Performance Comparison of WebAssembly Source Languages and Compilers for Running WebAssembly on the Server

Jan Föcking[1]

Abstract: WebAssembly (Wasm), a portable low-level bytecode format initially designed for targeting Web browsers, is experiencing an increasing adoption in server-side environments. Typically, a WebAssembly binary originates by the compilation of source code from one of many source languages that support Wasm as a compilation target. Unfortunately, insufficient study has been conducted to understand how the choice of a particular source language and compiler impacts the performance of the resulting Wasm binary. In this paper, we compare the performance of Wasm binaries by implementing three different workloads in different source languages, compile them to WebAssembly, and execute them in a standalone Wasm runtime on the server side. We collect several performance metrics and compare them with respect to the different source languages.

Keywords: WebAssembly; WASI; Performance comparison

1 Introduction

WebAssembly (Wasm) is a safe, fast, compact and portable low-level bytecode format. It was initially designed by the four major browser vendors to address restrictions in the standard client-side language JavaScript. Since 2017, all major browsers are supporting this standard. While the initial paper targets the Web as the primary target at first, it also mentions the existence of use cases beyond the Web due to the language-, hardware-, and platform-independence of the bytecode format [Ha17].

Since WebAssembly is an open standard with no Web-specific features or assumptions, the development of standalone runtimes was expected from the beginning [Ha17; Ro19]. In particular, security features (sandboxing and memory-safety), a near-native performance, and characteristics like compactness and portability of the bytecode format make it suitable for use on the server side. However, before the WebAssembly System Interface (WASI) was introduced in 2019, no standardized way was available to perform system calls. The introduction of WASI (including APIs for file I/O access, clocks, random) made it even more attractive to use WebAssembly on the server side [Cl19].

Initially, the focus was on supporting low-level languages, especially C and C++, for targeting the Wasm bytecode format [Ha17]. Today, there are many upcoming use cases

[1] Fachhochschule Münster, Fachbereich Wirtschaftsinformatik, Wirtschaftsinformatik, jf935878@fh-muenster.de

for WebAssembly, also on the server side, and multiple languages and compilers are able to target Wasm. However, no research is available that investigates the impact of different source languages and compilers to the performance of WebAssembly. This paper is, to our knowledge, the first one that evaluates the performance of different Wasm applications compiled from various source languages and executed in a non-Web, standalone Wasm runtime. Our goal is to propose which source languages should be used for building performant, Wasm based applications for execution in server-side environments.

In the following, section 2 describes backgrounds on Wasm in standalone environments and characterizes related work. Section 3 explains the experimental design to evaluate Wasm performance for different compilers, and section 4 presents and discusses the measurement results. Finally, in section 5 we present our conclusions and outline future work.

2 Background

2.1 WebAssembly basics

Despite its name, WebAssembly is not assembly code, but a safe and portable byte code format for efficient execution and compact representation. Although firstly implemented by Web browsers, Wasm's virtual instruction set architecture is designed to be embedded in other non-browser environments as well [Ro19]. According to the WebAssembly specification, a Wasm binary has the form of a so-called module that consists of multiple components such as functions, instructions, values and linear memory [Ro19]. Values can only be one of four numeric types, either integers (32 and 64 bit), or floating-point numbers. Functions organize code by taking values as parameters and returning a sequence of values as results. Functions contain code in the form of a sequence of instructions that are executed in order based on a stack machine. A Wasm module can load and store values by calling the linear memory, a mutable array of raw bytes, which can grow dynamically. WebAssembly is typically embedded into a host environment, typically a browser or a standalone Wasm runtime. A Wasm module can import functions that are provided by the host environment, the so-called host functions. In addition, a module exports arbitrary functions, which can then be called by the host environment.

For security reasons, WebAssembly does not provide any access to the host environment by default [Ro19]. Consequently, access to resources (like I/O or other operating system calls) can only be achieved if the host environment provides dedicated host functions for these purposes. This sandboxing concept enables the host environment with complete control about the capabilities it offers for Wasm modules. An approach to standardize the set of system-oriented host functions, which can be imported into a Wasm module, is WASI, a runtime-independent, non-Web, and system-oriented API for WebAssembly [Cl19].

Although initially seen as a replacement for JavaScript in the Web browser, Wasm is becoming increasingly popular for non-Browser use cases. Spies et al. [SM21] provide

an overview of some of those use cases, including TruffleWasm, an implementation of WebAssembly running on GraalVm that enables Wasm for use in polyglot programming. Another use case referenced by Spies et al. [SM21] is the execution of Wasm-based Smart Contracts on the Ethereum Blockchain, also known as Ethereum flavoured Wasm. Additionally, WebAssembly is especially suitable for applications requiring low-latency response, or hardware platforms with limited resources, for example in edge computing environments [HR19].

2.2 Runtimes and compilers

Multiple runtimes are available to execute WebAssembly workloads in non-Web environments. Wasmtime is a standalone runtime for WebAssembly and WASI maintained by the Bytecode Alliance. Other well-known standalone Wasm runtimes with WASI support are Wasmer, Wasm3 and WasmEdge.[2]

Support for compiling to WebAssembly is available in many programming languages, compiler toolchains and frameworks. In general, two different strategies can be identified. First, many compiler toolchains compile source code into native Wasm binaries. Second, for some other languages, their language runtime/interpreter is compiled into a Wasm binary in order to interpret ordinary programs written in that language at runtime. Some compiler toolchains are not suitable for compilation to WebAssembly if the resulting binaries are to run in a non-Web, standalone runtime. This is due to the fact that some toolchains assume the resulting Wasm binaries to be executed in a browser environment; these toolchains produce JavaScript glue code and make the resulting Wasm binaries depend on that code. Tab. 1 provides a classification of some well-known compiler toolchains[3].

2.3 Related work

Our work is closely related to other WebAssembly performance measurements. There have been prior studies evaluating the performance of WebAssembly in browser environments. Haas et al. [Ha17] compare the performance of WebAssembly to asm.js and native code by using PolyBenchC, a benchmark suite written in C. Yan et al. [Ya21] provide an understanding regarding the performance of Wasm applications and the impact of different optimization levels of a C-to-WebAssembly-compiler and give an overview of performance measurements prior to their work; this includes comparisons based on common C benchmark suites, PolyBenchC and the SPEC CPU suite, and an evaluation of the performance of WebAssembly based on a Sparse matrix-vector multiplication workload.

There has also been research regarding WebAssembly's performance in non-Web environments. Spies et al. [SM21] measure execution time, startup time, and resulting binary

[2] Overview of Wasm runtimes: https://github.com/appcypher/awesome-wasm-runtimes
[3] Overview of Wasm compilers: https://github.com/appcypher/awesome-wasm-langs

Compiler toolchain	Source language	Compilation strategy	Non-browser support	WASI support
AssemblyScript	TypeScript	Native Wasm binary	Yes	Yes
Blazor Wasm	C#	Language runtime/ interpreter in Wasm	No	No
CRuby	Ruby	Language runtime/ interpreter in Wasm	Yes	Yes
Emscripten	C/C++	Native Wasm binary	Yes	Partial
Go compiler	Go	Native Wasm binary	No	No
Pyodide	Python	Language runtime/ interpreter in Wasm	No	No
Rust compiler	Rust	Native Wasm binary	Yes	Yes
TinyGo	Go	Native Wasm binary	Yes	Yes
WASI SDK	C/C++	Native Wasm binary	Yes	Yes

Tab. 1: Classification of well-known compiler toolchains

size in different runtimes by using the PolyBenchC benchmark suite and the Clang and Emscripten compilers. There are a few more specialized studies evaluating the performance of WebAssembly on the ARM architecture [Me20], or in edge or IoT environments [HR19].

3 Experimental design

In this section, we outline our experimental setup that is made publicly available on GitHub[4]. In general, we performed our measurements in two phases that are shown in Fig. 1. First, we compiled the source code of our workload implementations into WebAssembly binaries. Afterwards, we executed the compiled binaries in Wasmtime (version 0.37), a standalone Wasm runtime, and collected several metrics about the runtime behavior of each binary.

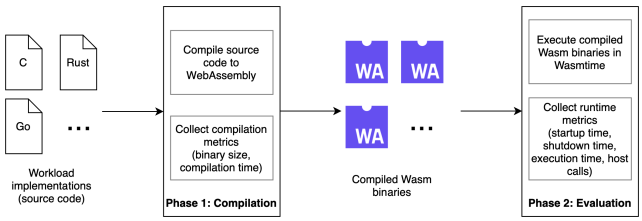

Fig. 1: Overview of the measurement process

As the goal of our measurements is to compare the performance with respect to different source languages and compilers, we used the following compilers from Tab. 1 that support non-browser runtimes and WASI: AssemblyScript (version 0.20.6), CRuby (based on version

[4] https://github.com/jfoe98/comparison-of-wasm-source-languages-for-running-on-the-server

3.2.0 Preview 1), Emscripten (version 3.1.13), Rust compiler (version 1.61.0), TinyGo (version 0.23.0), and WASI SDK (version 16.0 + Clang version 14.0.4). For each compiler setup, we compiled each workload twice, without and with full compiler optimizations enabled, to compare the efficiency of the optimizations.

3.1 Metrics

The first class of metrics, the development metrics, allow us to come to conclusions about the development process. Therefore, the compilation time and the resulting binary size were measured during the compilation process for each source language.

Second, we collected metrics about the runtime behavior of the compiled Wasm binaries. The main metric regarding the performance is the time taken for execution of the Wasm binary. However, since WebAssembly is often used for implementing small and ephemeral applications (like serverless functions in the cloud), the startup (e. g. the time for VM startup, code validation and module initialization) and shutdown time are important and were therefore included in our measurements. Additionally, we captured the amount and kind of WASI system calls to come to conclusions about the efficiency of the I/O implementations of different source languages and compilers.

3.2 Workload selection

To compare the source languages concerning various performance aspects, we implemented three programs, one stack-intensive, one compute-intensive, and one I/O-file-intensive application.

The first application, our stack-intensive program, is a recursive implementation of the Fibonacci sequence. For the compute-intensive workload, we implemented an iterative version of Fibonacci in all source languages. The third workload is intended to compare the I/O capabilities of the different source languages and compilers. As input, a file, containing random numbers from 1 to 100, is provided. The workload consists of reading the numbers from the given input file line by line and appending each number to another file based on the last digit of the number. If the file does not exist, it has to be created first. The output is a set of files, where each file contains all numbers from the given input file ending with the same digit.

We implemented each workload in each of the previously considered source languages. To establish comparability between the implementations, we backed our implementations on the examples from Rosetta Code [5]; for example, our I/O-intensive workload is based on two tasks, `Read a file line by line` and `Append a record to the end of a text file`.

[5] Rosetta Code: http://www.rosettacode.org/wiki/Category:Programming_Tasks

Since the input size of a program affects the amount of performed calculations, we used three input sizes for each workload: Small, Medium, and Large. Tab. 2 depicts the workloads' inputs for each size.

Workload Name	Input Small	Input Medium	Input Large
Fibonacci recursive (`fib-rec`)	5	30	50
Fibonacci iterative (`fib-it`)	5	50	90
Filesplit (`filesplit`)	1,000 lines	10,000 lines	100,000 lines

Tab. 2: Workload sizes for all workloads

3.3 Evaluation setup

Fig. 2 depicts our evaluation setup. We provisioned two virtual machines (VMs) (Ubuntu 22.04 LTS, 4 GB of RAM, Intel Xeon Gold 6150 CPU with one core and 2.7 GHz base frequency), one dedicated to perform the measurements, and the other one for managing the resulting metrics. We deployed multiple artifacts to the Measurement VM: The bash script `measurement.sh` is intended to start and orchestrate our measurements. As a first step, it triggers the compilation of the uncompiled source code to Wasm for all languages and compiler setups. Afterwards, it starts our evaluation tool in order to execute the previously compiled Wasm modules. The metrics that were collected during the compilation and execution of the Wasm binaries, were sent to a dedicated virtual machine, the Metrics VM, via HTTP. The Metrics VM was running two Docker containers, a Rust-based HTTP server we implemented to manage metrics, and a relational database to persist them. We separated out managing metrics into a dedicated VM to minimize overhead and to avoid disruptive impacts to our measurements.

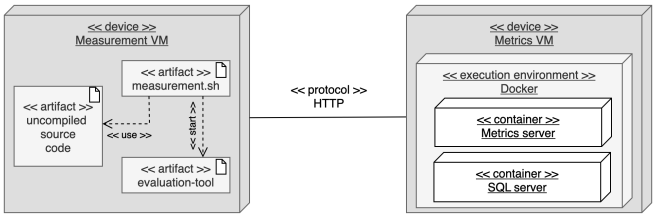

Fig. 2: Overview of the evaluation setup

3.4 Implementation details

3.4.1 Compiling code from source language to WebAssembly

In general, we identified two different approaches for compilation used by the different languages/compilers. On the one hand, most compilers compile to native Wasm binaries,

which can be executed directly by Wasm runtimes. On the other hand, within this study, Ruby with its CRuby runtime follows another approach, so that the CRuby runtime is compiled to WebAssembly. As a consequence, no compilation was necessary for all Ruby-based workload implementations, because the CRuby language runtime interprets ordinary Ruby files at runtime.

The compilation process was automated for each source language by using Docker containers. During the compilation process, the used time and the binary size were measured with the Linux standard tools `date` and `stat`. Each compilation was performed 15 times to achieve more significant results.

3.4.2 Evaluating WebAssembly binaries at runtime

In order to run the compiled Wasm modules and compare its runtime behaviors, we implemented our own evaluation tool in Rust based on the Wasm runtime wasmtime [6], and executed each workload 15 times in order to achieve significant results. The wasmtime Embedding API provides the ability to define and implement host functions that can be imported and called from within a Wasm module. We used this mechanism to define two host functions, `startup` and `finish`. The implementations of our Wasm workloads import the previously listed host functions and call them at the beginning and at the end of the execution. This allowed us to measure the time elapsed for the start and initialization of a module (i. e. the startup time), and the time needed for the termination (i. e. the shutdown time). This procedure is only possible for languages compiling into native Wasm binaries; consequently, we were not able to measure the startup and shutdown time for the CRuby approach. Additionally, we implemented Closures and registered them to the so-called `Call Hooks` of Wasmtime's Embedding API. Call hooks are triggered at certain events and we used them to start or stop our measurements when Wasmtime begins or terminates the execution of a module.

Another important aspect in our evaluation is to compare the efficiency of the WASI-based filesystem implementations between the different source languages. To capture the performed WASI calls, we performed a dedicated evaluation run, where we activated trace logging in order to make Wasmtime log all WASI calls. We collected metrics about the amount and types of performed WASI calls by implementing a tool for parsing the resulting log.

4 Results and discussion

In this section, we present and discuss our measurement results. Additional evaluations and plots are available on GitHub[7].

[6] Crate wasmtime: https://docs.rs/wasmtime/0.37.0/wasmtime/

[7] https://github.com/jfoe98/comparison-of-wasm-source-languages-for-running-on-the-server/blob/main/evaluation/Evaluation.ipynb

4.1 Compile time

In our measurements, WASI SDK and Emscripten stood out with compilation times near one second on average. However, TinyGo and AssemblyScript delivered solid results as well, providing compilation results in nearly 6.5 seconds. The Rust compiler took over two minutes on average for compilation, making it the slowest compiler by far. In our measurements, we did not find any significant impact of the different compiler optimization levels on the compile time; all compilers showed the same or even slightly better compilation performances with higher optimization levels enabled.

4.2 Binary size

As shown in Fig. 3, there were significant differences regarding the resulting binary sizes between the compared compilers. While AssemblyScript (mean binary size of 200 bytes) and Emscripten (600 bytes) produced very small binaries for the Fibonacci workloads, especially Rust's compilation process resulted in much larger binaries (1.9 mb). The source code of our I/O-intensive workload (i.e. `filesplit`) contains calls to system interfaces, like the filesystem. In this case, our results show that the binary size increased significantly because of a huge amount of WASI-specific bytecode added to the resulting Wasm binaries by the compilers. Regarding the different compiler optimization levels, TinyGo (binaries smaller by 55%), Emscripten (27%) and AssemblyScript (24%) produced significantly smaller binaries with high optimization levels enabled; Rust and WASI SDK showed no significant impact when varying the compiler optimization levels.

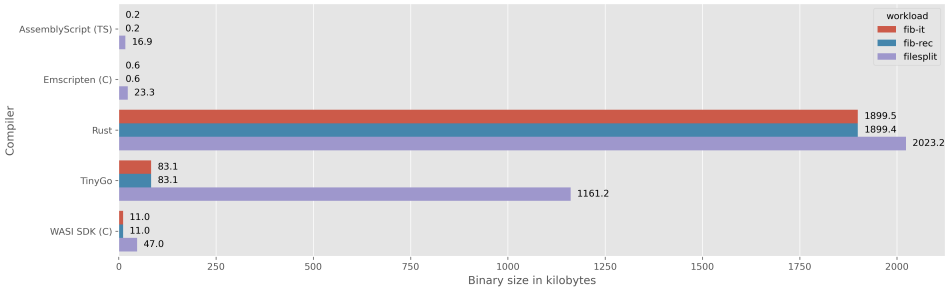

Fig. 3: Comparison of the resulting binary's size of different compilers

4.3 Startup and shutdown time

Fig. 4 shows the results of our measurements regarding the startup and shutdown time. In terms of startup time, the AssemblyScript compiler outperformed all other compilers with a median startup time beneath 1 µs for the Fibonacci workloads and 3 µs regarding the

I/O-intensive workload. Although the Fibonacci modules that were compiled from Go and Rust also delivered a median startup performance lower than 1 µs, the startup time increased massively when looking at the I/O-intensive workload (78 µs for TinyGo and 53 µs for Rust). The WASI SDK performed worst, since even the Fibonacci workloads showed a median startup time greater than 30 µs. An effect we could observe for all compilers is a significant increase of startup time when looking at the Filesplit workload due to larger binary size (as shown in Sect. 4.2) and additional WASI-specific initialization code that is executed during the initialization process of the Wasm module.

In comparison to the previously discussed startup performances, the shutdown of a Wasm module required less time across all compilers. For all compilers, the median shutdown time was under 1 µs for the iterative Fibonacci implementations. Looking at the recursive Fibonacci implementation, we could observe a significant increase in the median shutdown time and in the deviation as well. Due to the recursive behavior, especially for large input sizes, a huge amount of memory is allocated from the stack and needs to be freed when the shutdown occurs. Additionally, an increase of the shutdown time can be seen for the Filesplit workload, since additional system resources have to be released in this case as well.

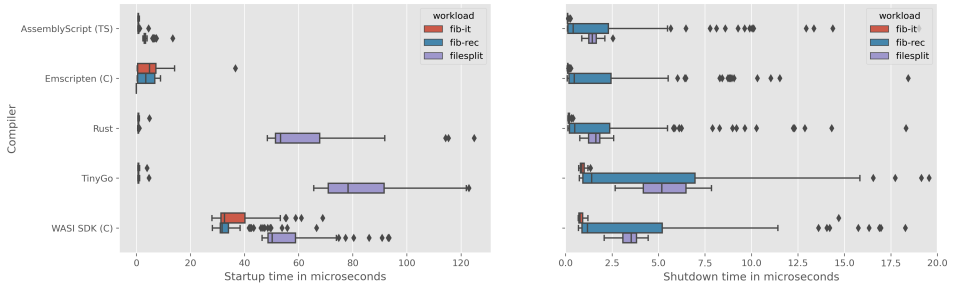

Fig. 4: Comparison of the startup and shutdown time of different compilers

4.4 Execution time

Since we assume high compiler optimizations to be used in production environments, Fig. 5 depicts the execution times we measured regarding the different input sizes for our three workloads that were compiled with full optimizations enabled. As Rust is the most used compiler in real-world use cases, the performance of the Rust-based Wasm binaries was chosen as the baseline and all other results are shown in relation to Rust's results.

Wasm binaries compiled by WASI SDK showed poor performance for workloads (`fib-it`, and `fib-rec` for small inputs) that are executed in the range of microseconds if compiled by other compilers, as WASI SDK's slow startup time (as shown in Sect. 4.3) had a significant effect here. In terms of the `fib-it` workload, the Emscripten-based binaries performed best

for all input sizes, but there were no big differences between the other compilers, except for TinyGo, which took 1.5 times longer. For the `fib-rec` workload, all compilers showed similar results, especially for the large input sizes. With regard to the `filesplit` workload, we identified significant differences between the compilers. While Rust-based binaries delivered the best results, binaries compiled by AssemblyScript and WASI SDK were 1.4 times slower. TinyGo-based binaries showed the poorest execution times by taking up to three times as much time as Rust-based binaries.

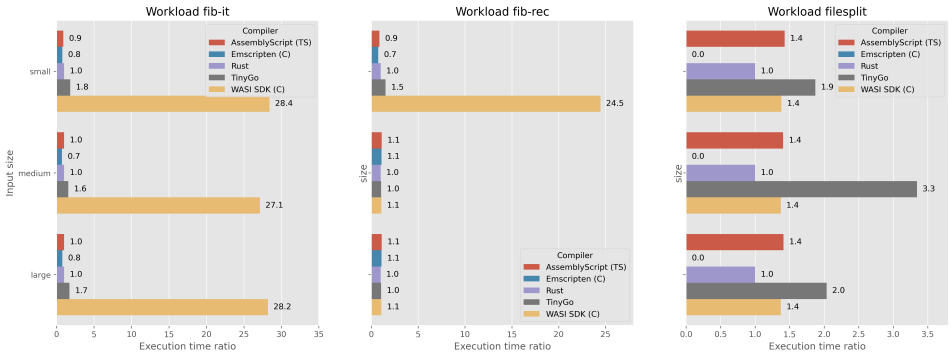

Fig. 5: Comparison of the median execution time for all workloads and input sizes

In Fig. 5, Ruby is not shown because its execution times showed such poor performance that the graph would have been distorted. While Ruby-based executions of the `filesplit` workload were 5 to 17 times slower compared to Rust-based Wasm binaries, the executions of the `fib-rec` workload were 68 times for large input sizes and 16,579 times slower for small input sizes. In median, running the `fib-it` workload took between 18,672 and 19,666 times longer than running the Rust-based binaries for different input sizes.

Depending on the compiler, the optimization level used during compilation had a significant impact on the execution time. On average, the median execution time was across all workloads and input sizes over 6.2 times for Emscripten and 3.9 times for Rust slower when compilation was performed without optimizations. WASI SDK (1.23x slower) and TinyGo (1.2x slower) showed smaller performance losses when deactivating compiler optimizations, while we could not determine any significant differences for AssemblyScript.

4.5 WASI calls

As shown in Tab. 3, the efficiency of the WASI-based implementations of the `filesplit` workload of different compilers differed. While TinyGo, Ruby and WASI SDK made excessive use of the call `fd_fdstat_get`, a call to get attributes of a file descriptor similar to POSIX's `fsync`, binaries compiled by other compilers did not execute this call at all. It is also noticeable that our AssemblyScript implementation called `fd_fdstat_set_flags` to set the append-flag before each write operation. Additionally, while all other implementations

buffered the input file while reading it line by line, binaries compiled with AssemblyScript seemed to be more inefficient and called `fd_read` multiple times for each line. According to the `fd_write` call, Rust-compiled implementations performed two write operations per line.

WASI call	AssemblyScript	Ruby	Rust	TinyGo	WASI SDK
fd_close	100,001	100,002	100,001	100,001	100,001
fd_fdstat_get	0	299,805	0	100,001	300,001
fd_fdstat_set_flags	100,000	1	0	0	0
fd_read	290,901	39	37	73	286
fd_write	100,000	100,003	200,000	100,001	100,000
path_open	100,001	100,050	100,001	100,001	100,001

Tab. 3: Performed WASI calls for the workload filesplit and input size Large

4.6 Reliability

With one exception, all implementations across all compilers always delivered correct results. The exception is the `filesplit` workload in combination with the Emscripten compiler, as it can also be seen in Fig. 5. The reason is that although Emscripten supports WASI in general, reading and writing to/from WASI file descriptors is only implemented for the default file descriptors STDOUT and STDERR at the time of writing[8].

4.7 Discussion

Tab. 4 summarizes our findings on the impact of different source languages and compilers on the performance of compiled Wasm binaries. In our evaluation, AssemblyScript performed well across all criteria and is therefore a good choice in many scenarios. Although Emscripten showed good results as well, it cannot be used for file-based workloads due to a limited WASI support. Due to its large feature set and very good runtime performance results, the use of Rust is recommended if a slow compilation time and a large binary size can be tolerated. TinyGo delivered solid performance results regarding all criteria. While WASI SDK provides a good development experience, its use is not suitable for short-lived, ephemeral tasks that are to be executed in a few microseconds due to the long startup times. The approach of compiling the CRuby language runtime to Wasm to interpret ordinary Ruby code in Wasm proved to be very inefficient and can therefore not be recommended.

5 Conclusions and future work

WebAssembly is experiencing an increasing adoption in server-side environments and more and more source languages and compilers support Wasm as a compilation target. In this

[8] https://github.com/emscripten-core/emscripten/issues/17167

	AssemblyScript	Emscripten	Ruby	Rust	TinyGo	WASI SDK
Compile time	+	++	n/a	--	+	++
Binary size	++	++	n/a	--	-	+
Startup / shutdown time	++	++	n/a	+	+/-	--
Execution time	+	++	--	++	+	-
Reliability	++	+/-	++	++	++	++

Tab. 4: Assessment of compilers regarding the compared performance criteria

paper, we gave an introduction into WebAssembly, presented an overview of use cases and compilers, and referenced related work. Afterwards, we compared the performance of Wasm binaries compiled from different source languages in a standalone, non-Web runtime. Significant differences were found between the compilers with respect to different criteria. We presented our results and made recommendations for the use of the different compilers and source languages.

In our study, we considered six source languages and three different workloads. In future work, the evaluation of additional source languages and compilers, and the measurement of additional, real-world workloads is a promising area.

References

[Cl19] Clark, L.: Standardizing WASI: A system interface to run WebAssembly outside the web, 2019, URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, visited on: 06/20/2022.

[Ha17] Haas, A. et al.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp. 185–200, 2017.

[HR19] Hall, A.; Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. ACM, 2019.

[Me20] Mendki, P.: Evaluating Webassembly Enabled Serverless Approach for Edge Computing. In: 2020 IEEE Cloud Summit. IEEE, 2020.

[Ro19] Rossberg, A.: WebAssembly Core Specification, https://www.w3.org/TR/wasm-core-1/, 2019.

[SM21] Spies, B.; Mock, M.: An Evaluation of WebAssembly in Non-Web Environments. In: 2021 XLVII Latin American Computing Conference (CLEI). IEEE, 2021.

[Ya21] Yan, Y. et al.: Understanding the performance of webassembly applications. In: Proceedings of the 21st ACM Internet Measurement Conference. ACM, 2021.