Using WebAssembly to make Serverless Applications more Portable

Report

Badrie Persaud
Zurich, Switzerland
18-796-557
badrieleonardas.persaud@uzh

Bill Bosshard
Zurich, Switzerland
12-933-255
bill.bosshard@uzh.ch

William Martini
Dürnten, Switzerland
13-765-334
william.martini@uzh.ch

Fall Semester 2019
Software Maintenance and Evolution
Department of Informatics
University of Zurich
Supervisor: Dr. Sebastiano Panichella
Advisor: Dr Seán Murphy

Table of contents

1 Introduction

WebAssembly (WASM) is a relatively new system for packaging software which can run in very diverse contexts, originally developed for a browser context, there is increasing interest in running it WASM binaries in a server context.

There are some reasons to believe that WASM is a good candidate for The Ultimate Portable Application Binary: many languages can compile down to a WASM binary, JavaScript runtimes are mature, stable and well optimized and offer powerful interfaces and abstractions of key Operating System concepts.

In concert with WASM development, interest in serverless technology [1] is increasing rapidly, providing an (arguably) new approach to software design. Serverless computing is an event-driven, stateless paradigm that supports loose coupling of platform services (eg database services, file storage services) delivering a new approach to application design. At present, most serverless platforms execute some variant of Docker containers, but there is interest in considering alternative runtimes for serverless: WASM is a candidate for consideration in this context and indeed Cloudflare is already offering it as one way of packaging applications for their platform [2].

1.1 Goal

The objective of this project is to determine **how WASM can be used in a serverless context and can be deployed on multiple cloud platforms**. In order to recommend a best practice on how to achieve this objective. We try to achieve this goal by getting an overview of the existing runtimes and approaches and compare possible advantages or disadvantages of the approaches.

2 Technology overview

First of all, we will give a short introduction in the following section to the most important terms in this project.

2.1 Serverless technologies

Serverless computing is a cloud-computing execution and event-driven model that gives fine-grain control of resources. The cloud provider dynamically manages the allocation of machine resources and the pricing is based on the actual amount of resources used by an application, in contrast to prepurchased units of capacity [3]. The serverless model relied on functions, where developers break down their applications into small stateless parts, which means it can be executed without any context regarding the underlying server. This is often referred to as functions-as-a-service [4].

The main advantage of serverless technologies is reduced costs. As you can see in Figure 1, you only pay for the resources you need, where in for traditional server environment you have to buy new hardware if more performance. Figure 1 shows how this leads to costs, that can be saved if serverless is used instead. This especially matters if the server doesn't have a steady load, and instead have peak times.

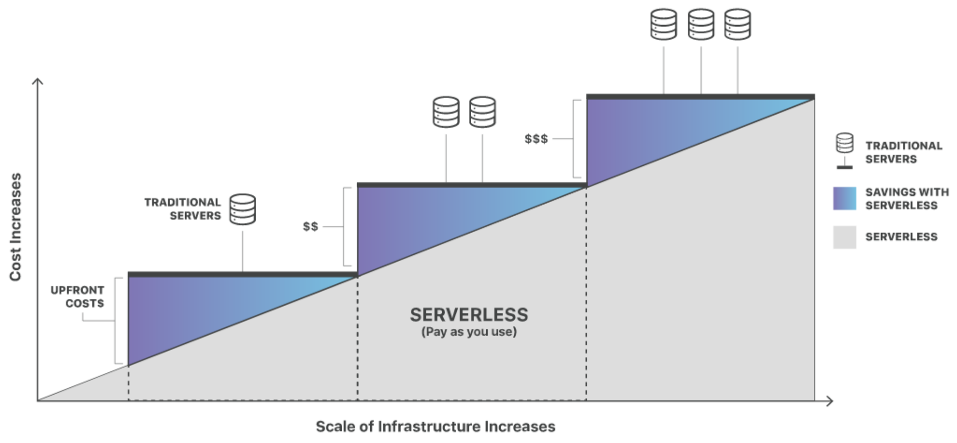
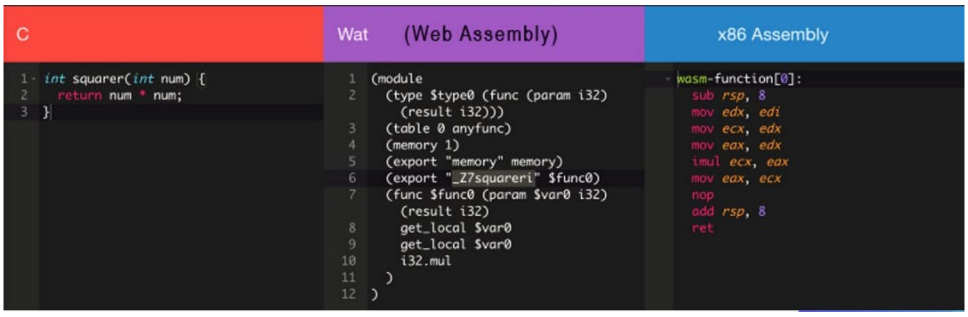


Figure 1 Cost benefits of serverless [5]

2.2 WebAssembly (WASM)

WASM is an instruction format for a stack-based virtual machine, first introduced as an MVP in March 2017. It has a binary format and is designed as a portable target for compilation of high-level languages, for example C or Rust. WASM enables deployment of code from those high-level languages on the web for client or web applications [6]. It is supported on all major browsers (Edge, Firefox, Chrome, Safari) and is designed quite secure with modules being executed in a sandboxed environment separated from the host runtime using fault isolation techniques.



```c
int squarer(int num) {
  return num * num;
}
```

```
(module
  (type $type0 (func (param i32)
    (result i32)))
  (table 0 anyfunc)
  (memory 1)
  (export "memory" memory)
  (export "_Z7squareri" $func0)
  (func $func0 (param $var0 i32)
    (result i32)
    get_local $var0
    get_local $var0
    i32.mul
  )
)
```

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

Figure 2 Comparison of C code, WebAssembly wat and x86 Assembly

2.3 WebAssembly System Interface (WASI)

WASI is the WebAssembly System Interface, it extends the WASM characteristic sandboxing to include I/O. It’s an API created by Mozella’s Wasmtime project and provides access to operating-system-like features, including filesystem interactions, Berkeley sockets, clocks, and random numbers. It tries to be independent of browsers, so it doesn’t depend on JavaScript or Web APIs, therefore it is not limited by the need to be compatible with JavaScript [7].

3 Comparison of different WASM runtimes options

In order to get insides and an overview of the existing runtimes, we decided to run benchmarks. There are many WASM runtimes available at least 26 of them [8]. We decided to reduce the number of runtimes for this project and decided to go with the three most popular and most advanced ones (among others due to the number of contributors and commits). These are also the runtimes that are continuously developed and frequently updated. The choice of runtimes has proven to be correct during the project but there are also other runtimes that could have a big potential. In the following section, we will first go into each runtime to show the differences of the runtimes.

3.1 Wasmer

Wasmer is a standalone WebAssembly runtime for running WebAssembly outside of the browser, supporting WASI and Emscripten. It is an open-source project from a startup from California. It is actively developed. Additional to the runtime, they work on libraries in many different languages like python or go to support integrating WASM files into them [9].

3.2 Wasmtime

Wasmtime is a standalone WASM-only optimizing runtime for WebAssembly and WASI. Like Wasmer, it can run code outside of the Web. It is built with Cranelift JIT. Further, it is the runtime of the Bytecode Alliance which consists of companies like Mozilla, Fastly, Intel, and Red Hat, therefore it has great support and is a very promising project [10].

3.3 Lucet

Lucet is a native WebAssembly compiler and runtime. It is designed to safely execute untrusted WebAssembly programs inside an application and was created by Fastly which is an American cloud computing services provider. It was announced on 28th March 2019 but Fastly has been working on this project behind the scenes since 2017. Lucet is designed to take WebAssembly beyond the browser, and build a platform for faster, safer execution on Fastly’s edge cloud. So, they mainly designed it to use it with their products. Even though it was created by a company it is open source. Lucet is built on top of the Cranelift code generator and supports the WebAssembly System Interface (WASI) [11].

Lucet differs fundamentally from Wasmer and Wasmtime since it requires the code to be compiled to WebAssembly and then to native x86-64 shared object files using the Lucetc compiler [12]. Only after this additional step, the shared object can then be run with the Lucet runtime. Besides this difference, there are other points in which the runtimes differ. Some of them are listed in Table 1. One point all runtimes have in common is that they all support WASI.

Runtime	**Written in**	**Compilation**	**Interoperability Support**	**Contributors**	**Platforms supported**
Wasmer	Rust/C++	JIT	Python, Go, PHP, Ruby, C#/.Net, R, Swift	49	Linux macOS Windows
Wasmtime	C++	JIT	Python	80	Linux macOS Windows
Lucet	Rust	AOT	-	19	Linux

Table 1 Differences between the runtimes based on [8]

3.4 Benchmark

To get a better inside in runtimes and to decide which runtime is the best-suited one for this project, we did benchmarks to see which one performs the best. We ran each test three times for each tool and used *time* to get the time needed for the execution. Therefore, we measured not only the time used by the algorithm but also the startup time of the runtime. Overall, we tried to get as much information out of the benchmarks as possible and therefore chose different types of benchmarks. Additionally, we did one benchmark not only in C but also in Rust. At this point in the project, it wasn't clear which language we will use to compile the WASM executable and with the different programming languages, we were able to get deeper insights into the runtimes. For the comparison, we used the average time calculated from the three runs for each runtime. To keep the results as comparable as possible we ran the benchmark on the same virtual machine in one session one runtime after the other.

3.4.1 C

We compiled different benchmarks in C to WASM with *wasm32-wasi-clang* and then ran it on the different runtime tools. For Lucet we needed a shared object to run the benchmarks. As already explained, we needed a shared object to run the Lucet runtime on. Therefore, an additional step was necessary and we used *lucetc-wasi* to get the shared object based on the C Code. Then for running the benchmark with *time*, we used *luect-wasi* on the shared object. Besides testing the runtime, we always ran the benchmark in native C and compiled it with *gcc*. We used for all runtimes and native C the *-Ofast* optimization.

Prime

The first benchmark we ran is a prime number one. It checks if a large integer prime number, in this case, 4294967029, is indeed a prime number.

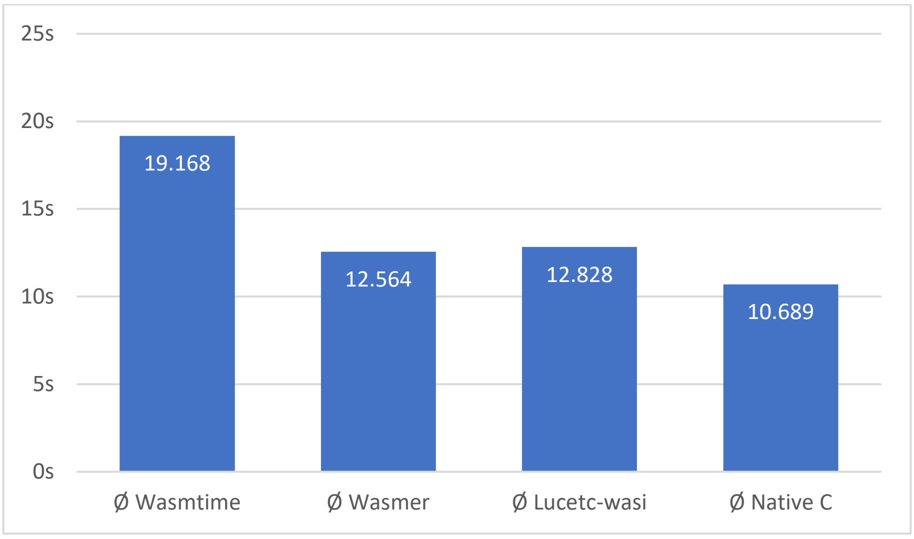


Figure 3 Results of the prime benchmark

As can be seen in Figure 3, we observed that Wasmer and Lucet are running about 7 seconds faster than Wasmtime. Lucet, Wasmer and native C are very close to each other but native is still the fastest one.

Fibonacci

For this benchmark, we used the calculation of the Fibonacci sequence. To test the difference between lighter and heavier workloads, we calculated the 42nd and the 52nd Fibonacci number.

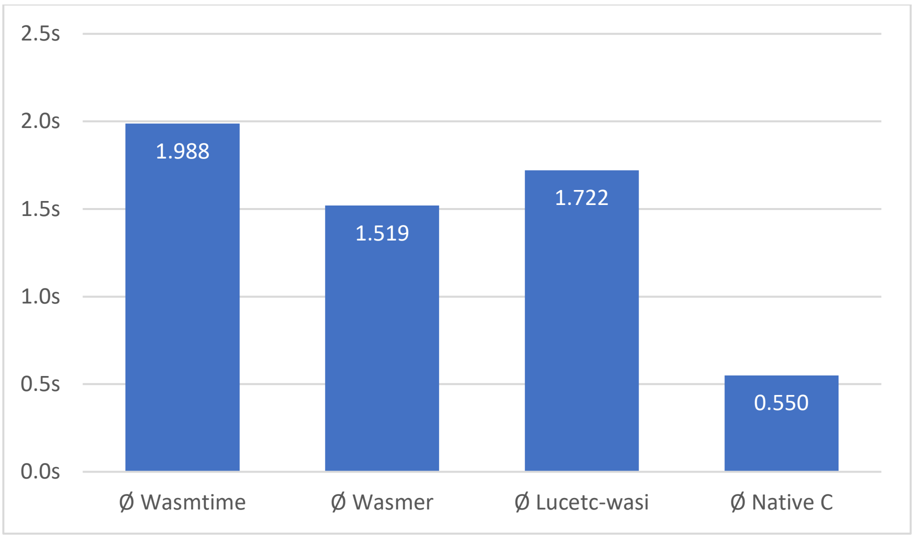


Figure 4 Results of the Fibonacci benchmark with n=42

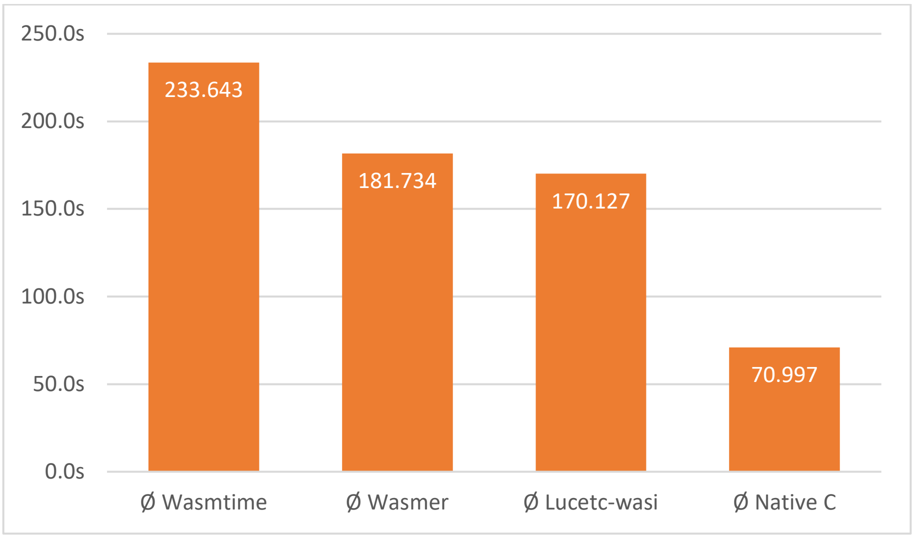


Figure 5 Result of the Fibonacci benchmark with n=52

In Figure 4 it can be observed that for the 42nd Fibonacci number Wasmer is the fastest with 1.5 seconds followed by Lucet with 1.7s and Wasmtime with 1.9 seconds. But for n = 52 (Figure 5) Lucet (2m50s) is about 10 seconds faster than Wasmer (3m) and more than a minute faster than Wasmtime (3m53s). Therefore, we suspect that Lucet performs better on bigger workloads and Wasmer is the better choice for small workloads.

Bubble Sort

For testing our theses that Lucet performs better for heavier workloads, we used the Bubble Sort algorithm to sort different long arrays. This is the only benchmark, that doesn't run with *time*. We used the *clock_t* of the *time.h* library inside the C program and measured therefore only the time was needed by the sorting function to sort the array. Like the other benchmarks, we ran it three times for each runtime and calculated the average. We sorted arrays of length 1’000, 5’000, 10’000, 50’000, 75’000, 100’000, 250’000 and 500’000. The arrays are filled with random numbers which are all created with the same seed.

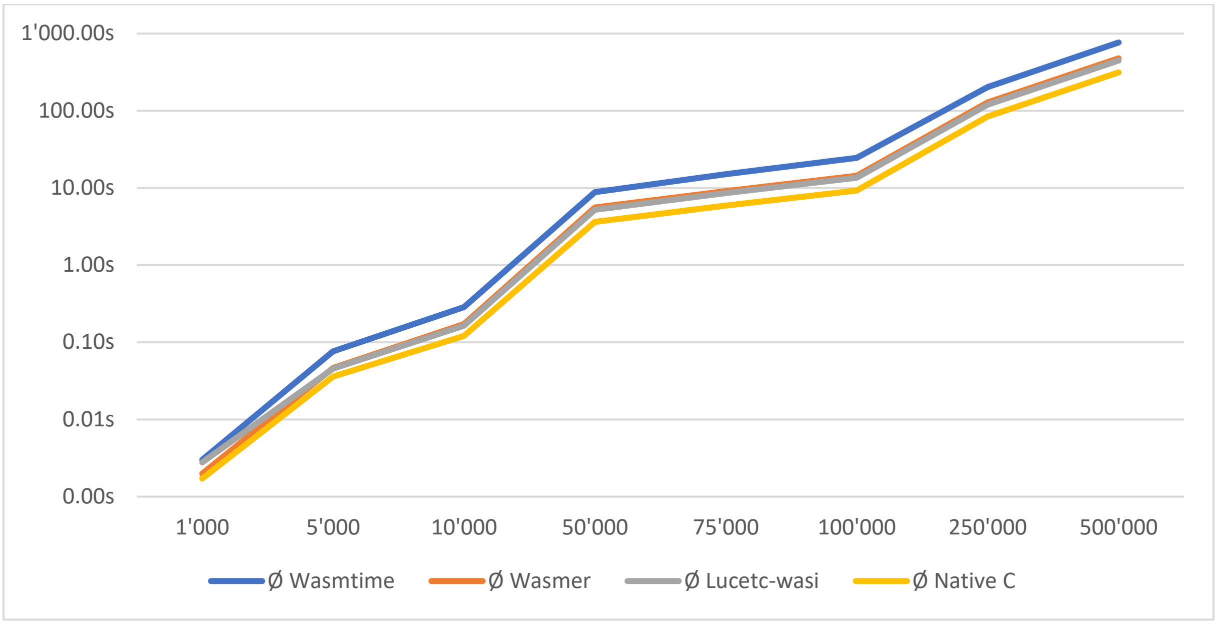


Figure 6 Results of the Bubble Sort benchmark with a log scale

As you can see in Figure 6 Wasmtime was slower than all other toolchain and native C except for the smallest array. Wasmer and Lucet performed significantly faster but not as fast as native C. Wasmer was faster when using smaller arrays but Lucet gained speed when increasing the length of the array. For the longest tested array, Lucet was half a minute faster than Wasmer. The startup time can't be the reason for this behavior because only the execution time of the function was measured which is not affected by the startup time.

Multiplication of square matrices

The last benchmark we ran based on C code is the multiplication of square matrices. The program multiplies two square matrices of size 800 to 800. These matrices are filled by random integers between 1 and 100. To ensure the comparability we used for the generation the same seed for all runs. This benchmark was again measured by the *time* command.

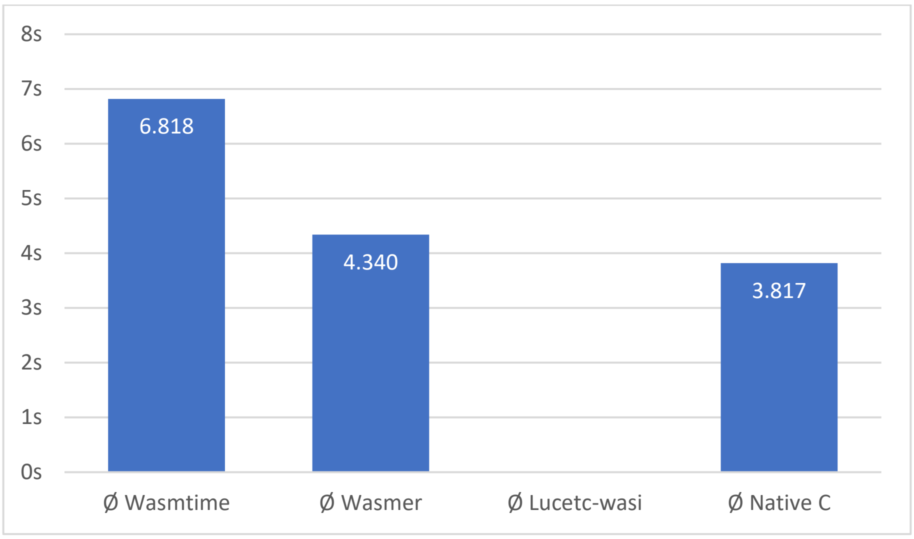


Figure 7 Results of the matrix multiplication benchmark

Wasmertime was again the slowest one by 2.5 seconds as can be seen in Figure 7. Wasmer and native C were quite close together, but the benchmark didn't work for Lucet because the main thread panicked. We tried to solve this issue, but this is a perfect example of the biggest challenge, we had during the project. Because WASM is such a young technology there aren’t many guidelines or for example Stack Overflow posts. This and missing documentation make debugging quite difficult. Also, we weren’t able to fix the problem because the only information Lucet gave us, is that the main thread panicked and not why it panicked. Investigating the code, we could not find the cause of the error.

C conclusions

From the Fibonacci benchmark, it seems that Lucet is faster for larger calculations being significantly faster than Wasmer and Wasmtime. But for smaller workloads, Wasmer seems to be a bit faster. This assumption was confirmed by the Bubble Sort benchmark. Lucet performed much better for heavy workloads. One reason for this behavior could be that Lucet has a longer startup time but this shouldn’t influence the Bubble Sort benchmark. So, the reason for this behavior must be something different but that needs further investigation. For C code in general, Wasmtime appears to be the slowest one. Lucet and Wasmer were quite close to each other depending on the workload.

3.4.2 Rust

After we ran the benchmarks based on C code, we decided to test in another programming language. This gives us a better overview and more information to decide which runtime we want to use and also in combination with which programming language. Therefore, we compiled some rust code to WASM. The procedure was the same as with the C code benchmarks. We ran it three times with each runtime tools and used *time* to get the used time of the program. For compiling the Rust code, we used *cargo build* with the *wasm32-wasi* target.

Fibonacci

We used the Fibonacci benchmark again to test the performance of WASM based on Rust. This gave us the ability to not only compare the different runtimes to each other but also to compare it to the results of the C Fibonacci benchmark. With this comparison, we can identify if the patterns we observed by the C code benchmark are also valid for Rust.

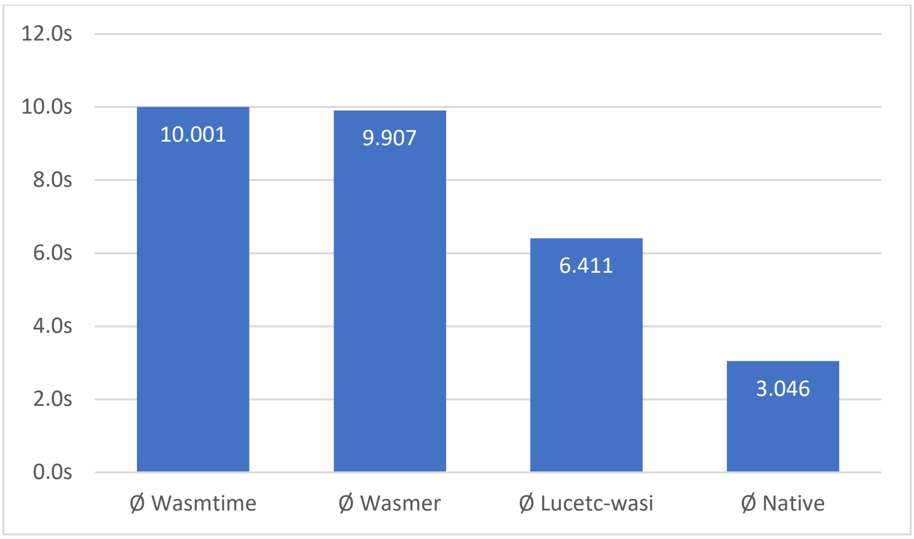


Figure 8 Results of the Fibonacci benchmark with n=42

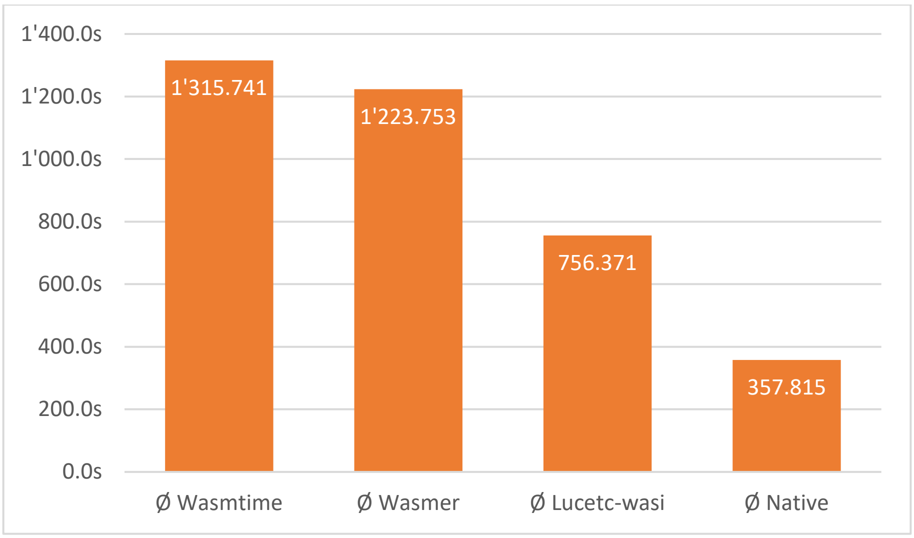


Figure 9 Results of the Fibonacci benchmark with n=52

Unexpected the results of these benchmarks are quite different from those with done in C. Compared to the same Fibonacci calculation, there wasn't a clear winner in C. Lucet performed best for n=52 and Wasmer for n=42. But the similar benchmark in Rust showed clearly that the best performing one native is as it can be seen in Figure 8 for the 42nd Fibonacci number and in Figure 9 for the 52nd. Wasmer and Wasmtime performed quite similar to Wasmer being a little bit better. The clear winner here is Lucet, however still two times slower than native c.

Rust conclusions

Lucet performed the best but it was still slower than native Rust. Wasmer and Wasmtime needed much more time to calculate the Fibonacci numbers. The benchmark showed that the performance depends on the programming language. Additionally, the patterns we observed with C doesn’t show up in Rust. But interestingly, native Rust needed much more time to perform the same calculation than native C. For n=42 and n=52 native C was more than five times faster than native Rust. This affected also the WASM runtimes like Lucet which needed four times less time for the same task with C than with Rust. This difference can be a result of using the optimization option -*Ofast* while compiling C code but there can also be other reasons.

3.5 Benchmarking conclusions

We have seen, that Lucet, in general, performed the best for bigger workloads in C, but fell short in small tasks, we suspect, that it has a longer startup time than the other runtimes. Wasmtime was in almost all cases outperformed by the other runtimes. But Lucet works a bit different than the other runtimes since they don’t use the WASM file directly, instead they transform it first to a shared object file which then can be run with Lucet. Because we wanted to just use WASM files we decided on working with Wasmer, since it outperformed Wasmtime in all benchmarks. Besides the results of the benchmarks, we experienced while working with the runtimes that Wasmer is the best-supported one and has also the bigger community. This makes working with Wasmer much easier than with the other runtimes. Additionally, we decided to use C instead of Rust because as already mentioned, C and C based WASM performed significantly faster than Rust and Rust based WASM. Even the runtimes when using C were faster than native Rust.

4 Deployment of WASM on serverless platforms

In this section, we will give a short overview of how we deployed and ran WASM files on a cloud service. We tried two different ways of deploying and running WASM files on different cloud provider platforms and compared them. The first approach was making use of the ability of JavaScript to load WASM files and use the exposed functions, the second approach was to use Docker container running a WASM runtime with a WASM file to execute the function. The first approach does not use one of the three prior investigated runtimes, therefore we did not use any knowledge that we gained before through the benchmark, but it provides a simple deployment solution for WASM modules. We followed this approach because it seemed very straightforward and portable and could be used as a fallback solution.

Our goal was to deploy the solution on Apache OpenWhisk, Bluemix of IBM Cloud and Lambda on Amazon Web Services (AWS). We chose these platforms to show that the solution is indeed portable at least for AWS Lambda the most used one serverless platform and IBM OpenWhisk as a comparison [13].

4.1 Deployment using JavaScript approach

In Figure 10 you can see the process needed to run a WASM file on a cloud platform. This approach is based on [14]. For our test we took C code and compiled it with Emscripten to a WASM file, doing this, we defined what functions were to be exposed. To run the WASM file, we used the index.js file, which loads the WASM file and is able to run the previously exposed function. To run this on a cloud service we zipped the WASM file together with the index.js and a package.json and created an action with the zip. By invoking the action, the cloud provider runs the JavaScript on the node runtime [14].

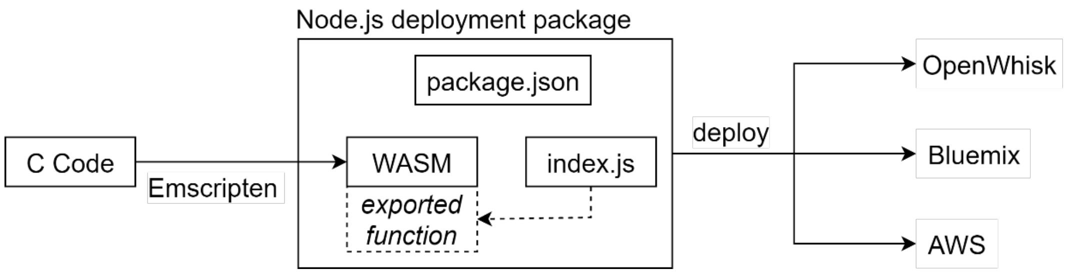


Figure 10 Pipeline with Node.js

4.2 Deployment using Docker approaches

Our second approach was trying to make use of Docker containers. For this approach, we used the openwhisk/dockerskeleton as the base image [15]. For Docker actions to be used you have to expose two endpoints for the container. Which will be called by the OpenWhisk platform. You have to implement an *"/init"* endpoint and a *"/run"* endpoint. The dockerskeleton provides a python flask implementation for these endpoints, which could be adjusted for our needs (we modified this skeleton to include our Wasmer runtime). *Init* allows us to inject an exec script and a post request to *run* executes the *exec* script.

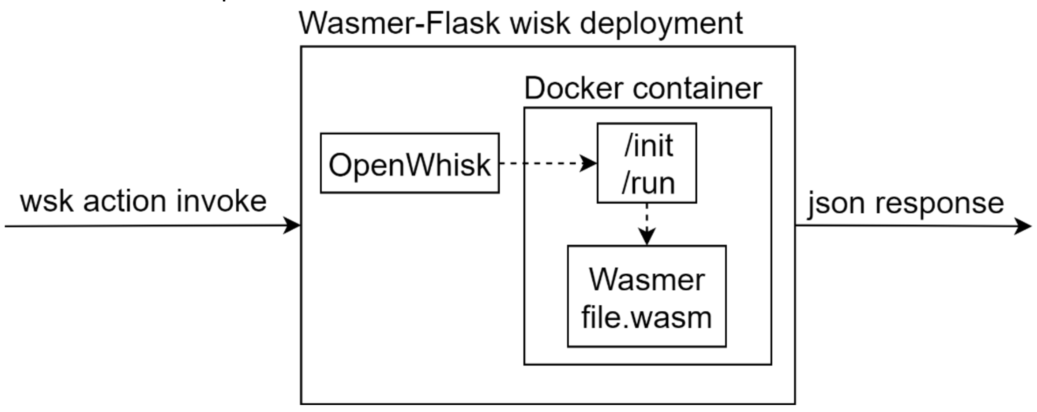


Figure 11 Wasmer-Flask deployment on OpenWhisk

The deployment flow can be seen in Figure 11. After creating an action specifying the dockerskeleton with the WASM and exec file, when we invoke an action, OpenWhisk deploys our Docker container. In the container an actionproxy runs the exec file. The exec file calls Wasmer run on the WASM file, then output is piped into a JSON string and displayed as a server response.

4.3 Comparison of performance of solution using JavaScript and solution using Docker

The biggest advantage of the Node.js approach is that it is very simple and straight forward. This allows a fast build and deployment on all three tested platforms.

We ran fib42 on in a docker image containing the Wasmer binary on OpenWhisk and compared the time with running Node.js on OpenWhisk. Also, for the same fib42 program, we ran it locally on vanilla Wasmer and native c to show the comparison. In Figure 12 you can see how the runtimes compare with the different technology

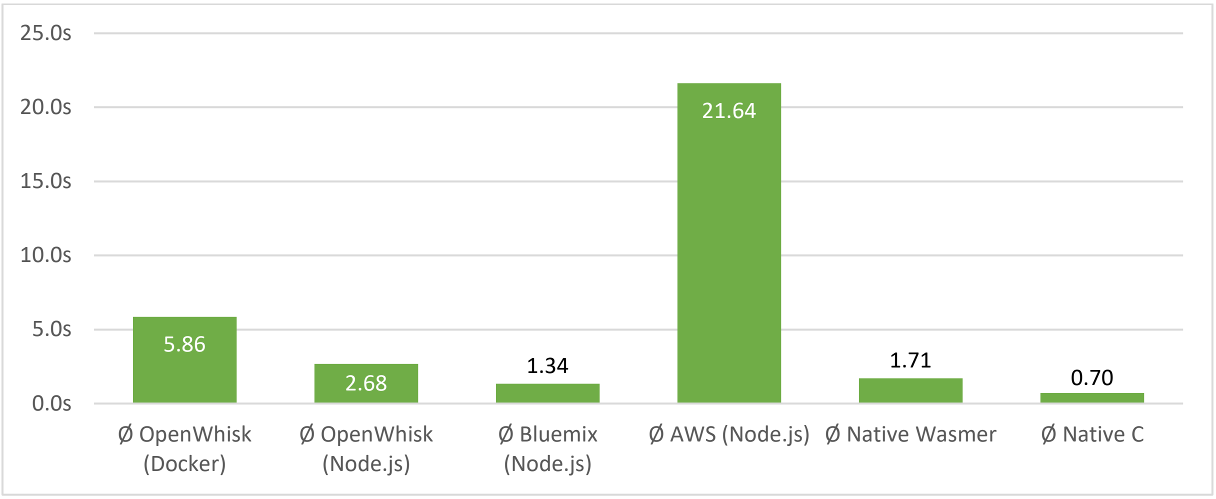


Figure 12 Serverless platforms compared to native

In the Docker solution, the whisk action runs an http post request to the endpoint *localhost:8080/run* which calls Wasmer run with the WASM file. we see that Wasmer in the Docker performs slower than Node.js on OpenWhisk. This may be due to having a heavy 1GB Docker container running on OpenWhisk.

4.4 Issues regarding integration

The Node.js approach currently supports only integers as input for the exported function of the WASM. This is due to that the exported functions only can receive and also return native WASM types, which are currently only integers [14]. But it can be assumed that the native WASM types will be expanded in the future and that will resolve this issue. Another restriction is that the used pipeline only works with Emscripten. At the moment it is the only practical way to export function with WASM [16]. It is possible to use other ways than Emscripten, but this will result in extremely more steps for the same result and is therefore not near to practical [16], [17].

One problem when using Docker is that AWS does not support Docker out of the box for serverless [18]. A workaround is needed to be able to use Docker on AWS Lambda. One way is to hack AWS that it runs Docker [18]. Another way is to use a framework called Serverless Container-aware Architectures (SCAR) [19]. Both ways include some drawbacks like sacrificing fast startup times every Lambda cold start or using much more additional computing power to run the workaround.

Regarding running Docker containers on OpenWhisk, it requires running a container with python compatibility (needed in the case with OpenWhisk) and libc to run binaries (to run Wasmer), this takes up quite some space. In our case, the container totaled 1GB of which is needless bloat. Also, the skeleton Docker provided by OpenWhisk cannot run binaries and needs modifying, then uploading to Docker hub, since OpenWhisk does not support the use of local containers.

5 Project conclusion

We started with the evaluation of three different WASM runtimes with numerous benchmarks. Based on the benchmarks we concluded, that Wasmer would be the best fit in the serverless context and therefore we used Wasmer for the later steps.

We were looking into different ways of making WASM portable and found and tested two different approaches. The first approach did not use any WASM runtime instead it makes use of the ability of JS to import and execute WASM modules. This approach turned out to be very straightforward and portable but also has some limitations on what can be used as an input. The second approach, was more complex to set up, making use of Docker technology to include the Wasmer runtime and file in a container and run the container on the cloud platform. The Docker approach is a great way to make WASM portable but has the major drawback that it isn't supported by AWS (without major hacks). In summary, both ways have their limitations and therefore there isn't a best practice at the moment.

From the benchmarks, we saw that Lucet was able to perform faster by compiling native x86_64 code. The runtimes, however, did not always perform near native speeds. We investigated further with Wasmer to run on serverless platforms and in the current state, the pipeline runs as a hack. However, we were able to make it work, proving the WASM's goal of compiling once and running everywhere is achievable.

Since WASM is such a new technology it is rapidly developing, and we expect that in future there will be even better approaches on how to deploy WASM in a serverless environment. We see a lot of potential in this technology and are certain that big cloud providers will adapt to it too. The work is still valuable since it provides two different ways that are currently working, it shows the current state of the WASM deployment process in a serverless context. It may need to be necessary to revisit aspects of this work since the technology moving forward quickly. Additionally, more tests on other serverless platforms could be done to assure that the proposed ways are indeed portable beside AWS and IBM Cloud. All the produced results, examples and interesting links for further reading are available on GitHub.

6 Bibliography

[1] E. Jonas *et al.*, “Cloud Programming Simplified: A Berkeley View on Serverless Computing,” *ArXiv E-Prints*, vol. 1902, p. arXiv:1902.03383, Feb. 2019.

[2] K. Varda, “WebAssembly on Cloudflare Workers,” *The Cloudflare Blog*, 01-Oct-2018. [Online]. Available: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/. [Accessed: 06-Dec-2019].

[3] R. Miller, “AWS Lambda Makes Serverless Applications A Reality,” *TechCrunch*, 24-Nov-2015. [Online]. Available: http://social.techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/. [Accessed: 11-Dec-2019].

[4] S. Carey, “What is serverless computing and which enterprises are adopting it?,” *Computerworld*, 13-Aug-2019. [Online]. Available: https://www.computerworld.com/article/3427298/what-is-serverless-computing-and-which-enterprises-are-adopting-it.html. [Accessed: 11-Dec-2019].

[5] “benefits of serverless.” [Online]. Available: https://www.cloudflare.com/resources/images/slt3lc6tev37/7nyIgiecrfe9W6TfmJRpNh/dfc5434659e31300d1918d4163dfb263/benefits-of-serverless.svg. [Accessed: 20-Dec-2019].

[6] “WebAssembly.” [Online]. Available: https://webassembly.org/. [Accessed: 11-Dec-2019].

[7] “bytecodealliance/wasmtime,” *GitHub*. [Online]. Available: https://github.com/bytecodealliance/wasmtime. [Accessed: 10-Dec-2019].

[8] S. Akinyemi, “Awesome WebAssembly Runtimes,” 08-Dec-2019. [Online]. Available: https://github.com/appcypher/awesome-wasm-runtimes. [Accessed: 11-Dec-2019].

[9] “Wasmer - The Universal WebAssembly Runtime.” [Online]. Available: https://wasmer.io/. [Accessed: 10-Dec-2019].

[10] L. Clark, “Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly,” *Bytecode Alliance*, 12-Nov-2019. [Online]. Available: https://bytecodealliance.org/articles/announcing-the-bytecode-alliance. [Accessed: 10-Dec-2019].

[11] P. Hickey, “Announcing Lucet: Fastly’s native WebAssembly compiler and runtime,” 28-Mar-2019. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime. [Accessed: 09-Dec-2019].

[12] A. Foltzer, “Lucet’s performance and lifecycle,” 15-May-2019. [Online]. Available: https://www.fastly.com/blog/lucet-performance-and-lifecycle. [Accessed: 23-Dec-2019].

[13] J. Demian, “Top 5 serverless platforms of 2018,” *Dashbird*, 25-Jun-2018. [Online]. Available: https://dashbird.io/blog/serverless-platforms-2018/. [Accessed: 11-Dec-2019].

[14] D. Thomas, “Serverless Functions with WebAssembly Modules - James Thomas,” 06-Aug-2019. [Online]. Available: http://jamesthom.as/blog/2019/08/06/serverless-and-webassembly-modules/. [Accessed: 09-Dec-2019].

[15] “openwhisk/dockerskeleton - Docker Hub.” [Online]. Available: https://hub.docker.com/r/openwhisk/dockerskeleton. [Accessed: 20-Dec-2019].

[16] J. Goldberg, “Notes on working with C and WebAssembly.” [Online]. Available: https://aransentin.github.io/cwasm/. [Accessed: 10-Dec-2019].

[17] Surma, “Compiling C to WebAssembly without Emscripten — DasSur.ma,” 28-May-2019. [Online]. Available: https://dassur.ma/things/c-to-webassembly/. [Accessed: 10-Dec-2019].

[18] V. Holubiev, “How Did I ‘Hack’ AWS Lambda to Run Docker Containers,” 03-Jun-2017. [Online]. Available: https://hackernoon.com/how-did-i-hack-aws-lambda-to-run-docker-containers-7184dc47c09b. [Accessed: 09-Dec-2019].

[19] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, “Serverless computing for container-based architectures,” *Future Gener. Comput. Syst.*, vol. 83, pp. 50–59, Jun. 2018, doi: 10.1016/j.future.2018.01.022.