An Evaluation of WebAssembly in Non-Web Environments

Benedikt Spies
Department of Informatics
Technical University of Munich
Munich, Germany
benedikt.spies@tum.de

Markus Mock
Department of Computer Science
University of Applied Sciences Landshut
Landshut, Germany
mock@haw-landshut.de

***Abstract*—In 2017, WebAssembly, a portable low-level bytecode, was released by the four major web browser makers to address the challenges presented by the maturation of the web and the rise of sophisticated and interactive applications such as 3D visualization, audio, and video streaming, and online games. JavaScript heretofore was the only built-in language of the web, and unfortunately, it is not well outfitted for the rich applications that have come to dominate the web today. Since its initial release, WebAssembly has made great strides on the web. With the WebAssembly System Interface release in March 2018, which allows WebAssembly to communicate with the operating system, it has become possible to run WebAssembly applications outside web browsers. This paper reviews the current state of WebAssembly and its system interface, describes the costs and benefits of these technologies for applications in different environments, and evaluates performance and portability. Our performance measurements demonstrate that WebAssembly is generally faster than JavaScript and, in some cases, can approach native code performance. Despite its limitations which make WebAssembly useless for specific applications domains, it nevertheless, has the potential to be beneficial in many environments and is likely to grow further even outside its original web environment.**

***Index Terms*—WebAssembly, JavaScript, bytecode, asm.js**

I. Introduction

JavaScript has achieved a virtual monopoly as a language for client-side code execution on the web, mainly because it is the only web standard technology running in all major browsers. Initially, JavaScript was not designed as a language for writing complex applications or high-performance algorithms. However, the capabilities of JavaScript improved considerably in the last two decades, enabling substantial applications, for example, games, as well as image and audio processing software. The browser is no longer just a program for displaying documents but has become a platform for rich applications, such as Google Maps.

Due to its portability and security model, JavaScript has since been adopted way beyond its initial context. For instance, Node.js [1] has become one of the most popular server-side application platforms. Unfortunately, JavaScript design restrictions make it generally impossible to execute it at native code performance. That is why the major browser vendors started developing a low-level bytecode called WebAssembly to address the standard client-side web language JavaScript restrictions. Since the release of WebAssembly (or short, WASM) version 1, WASM has gained popularity in web browsers. Like JavaScript, WASM provides a portable and secure foundation that might also expand to other environments. There are already some efforts utilizing WASM in various areas besides the web environment.

This work was performed while B. Spies was still a student at UAS Landshut.

WASM is a safe, portable, low-level bytecode format designed for efficient execution and compact representation. WASM is designed to solve some JavaScript issues, which is necessary to create high-performance web applications in browsers. WASM is a virtual Instruction Set Architecture (ISA) for a conceptual machine. It is the first truly cross-browser low-level code. In 2017 the Minimum Viable Product (MVP) of WASM was published, and all major browser vendors have implemented it in their browsers [2]. WASM 1.0 is specified by the WASM Core Specification [3]. Some of its core design principles are performance, compactness, security, and portability. Despite its name, WASM is technically not an assembly language; it is a bytecode format. It is designed to be platform-independent and can be embedded in browsers, run on a standalone Virtual Machine (VM), alternatively, be integrated into various environments.

WASM is a low-level bytecode, which executes with near-native code performance using standard modern hardware capabilities. It is also designed to be fast to transfer over the network, easy to decode, validate and compile. All of these steps are streamable and parallelizable. Most modern browsers already support streaming compilation, which compiles functions as soon as they arrive. Streaming means that there is no need to wait until the complete WASM code has been downloaded before starting the other steps. This technique minimizes page load latency [4].

Like JavaScript, WASM is likely to be used increasingly outside the web context. For example, the Shopify E-commerce platform uses it to allow shops hosted on its platform to extend functionality by executing WASM code provided by the hosted shops [5]. As the non-web use of WASM increases, WASM performance in such environments becomes increasingly essential. This paper is, to our knowledge, the first one that evaluates WASM's performance in such environments. It examines both execution time, which we compare to native execution times and since WASM is meant to be a compact representation, we also evaluate resulting code sizes. In our evaluation, we use different compilers to ensure as much as possible that the results are actual inherent properties of WASM and not of the used toolchain.

The rest of the paper is organized as follows. Section II presents background on WASM and related work attempting to characterize its performance. Section III explains the methodology we used to evaluate WASM performance and section IV presents our performance results. Finally, in section V we present our conclusions and outlines future work.

II. Background and Related Work

A. WASM Overview

WASM is a safe, portable, low-level bytecode format designed for efficient execution and compact representation. WASM is designed to solve some JavaScript issues, which is necessary to create high-performance web applications in browsers. WASM is a virtual ISA for a conceptual machine, and it is the first genuinely cross-browser low-level code. However, WASM is not the first attempt to bring near-native performance to the browser. Microsoft's ActiveX was based on executing native Windows binaries directly and relied on trust as the safety mechanism by using a code signing mechanism [6].

Another recent approach is Native Client (NaCl) [7], which relies on static validation of x86 machine code and a sandbox model; however, due to the reliance on the x86 architecture, it is inherently non-portable. Portable Native Client (PNaCl) is the portable version of NaCl, using a subset of LLVM [8] bytecode. However, PNaCl still exposes compiler- or platform-specific details such as the layout of the call stack. Also, the binary is not significantly smaller than JavaScript code [4]. Chrome was the only browser supporting NaCl and PNaCl. In 2017 the Chrome team announced the deprecation of these techniques in favor of WASM [9].

Asm.js is a strict subset of JavaScript, originally designed by Mozilla. It enables significant performance improvements compared to standard JavaScript. It is intended as a compilation target for statically typed languages, such as C. Asm.js code improves performance by eliminating dynamic types. A JavaScript engine with asm.js support can detect and optimize the code by treating values as statically typed. Firefox 22, released 2013, was the first browser using these optimizations [10]. Today all major browsers optimize asm.js. Since it is a subset of JavaScript, it can be executed on all JavaScript engines, in any case. However, it means that extending asm.js with new features always requires extending JavaScript first. Although asm.js code can be written by hand, it is usually generated by a compiler.

1) WASM Details: WASM is a low-level but platform-independent, i.e., portable bytecode format. Its instructions are an abstraction of the capabilities of modern hardware. It was designed with formal semantics from the beginning to enable fast and safe execution without requiring a separate trust mechanism. As such, it attempts to address all the problems of the previous ActiveX or asm.js approaches. The representation is designed to enable safety and efficiency by being compact, easy to decode, validate and compile. Also, it is designed to be *streamable*, i.e., validation and compilation can begin before an entire WASM code file has been transferred to the execution environment, e.g., to the browser. [4].

The code also is designed to be compact for fast transfers over the Internet. WASM's compact binary representation reduces load time and saves bandwidth. Because JavaScript is a plain text format, it cannot achieve such file sizes, even when minified and compressed [4].

Security is mandatory for web technologies since code originates from untrusted sources. The WASM code is executed in a memory-safe, sandboxed environment isolated from the host runtime. Applications cannot escape the sandbox, except through APIs provided by the host environment. Other web technologies like JavaScript are using a managed language runtime and garbage collection to enforce memory safety. However, garbage collection impacts performance negatively, which is why WASM's safety does not rely on it for memory safety but instead ensures it through its execution semantics-based design.

Another fundamental design goal of WASM is portability. Applications compiled to WASM are executable independent from machine architecture, Operating System (OS), and runtime platform. The same behavior can be expected in different environments. Once a program is compiled to WASM bytecode, it can be distributed and executed on all platforms as long as there is a WASM runtime available, similar to Java's Runtime Environment (JRE).

A WASM binary describes one module. A module is a distributable, loadable, and executable unit of code. It contains different sections, import, export, start, global, memory, data, table, elements, functions, and code. The module is a static representation. An instance of a module also consists of mutable memory and an execution stack, and instantiating a module requires that all its imports have to be satisfied.

WASM version 1 (MVP) is ideal for low-level languages, with minimal dynamic features at runtime. WASM does currently not provide any garbage collection. That is why statically typed languages with manual memory management like C, C++, and Rust are currently the most popular languages for WASM. Nevertheless, it is possible to compile higher-level languages to WASM, such as Go, C#, Java, and Python. Although, they currently have to bundle their runtime features, such as garbage collection or exception handling, into the binary, producing more bloated binaries. This bundling mechanism might change with a future version of WASM.

The WASM virtual machine is a stack machine, similar to the Java VM. Compared to a register machine, such as x86 or ARM, stack machines enable easier VM implementation, smaller binary encoding, and faster single-pass code verification. The virtual ISA supports only types and operations that are common on current hardware. WASM has only four value type: `i32`, a 32 bit integer type, `i64` a 64 bit integer type, `f32`, a 32 bit IEEE 754 floating-point type, and `f64`, a 64 bit IEEE 754 floating-point type. More complex types can be formed from these basic types.

WASM code has a binary and a textual encoding. Usually, WASM is encoded in binary format, which is more compact and can be transferred and parsed more efficiently. WASM Text Format (WAT), on the other hand, it is human-readable and is typically used for debugging, inspecting, or writing WASM code by hand. Both formats can be easily converted to each other via tools.

All major browser vendors have developed independent implementations of WASM. V8, SpiderMonkey, and JavaScript-Core, the JavaScript engines of Chrome, Firefox, and WebKit, reuse their optimizing JIT compilers to compile WASM modules ahead of time before instantiation [4], resulting in predictable high performance. Microsoft's Chakra engine lazily translates individual functions to an intermediate format and later compiles the hottest functions [4] producing faster startup times. V8 and SpiderMonkey are caching the compiled native code. The engine can avoid downloading, compiling, and optimization already processed modules, causing a noticeable startup time improvement.

Besides browsers, there are also implementations for other environments. Node.js is based on V8; therefore, WASM modules can be included in Node.js application. Wasmer is a WASM runtime that can be integrated into different language environments. Using the WASI interface, it is possible to execute WASM standalone without the necessity of another language.

B. WASM in Non-Web Environments

WASM is already being used in several non-web environments, namely mobile and desktop applications, server and cloud environments, microcontrollers, smart contracts, and polyglot programming, which we will discuss in turn.

1) Mobile and Desktop Applications: JavaScript is already a popular programming language for creating mobile and desktop applications. Using JavaScript beyond the web allows developments to use the same tools, libraries, and code on the web and mobile/desktop platform. The same applies to WASM. Cross-platform JavaScript frameworks like Electron can already embed and utilize WASM modules. However, WASM modules cannot only be embedded in JavaScript applications. Wasmer, for example, allows embedding in a wide range of languages (see section II-D). With WASM System Interface (WASI) standalone WASM applications can be created without the necessity to embed the module into another language's runtime system (see section II-C).

2) Server and Cloud Environemnt: Node.js is already a prevalent platform for backend development, such as web services. WASM modules can be easily embedded in a Node.js server application. Besides the reusability of WASM modules, the main advantage is the high performance of WASM code, which can be an alternative to C++ addons. Another emerging project is Lucet, a WASM compiler, and runtime for standalone WASM modules. It is designed for Function-as-a-Service (FaaS [11]) cloud applications. Lucet provides fast module initialization and low memory overhead (see section II-D). WASM is sandboxed. Cloud applications benefit from security by isolating applications from the host system and other applications. Therefore WASM is also a lightweight alternative to containers and virtualization technologies.

3) Microcontrollers: Usually, microcontroller programs are written with low-level languages like C. Disadvantages of using low-level languages are harder debugging and complicated porting of programs to other microcontrollers [12, section 1]. There are also various high-level programming languages available for writing microcontroller programs, e.g., Python and JavaScript. The disadvantages of high-level languages are slower execution and limited peripheral support [12, section 1]. WASM can be a middle ground between high-level and low-level languages [12, section 1]. It could be used as a low-level hardware abstraction to enable writing portable and performant programs for embedded systems and the Internet of Things (IoT) devices. Several projects are creating lightweight WASM interpreters and compilers for microcontrollers, such as WARDuino [12], WAMR [13] and Wasm3 [14].

4) Smart Contracts: WASM is gaining traction as a format for smart contracts in Blockchains. It enables near-native execution speed for smart contracts. Besides that, smart contract developers can use various programming languages and tools used by the WASM community. There is a proposal for Ethereum 2.0 to replace the current Ethereum Virtual Machine (EVM) with a new VM called Ethereum flavored WASM (eWASM), a deterministic subset of WASM [15]. WASM is closer to actual hardware instructions than EVM bytecode, resulting in more efficient code execution.

5) Polyglot Programming: WASM is designed to be embedded into JavaScript's runtime environment. Other languages and frameworks can also interoperate with WASM. Polyglot Programming is the approach of writing software in multiple languages. Advantages are reusability of code and free choice of language that best fits the problem to solve. The following implementations are in an early experimental phase. GraalVM announced GraalWasm, a WASM engine for GraalVM [16]. WASM modules will be able to interoperate with already supported GraalVM languages such as JavaScript, Python, Ruby, R, Java, C, and others. There are also more lightweight embedding efforts for many language environments. One ambitious project is Wasmer [17]. Wasmer is a WASM runtime and can be used as a library to embed WASM code in various languages. Wasmer currently supports Go, Rust, Python, Ruby, PHP, C, C++, C#, etc.

C. WASM System Interface

By default, WASM cannot talk to the operating system. A system interface is required for WASM applications beyond the browser. Such a system interface allows communication with the OS to access resources managed by the OS, like the file system, system clock, and network sockets. In March

2019, Mozilla announced WASI, which is planned to become the standard system interface for WASM [18]. In November 2019 the Bytecode Alliance has been founded. The Bytecode Alliance is an industrial partnership dedicated to creating secure software foundations, building on standards such as WASM and WASI [19]. Members of the Bytecode Alliance are Mozilla, Fastly, Intel and Red Hat. Just as WASM is an assembly language for a conceptual machine, WASM needs a system interface for a conceptual operating system, not any single operating system [18]. The reason not to choose an existing system interface is to perfectly fit WASM's fundamental principles: portability and security. WASI allows WASM to talk to the system securely. With WASI, the WASM application can run standalone. No embedding in other language environments is necessary. But in order to do so, the WASM runtime has to support the WASI interface. Mozilla also created a standalone runtime with WASI support, called Wasmtime (see section II-D).

D. WASM Implementations

There are several notable WASM implementations and compilers, which we will now discuss in turn.

Wasmtime is a standalone non-web runtime for WASM-WASI [20], is a project of the Bytecode Alliance. Wasmtime can be used as a command-line utility or a library embedded in a Rust or C application. The default JIT compiler is Cranelift. Cranelift supports all the functionality of WASM MVP. Wasmtime can also cache already compiled and optimized WASM binaries.

WAVM is a WASM VM, designed for use in non-web applications [21]. It uses the LLVM compiler backend. WAVM takes more time to tune the code for the exact CPU running the code. WAVM aims to achieve near-native performance. Besides WASI, WAVM fully supports proposed WASM features like SIMD, reference types, and exception handling. WAVM support specifying a directory to cache the compiled object code. Therefore huge modules do not have to be compiled a second time and can be loaded faster.

Wasmer is a standalone WASM runtime outside browsers [17]. It can be used on the command line and embedded in different languages. Supported languages are Go, Rust, Python, Ruby, PHP, C, C++, and C#. Wasmer supports multiple backends: singlepass, Cranelift and LLVM. Wasmer-JS is a Polyfill for running WASI modules in Node.js and browsers. Wasmer can also be used as an extension for PostgreSQL databases. WASM functions can be called within the PL/pgSQL language.

Lucet is a platform for executing WASM-WASI modules in a FaaS cloud infrastructure [22]. A significant goal of Lucet is the execution of a single request per WASM instance. Lucet is tuned for fast module instantiation and a low runtime footprint per instance. Hickey [23] claims Lucet can instantiate WASM modules in under 50 µs and handle tens of thousands of requests per second in a single system process. Lucet is currently used an an experimental platform of the cloud service provider Fastly.

Wasm3 is a high performance WASM interpreter [14]. The runtime supports the WASI interface. Wasm3 can be used on a great variety of system architectures, OSs, single-board computers, microcontrollers and browsers. It runs as a standalone runtime and can be used as a C, C++, Go, and Rust library. Wasm3 is not as fast as other runtimes using JIT compilers. Advantages of the interpreter over WASM compilers are a smaller runtime size and faster startup times.

III. Methodology

This paper evaluates WASM in non-web environments using code size, and startup and execution times as the evaluation metrics. We measure those for different WASM implementations and compare them to native code as well as optimized JavaScript using asm.js. All measurements were performed on an Intel x86-based Linux machine, with 16 GB of RAM, running on an i7-8565U CPU (1.8 GHz base frequency) with the Ubuntu SMP kernel, release 5.3.0-26-generic. A detailed listing of all specifications can be found in [24]. The used software package version can be found in appendix A.

We used the PolyBenchC benchmark suite as our primary benchmark. It is a benchmark suite of 30 numerical computations from various application domains such as linear algebra, image processing, physics simulation, dynamic programming, and statistics [25]. PolyBenchC was chosen because it contains standard algorithms used in many software applications and, as such, provides a representative evaluation of WASM performance characteristics. PolyBenchC benchmarks the single thread computational performance, which was the primary design objective of the first WASM version.

Native binaries are executed in the Linux as mentioned above. Asm.js is executed on the Node.js platform in V8, the most used JavaScript engine, which also drives, for instance, the popular Google Chrome browser. In addition, to ensure that the JavaScript results are not due to the specific execution engined, as second asm.js execution environment we used SpiderMonkey, the engine used in the Mozilla Firefox browser. Node.js and SpiderMonkey are additionally used to run WASM code. To evaluate WASM standalone execution, three WASI supporting runtimes are chosen: Wasmtime, Wasmer and WAVM. Notice that all of these three runtimes are in active development and performance is likely to improve further.

WASM code for JavaScript embedding and asm.js code is generated by the Emscripten compiler. Native x86-64 code and WASM-WASI code is generated using `clang`. The used compilers were called with similar arguments, optimizing for fast execution, time, e.g., using the `-O3` flag for the C compiler.

All steps of the benchmarking process are fully automated with Python scripts, which are available at [24].

A. Binary Size Measurement

Almost all compilation processes produce a single binary. This output binary is not dependent on any external libraries or data. Therefore the binary file sizes per target can be compared directly. The only exception is compiling with `emcc` (Emscripten) to WASM. It produces two files, one WASM file, and a JavaScript file, to load and execute the WASM code. For the Emscripten WASM version, the sum of the two file sizes is therefore used for comparison.

All modern browsers support HTTP Content-Encoding. Moreover, files are typically not transferred in plaintext. Instead, the content is compressed (using gzip, for instance) to speed up transfer and save bandwidth. Therefore we also compare the compressed file size and the compression rate of the code formats. Since gzip's compression level can be adjusted, we used a fixed level of 9, which is the highest compression level. 9 is a common default value and is used for example by Apache HTTP servers.

B. Execution and Startup Time Measurement

Most runtimes use a JIT compiler to translate WASM to native instructions. This method measures the execution time to evaluate the overhead of different algorithms, runtimes, and its JIT compilers. Following tests excludes time for VM startup, compilation, code optimization and additional data preparation steps. All PolyBenchC benchmarks are compiled with the *POLYBENCH_TIME* macro definition. This option makes every PolyBenchC executable output its execution time before exit. All PolyBenchC benchmarks were executed 15 times to achieve sufficient accuracy and generate confidence intervals.

Besides execution time, startup time is the main reason to use native binary applications. Like all other non-native code execution, WASM requires additional startup steps, e.g. VM startup, code validation, code compilation and code optimization. For JavaScript, all major browsers use a combination of an interpreter, for fast startups, and JIT compilers to optimize execution time. WASM runtimes use various approaches. Most WASM runtimes first JIT compile the whole bytecode to native code. In general, WASM code is also optimized right from the start to get predictable execution behavior. So we expect a more prolonged startup phase compared to asm.js.

The PolyBenchC benchmarks are modified to stop the process directly after the main function entry, and the startup time is determined by measuring the process time. The startup time should vary between benchmarks because larger codes need more time for validation and optimization. Moreover, it should also vary between runtimes because they perform different tasks at startup. All examined WASI runtimes can cache already compiled and optimized native code. To avoid that caching is affecting our timing, it was disabled to observe the behavior of a first execution; again, we performed 15 measurements.

IV. Evaluation

This section summarizes our findings for code size, and execution and start up times. Complete measurement data is available in [24].

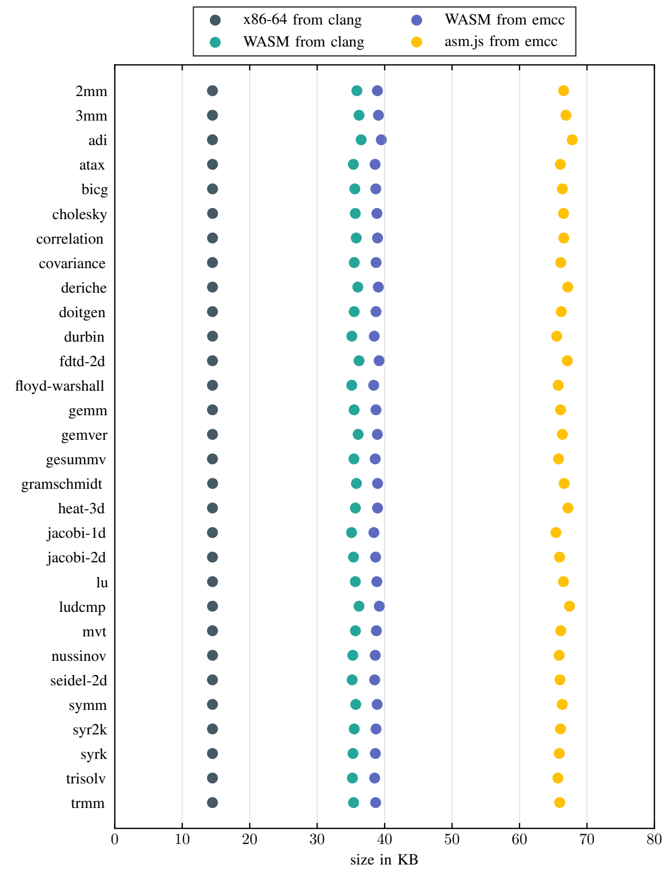

Fig. 1: Binary sizes of the PolyBenchC benchmarks compiled to different target platforms.

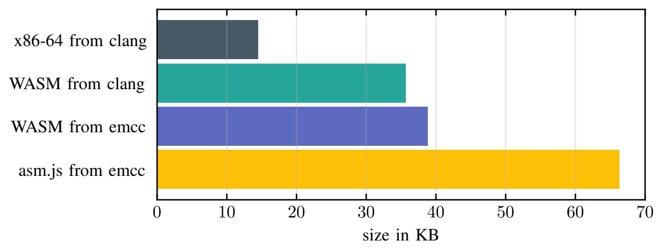

Fig. 2: Binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Geometric mean of 30 benchmark binaries per compile target.

A. Binary Size

1) Uncompressed: The measured uncompressed binary size of the PolyBenchC applications did not vary much across applications, for instance, the WASM generated by `emcc` were just under 40 kB.

As shown in figure 1, binary sizes of the different benchmark programs do not vary much, so we concentrate on geometric means since we are mostly interested in code size ratios. Figure 2, therefore compares the geometric mean of code sizes across the different benchmarks. The reason why

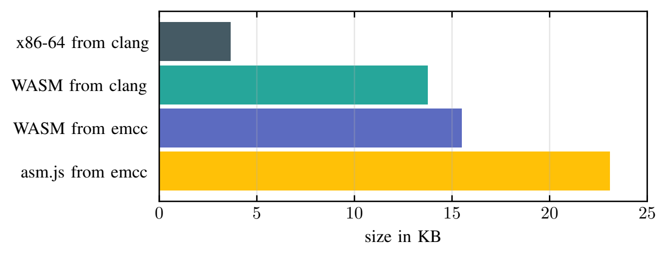

Fig. 3: Compressed binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Gzip is used for compression. Geometric mean of 30 benchmark binaries per compile target.

the code sizes vary so little, as shown in figure 1, is that the compute-intensive part (the focus of the benchmark suite) is performed by a few lines of code, and the overall code size is dominated by included library code, such as *stdio.h*, *unistd.h*, *math.h*, or the PolyBenchC specific *polybench.h*.

The average x86-64 binary is 14 kB, the `clang` WASM binary 36 kB, the `emcc` WASM binary is 39 kB (including JavaScript glue code), and the average asm.js code is 66 kB. We found that the WASM files are on average 246% of the native code size, and the WASM version is on average 58% of the asm.js JavaScript file. The measurements show that the WASM is roughly in the middle between x86-64 and asm.js code.

2) Compressed: Given the minimal variation of the uncompressed code sizes, the compressed binary sizes did not vary much and so we are again summizing data by showing geometric mean of all 30 benchmarks per compilation target in figure 3.

The average sizes for the compressed versions were for asm.js 23 kB, for x86-64 code 3.7 kB, for `clang` WASM 14 kB, and for `emcc` WASM 16 kB. The best average compression ratio was obtained for x86-64 with a ratio of 3.96 : 1, for the asm.js benchmarks it is 2.87 : 1, for `clang` WASM files it is 2.59 : 1, and for `emcc` WASM files it is 2.50 : 1. On average the `emcc` WASM code is 67% the size of the asm.js. Therefore the WASM format has a clear advantage in size, saves bandwidth and accelerates the loading of web applications compared to JavaScript. Size restricted environments like smart contracts in Blockchains or IoT devices can benefit from the compactness of WASM.

B. Execution Time

Figure 5 shows the geometric mean of all benchmark measurements per runtime environment in relation to x86-64 execution time. Focussing on the WASI standalone runtimes, WAVM, Wasmer and Wasmtime, the figure shows that execution time can vary greatly for different WASM runtimes. Peformance of WASM code depends heavily on the used runtime and its JIT compilation and code optimization. Although the average WAVM execution of PolyBenchC benchmarks is only 14% slower than x86-64, Wasmtime has significantly worse

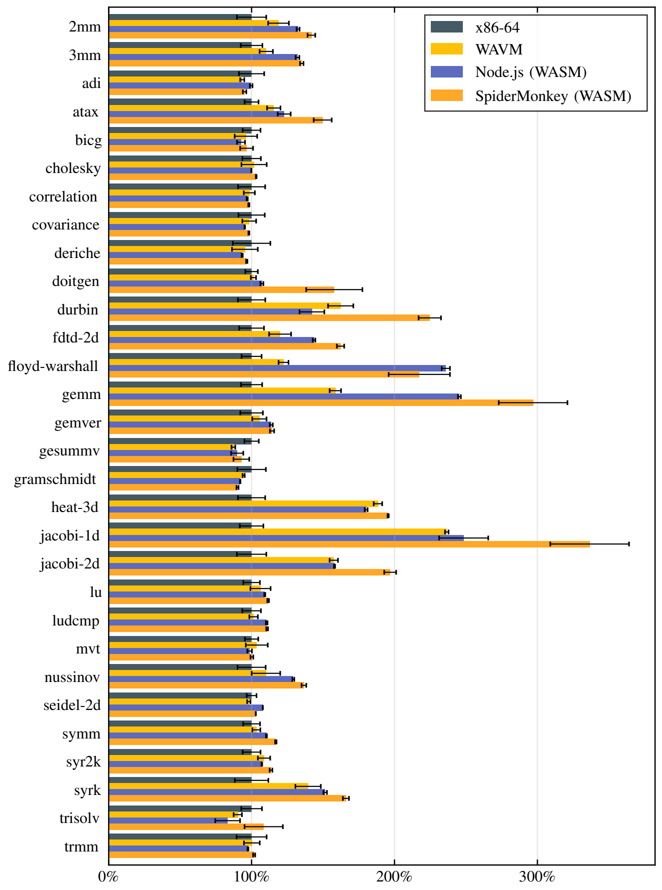

Fig. 4: Comparison of the top 4 runtimes with the best PolyBenchC execution times. Values are normalized to native x86-64 execution time. Arithmetic means of 15 runs and 95% confidence intervals. Small values are better.

values. The Wasmtime execution time is on average 345% of the native x86-64 time. This large difference as we will see in IV-C, is mostly due to startup time. We observe that Wasmer's execution time is on average 221% of the native x86-64 execution time.

WAVM shows overall the best results, but has the longest startup phase due to optimizations (cf. IV-C). 18 of 30 WAVM benchmarks are within 110% of native, 25 are within 150% and almost all are within 2× x86-64 execution time (see figure 6). The only exception is the *jacobi-1d* benchmark, with 237% of x86-64 execution time (see figure 6).

Figure 4 breaks out execution times for the four fastest runtimes. WASM appears to be faster than native x86 code for some benchmarks, however, this is due to measurements inaccuracies and execution time variance, and the confidence intervals overlap accordingly. Only two benchmarks, *gesummv* and *trisolv*, can achieve better performance with WASM (see figure 4). *Gesummv* and *trisolv* are one of the shortest benchmarks and the difference in speed is less than 1 ms.

After WAVM, Node.js and SpiderMonkey achieve the next best WASM performance. On average Node.js (WASM) is

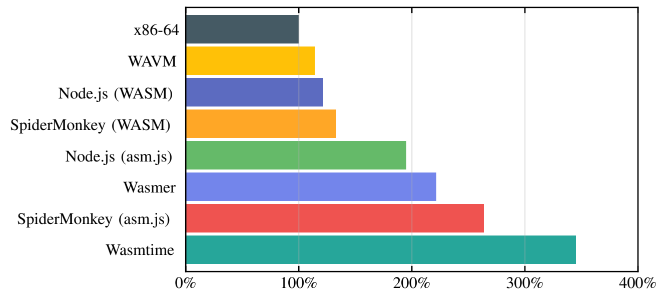

Fig. 5: Execution time of PolyBenchC benchmarks on different runtime environments. Values are normalized to native x86-64 execution time. Arithmetic means of 15 runs per benchmark. Geometric mean of all 30 benchmarks per runtime environment. Lower values are better.

122% of native execution time and SpiderMonkey (WASM) on average is 133% of native time. The WASM is almost always faster than the respective asm.js version (cf. figure 7), with the only exception of *gramschmidt* and *seidel-2d*.

On average WASM benchmarks in Node.js are 62.3% of the asm.js execution time. In SpiderMonkey the average WASM benchmark is 50.4% of the asm.js execution time, i.e, a significant performance improvement. The most significant improvement can be observed in SpiderMonkey with the *heat-3d* algorithm, which takes on average 30.0 s as asm.js and only 4.97 s as WASM code, this is an impressive 6.06× improvement.

WASM can outperform JavaScript and is the better choice when computational performance is the primary concern.

C. Startup Time

Figure 8 shows the geometric mean of all benchmark startup time measurements per runtime environment. On average x86-64 PolyBenchC startup takes 4.2 ms and the compared runtimes are at least 7× slower. Staring up JavaScript and WASM code requires many additional steps compared to native code. First of all, the runtime itself has to be started and the benchmark code has be be loaded, including steps such as validation, interpretation, JIT compilation and optimization, which clearly manifests itself in startup times.

Figure 9 and figure 10 show that the benchmarks per runtime are almost in line and have similar values. Exceptions are SpiderMonkey (WASM) and Node.js (asm.js). The values of those are really close to each other and do not clearly show which one is fastes. The geometric mean SpiderMonkey (WASM) startup time is 33.3 ms and for Node.js with asm.js it is almost identical with 33.5 ms.

The average SpiderMonkey (asm.js) startup time is 30.5 ms. The average Node.js (WASM) startup time is 40.6 ms. Comparing asm.js and WASM shows that the startup of WASM is slower than the asm.js version in the respective runtime. In Node.js the average asm.js startup is 7.1 ms (17%) faster than WASM, whereas in SpiderMonkey the average asm.js startup is 2.8 ms (8.4%) faster than WASM.

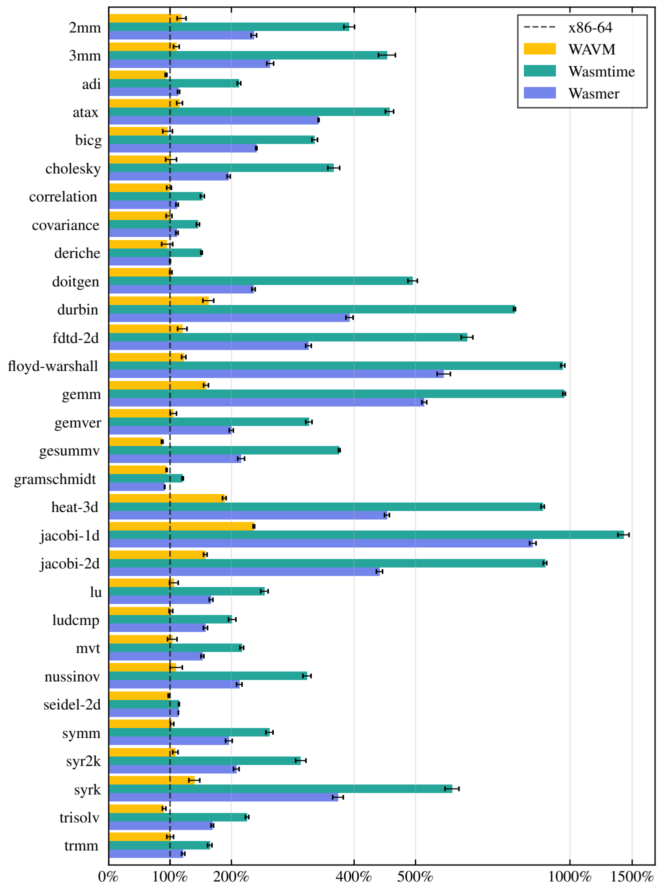

Fig. 6: Execution time of PolyBenchC benchmarks on WASI runtimes. Values are normalized to native x86-64 execution time and plotted on a logarithmic scale. Arithmetic means of 15 runs and 95% confidence intervals. Lower values are better.

All three benchmarked WASI standalone runtimes start slower than the JavaScript embedded runtime. The average Wasmer startup time is 62.5 ms, and the average Wasmtime startup time is 77.4 ms. With an average startup time of 366 ms WAVM is significantly slower than the other WASI runtimes (cf. figure 9). However, WAVM is the fastest WASM runtime in terms of execution time, as shown before. WAVM takes more time for compiling and optimizing and therefore achieves the best execution time.

The startup time of WASM is comparable to JavaScript. Fast startup is an essential property for responsive browser application as well as for fast container instantiation.

D. Portability

WASM is designed to be portable. This section briefly presents its requirements and introduces some existing runtime environments.

In order to run WASM modules the runtime has to offer deterministic behavior as specified in [3]. The WASM specification allows limited nondeterminism for some operations as a compromise, e.g. to achieve native performance [26].

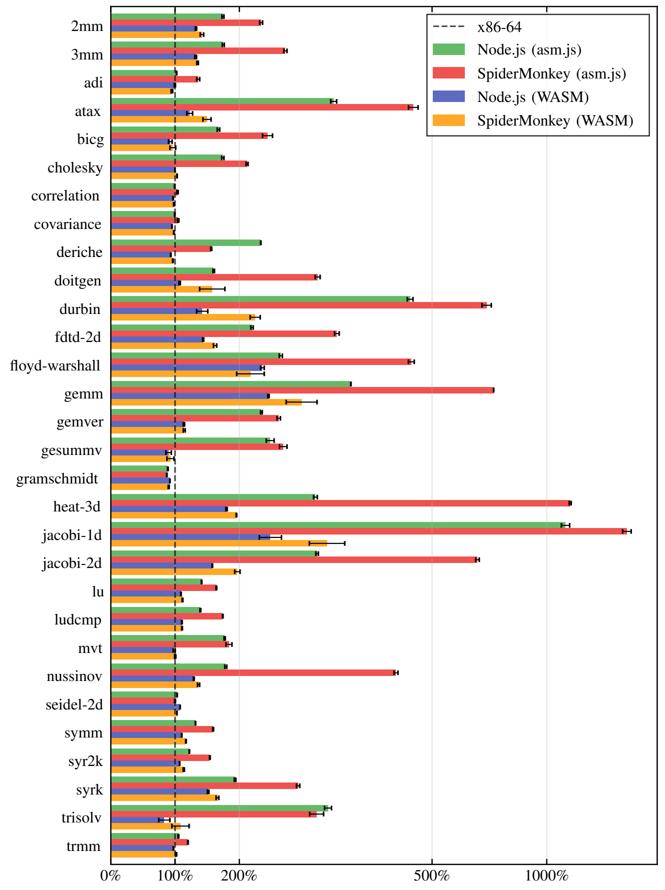

Fig. 7: Execution time of PolyBenchC benchmarks compiled to asm.js and WASM code. Values are normalized to native execution time and plotted on a logarithmic scale. Arithmetic means of 15 runs and 95% confidence intervals. Lower values are better.

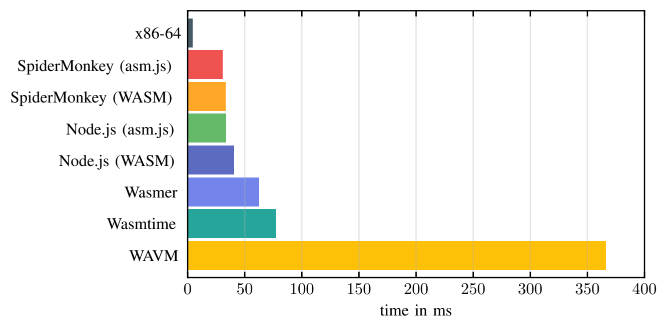

Fig. 8: Startup time (in milliseconds) of PolyBenchC benchmarks on different runtime environments. Arithmetic means of 15 runs per benchmark. Geometric mean of all 30 benchmarks per runtime environment. Lower values are better.

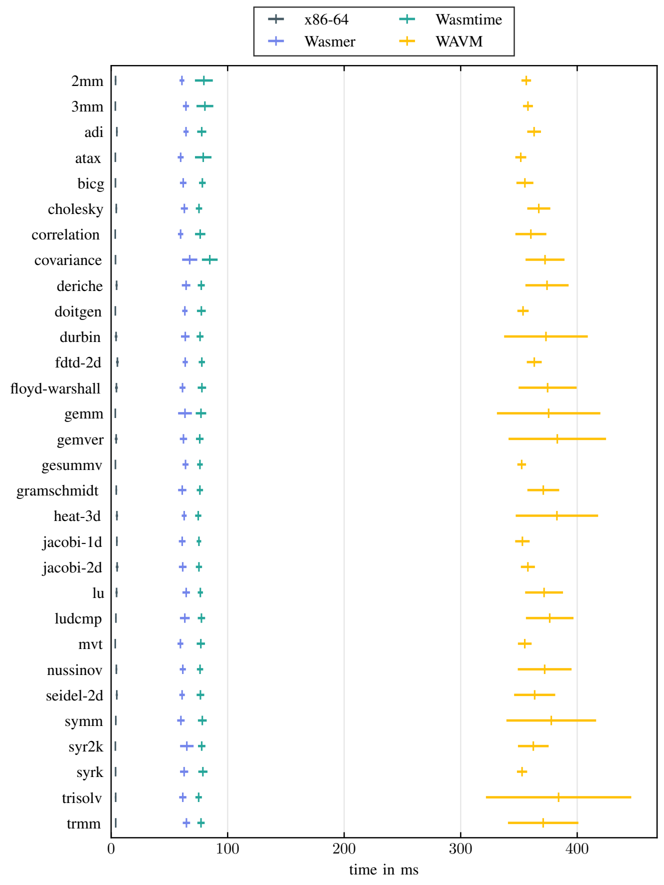

Fig. 9: Comparison of PolyBenchC benchmark startup time of native x86-64 and WASI standalone runtimes. Arithmetic means of 15 startups and 95% confidence intervals. Lower values are better.

The execution environment has to offer certain characteristics. Characteristics are e.g. addressable at a byte memory granularity, IEEE 754-2008 32 bit and 64 bit floating point support, little-endian byte ordering and efficiently addressability with 32 bit pointers [27]. If the host hardware, OS, or platform does not offer these required characteristics, it might be possible for the runtime to emulate this particular behavior [27]. This might lead to poor performance [27].

WASM can already be executed on a variety of ISA, OS and application environments. WASM runs in all major browser, e.g. Firefox, Chrome, Safari and Edge [28]. WASM runs on the most common processor architectures, e.g. x86, ARM, MIPS and RISC-V [14]. WASM runs on most system, e.g. Linux, Windows, macOS, Android and iOS [14]. WASM runs on Single-Board Computers and microcontroller units, e.g. Raspberry Pi, Orange Pi, Arduino and ESP8266 [14]. WASM runs embedded in different language environments, e.g. Rust, C, C++, C#, Python, Go, PHP and Ruby [17].

WASI enables WASM modules to access common system resources in a portable and secure way. WASI-compatible runtimes can execute WASI modules without the necessity

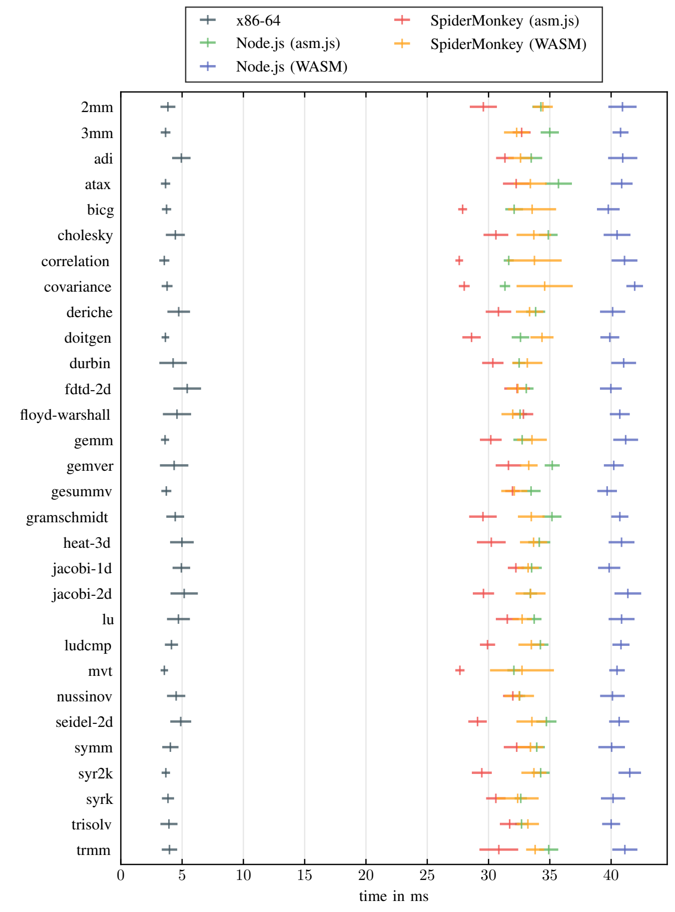

Fig. 10: Comparison of PolyBenchC benchmark startup time of native x86-64, asm.js and WASM embedded in JavaScript environments. Arithmetic means of 15 startups and 95% confidence intervals. Lower values are better.

of embedding the module into another language environment. Existing runtimes are e.g. Lucet, Wasmer, Wasmtime, WAVM and Wasm3 (see section II-D).

V. Conclusions and Future Work

One of WASM's stated goals is near-native performance. As shown in figure 5, performance strongly depends on the used runtime. While generally still significantly slower than compiled x86 code, WASM's performance is usually better than optimized JavaScript asm.js, with the V8, SpiderMonkey and WAVM runtimes achieving comparable results to native applications in many test codes (cf. figure 4).

Besides, we found that WASM is mature enough for browser applications. All major browsers have stable support for running WASM code, and several mature toolchains exist, which are essential for software development. Compilers, command-line tools, and debugging support in browsers have enabled productive use. Noteworthy examples are the Emscripten compiler, the WABT tools [29] and debugging capabilities in Google's Chrome browser DevTools [30].

Beyond web environments, the use of WASM is still somewhat experimental, and many typical programming features are missing so that currently, the usefulness of WASM for real-world software projects is rather limited. However, because of its versatility, there is a lot of experimentation going on in the most diverse areas (see II-B).

Since WASM performs better than JavaScript and even asm.js, it is a great embedded companion and possible replacement. Likewise other scripting languages might also benefit from the computing performance of WASM, without sacrificing its platform independence. WASM enables new kinds of applications on the web and non-web platform, e.g., image, audio, video processing apps, games, cryptography, and scientific applications. WASM is a better compilation target than JavaScript for low-level languages such as C and Rust, where the dynamic features of JavaScript are unnecessary and slow down execution. This makes WASM an excellent choice for applications that require high compute performance. For others, JavaScript is often fast enough, essentially if their work consists primarily of network data transfer and user interface-related tasks. As discussed, WASM is not yet a suitable compilation target for dynamic, garbage collected high-level languages (see II-A1).

Some recent and upcoming WASM features will require new performance benchmarks. Proposed features such as SIMD support, threads, bulk memory operations, garbage collection and exception handling need to be compared to native code performance. But PolyBenchC does not test these features, additional benchmarks need to be developed.

The WASI API extends the possibilities of WASM, by allowing direct access to system resources, in a secure way. WASI is not yet finalized and therefore is more suitable for prototyping and small projects. Important features like opening network sockets are still missing. Currently WASI is only capable of receiving data from sockets, sending data to sockets and closing sockets [31]. Therefore network connections have to be opened ahead of time by the host runtime. Future WASI might get a capability describing a set of possible sockets that can be created. A set of ports, addresses, or protocols could be allowed using specific capabilities [32]. Such a feature would be appealing for implementing a secure Function-as-a-Service serverless cloud service (for definition and background on FaaS, see [11]).

Another promising area for future work is research in the use of WASM in microcontrollers. Microcontrollers require lightweight WASM runtime, which created additional challenges. In addition and as mentioned in section II-B4, Ethereum might replace its current virtual machine for smart contracts, EVM, with eWASM. Evaluating the performance of WASM in the smart contract context is, therefore, another promising area.

Appendix

A. Software Versions Used in Experiments

cargo	1.38.0
clang	9.0.0-2
conda	4.7.12
Emscripten	emcc 1.39.2
Node.js	v13.7.0
PolyBenchC	4.2.1 (beta)
rustc	1.38.0
rustup	1.19.0
SpiderMonkey	JavaScript-C74.0
wasi-sysroot	8.0
Wasmer	0.7.0
wasmtime	0.8.0
WAVM	0.0.0-prerelease nightly (0cfad6a)

References

[1] nodejs.org, "Node.js," accessed: January 15, 2020. [Online]. Available: https://nodejs.org/

[2] J. McConnell, "WebAssembly support now shipping in all major browsers," Nov 2017, accessed: December 27, 2019. [Online]. Available: https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers/

[3] WebAssembly Working Group, "WebAssembly Core Specification," W3C, Oct 2019, accessed: December 27, 2019. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[4] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," vol. 52, no. 6, pp. 185–200, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140587.3062363

[5] "How Shopify Uses WebAssembly Outside of the Browser," 2020, accessed: April, 23, 2021. [Online]. Available: https://shopify.engineering/shopify-webassembly

[6] A. Rubin and D. Geer, "Mobile code security," *IEEE Internet Computing*, vol. 2, no. 6, pp. 30–34, 1998.

[7] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93.

[8] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[9] chrome.com, "WebAssembly migration guide," 2017, accessed: January 16, 2020. [Online]. Available: https://developer.chrome.com/native-client/migration

[10] Mozilla, "Mozilla Firefox release notes," Mozilla, accessed: February 9, 2020. [Online]. Available: https://www.mozilla.org/en-US/firefox/releases/

[11] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[12] R. Gurdeep Singh and C. Scholliers, "WARDuino: A dynamic WebAssembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 27–36. [Online]. Available: https://doi.org/10.1145/3357390.3361029

[13] Bytecode Alliance, "WebAssembly Micro Runtime," GitHub, accessed: February 15, 2020. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[14] "Wasm3," GitHub, accessed: January 6, 2020. [Online]. Available: https://github.com/wasm3/wasm3

[15] "Ethereum flavored WebAssembly (ewasm)," GitHub, Oct 2019, accessed: January 6, 2020. [Online]. Available: https://github.com/ewasm/design

[16] S. S. Salim, A. Nisbet, and M. Luján, "Towards a WebAssembly standalone runtime on GraalVM," in *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2019. New York, NY, USA: ACM, 2019, pp. 15–16. [Online]. Available: http://doi.acm.org/10.1145/3359061.3362780

[17] "Wasmer," 2019, accessed: December 29, 2019. [Online]. Available: https://wasmer.io/

[18] L. Clark, "Standardizing WASI: A system interface to run WebAssembly outside the web," Mozilla Hacks – the Web developer blog, Mar 2019, accessed: January 4, 2020. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[19] Bytecode Alliance, "About the Bytecode Alliance," Bytecode Alliance, accessed: February 27, 2020. [Online]. Available: https://bytecodealliance.org/

[20] "Wasmtime," GitHub, 2019, accessed: December 29, 2019. [Online]. Available: https://github.com/bytecodealliance/wasmtime

[21] "WAVM," GitHub, 2019, accessed: December 27, 2019. [Online]. Available: https://github.com/WAVM/WAVM

[22] "Lucet," GitHub, 2019, accessed: December 28, 2019. [Online]. Available: https://github.com/bytecodealliance/lucet

[23] P. Hickey, "Announcing Lucet: Fastly's native WebAssembly compiler and runtime," Fastly, Mar 2019, accessed: February 15, 2020. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[24] B. Spies, "Supplementary material," GitHub, 2020. [Online]. Available: https://github.com/HAWMobileSystems/Clei2021WebAssembly

[25] L.-N. Pouchet, U. Bondugula, and T. Yuki, "PolyBench/C," GitHub, 2016, accessed: January 4, 2020. [Online]. Available: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

[26] webassembly.org, "Nondeterminism in WebAssembly," accessed: March 2, 2020. [Online]. Available: https://webassembly.org/docs/nondeterminism/

[27] ——, "Portability," accessed: March 2, 2020. [Online]. Available: https://webassembly.org/docs/portability/

[28] ——, "WebAssembly," accessed: January 4, 2020. [Online]. Available: https://webassembly.org/

[29] "WABT: The WebAssembly Binary Toolkit," GitHub, accessed: February 25, 2020. [Online]. Available: https://github.com/WebAssembly/wabt

[30] I. Stepanyan, "Improved WebAssembly debugging in Chrome DevTools," Google Developers, 2019, accessed: February 24, 2020. [Online]. Available: https://developers.google.com/web/updates/2019/12/webassembly

[31] Bytecode Alliance, "WASI Core API," GitHub, 2019, accessed: December 21, 2019. [Online]. Available: https://github.com/bytecodealliance/wasmtime/blob/master/docs/WASI-api.md

[32] ——, "Additional background on Capabilities," GitHub, 2019, accessed: February 29, 2020. [Online]. Available: https://github.com/bytecodealliance/wasmtime/blob/master/docs/WASI-capabilities.md