Master of Science in Computer Science
June 2020

Considering WebAssembly Containers for Edge Computing on Hardware-Constrained IoT Devices

Jonah Napieralla

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

Contact Information:
Author(s):
Jonah Napieralla
E-mail: jonk15@student.bth.se

University advisor:
Dr. Emiliano Casalicchio
Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet	:	www.bth.se
Phone	:	+46 455 38 50 00
Fax	:	+46 455 38 50 57

Abstract

Background. Container technologies allow us to make our code portable enough to run properly and independently irrespective of the hardware architecture or operative system structure on which it is running. Thus developers are able to test their code running within containers on their personal development machines and if it runs there then they can assume it will also run wherever else it is deployed. The current status-quo in container technology is Docker, it is highly performant but it is still a large and complicated system and its shortcomings are more obvious on hardware-constrained IoT devices than they are when running on dedicated servers in a data-center. WebAssembly containers, which despite being new and not well researched, appear as a promising solution for running small lightweight containers. These provide most of the same benefits as Docker containers do while employing a simpler runtime that could be better suited for hardware-constrained IoT devices running on the edge of the internet.
Objectives. The aim of this project is to study the applicability of WebAssembly for use as containers on IoT devices, investigating how WebAssembly and its WASI interface can make moving from a centralized point of aggregation and computation to the edge easier, and comparing it to an alternative solution; the Docker container.
Methods. For analyzing the suitability of WebAssembly containers to be deployed at the edge; a systematic literature review of the capabilities of WebAssembly containers, compared to Docker containers, was performed. Practical experiments were also conducted to analyze various performance metrics through a quantitative comparison between WebAssembly containers, Docker, and native executables running on an ARM powered IoT device. Finally, we had planned to compare container orchestration frameworks available for WebAssembly and Docker.
Results. Through our systematic literature review we found no previous studies comparing WebAssembly to Docker, probably because of how recently the WebAssembly and WASI specifications came to existence. Our experiments showed that WebAssembly containers still suffered a severe performance overhead, only beating Docker containers through lower system memory consumption and startup-times. Our planned comparison between container orchestration frameworks could not be carried out due to the only WebAssembly framework that existed not support IoT devices, because they run on the ARM CPU-architecture.
Conclusions. We conclude that Docker containers are still the best choice for most cases, except for when sporadic execution of simple programs is necessary. This means that WebAssembly containers might be well suited for serverless computing at the edge where short functions, rather than an entire software-platform, are employed.

Keywords: WebAssembly, Container technologies, Virtualization, Distributed architectures, Cloud computing

Contents

Chapter 1

Introduction

In this chapter the study is presented through the exploration of the relevancy of the subject and the definition of the various fields and technology that comprise this study. Then the aim of this study and the objectives that were followed to accomplish said aim are presented, as well as three research question based on the aforementioned aim. Then finally the delimitations of the study, considering what parts are relevant and which are outside of the scope are considered, and an overview of the various chapters and parts of the study are presented.

1.1 Motivation

Following the advances in computer hardware, development of software has become less constrained by physical limitations, like processing data, and a greater focus instead lies in the logical difficulties and cost associated with deploying software broadly and making sure all parts of the stack, from hardware to platform to program, work well in tandem. Accompanying this change in the industry is the rise of cloud computing that has made large parts of the hardware and software stacks an abstraction that largely becomes irrelevant to developers. Using systems like Docker containers we still have to emulate an underlying operative system (OS) however, which gives developers an extra unwanted thing to consider and makes the virtualization layer more complicated. WebAssembly (Wasm) containers might provide a solution to this issues. This new container technology does not emulate an OS, it is nothing but program code written in a bytecode-format that is abstract enough to be interpreted and run on any system, yet specific enough to be convertible to native bytecode. Instead of emulating an actual OS through Docker, a Wasm container is executed through a simple runtime that promises to behave in the same way everywhere, irrespective of what other software libraries exist outside the container, or what OS the runtime is employed on. This might be especially well suited for IoT devices which are heterogeneous in the hardware they employ and computationally so weak that they might benefit from running a simple Wasm runtime that only interprets bytecode, instead of a Docker container that virtualizes a large, and potentially foreign, OS.

1.2 Aim and Objectives

The aim of this project is to study the applicability of Wasm for use as containers in edge computing, investigating how Wasm and WASI can make moving from a centralized point of aggregation and computation to the edge easier, and comparing it to alternative solutions like Docker containers. This will be accomplished by pursuing the following objectives:

- Conduct a systematic literature review of previous studies comparing Wasm to Docker, both in quantitative aspects, such as performance metrics, and qualitative aspects, such as security capabilities, and synthesize the conclusions drawn from these studies.

- Implement the same software that stresses the system in raw computational, filesystem I/O, network throughput, and measures program startup time, in a Wasm container, Docker container, and as a native (non-containerized) application and see how they compare.

- Investigate the possibilities of employing Wasm containers in a container orchestration schema for simple deployment and scaling of Wasm-based software stacks, and compare this to the orchestrations frameworks available on the Docker platform.

1.3 Research Question and Hypothesis

To best compare Wasm to Docker it is a good idea to not only do our own experiments, but also consider the previous work done by others. Therefore the first research question should be dedicated to reviewing earlier studies. To complement these earlier studies we should also perform our own experiments to produce something of new value, and for this a comparison of performance aspects between Wasm and Docker will be done. One of the most powerful applications of containers in recent years have been in their role in dynamic container orchestration schemes, for automatic deployment and scaling of entire software-stacks. Therefore the last research question is dedicated to looking at Wasm container orchestration frameworks for IoT devices. As such, the research questions become:

- RQ 1: What conclusions have previous studies been able to draw from comparisons between WebAssembly and Docker? Null hypothesis: previous studies have not been able to compare WebAssembly and Docker or have not found any differences.

- RQ 2: How do Wasm containers compare to running code natively or in Docker containers in CPU performance, memory consumption, network utilization, filesystem I/O speeds, and startup time performance metrics? Null hypothesis: there is no difference in the performance metrics between Wasm containers and native execution / Docker containers.

- RQ 3: How does the Wasm platform compare to that of the Docker platform in terms of support for container orchestration on IoT devices? Null hypothesis: the Wasm platform supports container orchestration to the same extent as the Docker platform does.

1.4 Delimitations

The delimitations of this project are to keep the study of other container technologies, like that of Docker, to a minimum. So that more time and focus can be dedicated to studying Wasm. This is because the Docker platform has already been extensively reviewed in academic literature while a large gap exists in the study of Wasm.

1.5 Thesis Overview

Following this introductory chapter come chapters on Background, Related Work, Methodology, Results and Analysis, Discussion, Conclusions, and Future Work. In the chapter on Background the subject of container technologies is explained. Virtualization is explained as well as the portability and overhead factors that are associated with virtualization. Container sandboxing and the security implications are covered. Container orchestration and how it allows a complex software architecture to be seamless deployed and managed is explained. After virtualization technologies are covered in general a closer look at Wasm specifically is taken, looking at how it differs from containers like Docker, its specific advantages, and considering how the newly introduced WebAssembly System Interface (WASI) brings Wasm containers a step closer to feature-parity with Docker containers. In the chapter on Related Work we look at previous studies involving Wasm specifically or container technologies in general as they relate to Wasm. In the chapter on Methodology we look at how the research questions of this study are to be answered through a systematic literature review as well as practical experiments in order to compare Wasm to its alternatives on both a high as well as a low level. In the chapter on Results we present the information obtained from the systematic literature review comparing Wasm to other container technologies like Docker, as well as looking at the quantitative data obtained from the practical experiments benchmarking Wasm containers. In the chapter for Discussion we consider the advantages of Wasm containers as they relate to edge computing on IoT devices. In Conclusions we answer the previously stated research questions and give some final remarks as to the viability of Wasm containers. Finally, in the section on Future Work we suggest possible research topics, associated to the topic at hand, that were thought of during the process of producing this thesis but laid outside its scope and our delimitations.

Chapter 2

Background

2.1 Edge Computing

The cloud computing architecture is commonly used to move computation from local servers and personal computers into remote machines, marking the most recent trend in server architectures, as discussed in [18]. This makes things easier for developers and systems administrators as the hardware becomes an abstraction of lesser importance, as does staying within performance boundaries due the possibility of dynamic up-scaling. Companies in the Internet of Things (IoT) business regularly use their edge devices for nothing but data collection where IoT devices serve no other than purpose than to manage the sensors they use. As small mobile devices have become ever more powerful in recent years this has become a wasted opportunity to do more at the edge (on the IoT devices) and leaving only the computations requiring aggregated values from multiple remote devices to the centralized cloud servers. That is what edge computing is about; bringing us the advantages of lower cloud computing costs as the remote hardware comes to better use, discussed in [10], lower network bandwidth utilization as non-relevant data can be discarded before it is sent to a centralized location for analysis, and faster response times for applications making use of real-time values, discussed in [16], as decisions can be taken at the edge where the data is collected. Difficulties in edge computing are often related to hardware constraints, as memory and processor limitations require software to run within these constraints and with as little overhead as possible to make the most of the available hardware, and the heterogeneity of devices, as different operating systems (OS) and CPU-architectures require software to compiled and tested against multiple platforms, as discussed in [19].

2.2 Serverless Computing

As cloud computing has started to make hardware an abstraction, to make deploying software easier for developers, we eventually reached what is called serverless computing, or function-as-a-service (FaaS) computing. This is the concept of not deploying programmatically managed software-stacks, but instead deploying several short functions with well delineated jobs. These functions are usually not called from other code, but rather through events. An event might be a certain request arriving to an interface that the cloud-provider is managing for the customer, and that event would be passed to a function written by the developer to handle that certain event. These are small programs that are started for every event they are bound to, and they usually run independently of any operating system or other execution environment. It is software-deployment distilled into its most simple form; several short functions that do not know each other and are managed by the cloud provider.

2.3 Container Technologies

Packaging software into containers makes deploying to heterogeneous systems easier as a virtual OS and file-system is ran on top of the native system. This allows developers to build, compile, and test only once for the virtualized platform, as the virtualization layer translates actions happening inside the container for the underlying system. At the same time these containers can be constrained more easily than programs running on the native platform, as every container serves as an isolated sandbox for the program inside it, saving the underlying system from being affected by misbehaving or exploited software inside containers. This allows many containers to be run in parallel on a single system without affecting each other, often under the oversight of a container orchestration platform monitoring their status. Such container orchestration frameworks allow for easy deployment and management of large container-based software-stacks. The downside of packaging software into containers is that it comes at a performance cost, as virtualization incurs reduced performance due to additional software overhead, as studied in [12]. This overhead has not hindered the proliferation of containers on high-performing desktops and servers, but it might on IoT devices, where streamlining all computation is a necessary practice. The Docker platform can be considered the current status quo in container technology. It is well researched and backed by a large community supporting its use in the virtualization of multiple platforms and services, but it suffers from the aforementioned performance overhead.

2.4 WebAssembly

WebAssembly is a new technology that allows software to be shipped in sandboxed, virtualized containers able to be deployed on a variety of platforms, diverse in OS and CPU-architecture, as introduced in [2], but it is new and suffers from an absence of studies investigating its potential. The Wasm specification became an official web-standard, anointed by the World Wide Web Consortium (W3C), in December of 2019 [23]. Before that is has been available in the Mozilla Firefox, Google Chrome, and Apple Safari web-browsers since 2017, but remaining largely unheard of. It was introduced as a way of running for-Wasm-compiled code-binaries in the browser. Wasm is a binary instruction format that can be executed in runtimes found in these web-browsers, where a virtualization-layer specific to each browser, but adhering to the Wasm standard, is employed, more info in [7]. Wasm can be written in a syntax similar to other assembly languages, but it is most commonly compiled to directly from other programming languages directly into Wasm binary instructions. Compilers from many common programming languages to Wasm exist.

2.4.1 WebAssembly System Interface

The newly introduced WebAssembly System Interface (WASI) takes the aforementioned virtualization specification a step further by providing a standardized approach to calling kernel-level system-calls from within Wasm, which are then translated to ones appropriate for the underlying system during runtime. The WASI takes Wasm from only being usable for computation to becoming a general purpose software runtime suitable for use as a lightweight container. The WASI specification was first introduced in 2019, but at the time it lacked any runtimes allowing it to be used in practice. Wasm runtimes supporting WASI have now recently come to existence, but most are run as community projects, lacking full-time developers and without official support from major browser-vendors.

2.4.2 Virtualization

The way Docker virtualizes away the underlying software and hardware architecture is through emulating an entire operating system (OS). This means the code written has to be taking this into account and external libraries have to be included in the image. Wasm simplifies this virtualization by ignoring the OS and the execution environment in general, and only focusing on the executable code, as the Wasm binary instruction-set file is read by the runtime and converted to the instruction-set native to the host. As such, a Wasm binary should be able to fully interface with any OS for which a runtime has been implemented, even allowing full system access through WASI, more so than that possible in Docker.

Portability

Because Wasm still relies on an underlying OS to be providing the Wasm binary a runtime, one might worry that the compiled binary stops working with the runtime if the Wasm versions for which the two were compiled differ too much. This is however not the case as the Wasm and WASI specifications promise full backwards compatibility. That is great for portability as it relates to time instead of hardware, as unlike old native code from many years ago which might not work on modern systems, a compiled Wasm binary will work in any Wasm runtime forever. The advantage over Docker here is that while a Docker image is portable across systems, the image itself will require updating as the virtualized OS becomes updated and patched for security vulnerabilities. Therefore a Docker image cannot always be used forever because of its complexity, but if a Wasm binary is well delimited to its use-case and there are no serious security bugs in the code written by the developer who created the binary, then it should in theory be valid forever.

Overhead

Like in any virtualization layer, the overhead stems from having to translate instructions from the containerized program to the actual host. One might assume that this would be lower in Wasm as there is no OS-level virtualization done there but this is often not the case. Docker enjoys many years of optimization work and is also able to utilize hardware virtualization resources or native OS-level virtualization, like that of Hyper-V on Windows. This gives Docker minimal overhead in practice, especially when the virtualized OS is the same as the host OS, as when running a Linux container on a Linux host. Wasm runtimes are more alike the language specific virtual machines like those of Python and Java, but also these have years of optimization behind them in their respective runtimes. So while the Wasm runtime does less in theory, as it only translates from virtual WebAssembly machine-instructions to native machine-instructions, it still suffers from the lack of optimization work done to the Wasm runtimes. But because the Wasm runtimes are similar to a native compiler, in the sense that the code can be recompiled to native-code offline (before execution), they should with time become more effective, just as native compilers have.

2.4.3 Sandboxing

The sandboxing capabilities of Wasm are very similar to those of Docker. The Wasm specification is designed to protect users from code they run while still allowing fine-grained controls over what access is allowed to the host OS. Both Docker and Wasm provide full sandboxing capabilities that stop anything unwanted from escaping either container, but while Docker almost solely interfaces with the underlying OS through shared files, the WASI specification allows most system-calls to be run by the Wasm containers. This does however not necessarily make the Wasm sandbox insecure, as it still provides a complete sandbox by default, with no outside access available. Any outside access through WASI that is to be available needs to be specifically defined when the Wasm container is initialized, like which files are to be accessible, and whether internet-access is provided, etc.

2.4.4 Orchestration

When discussing container orchestration, the practical question is really about the available frameworks, whether that be Docker Swarm or Kubernetes, for Docker containers. Technically, any new container technology can be employed in a container orchestration setup. But these are complicated pieces of software and writing your own orchestrator is for most not a viable option, so the question comes back to which preexisting orchestration frameworks are available for this container technology. During the first quarter of 2020 no frameworks supported Wasm, but Kubernetes looked like the most likely to be able to accomodate Wasm as it is not tied to any specific container technology. Then in April of 2020 the first project supporting Wasm containers was introduced, through Kubernetes, created and open-sourced by Deis Labs, a subsidiary of Microsoft. This project, called Krustlet [22], allows Wasm containers to be run as Kubelets in Kubernetes. However, it still suffers from many practical limitations, most notably the fact that it is not runtime independent and only supports the Wasmtime runtime, which does not support the ARM CPU-architecture, at the time of writing, and is thus not applicable to IoT-devices. So while orchestrating Wasm containers is possible, albeit poorly supported and only available on desktop and server computers, the Docker containers are still the most widely supported container technology, compatible with virtually every container orchestration framework.

Chapter 3
Related Work

In [8] the authors describe the motivations behind the creation of Wasm and to which use cases it might be applicable. It highlights the issues that make Wasm special and discusses how they could be utilized to evolve the web by turning it into a machine for all kinds of general-purpose computation that people do from day to day. This serves as a great introduction to WebAssembly but it is limited to the usage of WebAssembly within the web-browser and does not consider its use for edge computing.

In [5] a novel system for trusted resource accounting for cloud computing scenarios utilizing Wasm is suggested. This paper is one of the only ones that investigates WebAssembly for use in an actual scenario where its advantages over other container technologies, like Docker, are properly utilized. Their use case highlights how Wasm's portability and sandboxing capabilities produce a viable solution for running simple containers, but there is a general lack of attention to Wasm as the majority of the paper's focus is on the authors' novel solution for a resource accounting platform, and it also lacks a comparison to Docker and does not go into depth discussing how other container technologies could have been employed.

In [20] the authors argue that traditional container based isolation, for data processing in serverless architectures, leaves much to be desired. So they propose a more lightweight solution, based on Wasm containers, that with in-memory state sharing makes for a more efficient alternative. Utilizing their lightweight Wasm-based system they are able to speed up their specific use to make it twice as effective while utilizing only a tenth of the memory of what would otherwise be required with regular containers. This paper provides an interesting example of how Wasm containers can be utilized but the scenario for which they are using Wasm is highly specific and the discussion of how this could be achieved with other container technologies is limited. Therefore it is difficult to conclude whether or not Wasm containers really were the best choice here, but it does still highlights several aspects of Wasm specifically that make it a great alternative to Docker.

Very few studies exist yet that focus on how Wasm could be utilized on the edge or that look deeply into how it compares to Docker. Wasm and the WASI are still a very new technologies so most papers still look at them in a very generalized setting, not considering any specific areas where Wasm could provide unprecedented advantages over Docker. This is what we hope to do in this paper by studying Wasm's applicability specifically at the edge, on hardware constrained IoT devices, using the newly introduce WASI specification.

Chapter 4

Methodology

In order to investigate and answer the three research questions, a systematic literature review and quantitative experiments were performed to accomplish different objectives.

4.1 Literature Review

Performing a systematic literature review (SLR) comparing WebAssembly and Docker would provide us an overview of the current state of research into WebAssembly and help us identify existing research gaps. Some of these gaps can then be addressed in the quantitative experiments performed in addition to this review. The aim of this SRL is to answer the first research question assessing previous studies comparing WebAssembly to Docker. This review should be transparent and reproducible, so the steps taken throughout this review will be well delineated and precisely explained.

4.1.1 Search Criteria

In the following section we are going to present the criteria followed when searching for studies comparing WebAssembly to Docker.

Platforms and Search Engines

For finding the greatest amount of published studies several different platforms were used, these were:

- ACM Digital Library
- IEEE Xplore
- ScienceDirect
- SpringerLink
- Google Scholar

Google Scholar was expected to contain lots of duplicates of studies already found in the other engines, as well as many studies of lesser quality, but considering the need for a broad scope, it was still included.

Terms

For finding the relevant studies, through the search engines, we will be filtering by terms included in the entirety of the published text, to the extent that is available to the search engine. To keep a broad scope the criteria will only be the inclusion of the terms 'WebAssembly' and 'Docker'. Any study including both of these will be saved for manual review of its content and assessed for its relevancy to answering the research question.

Date of Publication

While WebAssembly first came into practical existence through its Mozilla Firefox and Google Chrome implementations in March of 2017, it was still lacking the WASI capabilities, these came in 2019, at that time that make it a proper contender against Docker in the container technology space. However, to not limit the scope of studies found we are including those published in 2017 and later. Because those studies considering WebAssembly without WASI might still provide valuable information in comparison to Docker.

4.1.2 Inclusion Criteria

After following the aforementioned search criteria the following inclusion criteria were used to filter articles found in the search to those with comparisons between WebAssembly and Docker relevant to the research question.

- The results of the study must be published in English.
- The study must be peer reviewed unless the results are purely quantitative and reproducible from thorough descriptions of the experiments performed.
- The study is based on facts and not opinion, it is thorough in its consideration of both WebAssembly and Docker, and it follows scientific practices and reasoning.

4.1.3 Search and Selection Procedure

The following procedure will be followed in searching and selecting studies for inclusion in this review, presented here in the order it will be performed.

1. **Initial search**: Each search engine will be searched using the two aforementioned terms and filtered by nothing but the aforementioned publication date.
2. **Exclusion based on title**: Studies with titles that very clearly do not match the kind of comparison between WebAssembly and Docker that is required to answer the research question are excluded.
3. **Exclusion based on abstract**: If the title is ambiguous or possibly relevant then the abstract will be assessed for its relevance to our comparison. If there is no mention of a comparison between WebAssembly and Docker here, in any sense, then the study is excluded from further consideration.

4. **Aggregation of filtered studies**: At this point a number of relevant studies have been found and the full papers will be obtained for reading. Duplicates will also be identified and removed at this point, before the papers are obtained.

5. **Late exclusion based on full paper**: If upon reading the full paper it is obvious that the paper is not relevant to our review or does not meet our inclusion criteria, then this is the last chance for it to be excluded from the review.

At each stage of the search and selection procedure the number of papers from each platform will be counted, apart from during the last two stages when full papers will have been obtained and duplicates will have been removed. Papers from stages 1, 2, and 3 will not be referenced or mentioned in this review because they will be too many and too irrelevant to our study. Every paper acquired in stage 4 and considered in stage 5 will however be referenced and it it is excluded in stage 5 then the reason for exclusion will be mentioned.

4.1.4 Quality Assessment

After selecting studies for review, to better know how much weight and consideration to give every selected study, a quality assessment would be performed following the guidelines set forth by Kitchenham and Charters [11]. This assessment would consider the selected studies's relevancy to answering the research question and the validity of the data and conclusions provided by them. The following criteria were involved for making the aforementioned assessments:

- **Container technologies**: The study's primary focus must be on container technologies or solving a problem revolving around the use of container technologies.

- **Application**: Enough information about the application of, and context around, the study must be provided. This is so that this review will be able to assess the applicability of the study's conclusion to the general use of container technologies.

- **Approach**: Detailed information about the approach taken to procuring the study's results must be provided. This is so that an assessment of the validity of the gathered data, and conclusions drawn, can be made.

4.1.5 Data Extraction and Synthesis

To extract the relevant data and synthesize the conclusions drawn from the selected studies, the following steps would be taken. This is the process by which the results of this systematic literature review would be produced and presented for answering the research question.

1. Read all full papers and produce a list of categories for the data and conclusions they provide. This could be performance, with sub-categories for different performance-metrics, sandboxing capabilities, container orchestration capabilities, etc.

2. Divide the papers up into the previously produced categories, but do no limit papers to belonging to a single category. A paper considering both performance and security capabilities can belong to both categories.

3. Study the results and conclusions of each category, consider which papers draw opposing or similar conclusions, and synthesize all of this into a summary of each category where all results are aggregated with consideration for the quality-weights produced in the quality assessment.

4.2 Quantitative Comparisons

A quantitative comparison between Wasm, Docker, and a native software implementation must be performed to compare the performance metrics of these platforms. In addition to this a quantitative comparison of the performance metrics between different third-party Wasm runtimes was planned. This could however not be carried out because despite every Wasm runtime announcing their plans to support the AArch32/64 ARM CPU architecture, only one of them, the Wasmer runtime, actually does this at the time of writing, highlighting the current immaturity of this ecosystem. Other runtimes were not considered to keep the focus of this study on IoT devices, which almost solely run on ARM architectures. Therefore all experiments performed here are using the Wasmer runtime [24] with Wasm-binaries compiled either with Emscripten [3] for the C++ code, or Rust's dedicated Wasm tool-chain [13] for the Rust code.

The quantitative comparisons performed were done so using various implementations of common CPU intensive algorithms as well as some programs designed purely for the purpose of this thesis. No preexisting benchmarking tools were compiled to Wasm or exist for Wasm, at this time, because these programs often interface heavily with the underlying system, and the WASI interface that handles this is still weak and error prone in most Wasm runtimes. Therefore we expected the process to become clearer and the results easier to reason for if we ran simple and well defined algorithms where knew exactly what was done in our tests.

The specific tests, namely CPU performance, memory utilization, and filesystem I/O performance were chosen because those are some of the most common metrics in comparisons between container technologies. Not many studies comparing these exist but [1] is an example of a paper comparing Docker to another container technology by comparing them through the three aforementioned performance criteria. A more recent study, from 2019, discussing a new Wasm runtime for serverless computing [9], also benchmarks its own runtime by startup time. This is very relevant for systems employing serverless/FaaS computing, for which Wasm is highly appropriate, so therefore this metric was chosen for as our fourth performance metric.

A benchmark for the networking was designed and planned, but because the Wasmer runtime still does not support sockets, this could not be carried through. The plan was for four tests, utilizing a server running on the same local network as the testing-device, to be run, so that network latency would have the minimum possible effect. These would have involved uploading and downloading a single large file, to measure single-file performance, as well as uploading and downloading lots of medium sized files at once to measure asynchronous networking performance.

Every test was carried out 10 times so that averages results could be presented and abnormalities avoided. This 10 executions were then used in the tests for calculating statistical significance. These significance tests were calculated using Friedman tests with a significance threshold of p=0.05, where any result below p=0.05 would be considered significant. The Friedman test works by converting the data-points of each test into ranks from 1 to 3 and then testing if there is a significant difference in the average rank of any of the solutions (native/Docker/Wasm).

4.2.1 Hardware

The quantitative experiments were performed on a stripped-down server-version of the Ubuntu Linux OS on version 20.04 LTS running on the Raspberry Pi 3 Model B+. They were first performed on the OS version 18.04 but during the process of writing this thesis a new Long Term Support (LTS) version was released so they were redone on that, so that the results would be applicable to the current OS for at least two more years, until in 2022 when the next LTS version releases. No noticeable difference could however be seen in the results between these two version. //

The Raspberry Pi 3 Model B+ was chosen specifically because its hardware seems to be appropriate for an average IoT device in the decade 2020-2030, albeit for the time of its release in 2012 (Model B+ was released later, but with similar hardware) it was definitely a high-end IoT device. This device runs on 1GB of LPDDR2 SDRAM with a Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC processor at 1.4GHz. All tests were run with the device plugged in (no battery) and without any restrictions for limiting power-consumption.

4.2.2 CPU Benchmarks

The CPU benchmarks were written in C and compiled to Wasm using Emscripten and executed within the Wasmer runtime. Eight tests were run for measuring CPU performance, all single-core, and the computations were for common well known algorithms. The performance metric considered was for the real-time it took for the algorithm to complete, with no other active processes, bare the minimum necessary, running on the system at the same time. A single warm-up run was performed for each test before the actual run, that was listed, was made. The eight algorithms were for AES encryption, Base64 encoding, Dijkstras algorithm, Huffman compression, matrix multiplication, the recursive quick-sort algorithm, SHA-256 hashing, and XOR encryption. This was considered a good blend between different computationally intensive computing scenarios; compression, cryptography, string encoding, graphing algorithms, mathematical equations, and sorting algorithms.

4.2.3 Memory Benchmarks

The memory benchmarks were also written in C, compiled to Wasm using Emscriptem and executed within the Wasmer runtime. For this a minimal application was used that did nothing but block the program. A that point, upon blocking execution, the memory consumption was then measured using the Unix free-command. This is a common and well regarded tool for measuring system memory consumption and the same one used in the aforementioned paper comparing Docker to another container technology [1]. Here, the native version was again used as the baseline, and the Wasm and Docker versions were measured by seeing how much more memory than the native version it consumed, ie. how much memory the runtimes consumed.

4.2.4 Filesystem I/O Benchmarks

The memory benchmarks were written in Rust, compiled to Wasm using the official Rust Wasm toolchain, and executed within the Wasmer runtime. C++ was first used but the Emscripten compiled code would not run on Wasmer, so the program was rewritten in Rust. However, because of the immaturity of the Wasmer runtime's support for the Aarch64 architecture, the Wasm binary would not run on the Raspberry Pi IoT device, the Wasmer runtime could however run the Wasm I/O code on an x86-64 processor. Therefore this test was the only one that could not be run on the originally considered hardware but was instead run on a desktop computer with an Intel i5-2500k processor, clocked at 4.8GHz, with 16 GB of RAM memory. The relative filesystem I/O performance differences between native-Docker-Wasm should however be similar to what they in the future will be on an Aarch64 IoT device, once they are better supported by the available Wasm runtimes.
For this test both read and write times were compared. Measuring ten times for each of the three platforms (native, Docker, Wasm) how long it took the program to read and to write a 1 gigabyte file. For these tests no warm-up runs were made and different files, albeit with the same content, were used to minimize the unexpected difference that caching might have on the results. This was done to increase uniformity among the two tests which would otherwise have ran for different lengths of time.

4.2.5 Startup time Benchmarks

The memory benchmarks were also written in C, compiled to Wasm using Emscriptem, and executed within the Wasmer runtime. Like for the memory benchmarks, a simple program was used with no other purpose than to initialize a runtime and then exit, for either native execution, for the Wasm runtime, or for a Docker container. The time in milliseconds was then measured, using the Linux time-command, it took the programs to start up. The Linux time-command was used for being a highly accurate and very well known option for measuring the execution times of programs on Linux systems.

Chapter 5

Results and Analysis

5.1 Literature Review

5.1.1 Search and Selection Procedure

Initial Search

Following the search criteria described in Methodology the initial search, on the five search engines, yielded 140 results in total. Of these 140, 14 were from the ACM Digital Library, 11 from IEEE Xplore, 1 from ScienceDirect, 9 from SpringerLink, and 105 from Google Scholar.

Exclusion Based On Title

After filtering based on title, where studies that did not mention anything even slightly related to either WebAssembly, Docker, or container technologies, etc. were removed, 37 studies were left. Of these 37, 7 were from the ACM Digital Library, 5 from IEEE Xplore, 0 from ScienceDirect, 3 from SpringerLink, and 22 from Google Scholar.

Exclusion Based On Abstract

After filtering based on abstract, where any studies with a focus clearly irrelevant to answering the first research question were removed, 13 studies were left. Of these 13, 2 were from the ACM Digital Library, 4 from IEEE Xplore, 0 from ScienceDirect, 0 from SpringerLink, and 7 from Google Scholar.

Removing Duplicates

At this point duplicates were identified and because the search on Google Scholar yielded all 6 papers from the ACM Digital Library and IEEE Xplore, 6 of the 7 remaining studies found through Google Scholar were removed. No other duplicates were found and thus 7 papers remained. Each of these 7 full papers were then acquired for the last step of performing a late exclusion based on the full paper.

Late Exclusion Based On Full Paper

Then it was time for the final exclusions based on the full contents of the acquired papers. At this stage any papers missing any sort of direct comparison between WebAssembly and Docker would be discarded. To not make the mistake of prematurely excluding any papers during our aforementioned exclusion stages, we were very lenient before and included several papers only based on the possibility that they could provide a comparison between the two container technologies. This did however mean that all of the remaining papers would be excluded because they only considered one of the two container technologies but provided nothing but some mentions of the other. This is what caused the papers to appear in our searches. One of the papers [15] did, for example, mention one of the other papers [9] chosen for our review as related work. The two papers which considered container technologies in the context of serverless did provide comparisons but not between WebAssembly and Docker directly. The comparisons were instead between WebAssembly and a serverless platform utilizing Docker [9] as well as between Docker and a novel runtime for serverless computing utilizing WebAssembly [21]. Because the serverless platform utilizing Docker and the serverless runtime utilizing WebAssembly considered in those studies are only abstractions of either technology, it is not possible to directly judge either Docker or WebAssembly by their results. The 7 papers excluded based on a review of the full papers, together with the search platform from which they were acquired and their reason for exclusion, are listed in Table 5.1.

Based on this lack of any studies comparing WebAssembly to Docker the remainder of the planned systematic literature review had to be cut. The papers presented in Table 5.1 do however serve as a relevant addition to the papers mentioned in the earlier section Related Work. It should also be easy to replicate this systematic review later when more studies exist, following the methodology defined in this thesis.

5.2 Quantitative Experiments

Now follow the results of the quantitative comparisons between natively executed code, Docker containerized code, and in Wasm container executed code.

5.2.1 CPU Benchmark

At greatly increased execution times in our benchmarks, visualized in Figure 5.1, the Wasm containers fared very poorly in comparison to the binaries compiled for the native architecture and Docker, which performed similarly. There was a significant difference $p<0.05$ between Wasm and Docker, in Docker's favor.

5.2.2 Memory Benchmark

In our memory benchmarks, visualized in Figure 5.2, where the total amount of consumed system memory while the program was running was measured, Wasm did better. The native solution had the lowest memory cost but the Wasm runtime still

Title	Platform	Reason for exclusion
AccTEE: A WebAssembly -based Two-way Sandbox for Trusted Resource Accounting [6]	ACM Digital Library	Utilizes WebAssembly but does not include any comparisons to Docker.
An execution model for serverless functions at the edge [9]	ACM Digital Library	WebAssembly is not compared to Docker, but to a platform utilizing Docker.
A Method of Dynamic Container Layer Replacement for Faster Service Providing on Resource-Limited Edge Nodes [14]	IEEE Xplore	Utilizes Docker but does not include any comparisons to WebAssembly.
A Survey of IoT Security Threats and Solutions [17]	IEEE Xplore	Considers WebAssembly but not Docker.
Challenges and Opportunities for Efficient Serverless Computing at the Edge [4]	IEEE Xplore	Utilizes WebAssembly but does not include any comparisons to Docker.
tinyFaaS: A Lightweight FaaS Platform for Edge Environments [15]	IEEE Xplore	Utilizes Docker but does not include any comparisons to WebAssembly.
Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing [21]	ArXiv.org	Docker is not compared to WebAssembly, but to a novel runtime utilizing WebAssembly.

Table 5.1: Papers excluded from the literature review after a full paper review.

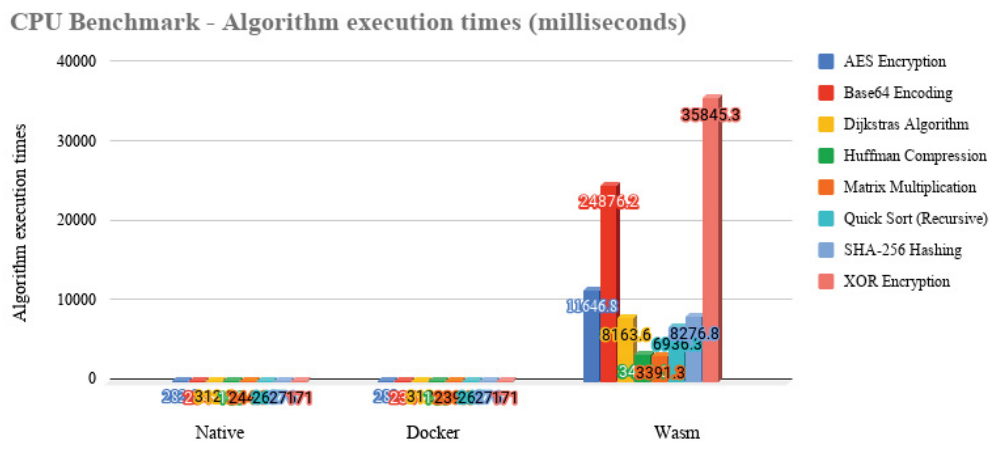

Figure 5.1: CPU benchmarks showing results for the ten tests. Normalized after the native implementation which is not shown but was normalized to 1.

consumed much less than the Docker runtime. There was a significant difference p<0.05 between Wasm and Docker, in Wasm's favor.

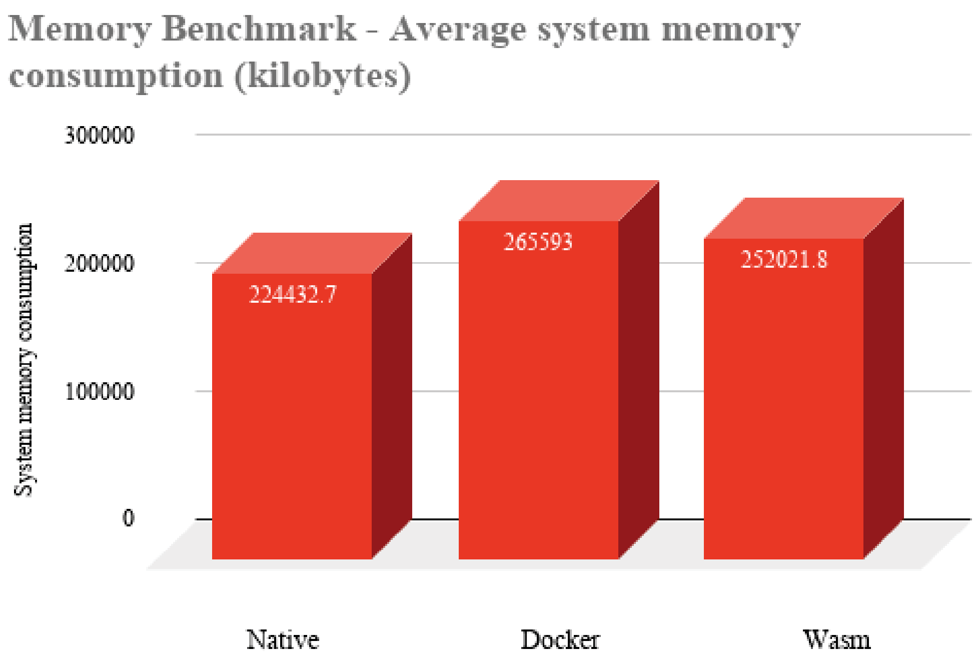

Figure 5.2: Memory benchmarks showing results for the ten tests. Normalized after the native implementation, showing increased memory usage of Wasm.

5.2.3 Filesystem I/O Benchmark

In the filesystem I/O benchmarks, visualized in Figure 5.3, the Wasm containers again fared very poorly in comparison to the binaries compiled for the native architecture and Docker. These results are similar to those observed in the CPU benchmarks. But most noticeable is the much higher penalty that the Wasm container took while reading instead of writing. There was a significant difference $p<0.05$ between Wasm and Docker, in Docker's favor.

5.2.4 Startup Time Benchmark

With Wasm containers doing much better than Docker containers in the startup time test, visualized in Figure 5.4, this experiment is the first to cast Wasm in a favorable light, as Wasm containers take only a fraction of the time to start that Docker containers do. There was a significant difference $p<0.05$ between Wasm and Docker, in Wasm's favor.

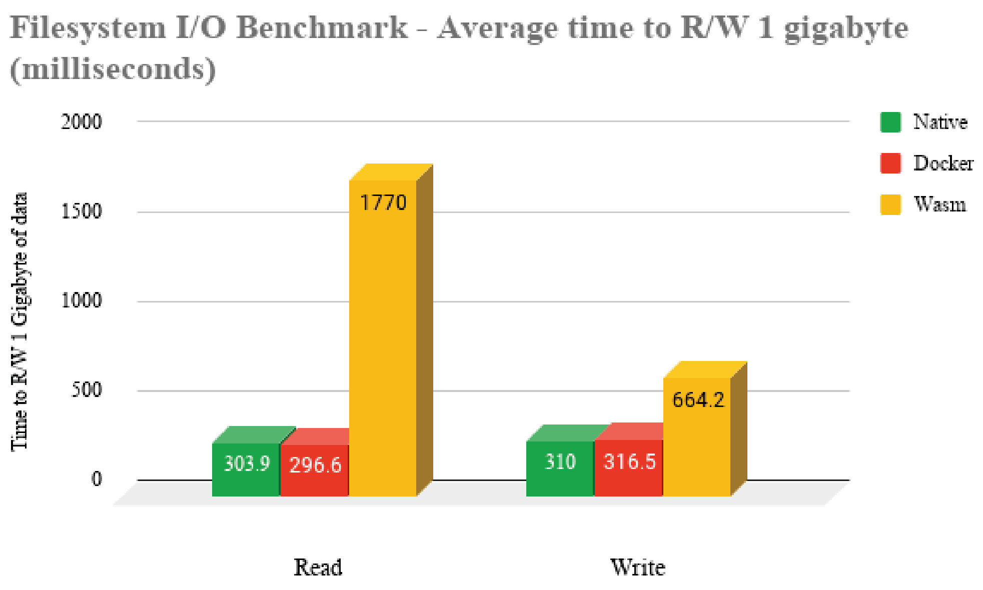

Figure 5.3: Filesystem I/O benchmarks showing results for the two tests. Normalized after the native implementation which is not shown but was normalized to 1.

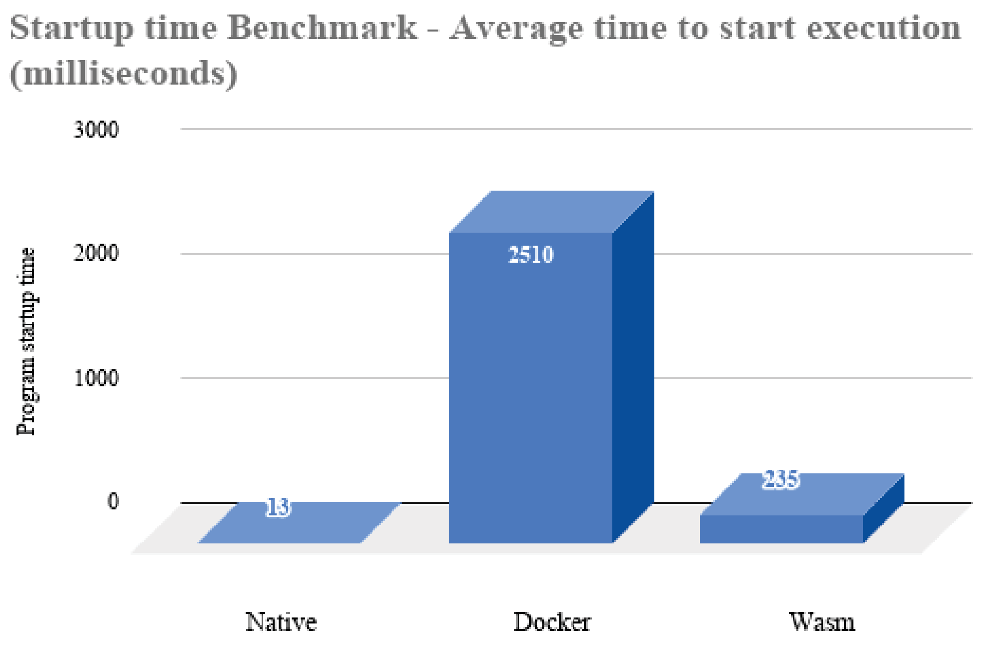

Figure 5.4: Startup time benchmarks showing results for the ten tests. Real-time values in milliseconds displayed without normalization.

Chapter 6
Discussion

Having performed our literature review and the quantitative experiments the still prevalent shortcomings of the Wasm containers have been made clear. As previously mentioned, there are two primary concerns relating to IoT devices:

- Hardware constraints, making highly optimized code important not only because every program will take several magnitudes longer to finish than on a server, but also because these devices often run on a battery and the cost of manually having to replace them can be immense.
- Issue of heterogenity, as devices will be running on different hardware architectures and running different operative systems, making portability crucial especially when fast deployment is important, as binaries should not have to be modified for different devices and code from many years ago should still be able to be ran on any version of the operative system.

So how has Wasm been able to address these issues and does it compare favorably to Docker?

Performance wise we saw the for-Wasm-compiled binaries take twice as long as long to finish execution as the for-native-compiled binaries. This shows poor utilization of the hardware capabilities, as the overhead associated with the Wasm runtime, for virtualizing instructions, was as large as the computation itself. This makes Wasm not viable on IoT devices with CPU-intensive applications meant to be ran for a longer period of time. But there is a silver lining; the startup time for each Docker container was a magnitude of x10 times higher than those of the Wasm containers, of over 2000 milliseconds on average. This difference is so immense that it must be considered and even puts Wasm at an advantage over Docker in numerous scenarios. Proper use of containers dictates that each container should serve a distinct task, a job well contained, this implies keeping services separated or spawning up new containers for new jobs, as done by a container orchestration framework. For long running applications such as database or REST-server that might be running on a data-collecting IoT device this startup time does not make a difference, as such programs will only start once. But for smaller jobs that run sporadically, possible triggered by a sensor connected to the IoT device, or a few times a day on a schedule, the startup time of a Docker container might be longer than the actual execution time of a containerized program. Some programs are also not compute-heavy, but rather sit idle most of the time, waiting for input. Even a severely hardware constrained device has time to get a lot of work done in 2000 milliseconds, which might be more than it needs, and considering that the performance penalty of Wasm is very high on average, it would take about 2-3 seconds for a Docker container based application to catch up with a Wasm container, considering its faster startup time. So if a program requires under a couple of seconds of computation time before it is finished, then the Wasmer container would be the faster alternative when startup times are considered in the time to completion.

When it comes to the heterogeneity of IoT devices and deploying to different hardware and operating systems, Wasm and Docker are approximately on par. Both platforms support Windows, Mac, and Linux running on both x86 and ARM chips, so there is no difference in the size of supported devices. Wasm does however have a slight advantage here in the sense that its binaries can be used for longer because they package less code with potential vulnerabilities. A Docker containers virtualized OS might have to be patched for security vulnerabilities if it is public facing or at the risk of other kinds of malicious tampering, whereas a Wasm container only has to rely on its to-binary-compiled Wasm code not containing bugs - the attack surface there is significantly smaller.

Having considered these issues now it seems the answer is not yet clear and the matter of which container technology is better comes down to a matter of circumstance. The Wasm platform and ecosystem does however still suffer from serious immaturity, this was most noticeable by the lack of supports for opening ports through our runtime, the only Wasm runtime we were able to test as the others did not support the AArch32/64, the only relevant one for IoT devices. Supports for handling ports is however included in the WASI specification so it should only be a matter of time before this functionality becomes useable in practice. Ending on that note, it seems time is still the major obstacle, and given some more time we should see more runtimes support the AArch32/64 architecture, full WASI support in runtimes, and hopefully further optimizations being made in the conversion of Wasm bytecode to native bytecode by the compilers resulting in better CPU utilization.

Chapter 7
Conclusions and Future Work

This study investigated the potential for Wasm containers to make deploying to heterogenous and hardware-constrained IoT devices on the edge easier. A literature review was done, examining the official Wasm and WASI specifications and documentations. The Wasm container was compared to the Docker container both through the aforementioned literature review, as well as through concrete, quantitative comparisons to natively executed code and Docker containerized code. Finally, a discussion brought all this together in the context of deploying on IoT devices on the edge, and we concluded that the Wasm container can be a viable alternative to the Docker container given the right requirements and circumstances, but that the Wasm ecosystem ultimately still is in an immature shape and some issues will have to be addressed before Wasm can be broadly used in edge computing. The most important ones being feature parity with the official WASI specification as well as further improvements to the Wasm to native bytecode compiler to reduce CPU-performance costs of running in Wasm container.

RQ 1: What conclusions have previous studies been able to draw from comparisons between WebAssembly and Docker? Null hypothesis: previous studies have not been able to compare WebAssembly and Docker or have not found any differences.

Answer: The null hypothesis could not be rejected. There exist no peer-reviewed studies comparing WebAssembly to Docker and thus no conclusions could be drawn from previous research comparing these two container technologies.

RQ 2: How do Wasm containers compare to running code natively or in Docker containers in CPU utilization, memory utilization, network utilization, and filesystem I/O utilization performance metrics? Null hypothesis: there is no difference in the performance metrics between Wasm containers and native execution / Docker containers.

Answer: The null hypothesis could be rejected. There is a significant difference in performance metrics but, in most cases, not in the favor of the Wasm container. This could however not be answered in relation to network related performance due to a lack of support from the runtime.

RQ 3: How does the Wasm platform compare to that of the Docker platform in terms of support for container orchestration? Null hypothesis: the Wasm platform supports container orchestration to the same extent as the Docker platform does.

Answer: The null hypothesis could be rejected. The Wasm platform supports orchestration only through an immature and feature constrained Kubernetes project that utilizes a Wasm runtime incompatible with the AArch32/64 ARM CPU-architecture on which IoT devices run.

7.1 Future Work

Two topics will be suggested for future work, resolving both of these can be seen as an integral part of making the Wasm platform viable for widespread use.

Due to one of the primary issues with bringing Wasm containers to IoT devices found to be the poor CPU-performance, we suggest a study of the Wasm to native bytecode compiler be made to analyze where the bottlenecks lie. This could be for both the purpose of helping compiler developers see where their attention is required, as well as helping developers by showing them how they can circumvent known bottlenecks, thus producing more performant code through deliberate practices.

As container orchestration lies at the core of most container based software set-ups a proper orchestration framework is required for the Wasm platform to flourish. Thus we suggest a study focusing solely on this issue of what would be required for Wasm-based container orchestration to reach feature parity with that of the Docker-based orchestration frameworks. This could include an implementation of the suggested framework, although it might be too large of a job for a small team of researchers to complete and would probably require the collaboration of an organization with expertise in these frameworks.

References

[1] MinSu Chae, HwaMin Lee, and Kiyeol Lee. A performance comparison of linux containers and virtual machines using docker and kvm. *Cluster Computing*, 22(1):1765–1775, 2019.

[2] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web [On-line; accessed 7-February-2020].* Mozilla Hacks Blog, https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, 2019.

[3] Emscripten Contributors. *Emscripten Homepage [On-line; accessed 3-May-2020].* https://emscripten.org/, 2020.

[4] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615, 2019.

[5] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 123–135, New York, NY, USA, 2019. Association for Computing Machinery.

[6] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 123–135, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[9] 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, page 225–236, New York, NY, USA, 2019. Association for Computing Machinery.

[10] N. Hassan, S. Gillani, E. Ahmed, I. Yaqoob, and M. Imran. The role of edge computing in internet of things. *IEEE Communications Magazine*, 56(11):110–115, November 2018.

[11] B. Kitchenham and S Charters. Guidelines for performing systematic literature reviews in software engineering, 2007.

[12] R. Morabito. Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, 5:8835–8850, 2017.

[13] The Rust Language Organization. *WebAssembly - Rust Programming Language [On-line; accessed 3-May-2020]*. https://www.rust-lang.org/what/wasm, 2020.

[14] J. Park, Y. Kim, A. Son, Y. Lim, and E. Huh. A method of dynamic container layer replacement for faster service providing on resource-limited edge nodes. In *2019 IEEE 2nd International Conference on Electronics and Communication Engineering (ICECE)*, pages 434–437, 2019.

[15] T. Pfandzelter and D. Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24, 2020.

[16] G. Premsankar, M. Di Francesco, and T. Taleb. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal*, 5(2):1275–1284, April 2018.

[17] A. RADOVICI, C. RUSU, and R. ŞERBAN. A survey of iot security threats and solutions. In *2018 17th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–5, 2018.

[18] S. Rajan and A. Jairath. *Cloud Computing: The Fifth Generation of Computing.* IEEE, 2011 International Conference on Communication Systems and Network Technologies, June 2011.

[19] D. Schäfer, J. Edinger, S. VanSyckel, J. M. Paluska, and C. Becker. Tasklets: Overcoming heterogeneity in distributed computing systems. In *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 156–161, June 2016.

[20] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing, 2020.

[21] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing, 2020.

[22] Ralph Squillace. *WebAssembly meets Kubernetes with Krustlet [On-line; accessed 7-May-2020].* Microsoft Open Source Blog, https://cloudblogs.microsoft.com/opensource/2020/04/07/announcing-krustlet-kubernetes-rust-kubelet-webassembly-wasm/, 2020.

[23] W3C. *World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation [On-line; accessed 7-February-2020].* W3C Press Releases, https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en, 2019.

[24] Inc. Wasmer. *Wasmer - The Universal WebAssembly Runtime [On-line; accessed 3-May-2020].* https://wasmer.io/, 2020.