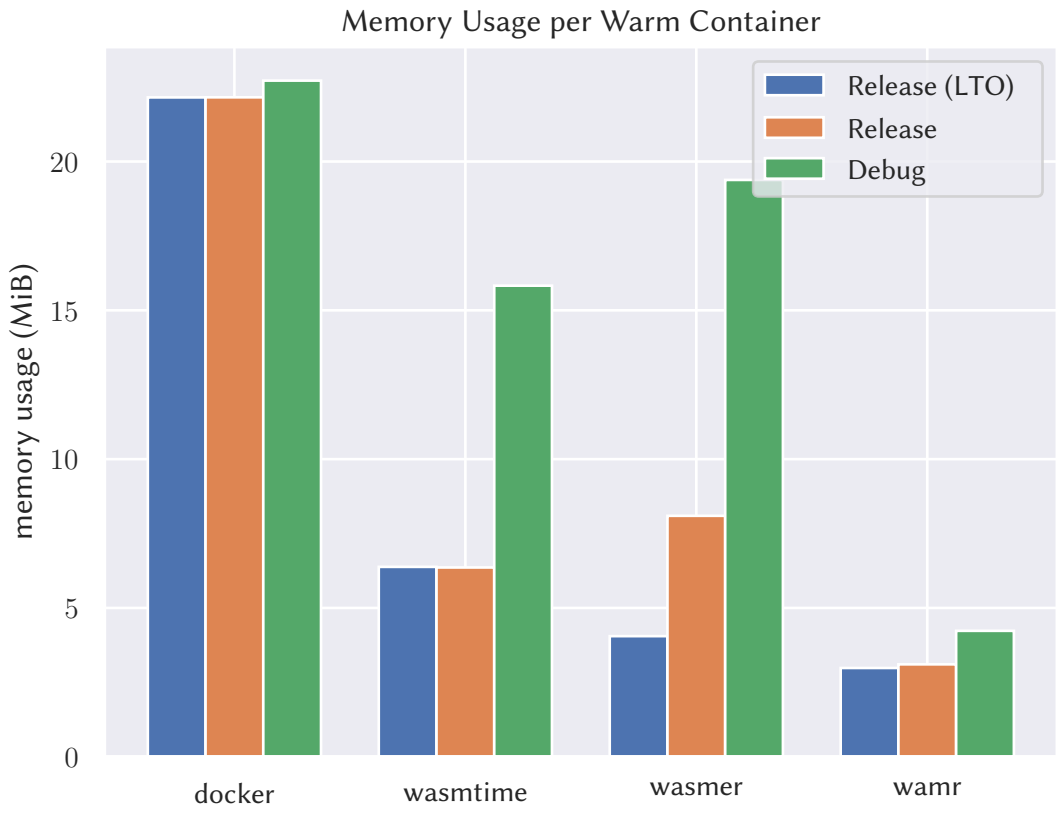


Figure 4.10: The average memory usage of Docker and WebAssembly containers given different module sizes.

LTO does not seem to make a large difference to `wasmtime` or `wamr`, but `wasmer` does not seem to apply optimizations when precompiling. The results show that for `wasmer` and `wasmtime` the container size primarily depends on the module size, except for LTO, where we suspect `wasmtime` to apply optimizations when precompiling and thus coming out at the same size. Due to its JIT nature, it is probably more likely to trade memory for compilation speed or code performance. While `wamr` seems to be very memory efficient, it is important to note that due to its architecture, we do not hold instantiated modules in memory in-between `run` invocations, but only the module bytes themselves.

Given these results and our minimal but realistic WebAssembly module size, we could hold more WebAssembly containers in the same amount of memory than Docker. Especially more of the less popular functions could be kept warm resulting in less cold starts. Although, with cheap cold starts, there is an interesting trade-off between using resources to keep functions alive,

Runtime	Release (LTO)	Release	Debug
`wasmer`	5.5×	2.7×	1.2×
`wasmtime`	3.5×	3.5×	1.4×
`wamr`	7.4×	7.1×	5.3×

Table 4.3: The ratio of memory usage for the different module sizes in WebAssembly executors compared to the Docker container runtime.

and saving memory but paying the small, but nonetheless non-zero latency cost for the cold start. Table 4.3 shows how many more containers could be kept warm in the respective Wasm executors compared to Docker. When optimizing the module size with LTO or other techniques, we can fit more than five times as many containers in the same amount of memory compared to Docker when using `wasmer`. Even for big modules we stay below the consistently higher memory usage of Docker. In more constrained environments with lower memory availability, `wamr` – primarily due to how we use it in the executor – would be a good choice given its much lower memory usage, but at the cost of performance as we have seen in the previous measurements. Ultimately, the overhead introduced by the executors themselves is minimal and mostly dependent on the module size. This puts the control in the action developer's hands. If they manage to strip their binary down to the necessary contents, they can expect a small memory footprint at execution time. With Docker, there is a consistent memory usage but higher than the Wasm executors and less opportunity for developers to influence its footprint.

If memory is a priority, other optimizations such as using a smaller memory allocator are also possible. Written for the use case in browsers, `wee_alloc` [8] is an allocator that trades speed for size. Rust also allows writing `no_std` programs, i.e. programs without its standard library. Since this is not how we anticipate most functions would be written, we did not employ this technique. However, without the standard library only the code and libraries the developer brings in end up in the binary, granting a maximum of control over the program's size. Using other languages may also come with size benefits. The prime number example we wrote in AssemblyScript is very small in comparison to the Rust-produced modules and comes out optimized at about 22 KiB. It shows that modules can get very small and the in-memory container sizes would benefit accordingly, for most of the runtimes.

It would also be possible to accommodate systems with very constrained memory settings, by fully exploiting the WebAssembly runtime's design. Wasm runtimes do not fundamentally have to keep modules in memory. The command line versions of `wasmtime` or `wasmer` cache compiled modules on the filesystem by default, so the support already exists. `wasmer` would be a prime-candidate for this type of optimization. The executor instantiates a module from in-memory bytes it receives from OpenWhisk. However, `wasmer` – in its AoT mode with `LLVM` and the `NativeEngine` – needs to write the bytes of the shared object it gets to a temporary file, because `dlopen` only works on files. Thus it already needs to write to a file internally because of this restriction. Doing this procedure in the executor itself would be possible, since the API allows us to create a module directly from a file to skip this process. It means we

[8] `https://github.com/rustwasm/wee_alloc`

would have to create the `Module` from that file on every `run` invocation, in order to save on memory. Given that the `init` time of `wasmer` is currently around 20 ms, according to Figure 4.4, it is not infeasible to incur this cost on every `run` if memory is a priority. Alternatively, `wasmtime`'s deserialization cost is just half, and thus could be similarly cached to the file system. Again, there is a trade-off between memory and performance, but most importantly, it *exists* and can be made by the implementor for whichever property is more essential.

CHAPTER 5

Related Work

In this chapter, we look at approaches for reducing cold starts such as improvements to the traditional container runtime but also including WebAssembly.

5.1 · Incremental Approaches

These approaches attempt to work around the limitations of OS-level virtualization with additional techniques, typically implementing some form of caching or pre-creation. Thus, they can be seen as incremental improvements.

5.1.1 Pre-Warming

Lin and Glikson implement a pooling solution for Knative, an open source platform built on Kubernetes [LG19]. Pods are pre-warmed and added to a management pool. When a request comes in, instead of cold starting, a pod from the pool is used for execution. According to them, the migration takes around 2 seconds. The response to first request for a simple HTTP server example is improved by 2.3×, while a more complex image classifier is sped up by 5.2×. In absolute terms, however, the response times are still in the range of multiple seconds. Notably, this approach also includes pre-warming not just the container and language runtime, if applicable, but also the function itself. That in particular explains the large gain in the image classifier example, which needs to load its machine learning model from disk in the first invocation. This example also shows that our decision against a Kubernetes-based approach due to performance concerns is warranted.

Thömmes describes the pre-warming approach used in OpenWhisk [Thö17]. The platform operator needs to configure the pre-warming system by specifying which containers are pre-created. For instance, if the anticipated load is primarily JavaScript functions, then the operator

configures some number of `node.js` containers to be started. In contrast to the previous approach, only the container setup itself and that of the language runtime is taken out of the hot path. Any function initialization still needs to happen during `init`. Hence, the only difference between pre-warmed and warm containers is that the initialization still needs to happen before execution. The upside is that the container is generic over any potential JavaScript function and not bound to any specific action. While JavaScript is interpreted or just-in-time compiled, and therefore does not add much to the initialization, for some languages it can be much longer. As we mentioned before, the reason we used the `dockerskeleton` rather than the official Rust runtime is that the source code needs to be compiled during the initialization. While this allows the portable source code to be compiled for any underlying hardware and then executed at native speeds, it comes with a significant compilation cost. Our approach improves on that, since WebAssembly itself is a portable, yet more efficient and compact code representation and could be executed immediately, but also compiled again for more performance.

While these attempts manage to reduce the cold starts, they come at a high cost of resources. Pre-warming a sufficient amount of containers to avoid cold starts even during bursts of requests requires seizing a significant amount of memory. This is especially problematic at the edge.

5.1.2 Pre-Creation

One approach that alleviates cold starts while consuming only negligible amounts of resources, is proposed by Mohan et al. [Moh+19]. Their approach avoids the specialization of the container for either one particular function or the language runtime. Rather, it can be used for any container. First, they analyze the core issue for the high cold start times of containers and their rise under increasing concurrency. The startup of a Docker container requires setting up its network namespace in the Linux kernel, which is responsible for more than 90% of the startup time. The kernel uses a single global lock for this task, which is the reason for the decline in performance under concurrency. Given these findings, they propose an approach that pre-creates these namespaces using so-called pause containers. Their initialization is effectively paused after the network namespace has been created. Subsequently, they can be attached to a Docker container, such that both share the namespace and the Docker container itself avoids the namespace creation. On top of this, they introduce a pool manager, which holds a number of pause containers from where they can be attached as needed, but also returned to after the attached container terminates. Due to only requiring memory on the order of kilobytes, pre-creating a large number of pause containers is reasonable and can thus accommodate bursty workloads, too. The evaluation shows a reduction in cold start time of up to 80%. Given these numbers, this approach is a viable alternative to a more radical approach like ours, at least for the cloud and perhaps also on edge devices, although the latter has not been evaluated.

5.1.3 Prediction

Shahrad et al. characterized the serverless workload at Microsoft Azure and use their findings to propose an improved keep-alive policy [Sha+20]. They first point out that most systems such as OpenWhisk, but also the large cloud providers, use a fixed policy where every function's execution environment is kept alive for the same amount of time. Given that they find an 8 order of magnitude difference in the frequency with which functions are called, this can be a very wasteful policy for many of the rarely invoked functions.

They characterize policies by a set of rules. One is the pre-warming window, which is the time the policy waits after an execution before pre-warming the container for the next anticipated invocation. The other is the keep-alive window, which is the time it keeps containers warm after either pre-warming or an actual execution. Their proposed policy works by learning a function's invocation frequency and adjusting the window parameters. It uses a histogram to track the idle times of functions, from which the 5th percentile is used as the pre-warming window, and the 99th percentile used as the keep-alive time. Using real invocation traces as input, they simulate workloads against an implementation of their policy in OpenWhisk. The standard fixed keep-alive policy in OpenWhisk generates 2.5× more cold starts than the histogram-based policy while using more memory. Overall, the histogram-based policy reduces the number of cold starts while minimizing the used memory.

This approach is independent of the underlying container runtime. Although Wasm executors can reduce the cold start times significantly, for efficient successive invocations some amount of keep-alive is needed. Since the keep-alive policy in OpenWhisk is a fixed one, this approach would evidently be a better choice to reduce cold starts and resource waste. While pre-warming is less important when cold starts are cheap, having a keep-alive time based on the function's invocation pattern still reduces memory consumption compared to a fixed policy. Future work might examine this policy for Wasm executors and find a suitable percentile to choose the pre-warming window from, given that cold starts are cheaper.

5.2 · WebAssembly in Serverless

5.2.1 Execution in node.js

We built our work on previous research that built a WebAssembly-based serverless runtime. Hall and Ramachandran use the V8[1] engine's WebAssembly support within `node.js` as their basis. Since V8 implements the WebAssembly standard like any other runtime, it has the same guarantees in terms of security. In particular, V8 *contexts* provide isolation between different executing WebAssembly modules. Thus, for each request their implementation creates a new context where the WebAssembly module is loaded, executed via the V8 API, the result returned and the context destroyed.

[1] `https://v8.dev/`

For single clients sending multiple requests, they find their WebAssembly platform to have a small initial cold start penalty due to context creation, but performs more predictably than Docker afterwards. Their test against native binaries in Docker containers shows them to suffer from a high cold start latency but are faster over time. The examples they use are image recognition applications and thus CPU-bound. In that regard, these are similar results as our CPU-bound workload, where Docker's slow initial startup was amortized over many invocations. In the best case, their numbers indicate the WebAssembly module to reach 56% of the throughput of Docker, while our precompiled WebAssembly reached 88% of throughput with `wasmer`. Recall that `wasmtime`, our JIT-configured runtime reached 57% of throughput in our test. This confirms that WebAssembly execution in `node.js` is not the best option for pure speed due to its JIT compilation strategy.

They note that the model of single clients making multiple requests is not fully representative of a real workload. Rather, multiple users would make requests, each of which needs to be isolated and consequently result in a cold start. Their findings for this access pattern are in line with our mixed and realistic workloads, namely that WebAssembly-based executors exhibit more consistent performance. Whether or not we have improved upon the cold start times of their approach is unclear, since no specific cold start measurements are given, and only the latency as a whole is examined. Since they use `node.js`, their solution would most likely be able to achieve a much higher throughput for an I/O-bound workload, since the runtime uses an event loop to provide efficient non-blocking I/O. As previously mentioned, once WebAssembly runtimes and the corresponding Rust libraries have added support for asynchronous I/O, they should be at least as fast.

In summary, our work builds on top of this research by implementing support with three standalone WebAssembly runtimes, some with higher throughput. In Chapter 3, `wasmtime`'s JIT proved to be much slower than precompilation, and we can assume `node.js` to behave similarly. Thus, we can assume that our ahead-of-time approach improved the cold start times in comparison to their on-demand approach, in particular on resource-constrained edge devices. Additionally, our work adds capabilities to the system, which shows how functions could share system resources across invocations or make requests over the network with fine-grained acccess control.

5.2.2 Cloudflare Workers

Cloudflare is a cloud service provider, whose *Workers* offering enables end-users to run code on its Edge Network. The Workers also use Google's V8 JavaScript and WebAssembly engine to execute that code. Instead of running each serverless function in a separate Docker container or even `node.js` instance, the Workers use V8 *isolates* for lightweight sandboxing. Isolates start within the already running V8 engine, so the start is almost instantaneous. Workers allow execution of JavaScript directly in an isolate as well as Rust, C and other languages via Wasm support [Clo21]. Similar to the preceding approach, a lighter-weight sandboxing is introduced to cut down on the cold start latency, only this time isolates instead of contexts. This execution model eliminates the costly startup of a `node.js` process, but it would still need to parse

and compile the JavaScript or WebAssembly before execution, which is what our approach eliminates.

5.2.3 Fastly's Lucet

Cloud provider fastly offers its experimental fastlylabs[2], where its previously mentioned `lucet` WebAssembly runtime can be used inside *Terrarium*, to run Wasm code at the edge [fas19]. The `lucet` compiler first translates WebAssembly to native code, after which it can be executed in the complementary runtime. Due to this design – which spawned the idea for our work – `lucet` can execute Wasm efficiently on every request, with module instantiation times of 50 microseconds and only kilobytes of memory, according to them.

Even though we did not leverage `lucet` due to security concerns as well as platform and ISA support, we did use the idea of precompiling to native code ahead-of-time. This payed off, given that we measured module instantiation times of 340 microseconds to set up a `wasmtime` instance from a module. Without this, we also would not have achieved cold starts of around 10 milliseconds in that runtime. Fastly has joined the Bytecode Alliance[3], under whose umbrella `wasmtime` is developed, and `lucet` is being integrated into `wasmtime` to provide ahead-of-time compilation [Hic20]. This might bring its AoT performance into the realm of `wasmer`'s.

5.2.4 Faasm

Shillaker and Pietzuch introduce `faasm`, a serverless runtime using WebAssembly [SP20]. One or more functions is executed in a `faaslet`, which uses WebAssembly's software fault-isolation to restrict memory access to its own address space. Every `faaslet` is run in a new thread which allows limiting CPU cycles with `cgroups` and network with Linux namespaces. We did not provide a CPU limitation mechanism since we believe our threading model to be only an intermediate solution until WASI functions can be executed asynchronously, but it could use the same `cgroup` mechanism in principle. When executing asynchronously, however, `cgroups` cannot be used to limit an individual function's CPU time, which is therefore an open problem. Memory on the other hand can be limited through appropriate APIs of the WebAssembly runtimes.

Running multiple functions together allows for memory sharing and statefulness – an explicit goal of the runtime – also across hosts via a state management API. Even though in traditional serverless platforms functions can be chained together, this process requires serialization which can be omitted when sharing memory directly. This is implemented on top of the Wasm memory model with *shared regions*, where the linear memory of a WebAssembly instance is extended by additional pages which are mapped onto that region. Statefulness was not a goal of our work, given that this limitation of serverless platforms has not been a primary one for

[2] `https://www.fastlylabs.com/`
[3] `https://bytecodealliance.org/`

developers, according to our literature review. For applications that make heavy use of state sharing, this may prove beneficial, although the increase in implementation complexity needs to be considered.

Fast cold starts are achieved by initializing a `faaslet` ahead-of-time as a snapshot of a function's stack, heap function table, stack pointer and data – in WebAssembly terminology. This is generated at function-creation time and stored in an object store. When cold starting a function, the snapshot is retrieved and restored to produce a `faaslet` instance. The restoration procedure guarantees no data from previous invocations is shared with future invocations. This is similar to how we use WebAssembly `Module`s as our templates and produce `Instance`s on every invocation for isolation. However, due to their pre-creation and restore procedure, the cold start times are at around 1 millisecond. This technique could be adopted in our work as well to provide even faster cold starts.

5.3 · WebAssembly in Smart Contracts

Smart contracts are programs deployed to distributed ledgers from where they can be run by users on-demand. As such, smart contracts are akin to serverless functions, since developers do not need to manage the machines they run on. Users of the contract often pay a per transaction fee similar to the pay-as-you-go model of serverless and the node running the untrusted contract needs to isolate its execution to enable secure multi-tenancy. The transaction fee is usually based on the amount of CPU the smart contract consumes, which incentivizes developers to produce lightweight code. That in turn means it is very likely that the average smart contract executes much faster than the average serverless function, making OS-level virtualization even less suitable for isolation, since it would be responsible for an even larger proportion of the latency. The node operator running the smart contract has an interest in achieving high throughput to make the most of the available resources, so performance cannot be neglected. Additionally, smart contract nodes likely need to handle bursty request patterns and thus need to be elastic, too. These requirements are similar to what a serverless container runtime needs to offer.

Fittingly, WebAssembly has been recognized as a suitable target format for smart contracts. The Ethereum-flavored WebAssembly restricts the instruction set to a suitable subset for smart contracts and one of the cited reasons for the switch from the previous custom virtual machine is performance [Dev21a]. The conforming `hera`[4] implementation can use WebAssembly runtimes that use interpretation, JIT or AoT compilation strategies. One of the options includes `WAVM`[5], a runtime written in C++ with a focus on performance and claimed near-native execution speed. Other ledgers like near[6] use `wasmer` with caching, while IOTA[7] uses `wasmtime` and EOSIO[8]

[4] `https://github.com/ewasm/hera`
[5] `https://github.com/WAVM/WAVM`
[6] `https://github.com/near/nearcore/tree/master/runtime/near-vm-runner/`
[7] `https://github.com/iotaledger/wasp`
[8] `https://github.com/EOSIO/eos-vm`

implemented their own WebAssembly VM, citing a lack of performance in existing runtimes. Many more ledgers have adopted what seems to have become the de-facto standard target format for smart contracts.

Overall, these approaches validate the fundamental choice of WebAssembly as a performant option in scenarios where lightweight isolation is required. The smart contract space will be interesting to watch and has already given rise to some WebAssembly synergies. For example, both near and fastly are listed as platinum sponsors on AssemblyScript's GitHub page[9], which might make AssemblyScript a more viable and mature alternative to write serverless functions in WebAssembly.

[9] https://github.com/AssemblyScript/assemblyscript/

CHAPTER 6

Conclusion

In this work, we implemented a WebAssembly container runtime for Apache OpenWhisk. The main goal was to examine the performance differences between the WebAssembly and Docker container runtimes in terms of cold start latency and execution performance. In this chapter, we conclude the work by answering the research questions posed in Chapter 1.

6.1 · Research Questions

RQ1: By how much can WebAssembly runtimes alleviate the cold start latency?

Our WebAssembly executors can reduce the cold start latency, on average, by more than 99.5% on a Raspberry Pi compared to OpenWhisk using Docker. This is in part due to Docker reaching the memory limits of the Pi and swapping. However, even on our server machine, where enough memory is available, the cold start time can be reduced by around 94%, on average. The executors behave more consistently in the cold start than the Docker container runtime. Among the Wasm runtimes, `wasmer` takes more time for the cold start than the others, being between 2 - 3× slower. The `wamr` and `wasmtime` runtime behave very similar, with `wamr` taking the lead due to its computationally cheap initialization procedure.

When only accounting for the overhead introduced by the executors themselves, `wasmtime` introduces less than 10 ms, and `wasmer` less than 20 ms on the server machine. In comparison with Docker startup times, which are always far north of 100 ms [Wan+18; Man+18], this is an order of magnitude lower. On the Raspberry Pi, the WebAssembly runtimes bring the cold start overhead into the range of 112 ms to 274 ms, which is on the same order of magnitude as Docker on a server machine. Furthermore, we have seen in our tests – what is also confirmed by other research [Moh+19] – that Docker startup times rise under increasing concurrency, while the Wasm executors behave consistently across the number of concurrent requests. Since

bursty workloads are especially prevalent in serverless, this is an equally important property. With these results, we expect Wasm executors to perform similarly outside of the OpenWhisk context.

RQ2: How do the performance characteristics of the WebAssembly and Docker container runtimes differ?

In the mixed and realistic workload we can best see the holistic picture, because it shows the execution performance including cold starts. Here, the WebAssembly executors are always faster than Docker. In the mixed workload, the WebAssembly executors achieve between 1.6 and 3× the throughput on the server, and 2.4 to 4.2× the throughput on the Pi. In the realistic workload, we find Docker's overall performance to be strongly linked to the cold start time. The WebAssembly executors, however, are not noticeably susceptible to cold starts, showing much more consistent behavior. In the realistic workload, `wasmer` takes the lead and processes around 95% of requests strictly faster than all other container runtimes. For sufficiently long requests, either CPU- or I/O-bound, the cold start time of `wasmer` does not significantly affect its performance, even though it is twice that of `wasmtime` or `wamr`.

For I/O-bound workloads, the used Docker image shows a higher overhead even when the execution time itself is fixed. It confirms that Docker has a higher inherent management cost due to implementing isolation with OS-level virtualization, rather than in user space. The Wasm executors scale linearly for I/O-bound workloads, up to the limits imposed by OpenWhisk and later the underlying operating system, due to their threading model. Once Wasm runtimes allow handling I/O asynchronously – which is starting to appear – we expect this performance to be even better while consuming fewer resources.

On the other side, Docker handles pure CPU-bound workloads very well. It proves once more that Docker containers can be very performant – once they are running. The issue is getting to that point. In terms of pure CPU utilization, `wasmer` managed 88% of the throughput compared to the native binary in a Docker container, with the other runtimes falling behind more quickly. Our AssemblyScript experiment showed, however, that the prevalent `node.js` runtime would perform much worse under such workloads, indicating that they are either not very common in today's FaaS or that this level of performance is good enough. Since WebAssembly shows to be several factors faster, this is an overall improvement.

RQ3: What are the resource costs for keeping containers warm in both container runtimes?

The cost for keeping containers warm is relatively consistent for Docker independent of module size, for the image we tested. The in-memory size of WebAssembly containers is primarily determined by the module they hold. The `wasmer` runtime shows the strongest such effect closely followed by `wasmtime`. Due to not keeping modules in memory between requests, `wamr` shows very little memory usage and exemplifies the use case when memory would

be very constrained and a priority over speed. WebAssembly modules can be optimized to contain only the necessary code with LTO compilation in Rust or with AssemblyScript by default, in which case the module is in the lower kilobytes range. Overall, for these low-sized WebAssembly modules, the Wasm executors can be expected to consume less memory than the Docker image we tested and thus hold multiple containers in the same amount of memory.

RQ4: Which WebAssembly runtime is the most suitable for use on edge devices?

Both `wasmtime` and `wasmer` have an API that allows us to efficiently implement our main requirements for modules: An initialize once, run often lifecycle. Unfortunately, `wamr`'s API is less suitable to that task and it may be part of why it lacks in performance. It has the lowest throughput of Wasm executors in I/O- and CPU-bound workloads, as well as the mixed and realistic workload. The memory usage, on the other hand, is lowest in terms of idle footprint and container size, from a pure numbers perspective. Its API also showed limitations that the others did not have, that is, specifying function imports on a per-module basis is not possible. A certain upside is its support for more instruction set architectures than the others, which may become relevant for heterogeneous edge devices. From a performance perspective however and unless memory is very constrained, it is the least suitable of our selected runtimes.

It then comes down to the other two runtimes. The main advantage of `wasmer` is its ahead-of-time compilation that produces high quality native code, evident by its performance in CPU-bound workloads. This seems to be coupled with taking twice the time for cold starts compared to `wasmtime`, which handles this task particularly well. On the Raspberry Pi and in a mixed workload, the margin to `wasmer` is slimmer than on the server. In our realistic workload, `wasmer` showed that the slower cold start did not significantly affect its throughput, being even strictly faster than `wasmtime`. Both runtimes support primarily `x86_64` and `aarch64`, which covers a wide range of devices, but particularly `arm32` support is missing as yet. How relevant that is depends on the planned use of the serverless platforms and whether its worker nodes would run on devices with that or other ISAs. Both of these runtimes perform well under various circumstances, but based on our general-purpose mixed and realistic workloads, `wasmer` takes the lead, even with a slightly higher memory usage.

RQ5: How suitable is WebAssembly for serverless execution?

We find WebAssembly to be a very suitable target format for serverless functions. This comes down to a number of points.

Language Support The high-performance languages C, C++ and Rust already have excellent support for Wasm as a compilation target. Based on a curated list[1], many more languages such as AssemblyScript, C#, Lua, Go and Zig are considered production-ready. Work for Python, Java and Kotlin support is also underway.

[1] `https://github.com/appcypher/awesome-wasm-langs`

Ultimately, that means developers do not have to learn a new language to write serverless functions, which was previously enabled by packaging languages' runtimes in Docker images. With WebAssembly, serverless operators can polish their support for WebAssembly instead of maintaining multiple Docker images with different language runtimes in different versions and for different ISAs. This reduction in complexity also reduces the attack surface.

Performance & Cold Start The performance of WebAssembly primarily depends on the compilation strategy used by the runtime. We have seen up to 88% of the CPU-bound performance of Docker and better performance in I/O-bound workloads. The latter will improve even more once Rust runtimes are able to execute the WASI I/O functions asynchronously. With AssemblyScript, we have seen that the same program code will run faster when compiled to WebAssembly than in a `node.js` runtime, making this target even more attractive to TypeScript developers. The lightweight isolation of WebAssembly means a fast cold start without sacrificing security. The APIs of our Rust runtimes proved powerful enough to implement a fast startup procedure with precompilation. We can essentially use WebAssembly as a universal intermediate representation of a program, that can be compiled to a more performant version, which is itself a useful merit.

Scale-to-zero While WebAssembly does not get us all the way to the ideal of scale-to-zero, it gets us much closer than Docker. It would allow for rarely invoked functions to never be kept-alive and always cold start with negligible performance costs, while keeping the frequently accessed ones in memory with little memory cost. This would require a suitable keep-alive policy like the one introducded by Shahrad et al.

System Access The WebAssembly system interface allows for secure access to resources on the host. It enables more fine-grained access control than Docker without a significant overhead. We have seen how a Wasm executor can give a module access on a per-function basis and per-file for fileystem access.

The key takeaway is that WebAssembly proves to be a suitable target format due to a performant isolation mechanism with fast startup times.

6.2 · Open Challenges & Future Work

There are a number of open challenges to create a fully production-ready WebAssembly container runtime.

Even though WebAssembly has left the experimental phase and is standardized, so far it is only a *minimum viable product*. Some important features are proposed but not yet implemented.

These include interface types, i.e. passing complex types from host to module or module to module, multi-threading and atomics or a garbage collector (GC), which will make it easier for GC'ed languages to be compiled to Wasm.

We also previously mentioned running WASI functions asynchronously, which would allow for non-blocking I/O and thus much better performance for I/O-bound workloads. This is planned to become available as `wasi-cap-std-async` and will then be usable by WebAssembly runtimes that implement support for it, like `wasmtime`. For other runtimes, which do not build on this library, it is unclear if and when support will land.

Sharing resources fairly among tenants is important for serverless frameworks. While memory can be limited by the host of a Wasm module, limiting CPU shares is not possible. Due to popularity in smart contracts, WebAssembly runtimes have added gas metering, i.e. limiting the number of instructions that a module can execute before it is interrupted. It achieves a limitation of the CPU but the technique is unsuitable for serverless since the typical model is to pay by execution time rather than instructions and it adds book-keeping overheads.

Our executor is stateless without exception, that is, each function is executed in a new `Instance`. Some functions, however, may wish to cache data from an external service in memory to improve latency. A fully stateless system does not allow for such optimizations, while caching `Instance`s rather than `Module`s and invoking them repeatedly would.

At the end of Chapter 3, we concluded that neither of the threading models we examined was strictly superior. If a priori knowledge exists about the type of workload, then we may wish to run multiple `Invoker`s with different threading models. If the workload type is annotated on the function, the load balancer could dispatch to the most appropriate `Invoker` and improve performance.

While we have seen great leaps in performance on a Raspberry Pi, the cold start times of just the executor are still beyond 100 ms. This is on the same order of magnitude that current FaaS offerings perform on, although on more powerful hardware. Regardless, it may not be fast enough for latency-critical applications at the edge. Therefore, other techniques like those explored in the related work may need to be implemented in conjunction with our approach to bring the startup latency further down.

List of Figures

List of Tables

List of Listings

References

[Asl+21] Mohammad S. Aslanpour et al. “Serverless Edge Computing: Vision and Challenges”. In: *2021 Australasian Computer Science Week Multiconference*. ACSW ’21. Dunedin, New Zealand: Association for Computing Machinery, 2021. ISBN: 9781450389563. DOI: 10.1145/3437378.3444367. URL: https://doi.org/10.1145/3437378.3444367.

[BF19] Luciano Baresi and Danilo Filgueira Mendonça. “Towards a Serverless Platform for Edge Computing”. In: *2019 IEEE International Conference on Fog Computing (ICFC)*. 2019, pp. 1–10. DOI: 10.1109/ICFC.2019.00008.

[Cas+19] Paul C. Castro et al. “The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry”. In: *CoRR* abs/1906.02888 (2019). arXiv: 1906.02888. URL: http://arxiv.org/abs/1906.02888.

[Cow+98] Crispin Cowan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. In: *Proc. 7th USENIX Security Symposium*. 1998.

[Eis+21a] S. Eismann et al. “Serverless Applications: Why, When, and How?” In: *IEEE Software* 38.1 (2021), pp. 32–39. DOI: 10.1109/MS.2020.3023302.

[Eis+21b] Simon Eismann et al. *A Review of Serverless Use Cases and their Characteristics*. 2021. arXiv: 2008.11110 [cs.SE].

[Fox+17] Geoffrey C. Fox et al. “Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research”. In: *CoRR* abs/1708.08028 (2017). arXiv: 1708.08028. URL: http://arxiv.org/abs/1708.08028.

[Haa+17] Andreas 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*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200. ISBN: 9781450349888. DOI: 10.1145/3062341.3062363. URL: https://doi.org/10.1145/3062341.3062363.

[HR19] Adam Hall and Umakishore Ramachandran. “An Execution Model for Serverless Functions at the Edge”. In: *Proceedings of the International Conference on Internet of Things Design and Implementation*. IoTDI ’19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 225–236. ISBN: 9781450362832. DOI: 10.1145/3302505.3310084. URL: https://doi.org/10.1145/3302505.3310084.

[HLL12] Bai-Ruei Huang, Chang Hong Lin, and Chia-Han Lee. “Mobile augmented reality based on cloud computing”. In: *Anti-counterfeiting, Security, and Identification*. 2012, pp. 1–5. DOI: 10.1109/ICASID.2012.6325354.

[Jan+19] Abhinav Jangda et al. “Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, 2019-07, pp. 107–120. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jangda.

[LKP20] Daniel Lehmann, Johannes Kinder, and Michael Pradel. “Everything Old is New Again: Binary Security of WebAssembly”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020-08, pp. 217–234. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[Lei+19] Philipp Leitner et al. “A mixed-method empirical study of Function-as-a-Service software development in industrial practice”. In: *Journal of Systems and Software* 149 (2019-03-01), pp. 340–359. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.12.013. URL: https://www.sciencedirect.com/science/article/pii/S0164121218302735.

[LG19] Ping-Min Lin and Alex Glikson. “Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach”. In: *CoRR* abs/1903.12221 (2019). arXiv: 1903.12221. URL: http://arxiv.org/abs/1903.12221.

[Man+18] J. Manner et al. “Cold Start Influencing Factors in Function as a Service”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[MB17] G. McGrath and P. R. Brenner. “Serverless Computing: Design, Implementation, and Performance”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36.

[Men20] P. Mendki. “Evaluating Webassembly Enabled Serverless Approach for Edge Computing”. In: *2020 IEEE Cloud Summit*. 2020, pp. 161–166. DOI: 10.1109/IEEECloudSummit48914.2020.00031.

[Moh+19] Anup Mohan et al. “Agile Cold Starts for Scalable Serverless”. In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, 2019-07. URL: https://www.usenix.org/conference/hotcloud19/presentation/mohan.

[Nas+17] S. Nastic et al. “A Serverless Real-Time Data Analytics Platform for Edge Computing”. In: *IEEE Internet Computing* 21.4 (2017), pp. 64–71. DOI: 10.1109/MIC.2017.2911430.

[ND18] Stefan Nastic and Schahram Dustdar. “Towards Deviceless Edge Computing: Challenges, Design Aspects, and Models for Serverless Paradigm at the Edge”. In: *The Essence of Software Engineering*. Ed. by Volker Gruhn and Rüdiger Striemer. Springer, 2018, pp. 121–136. ISBN: 978-3-319-73897-0. DOI: 10.1007/978-3-319-73897-0_8. URL: https://doi.org/10.1007/978-3-319-73897-0_8.

[Rau+19] Thomas Rausch et al. “Towards a Serverless Platform for Edge AI”. In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. Renton, WA: USENIX Association, 2019-07. URL: https://www.usenix.org/conference/hotedge19/presentation/rausch.

[Rau+20] Thomas Rausch et al. “Synthesizing Plausible Infrastructure Configurations for Evaluating Edge Computing Systems”. In: *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, 2020-06. URL: https://www.usenix.org/conference/hotedge20/presentation/rausch.

[Sha+20] Mohammad Shahrad et al. “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 2020-07, pp. 205–218. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[SD16] W. Shi and S. Dustdar. “The Promise of Edge Computing”. In: *Computer* 49.5 (2016), pp. 78–81. DOI: 10.1109/MC.2016.145.

[SP20] Simon Shillaker and Peter Pietzuch. “Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 2020-07, pp. 419–433. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

[Wan+18] Liang Wang et al. “Peeking Behind the Curtains of Serverless Platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018-07, pp. 133–146. ISBN: ISBN 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.

Online Resources

[All21] Bytecode Alliance. *Cranelift Code Generator*. 2021-04-23. URL: https://github.com/bytecodealliance/wasmtime/tree/main/cranelift (visited on 2021-04-27).

[Aut21] The Kubernetes Authors. *Kubernetes*. 2021-03-04. URL: https://kubernetes.io/ (visited on 2021-03-04).

[Cla19] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. 2019-03-27. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (visited on 2021-02-14).

[Clo21] Cloudflare. *How Workers works*. 2021-02-12. URL: https://developers.cloudflare.com/workers/learning/how-workers-works (visited on 2021-02-18).

[Cui18] Yan Cui. *I'm afraid you're thinking about AWS Lambda cold starts all wrong*. 2018-01-15. URL: https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f (visited on 2021-04-14).

[Dei21] Deislabs. *Krustlet*. 2021-03-04. URL: https://github.com/deislabs/krustlet (visited on 2021-03-04).

[Den19] Frank Denis. *Memory management in WebAssembly: guide for C and Rust programmers*. 2019-02-07. URL: https://www.fastly.com/blog/webassembly-memory-management-guide-for-c-rust-programmers (visited on 2021-04-25).

[Dev19] Apache OpenWhisk Developers. *OpenWhisk system overview*. 2019-08-14. URL: https://github.com/apache/openwhisk/blob/b5b88ece0ae08c87948796b38ab18e55ab4b70a7/docs/about.md (visited on 2021-02-19).

[Dev20] Apache OpenWhisk Developers. *OpenWhisk Deploy Kube: Configuration Choices*. 2020-08-19. URL: https://github.com/apache/openwhisk-deploy-kube/blob/281c0ace1dbede3cbd18b0aecf0775fb3dff4bdb/docs/configurationChoices.md (visited on 2021-03-04).

[Dev21a] Ethereum Developers. *Ethereum WebAssembly (ewasm)*. 2021-05-09. URL: https://ewasm.readthedocs.io/en/mkdocs/ (visited on 2021-05-09).

[Dev21b] Rust Language Developers. *Asynchronous Programming in Rust*. 2021-04-27. URL: https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html (visited on 2021-04-27).

[Far+19] Manzoor Farid et al. *Edge Computing for Serverless Applications*. 2019-07-08. URL: https://www.ibm.com/cloud/blog/edge-computing-for-serverless-applications (visited on 2021-04-25).

[fas19] fastly. *Announcing Lucet: Fastly's native WebAssembly compiler and runtime*. 2019-03-29. URL: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime (visited on 2020-10-25).

[HWZ14] David Herman, Luke Wagner, and Alon Zakai. *asm.js Working Draft*. 2014-08-18. URL: http://asmjs.org/spec/latest/ (visited on 2021-02-14).

[Hic20] Pat Hickey. *How Lucet and Wasmtime make a stronger compiler, together*. 2020-03-03. URL: https://www.fastly.com/blog/how-lucet-wasmtime-make-stronger-compiler-together (visited on 2021-05-07).

[Hyk19] Solomon Hykes. *WASM+WASI Tweet*. 2019-03-27. URL: https://twitter.com/solomonstre/status/1111004913222324225 (visited on 2020-10-25).

[IBM21] IBM. *Originless responses*. 2021-04-27. URL: https://cloud.ibm.com/docs/cis?topic=cis-edge-functions-use-cases#originless-responses (visited on 2021-04-27).

[Reh19] Taavi Rehemägi. *State of Lambda functions in 2019 by Dashbird*. 2019-06-20. URL: https://dashbird.io/blog/state-of-lambda-functions-2019/#ok-pretty-cool-but-what-about-their-runtime (visited on 2021-04-24).

[Shi21] Mikhail Shilkov. *When Does Cold Start Happen on AWS Lambda?* 2021-01-05. URL: https://mikhail.io/serverless/coldstarts/aws/intervals/ (visited on 2021-03-08).

[Tea21] WasmCloud Team. *Wasmcloud Actor Interfaces*. 2021-03-05. URL: https://github.com/wasmCloud/actor-interfaces (visited on 2021-03-05).

[Thö17] Markus Thömmes. *Squeezing the milliseconds: How to make serverless platforms blazing fast!* 2017-04-20. URL: https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0 (visited on 2021-05-04).

[TCW21] Aaron Turon, Wonwoo Choi, and Yoshua Wuyts. *Tide*. 2021-02-26. URL: https://crates.io/crates/tide (visited on 2021-02-26).

[W3C20] W3C. *WebAssembly Website*. 2020-10-25. URL: https://webassembly.org/ (visited on 2020-10-25).

[Win20] Andy Wingo. 2020-10-15. URL: http://wingolog.org/archives/2020/10/15/on-binary-security-of-webassembly (visited on 2021-04-26).

[ZN13] Alon Zakai and Robert Nyman. *Gap between asm.js and native performance gets even narrower with float32 optimizations*. 2013-12-20. URL: https://hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-optimizations/ (visited on 2021-02-14).