Linköpings universitet/Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16 ECTS | Programming
Spring 2021 | LIU-IDA/LITH-EX-G--21/036—SE

Containerizing WebAssembly

Considering WebAssembly Containers on IoT Devices as Edge Solution

Fredrik Eriksson
Sebastian Grunditz

Supervisor: Jody Foo
Examiner: Peter Dalenius

LINKÖPINGS UNIVERSITET

Linköpings universitet/Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16 ECTS | Programming
Spring 2021 | LIU-IDA/LITH-EX-G--21/036—SE

Upphovsrätt

Copyright

Considering WebAssembly Containers on IoT Devices as Edge Solution

Fredrik Eriksson
Linköping University
Linköping, Sweden
freer320@student.liu.se

Sebastian Grunditz
Linköping University
Linköping, Sweden
sebgr273@student.liu.se

ABSTRACT

This paper will explore the speed of execution, memory footprint and the maturity of WebAssembly Runtimes (WasmRT). For this study, the WasmRT will be Wasmer[1] and Wasmtime.[2] Initially, benchmarks were run on a Raspberry Pi 3 model B to simulate a more hardware capable IoT-device. Tests performed on a Raspberry Pi shows that there are many instances where a WasmRT outperforms a similar Docker+C solution.

WasmRT has a very clear use case for IoT devices, specifically short jobs, the results from our research will show that WasmRT can be up to almost 70 times as fast as a similar Docker solution. WasmRT has a very strong use case that other container solutions can not contend with. This paper will show how effective a lightweight, portable, and fast WasmerRT can be, but also to highlight its pain points and when other container solutions may make more sense.

Author Keywords

WebAssembly, WASI, Wasmer, WasmTime, native, benchmark, IoT, Docker, Edge, Containerization

INTRODUCTION

Today, IoT-devices vary in operating systems and hardware configurations. One solution for developing applications on these devices, that does not include extensive knowledge on available resources, is abstracting the development to using an Software Development Kit (SDK) and Docker containers. Docker [3] and its' containers provides developers an easier way to ship software and configurations. It is similar to a Virtual Machine, but more lightweight as Spoiala et al.[20] shows, and allows users to run software as the developer intended, in a safe environment. Docker allows for the same program to be run on several different OS and hardware configurations without rewriting the program for each permutation[14].

Docker containers are very good for long running programs such as http-servers or databases. However, when running short, sporadic, burst-like jobs, Docker containers can impose overhead in both memory and startup time.

For network cameras, the programs are often event driven, where the program might listen to webhooks or similar events, and when an event is triggered, a single job (calculation et cetera) starts. If the job is called with varying frequency , a Docker container or virtual machine that is up and running at all times might not be feasible on restricted hardware but also unnecessary to have an OS environment per event even for hardware that can handle it. However running a native executable might not be desirable not only for security reasons but also because having the abstraction layer for "compile once and run everywhere" is sought after. For this, a WebAssembly Runtime (WasmRT), is a great fit. A WasmRT is great for when a job can be directly replaced with one binary. It is not meant to replace containers such as Docker entirely, but to be a alternative when a whole OS-virtualization is not needed.

A runtime is a sandbox environment, that allows the user to run code in a safe setting. Code executed in a runtime has no access to the OS functions, like reading and writing files. Wasm needs a runtime to be executed because it originally was created for use in webbrowsers, where all code is executed in virtual machines. The WasmRT is essentially the middle man between the OS and the Wasm executable.

Research objective

The objective of this article is to evaluate the performance, execution time and max memory allocation, for different WasmRTs, comparing them to a similar Docker+C solution, to see if a WasmRT can be a valid alternative to Docker solutions on an IoT device.

Research questions

The more specific research questions to be answered is:

- How does various C-programs compiled to WebAssembly, ran in a WebAssembly Runtime, differ from the same C-program running on a similar Docker solution. In terms of startup time, execution time and memory usage?

- How does the different Wasm Runtimes differ in terms of memory and execution time?

[1] https://wasmer.io/
[2] https://wasmtime.dev/
[3] docker.com/

Delimiters

There are many different ways to compile a file to Wasm binary-format, there was not enough time to test all available compilers. Similarly, there exist many different WasmRTs. Due to time constraints, two where chosen as representatives for WasmRTs in general.

All tests were ran on a Raspberry Pi model 3; this does not represent all IoT-devices, although the hardware limitation of a Raspberry Pi model 3 does capture a wide range of IoT devices.

Our research will look closely at startup time, execution time and memory usage, but the scope will end there. We did not have time to cover qualitative aspects of development of programs with Wasm. Therefore aspects such as maintainability, ease of use and other qualitative aspects will not be taken into account.

BACKGROUND

While there are many potential uses for WasmRTs and Wasm in general, it has specific applicability for containerization on restricted hardware. One of the main benefits of a container solution is that it allows for portability.

Consider an example from the company Axis Communications that develops IoT devices such as network cameras. These network cameras can be quite constrained hardware wise, and are often varying in their hardware configurations. A container solution, allowing the same code to be executed on different hardware configurations, would be beneficial. However due to memory constraints, containerization is not always possible.

Looking at alternative solutions, such as WasmRTs, may provide less memory usage and in general can be combined with code that is running natively on the OS, instead of an abstraction layer above the OS. WasmRT could then provide the company with a way to increase portability of code, and make the development for uniform applications across platforms easier.

Axis Communications

Axis Communications is a manufacturer and provider of network based solutions in physical security and video surveillance. Their current solution for developing applications to run on their network cameras are based on Docker+C/Python. This limits who can develop applications for the cameras, or extends the development time if the developer is not proficient in C or Python. But more importantly, because Axis Communications provide a array of network cameras which has different hardware solutions, providing a "compile once and run everywhere" solution such as Wasm could be beneficial.

Because of this, they would like to explore how Wasm can be integrated in the workchain to make the development process faster and hopefully easier.

THEORY

First we will talk about WebAssembly and Wasm runtimes, to then move over to IoT devices, network cameras, and benchmarking.

```c
int add(int a, int b)
{
  return a + b;
}
```

(a) source code in C

```
; setup functions
(function $add
  (param $0 i32)
  (param $1 i32)
  (result i32)
 (i32.add
  (get_local $1)
  (get_local $0)
 )
)
```

(b) WebAssembly text format

```
wasm-function[0]:
  sub rsp, 8
  mov ecx, esi
  mov eax, ecx
  add eax, edi
  nop
  add rsp, 8
  ret
```

(c) WebAssembly binary

Figure 1: Add function compiled to WebAssembly

WebAssembly

WebAssembly (Wasm) is a portable compilation target for programming languages created by World Wide Web Consortium (W3C)[19], that was first introduced to the web to provide an alternative to Javascript for high performance code. Since Wasm is a binary instruction format, it is not meant to be written by hand, although possible, but rather it is intended to be used as a compilation target for various established languages such as C, Rust, etc.

Even though Wasm is a compilation target for languages that can be run natively or in an already existing runtime, Wasm itself should not run standalone. It has no IO capabilities on its own, and can not do anything other than heat up the CPU. Wasm files requires a runtime to execute its instructions, in a browsers internal virtual machine, Node.js, or a WasmRT for example.

Compiling Wasm

There are many different ways of compiling Wasm, and it largely depends on what language is used to write the source code. For C, the Clang+LLVM compiler has built-in support for compiling C-code to Wasm-binaries. An example showing the translation from C to Wasm-binary can be seen in fig. 1. The first step shows the source code in C, adding a and b together and returning the result. The second step shows the same code translated to Wat (WebAssembly text format), loading addresses and so forth. The last step shows the binary code, moving data between registries and using near-hardware instructions.

Wasm and the world

The initial aim of Wasm was to create a language that could deliver near native performance to the browsers. This gives Wasm the property of being containerized by default, protecting the host. Wasm can not access any resource from the host computer by design.

In order for Wasm to gain access to resources from the host, WASI (WebAssembly System Interface) was created. WASI creates a connection between the Wasm-container and the host OS, similar to how a Virtual Machine, or Docker, works. However, Wasm has interesting properties such as having complete control of what the binary can access from the host. It allows for full access to the real system while being in a isolated sandbox. This means that Wasm+WASI could be a contender to Docker. Docker's founder Solomon Hykes said

"If WASM+WASI existed in 2008, we would not have needed to created Docker."[4]

WasmRT

WebAssembly Runtime is essentially an environment where Wasm can be executed safely, it provides an abstraction layer above the OS, containerizing the execution.

There are different implementations of a WasmRT, that offers different functionalities. Two of them are Wasmtime, a open-source runtime governed by the Bytecode Alliance, the same people that first developed the Wasm-standard, and Wasmer, also open-source, but governed by a private company. The main difference between these two is the compiler converting Wasm-binary code to executable machine code: Wasmtime uses CraneLift compiler, while Wasmer has the option to choose between Singlepass, CraneLift, and LLVM.

IoT Devices

Internet of Things, or IoT, simply means to connect devices to the internet. These devices could be sensors, home appliances, cameras and so on. This creates a way to exchange data with other systems and devices through the internet without the need of manual human interaction.

A good example would be to send an action to a camera to start recording or to pan, zoom or tilt the camera. These devices have more often than not limited processing capability caused by a constrained hardware configuration to save battery. Many IoT-devices are made to be powered by Power-over-Ethernet (PoE), which enables a single ethernet cable to act both as a internet connection and as a power delivery cable.

Network Cameras

Network Cameras (NC) are commonly used for surveillance, these cameras behave differently than normal analog cameras. Most network cameras are not required to record at all times since NC can be highly configured. The standard function of a NC can be seen as a webcam where the camera is not tasked with recording to one big file but instead to stream the video directly over the internet. If surveillance is needed, multiple NC can be setup and can be handled remotely.

Because NC can be handled remotely, it lends to easier handling of multiple camera streams but also high configurability. With internet connection, the NC can easily decide on what action it should take according to what the NC sees or hears. Not only can the camera take local action such as playing a sound or start recording when it detects movement but the NC could also alert a fire department if fire is detected, since it is connected to the internet. It is just a question of implementation of software.

Benchmarking

Since this study only looks at the performance of different container solutions, it is important to understand that the result does not take factors such as time to develop, maintainability, lines of code, and so on into account. Such factors are obviously important for the development of programs but will not be investigated in this study.

[4] https://twitter.com/solomonstre/status/11...

The first part of the benchmark is speed, which is measured in seconds wall clock time per execution. Wall clock time is the real time an execution takes, measured from a timer or wall clock. This is common practice in benchmarking [9, 10]; it records the execution time of a program as it would be perceived by a user.

The second part is the memory, which is measured in max resident set size in KB per execution. Measuring the memory is important to due to the fact that IoT devices can usually be quite memory constrained, by measuring the absolute max memory usage of a process during its lifetime, a good understanding of the memory usage of containers can be understood.

RELATED WORK

Previous work from Napieralla [16] has laid a good foundation for comparison of WasmRT versus Docker, ultimately the research is similar but our work should provide a wider suite of test cases and a larger collection of WasmRTs is tested.

One early account of Webassembly usage outside of the browser in a WasmRT was by Goltzshe et al [3]. The authors utilized Wasm with Node.js as an WasmRT, were the aim was to have a light-weight safe execution model for remote code. Where the authors utilized Wasm to execute code on an abstraction layer above the OS, to safely run workloads. This research gives a sample use case for Wasm, however it only runs Wasm in the context of Node.js (V8 Engine).

From previous work it was extrapolated that there was sample implementations of WasmRT and use cases, this paper seeks to extend the knowledge of Wasm as an container and its applicability on hardware constrained IoT devices. While there is very little research done for this subject there are some research done on the comparison of WasmRT to a similar solution with Docker on IoT devices, as shown from Napieralla [16].

Speed of Wasm

The output of the Wasm compiler is very close to instructions of a modern processors instruction set, just a small abstraction above final instruction set, this allows for portability of the format. The WasmRT is then tasked with compiling the Wasm format to actual instructions for the underlying hardware. Moreover if Wasm needs to interact with the host in some way those functions need to be imported as stated from Nießen et al [17]. Extensive benchmarking has been done in Wasm comparing it to other languages. The results from Herrera et al [5], it is clear that Wasm is generally in the middle between JS and native code. There is a clear speed difference when tested with different browsers: most of the time native is faster than Wasm, and likewise Wasm is faster than JS. Which leads to the conclusion that Wasm may have increased portability and safety but it may come with a performance cost.

Rossberg et al. [19] measured the execution time for a suite of benchmarks in C called PolyBenchC, running both in *asm.js*, Wasm, and native. The results shows Wasm averages a 33.7% increase in execution time compared to native. It also shows

that the code size of Wasm is 62.5% of the size of corresponding *asm.js* code, and 85.3% of the size of corresponding native x86-64 code.

On the edge with Wasm

There are quite a few articles on the subject of using Wasm as a serverless solution for edge computing.

Hall and Ramachandran[4] puts Wasm up against OpenWhisk, an open source platform using the Docker engine, in three scenarios: single client multiple access, multiple client single access, and multiple client multiple access. They discovered that the cold-start performance is much better on Wasm, and also that Wasm seems to have a more stable performance, with less variance in latency between runs.

Murphy et al.[15] compared both different runtimes of Wasm+WASI, and compared Wasmer, a WasmRT, against other serverless solutions. The first test showed that, aside from native C, Wasmer had the best runtime, with Lucetc-wasi as a close third, and WasmTime lagging behind at fourth. Comparing serverless solutions showed that Wasmer performed better than OpenWhisk, and much better than AWS (Amazon Web Services), while getting beaten by Bluemix (IBM solution) and native C; further building a case for using Wasm+WASI as a serverless solution.

Although many articles of using Wasm on the edge exists, few compare them against Docker, we only found one [16]. The author compares Wasmer to Docker, and finds that Wasmer more often than not loses. The comparison is made by running a host of C-programs compiled to Wasm with emscriptem, a toolchain for compiling to Wasm. It does not seem however, that the author treats Wasmer as executing a binary, instead of starting a sandboxed environment to run a file in, since the author meassures only the execution time of the programs in Docker once it has started, thus disregarding the startup time that is imposed by running in a sandboxed environment. Worth noting is that when the author wrote the article, WasmTime could not be run on the ARM architecture, thus disabling that comparison.

Further on, the large difference in native versus Wasmer execution time seems to be remedied with updates to Wasmer.

Docker on the edge

There is work done in the field of evaluating how Docker performs in an edge computing context on IoT-devices; both in terms of evaluating how different images perform on Docker [6] and comparing performance of native vs Docker [11, 12, 13].

The performance of different Docker images is an interesting subject, since it is more about optimizing Docker, more than evaluating Docker as a technology. Islam et al. [6] show that there is much improvement to be done from the official Docker image; by switching to an Alpine image they reduce the deployment time by ca 50%.

Comparing Docker against native shows a negative performance impact, but according to Morabito et al. [11], Dockers performance is good enough for usage in an edge computing context. Mendki [12] shows that Docker is capable, by running an image processing program on a Raspberry Pi and comparing the native performance with the performance of a Docker container runnng the same program. The authors also concludes that Docker is lightweight enough to run on an edge computing IoT-device. Morabito [12] performs some synthetic benchmarks to benchmark the CPU, Memory, Network I/O bandwidth, and Disk I/O bandwidth, as well as some more real life benchmarks. The findings also here show that Docker is a viable solution for edge computing. Noteworthy is that all these benchmarks are longer in nature, which could bias the results towards Docker.

METHOD

First we describe Polybench-C, test bias, and selection of WasmRTs to then move on to how the benchmarks were performed and how the results were recorded.

PolyBench/C

PolyBench-C is a suite of 30 different numerical computations, ranging all the way from linear algebra to image processing and physics simulations, written in C [18, 22] and widely used in benchmarking [1, 2, 8, 21]. It provides a wide range of tests to ensure a good coverage of language capability is reached.

PolyBench/C has both shorter, burst like jobs, as well as longer; it also has workloads that uses less memory, and workloads that uses substantially more. This shows both the strengths and weaknesses of Docker and Wasm, as they can show how well they perform in the different subsets of these workloads.

The drawbacks of using PolyBench/C is that it is more of a simulated workload, rather than what could actually be produced in a real production environment. However, it can still be indicative of the performance of the different solutions.

Test Bias

We have tried our best not to introduce any bias towards any particular solution. There had to be some selectivity in how we chose our tests though; they all needed to be able to compile to WebAssembly.

The image used for the Docker tests was a Debian image, provided by Dockers official images [5]. Other images were tested, such as Ubuntu and Alpine, however Debian was faster than the other two, therefor Debian was chosen to be used as the default Docker image for the Docker tests.

The choice for PolyBench/C test suite was because it is compilable to Wasm, but also allowed for a good overlap between the kernels and calculations that would be done on a network camera.

Ultimately this report was requested by Axis Communications. They have not tried to change the outcome of the study by providing specific benchmarking tools or suites; they have provided network cameras, but these could unfortunately not be used in the study. Axis Communication were not seeking

[5] hub.docker.com/debian

a specific result but rather to see the feasibility of WasmRT and Wasm usage on their hardware in general. Therefore not incurring any bias towards any particular solution.

Selection of WasmRT

For our thesis, we chose to work with WasmTime and Wasmer as representatives for WasmRTs. This choice was based on two reason. First, both Wasmer and WasmTime has support for WASI, which we consider a necessary requirement for a WasmRT in order to be competitive against Docker at all. Second, Wasmer is comparing it self favorably against WasmTime[6]in a way that could seem unfair and unrealistic. This comparison incites us to research if there are substance to their claims.

For WasmTime, we found no relevant flags to be appended to the run argument; there is a selection of different proposal-flags that enables different experimental parts of Wasm, such as threads, but these were deemed non-essential for our tests.

Meanwhile, Wasmer offers many permutations of compilers and engines to use when running the Wasm-program. We chose the default configuration, JIT-engine and Cranelift-compiler, since this is what is recommended for development by Wasmer and the LLVM-compiler with Native-engine is recommended for production.

Benchmarking WebAssembly Runtimes

Memory usage was measured using Linux built-in Time module, called by **/usr/bin/time -f '%M'**, to get the max momentary memory usage in KB. Time was measured using pythons `datetime` module, timing how long each benchmark took in seconds, measured in eight significant digits.

These results were continuously appended to files so that for each test-run execution time was automatically recorded, enabling multiple sequential runs of the benchmark suite without supervision. The source code can be found at our github[7].

Benchmarking Startup behaviour

The startup behaviour of containers were measured in a similar method to the previous benchmark. Here, both startup time and memory usage was measured of the previously listed WasmRTs and a Debian Docker container. All test of the startups were performed by running a C main function that returns 0, and does nothing else. This was done to test the startup behaviour, how fast and how much memory did it take to reach the main function. The metrics were measured in similar fashion to the previous section.

Benchmarking on IoT

For the benchmarks, all test were ran on an Raspberry pi 3 model B. The devices has 1 GB of RAM and an Quad Core 1.2GHz 64bit CPU (Broadcom BCM2837), running ARM64. The Raspberry Pi was running Manjaro Linux 5.10.31-1-MANJARO-ARM in TTY mode.

[6] wasmer.io/wasmer-vs-wasmtime
[7] http.github.com/FreerGit/Ax...

Comparison of benchmarks

Each benchmark described in the previous sections was run 10 times to produce a larger dataset. These where then averaged, and the minimum and maximum value was also recorded, so that standard, minimum, and maximum deviation could be identified.

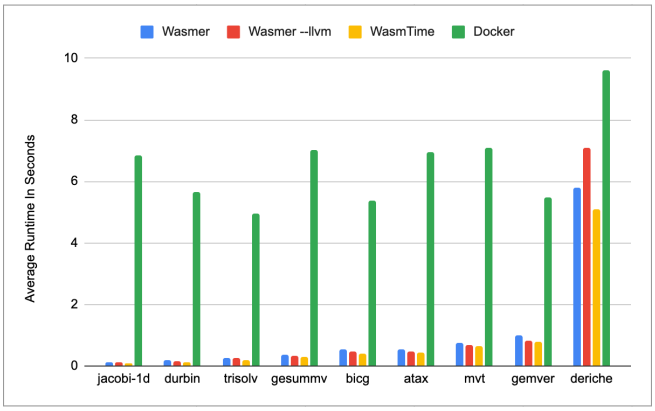

Figure 2: Lower part of execution time graph

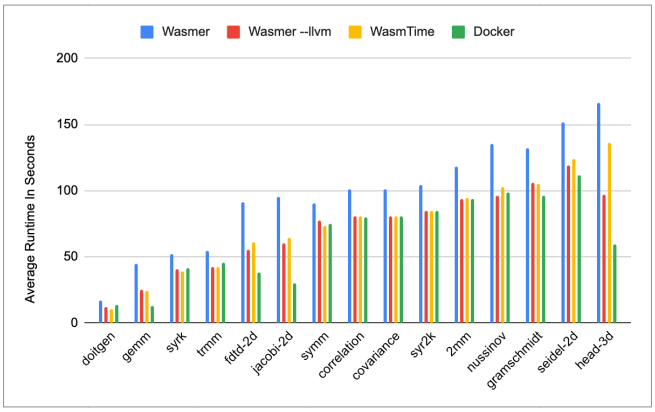

Figure 3: Middle part of execution time graph

RESULTS

The time and memory results from the benchmarks on the Raspberry Pi 3 will be shown in here.

PolyBench-C Benchmarks for Raspberry Pi

As can be seen in figs. 2 to 4, the execution times wildly differed depending on the kernel. On the short jobs (sub 20 seconds) Docker is significantly slower than all three other; being at most 69 times slower than WasmTime when executing `jacobi-1d`. The longer jobs saw a shift in the numbers, with Docker and WasmTime trading blows, and Wasmer with the `--llvm --native` flags occasionally getting top position. All the while, Wasmer without extra flags was the slowest of the bunch, with the worst being almost 3.5 times slower than Docker at `gemm`.

The memory is a different story. Wasmer, both with and without the `--llvm --native` flags, has a max memory utilization of about 1.27 times more than WasmTime. Docker

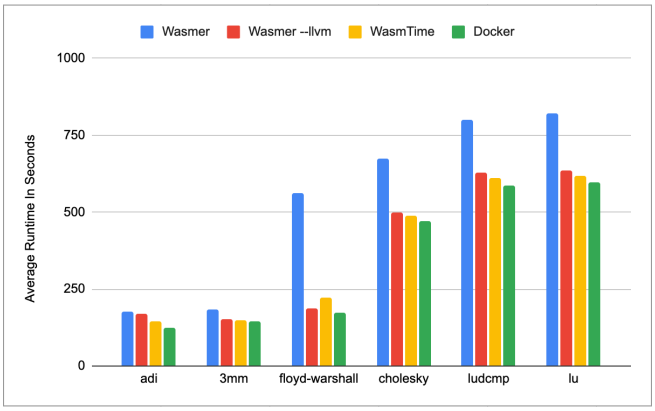

Figure 4: Higher part of execution time graph

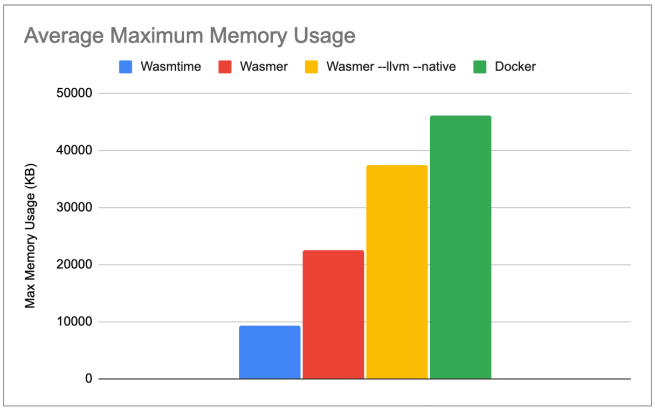

Figure 6: Average max momentary startup memory usage for each container type

has a consistent max memory usage of about 46000 KB +- about 1000 KB, taking more resource than the others on the light kernels while allocating less on the heavier parts.

Startup Benchmark for Raspberry Pi

This section will present the results from the startup benchmarks over Docker and the chosen WasmRTs, the results from the Raspberry Pi will be presented.

As can be see in fig. 5, there were a noticeable difference between startups of the containers, note the logarithmic scale. The Docker container had an average startup time of 6300ms which was the highest of the four, with Wasmtime averaging a 50ms startup, which was the lowest.

The memory usage of the containers were also recorded, as can be seen in figure fig. 6. Docker reaching the highest peak of memory usage with roughly 46 MB and Wasmtime show the lowest peak with a little above 9 MB of memory.

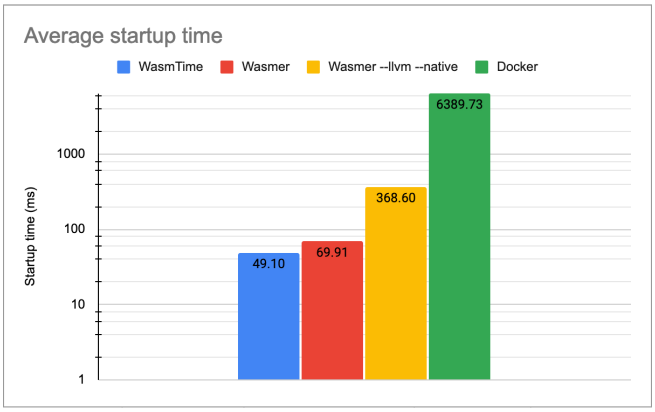

Figure 5: Average startup time for each container type, in logarithmic scale

DISCUSSION

First we discuss the methodology, testresults of benchmarking and the startup time for the different WasmRTs and Docker to later go into the PolyBench/C results.

Due to close consideration of the benchmarking method, the results from our research does provide a high certainty of the use case for Wasm on IoT devices. While a Raspberry Pi is not by definition an IoT-device, it can be used as such, and is a good platform for simulating the constrained hardware environment that is often present. Since IoT-devices can vary wildly in hardware configurations, there is no one device to run all tests on to get a definitive result. Our research is instead a step on the way to understanding how Wasm and it's different runtimes can be used in a IoT context.

From the startup tests that were ran on the WasmRTs and the Docker container, it was clear that Docker has a larger memory overhead than WasmRTs in general. Quite unsurprisingly so, since Docker requires virtualization of the OS. Another interesting difference between the WasmRTs is that they have quite a varying amount of memory usage. When run with the --llvm and --native flags, changing compiler and engine, Wasmer had almost 4 times the max memory usage of Wasmtime. We believe this is because Wasmer with these flags incur a larger overhead because of the compilation and optimization that llvm and the native engine does. The difference between Wasmtime and Wasmer is probably caused by the efficiency of JIT compiler of Wasmtime.

As was hypothesised in the introduction, the startup time of Docker can cause problems for short jobs; fig. 5 shows this clearly. Docker took rougly 130 times longer to start than Wasmtime, again this is due to Docker requiring actual virtualization of the OS. However, this shows an interesting point: WasmRTs in general greatly outperform Docker in terms of startup time and if the jobs are short in nature; the overall execution time would decrease a lot by WasmRT. A similar distribution between the WasmRTs was found in startup time as in memory usage shown in fig. 6. This distribution is likely due to the fact that Wasmer with `--llvm --native` flags has to compile and optimize, which will lead to a slow down in startup but a speed up in runtime. We can see from figs. 2 to 4 that for the shorter kernels, the default configuration for Wasmer often outperforms the version with flags recommended for production. When the overall execution time extends, Wasmer with `--llvm --native` takes over Wasmers spot, because

the startup time becomes less relevant compared to the total execution time.

As can be seen in fig. 3, Docker sometimes has a significantly lower execution time than the others. We are not exactly sure what causes these inverted spikes, but the most likely explanation is that as the overall execution time of the program increases, the startup time is less and less relevant; therefore the execution time will more closely resemble the execution time of native C.

A fair point in favor of using Docker containers would be to always keep the container running and have it wait for jobs, thus eliminating the startup time, except the first startup of course. This would certainly change the variability in our findings, however, having many Docker containers running at all times would need a lot of memory; something that most IoT devices have a short supply of. In the end, it simply depends on the architecture and system that is being built.

To give a concrete example of when WasmRT would make sense, consider a network camera that should save a recording of any movement it sees. This feature would be needed on all network cameras no matter the model, the image processing or recording could certainly be done by a module compiled to Wasm. The recording would need to start immediately and it would most likely be infrequent, or with at least with irregular intervals. It would then make sense to have this module compiled to Wasm for portability to all NC models and also fast enough that the client will be satisfied.

The performance numbers procured from benchmarking are a good indication that Wasm is a contender in certain context. Unfortunately, they do not show the whole picture. Since Polybench/C is considered a synthetic workload by some [7], an argument could be made that our study is not applicable for real life applications. We believe that, even though PolyBench is not universally thought of as a good benchmark , it still shows the potential of Wasm. It is very hard to create a benchmark or suite of benchmarks, that has everything covered.

Unfortunately, we did not get a chance to perform the benchmarks on the network cameras provided by Axis Communications due to time constraints, and incompatability between the public toolchain and Wasm. The internal toolchain would have to be modified to be compatible with Wasm for usage on their operating system.

CONCLUSION

Combining our results with the findings from our literature study, we see that Wasm could be a viable contender on the edge as a container. The Wasm System Interface (WASI) need to mature further before the Wasm/WASI ecology is able to stand on it's own; the fact that WASI does not have support for communication over sockets yet could be crippling for the usefulness, depending on application and usage. Although networking, and other parts of what should be in WASI to consider it standalone, it has many people actively developing[8]it at the time of writing.

[8] github.com/WebAssembly/WASI

The max memory usage of the docker container was consistently 46 MB. This leads us to believe that Docker seems to allocate a few pages memory at once to have overhead reducing the need to allocate memory on the fly, while WasmRTs allocates memory on the fly, resulting in higher highest but also lower lowest. This supports the hypothesis of WasmRT being better at short-term jobs where the memory usage is a lot more fine tuned to the job and the allocation has higher constraints, whereas Docker seems to allocated larger chunks, presumable because it is faster, and memory usage usually is not an issue for most systems that uses Docker.

For companies such as Axis, they absolutely can integrate Wasm into their development process; especially for code that is not performance-critical. Allowing for greater portability of the code, and for a wider array of developers to write code in a language they feel comfortable with should provide beneficial. However, Wasm+WASI is not mature enough at the time of writing to completely replace existing eco-systems. The lack of modules such as threads and networking makes WASI too immature.

Lastly, with the results it is very clear that WasmRT really shines on fast jobs, specifically programs that run for 10 seconds or less. There is a very clear fit for WasmRT for short, burst-like jobs on IoT devices. Combined with the high portability of Wasm, integrating Wasm into existing code bases that will be ran on different hardware and operating systems, makes a lot of sense.

Future work

Future work should look into performances between different WasmRTs more, and include others that we unfortunately could not. It is also interesting how different WasmRTs run when used in existing programming languages such as C or Rust; a path worth exploring.

We compared WasmRTs against C, because of its high baseline performance. However it would be interesting to compare it against others, perhaps slower, languages to see how well WasmRTs stack up against other popular choices.

It would also be interesting to see how Wasm performs on actual IoT hardware, since we only got the opportunity to test it on a RPI. Because we did not have the time to run benchmarks on an actual network camera, with more restricted hardware, it would be interesting to see how the results changes.

Finally, qualitative research on developing with Wasm would be very interesting, as stated earlier in the paper, metrics such as developing speed and satisfaction was not taken into account in our research. While Wasm and WasmRT has great use cases in theory, the technology has to be usable in production grade systems, where metrics such as developing speed is very important.

REFERENCES

[1] Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. PolyBench/Python: Benchmarking Python Environments with Polyhedral Optimizations *(CC 2021)*. Association for Computing Machinery, New

York, NY, USA, 59–70. DOI: http://dx.doi.org/10.1145/3446804.3446842

[2] Xing Fan, Rui Feng, Oliver Sinnen, and Nasser Giacaman. 2016. Evaluating OpenMP Implementations for Java Using PolyBench. In *OpenMP: Memory, Devices, and Tasks*, Naoya Maruyama, Bronis R. de Supinski, and Mohamed Wahib (Eds.). Springer International Publishing, Cham, 309–319.

[3] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 123–135. DOI: http://dx.doi.org/10.1145/3361525.3361541

[4] Adam Hall and Umakishore Ramachandran. 2019. An Execution Model for Serverless Functions at the Edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI '19)*. Association for Computing Machinery, New York, NY, USA, 225–236. DOI: http://dx.doi.org/10.1145/3302505.3310084

[5] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. 2018. WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices. *University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2* (2018).

[6] Johirul Islam, Erkki Harjula, Tanesh Kumar, Pekka Karhula, and Mika Ylianttila. 2019. Docker Enabled Virtualized Nanoservices for Local IoT Edge Networks. In *2019 IEEE Conference on Standards for Communications and Networking (CSCN)*. 1–7. DOI: http://dx.doi.org/10.1109/CSCN.2019.8931321

[7] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 107–120. https://www.usenix.org/conference/atc19/presentation/jangda

[8] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 272–283. DOI: http://dx.doi.org/10.1109/HPCA.2014.6835938

[9] Samson Lee, John Leaney, Tim O'Neill, and Mark Hunter. 2005. Performance benchmark of a parallel and distributed network simulator. In *Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*. IEEE, 101–108.

[10] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *J. Parallel and Distrib. Comput.* 70, 12 (2010), 1204–1219.

[11] Pankaj Mendki. 2018. Docker container based analytics at IoT edge Video analytics usecase. In *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*. 1–4. DOI: http://dx.doi.org/10.1109/IoT-SIU.2018.8519852

[12] Roberto Morabito. 2016. A performance evaluation of container technologies on Internet of Things devices. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 999–1000. DOI: http://dx.doi.org/10.1109/INFCOMW.2016.7562228

[13] Roberto Morabito and Nicklas Beijar. 2016. Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies. In *2016 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops)*. 1–6. DOI: http://dx.doi.org/10.1109/SECONW.2016.7746807

[14] Roberto Morabito, Ivan Farris, Antonio Iera, and Tarik Taleb. 2017. Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge. *IEEE Internet of Things Journal* PP (06 2017), 1–1. DOI:http://dx.doi.org/10.1109/JIOT.2017.2714638

[15] Seán Murphy, Leonardas Persaud, William Martini, and Bill Bosshard. 2020. On the Use of Web Assembly in a Serverless Context. In *Agile Processes in Software Engineering and Extreme Programming – Workshops*, Maria Paasivaara and Philippe Kruchten (Eds.). Springer International Publishing, Cham, 141–145.

[16] Jonah Napieralla. 2020. *Considering WebAssembly Containers for Edge Computing on Hardware-Constrained IoT Devices*. Ph.D. Dissertation. http://urn.kb.se/resolve?urn=urn:nbn:se:bth-20112

[17] Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth B. Kent. 2020. Insights into WebAssembly: Compilation Performance and Shared Code Caching in Node.Js. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (CASCON '20)*. IBM Corp., USA, 163–172.

[18] Louis-Noël Pouchet and others. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012).

[19] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. 2018. Bringing the Web up to Speed with WebAssembly. *Commun. ACM* 61, 12 (Nov. 2018), 107–115. DOI: http://dx.doi.org/10.1145/3282510

[20] Cristian Constantin Spoiala, Alin Calinciuc, Corneliu Octavian Turcu, and Constantin Filote. 2016. Performance comparison of a WebRTC server on Docker versus virtual machine. In *2016 International Conference on Development and Application Systems (DAS)*. 295–298. DOI: http://dx.doi.org/10.1109/DAAS.2016.7492590

[21] Claudius Sundlöf. 2018. Improving performance of sequential code through automatic parallelization. (2018).

[22] Tomofumi Yuki and Louis-Noël Pouchet. 2015. Polybench 4.0. (2015).