



**HOCHSCHULE LANDSHUT**  
UNIVERSITY OF APPLIED SCIENCES

FACULTY OF COMPUTER SCIENCE

## **BACHELOR THESIS**

# UTILIZATION OF WEBASSEMBLY IN NON-WEB ENVIRONMENTS

Benedikt Spies

March 2020

supervised by Prof. Dr. Markus Mock

# ERKLÄRUNG ZUR BACHELORARBEIT

(gemäß § 11, Abs. (4) 3 APO)

Benedikt Spies

## Hochschule für angewandte Wissenschaften Landshut Fakultät Informatik

Hiermit erkläre ich, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

.....

(Datum)

.....

(Unterschrift der/des Studierenden)

# *Abstract*

*WebAssembly is a young technology with great potential for the web platform. This technology is necessary to bring certain kinds of applications to the browser. At the beginning JavaScript was not meant to handle heavy processing, but today it is also used for audio, video and 3D graphics software. Despite remarkable performance improvements in the last years, JavaScript cannot compete with machine level code. WebAssembly is a portable, low-level bytecode, providing nearly no-overhead code execution. In March 2019 Mozilla announced the WebAssembly System Interface, allowing WebAssembly code to communicate with the operating system. This allows WebAssembly applications to run outside browsers. The thesis shows the current state of WebAssembly and its System Interface, describes the benefits and costs of these technologies for applications in different environments and compares performance, portability and security to current technology standards. It turns out that using WebAssembly does not necessarily mean high performance. Missing features make WebAssembly useless for some application. Nevertheless WebAssembly has the potential to be beneficial in many environments.*

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Background . . . . .	1
1.2. Scope . . . . .	2
1.3. Outline . . . . .	3
<b>2. Theory</b>	<b>4</b>
2.1. Timeline . . . . .	4
2.2. JavaScript . . . . .	5
2.2.1. Introduction . . . . .	5
2.2.2. Utilization in Non-Web Environments . . . . .	5
2.3. asm.js . . . . .	7
2.4. Emscripten . . . . .	8
2.5. Node.js . . . . .	9
2.6. WebAssembly . . . . .	10
2.6.1. Introduction . . . . .	10
2.6.2. Prior Attempts . . . . .	11
2.6.3. Performance . . . . .	11
2.6.4. Compactness . . . . .	12
2.6.5. Security . . . . .	12
2.6.6. Portability . . . . .	12
2.6.7. Module Structure . . . . .	13
2.6.8. Language Support . . . . .	13

2.6.9.	Conceptual Machine . . . . .	13
2.6.10.	Binary and Text Format . . . . .	14
2.6.11.	Current State . . . . .	15
2.6.12.	Implementations . . . . .	16
2.6.13.	Planned Features . . . . .	16
2.6.14.	Utilization in Non-Web Environments . . . . .	19
2.7.	WebAssembly System Interface . . . . .	22
2.7.1.	Introduction . . . . .	22
2.7.2.	System Calls . . . . .	23
2.7.3.	Portability . . . . .	23
2.7.4.	Security . . . . .	24
2.7.5.	Current State . . . . .	24
2.7.6.	Implementations . . . . .	25
<b>3.</b>	<b>Method</b>	<b>27</b>
3.1.	Benchmarking WebAssembly . . . . .	27
3.1.1.	Choosing Benchmark Suite . . . . .	28
3.1.2.	Choosing Runtime Environments . . . . .	28
3.1.3.	Compiling Benchmarks . . . . .	28
3.1.4.	Binary Size Measurement . . . . .	29
3.1.5.	Execution Time Measurement . . . . .	30
3.1.6.	Startup Time Measurement . . . . .	31
3.2.	Determining Portability . . . . .	33
3.3.	Utilizing the Same WebAssembly Binary in Different Environments	34
3.3.1.	Choosing an Algorithm and Tools . . . . .	34
3.3.2.	Compiling Implementation to WebAssembly . . . . .	36
3.3.3.	Running on Command Line . . . . .	36
3.3.4.	Running in Browser . . . . .	37
3.3.5.	Running in Node.js . . . . .	39
3.4.	Evaluating Security . . . . .	41

<b>4. Analysis</b>	<b>42</b>
4.1. Binary Size . . . . .	42
4.1.1. Uncompressed . . . . .	42
4.1.2. Compressed . . . . .	45
4.2. Execution Time . . . . .	47
4.3. Startup Time . . . . .	53
4.4. Portability . . . . .	57
4.5. Security . . . . .	59
<b>5. Discussion</b>	<b>62</b>
5.1. Near-Native Performance . . . . .	62
5.2. Usage of WebAssembly in Production . . . . .	62
5.3. When to Prefer WebAssembly . . . . .	63
5.4. When Not to Prefer WebAssembly . . . . .	63
5.5. Future Work . . . . .	64
5.5.1. Comparing WebAssembly to Other Bytecodes . . . . .	64
5.5.2. Performance Benchmarking with Added Features . . . . .	64
5.5.3. Benchmarking Further Runtimes . . . . .	64
5.5.4. Security of WASI Sockets . . . . .	65
5.5.5. Evaluating Benefits and Costs on Microcontrollers . . . . .	65
5.5.6. Comparing ewasm to EVM . . . . .	65
<b>6. Conclusion</b>	<b>66</b>
<b>Bibliography</b>	<b>67</b>
<b>List of Figures</b>	<b>74</b>
<b>List of Tables</b>	<b>78</b>
<b>Glossary</b>	<b>79</b>
<b>Acronyms</b>	<b>84</b>

<b>Appendices</b>	<b>86</b>
A. PolyBenchC Measurements . . . . .	86
B. System Specification and Software Versions . . . . .	91

# Chapter 1

## Introduction

### 1.1 Background

JavaScript (JS) has a monopoly in client-side web development, mainly because it is the only web standard technology running in all major browsers. At the beginning JS was not designed as a language for writing complex application or high-performance algorithms. But the capabilities of JS improved a lot in the last two decades, enabling heavy applications, for example games, image and audio processing software. The browser is no longer just a program for displaying documents, but also a platform for rich applications, such as Google Maps.

Due to its portability and security JS also majored in other environments. Node.js is one of the most popular server-side application platforms. Unfortunately design restrictions prevent JS from reaching near-native execution performance. That is why the major browser vendors started developing a low-level bytecode called WebAssembly (WASM) to address the restrictions of the standard client-side web language JS. Since the release of WASM version 1, WASM is gaining popularity in web browsers. Similar to JS, WASM provides a portable and secure foundation that might also spreading to other environments. There are already some efforts utilizing WASM in various areas besides the web environment.



## 1.2 Scope

The main focus of this thesis is to introduce the technology WASM with its current state of development and to evaluate benefits and costs of using these technologies in a variety of environments. This thesis does not concentrate specifically on one runtime environment. WASM is treated as a bytecode format that can be an alternative or extension to existing environment-specific technologies. WASM claims to provide near-native execution performance, which is attempted by this study to be validated. WASM could also bring web security standards and portability to multiple platforms, but at what cost? In addition, the thesis demonstrates the usability of the relatively young technology.

## 1.3 Outline

This thesis start by introducing the technologies JavaScript (JS), asm.js, WebAssembly (WASM) and WebAssembly System Interface (WASI). In addition, a chronological overview of the latest events, developments and announcements concerning these technologies is given. The introduction also presents the current state of WASM and planned features. After that, the focus shift towards non-web environments where WASM is already utilized or where is a legitimate interest to use it.

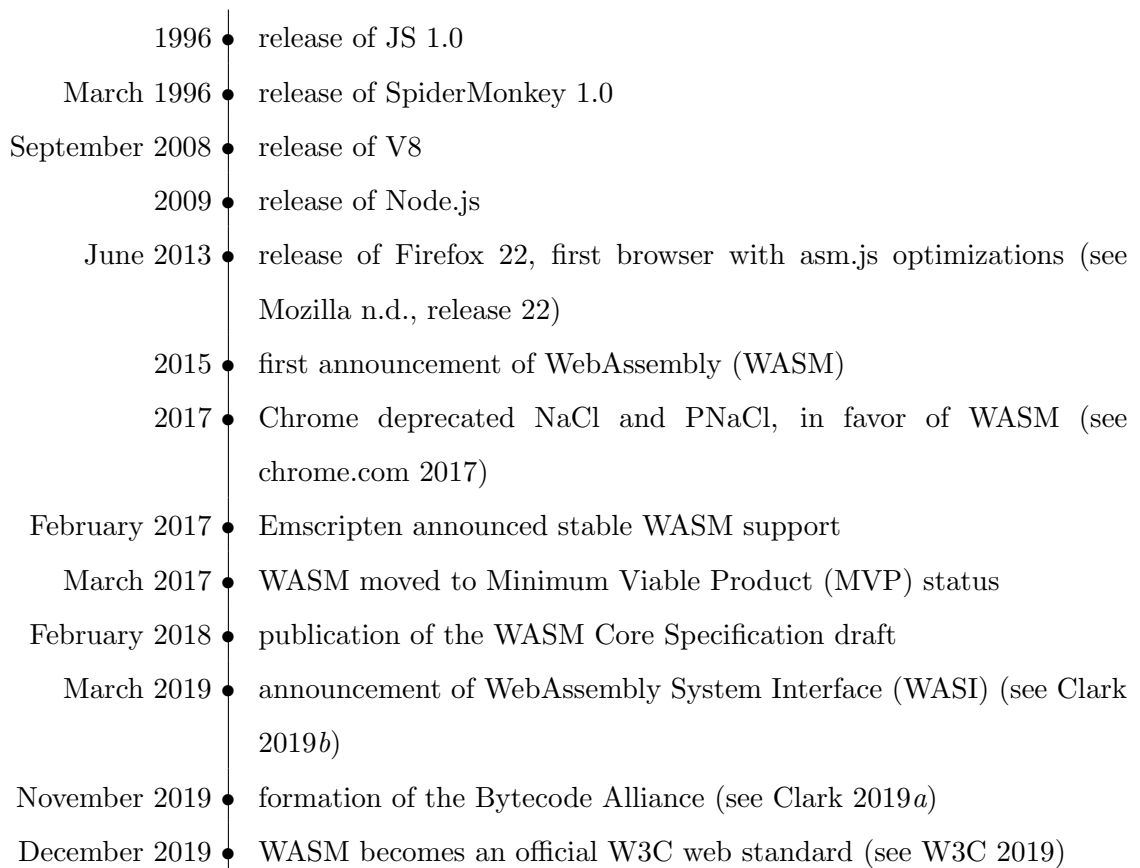
To quantify the benefits and costs of using WASM, the following chapter presents suitable methods. Hereby native machine code and JS set the reference frame for comparison. In this regard, the most important issues are portability, runtime behavior and security. Performance is compared with suitable benchmark suits, called PolyBenchC. Other compared criteria are binary size and startup time. Portability is showcased by deploying a gzip algorithm implementations to different environment. Relevant publications are used to evaluate WASM's security.

Afterwards the gained data is analyzed and made clear by visualizing it. In the following chapter, the findings are viewed from different perspectives, trying to clarify the questions that came along during working on this study. Finally the thesis concludes with some personal thoughts.

# Chapter 2

## Theory

### 2.1 Timeline



1996	•	release of JS 1.0
March 1996	•	release of SpiderMonkey 1.0
September 2008	•	release of V8
2009	•	release of Node.js
June 2013	•	release of Firefox 22, first browser with asm.js optimizations (see Mozilla n.d., release 22)
2015	•	first announcement of WebAssembly (WASM)
2017	•	Chrome deprecated NaCl and PNaCl, in favor of WASM (see chrome.com 2017)
February 2017	•	Emscripten announced stable WASM support
March 2017	•	WASM moved to Minimum Viable Product (MVP) status
February 2018	•	publication of the WASM Core Specification draft
March 2019	•	announcement of WebAssembly System Interface (WASI) (see Clark 2019b)
November 2019	•	formation of the Bytecode Alliance (see Clark 2019a)
December 2019	•	WASM becomes an official W3C web standard (see W3C 2019)

## 2.2 JavaScript



Figure 2.1.: JavaScript (JS) logo  
(source *JS Logo By The Community* n.d.)

### 2.2.1. Introduction

The web began as a simple document exchange network but has now become the most ubiquitous application platform ever (see Gallant 2019, section 1). JavaScript (JS) was created by Brendan Eich in 1995. JS is a object oriented, loosely-typed dynamic scripting language. JS was the only programming language of web browsers until now. Other technologies have been only available via plugins or not supported by all major browsers. Because of JavaScript’s ubiquity, rapid performance improvements and perhaps through sheer necessity, it has become a compilation target for other languages (see Haas et al. 2017, section 1). Emscripten (see Section 2.4) can be used to compile C and C++ programs to `asm.js`, a superset of JS (see Section 2.3).

### 2.2.2. Utilization in Non-Web Environments

JS has also become a major technology beyond browsers. Node.js can be used for server-side development (see Section 2.5). With React Native and NativeScript, mobile apps can be written in JS. Electron is a framework to create cross-platform desktop applications. These technologies and many more bring the JS language nearly into every environment. Developers can use the tools they are familiar with

---

everywhere. The ability to use the same language and its ecosystem everywhere made JS one of the most popular and most used programming languages.

## 2.3 asm.js

Asm.js is a strict subset of JS, originally designed by Mozilla. It enables significant performance improvements compared to standard JS. It is intended as a compilation target for statically-typed languages, such as C. Asm.js code improves performance by eliminating dynamic types. Figure 2.2 shows an example of how asm.js can treat values as integers, even if JS has no integer type. Dynamic typing requires languages to perform type checking at runtime and prevents Just-in-Time (JIT) compilers from generating the right machine instructions upfront. A JS engine with asm.js support can detect and optimize the code by treating values as statically-typed. Firefox 22, released 2013, is the first browser using asm.js optimizations (see Mozilla n.d., release 22). Today all major browsers optimize asm.js. Since it is a subset of JS, it can be executed on all JS engines, even if they do not improve performance. However, it means that extending asm.js with new features always requires extending JS first. Also asm.js can be written by hand, it is usually generated by a compiler.

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

(a) C code

```
function add(a, b) {  
    a = a | 0;  
    b = b | 0;  
    return (a + b) | 0;  
}
```

(b) asm.js code

Figure 2.2.: Integer add function written in C, compiled to asm.js. The bitwise OR suffix `| 0` does not modify the value, but is a type hint for the JS compiler to treat  $a$  and  $b$  as 32 bit signed integers.

## 2.4 Emscripten

Emscripten is a LLVM-to-web compiler mainly for compiling unmodified C and C++ applications to asm.js. In February 2017 Emscripten announced official stable WASM support as an alternative to asm.js. Since version 1.38.1, released in May 2018, WASM is emitted by default (see [emscripten.org](https://emscripten.org) 2019).

```
$ cat hello.c
#include<stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
$ emcc hello.c -o hello.js
$ ls
hello.c  hello.js  hello.wasm
$ node hello.js
Hello World
```

Figure 2.3.: Command line showing content of *hello.c*, a Hello World C program.

Code is compiled with emcc to JS and additional WASM file. Output files are listed in the current file system directory. Node.js executes the *hello.js*.

## 2.5 Node.js



Figure 2.4.: Node.js logo  
(source nodejs.org n.d.)

Node.js is a runtime environment for executing JS code outside browsers. The platform is built on Chrome's JS engine V8. Node.js uses an event-driven, non-blocking input/output (I/O) model that makes it lightweight and efficient. Node.js is mainly used for fast, scalable network applications. Due to JS' popularity and its performance improvements, JS became suitable for applications outside browsers. Node.js was first introduced in 2009 by Ryan Dahl. Today it is one of the most commonly used technologies by developers (see Stack Overflow 2019).

Because of JS, Node.js has some advantages for server-side programming over other technologies. The following advantages come from Cantelon et al. (2013). Developers can write client and server application in one language. Code can be shared between server and client. JSON is a popular data interchange format and is native to JS. Various NoSQL databases use JS, so interfacing is easy. JS can be a compilation target for a number of languages. Because V8 is kept up to date to the latest ECMAScript standard, Node.js can benefit from the newest features.

Through the usage of V8, Node.js can take advantages of asm.js (see Section 2.3) and WASM (see Section 2.6). Node.js is also the foundation of Electron, a framework for the development of cross-platform desktop graphical user interface (GUI) applications.



## 2.6 WebAssembly



Figure 2.5.: WebAssembly (WASM) logo  
(source [webassembly.org](https://webassembly.org) n.d.*d*)

### 2.6.1. Introduction

WebAssembly (WASM) is a safe, portable, low-level bytecode format designed for efficient execution and compact representation. WASM is intended to solve some issues of JS, what 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 has been published and all major browser vendors implemented it in their browsers (see McConnell 2017). WASM 1.0 is specified by the WASM Core Specification (WebAssembly Working Group 2019). WASM is an open and standardized bytecode designed to be embedded in different environment, also besides the web. Some of its core principals are performance, compactness, security and portability. The term WebAssembly is slightly misleading. Actually WASM is technically not an assembly language, it is a bytecode. And WASM is in no way bound to the web. WASM is designed to be platform independent from the start. It can be embedded in browsers, run on a standalone Virtual Machine (VM), or be integrated in other environments.

### 2.6.2. Prior Attempts

#### ActiveX

WASM is not the first attempt to bring near-native performance to the browser. Microsoft's ActiveX used code-signing of x86 binaries. It relied entirely upon code signing and therefore did not achieve safety through technical construction, but through a trust model (see Haas et al. 2017, section 1.1).

#### NaCl

NaCl is a technique that relies on x86 machine code. The used x86 code has to follow certain patterns for validation, such as bitmasks before memory accesses and jumps (see Haas et al. 2017, section 1.1). Because NaCl is a subset of x86 machine code, it is not portable to other machine architectures. PNaCl is the portable version of NaCl, using a subset of LLVM bytecode. But PNaCl still exposes compiler- or platform-specific details such as the layout of the call stack. Also the binary is not significantly smaller than JS code (see Haas et al. 2017, section 1.1). Chrome was the only browser supporting NaCl and PNaCl. In 2017 the Chrome team announced the deprecation of these techniques, in favor of WASM (see [chrome.com 2017](https://www.chromium.org/blink/depriorizing-nacl)).

#### asm.js

Another approach is asm.js, which is already introduced in Section 2.3.

### 2.6.3. Performance

WASM is a low-level bytecode, which executes with near-native code performance, by using capabilities of common modern hardware. Efficiency is one of WASM's core principals. The bytecode provides fast execution times, independent from language, hardware and platform. The code is also designed to be fast to transfers, easy to decode, easy to validate and easy to compile. All of these steps are

streamable and parallelizable. Most modern browsers already support streaming compilation, which compiles functions as soon as they arrive. There is no need to wait until the entire file is downloaded. This technique minimizes page load latency (see Haas et al. 2017, section 5). Section 3.1.5 shows how WASM’s performance is measured and compared to other technologies.

#### **2.6.4. Compactness**

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 JS is a plain text format, it cannot achieve such file sizes, even when minified and compressed (see Haas et al. 2017, section 5). Section 3.1.4 shows how binary size is compared to other binary formats.

#### **2.6.5. Security**

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 Application Programming Interfaces (APIs) provided by the host environment. Other web technologies like JS are using a managed language runtime and Garbage Collection (GC) to enforce memory safety (see Haas et al. 2017, section 1). But Garbage Collection (GC) impacts performance. That is why WASM’s safety does not depend on GC. Section 3.4 shows how WASM’s security is evaluated for this thesis.

#### **2.6.6. Portability**

Another important 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 Java Runtime Environment (JRE). Section 3.3 showcases how a single WASM module can be utilized in different environments.

### 2.6.7. Module Structure

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. To instantiate a module all its imports has to be satisfied.

### 2.6.8. Language Support

WASM version 1 (MVP) is ideal for low-level languages, with very little dynamic features at runtime. WASM does currently not provide any Garbage Collection (GC). 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, like Go, C#, Java, Python, etc. But they currently have to bundle their runtime features, like GC or exception handling into the binary, producing more bloated binaries. This might change with future version of WASM (see Section 2.6.13).

### 2.6.9. Conceptual Machine

WASM is an assembly language for a conceptual machine. The computational model of WASM is a stackmachine. Compared to a register machine, like x86 or ARM, stackmachines enables easier VM implementation, smaller binary encoding and faster single-pass code verification. The virtual Instruction Set Architecture (ISA) supports only types and operations that are common on current hardware. WASM has only four value types (see WebAssembly Working Group 2019, section 2.3.1):

- `i32`: 32 Bit integer type
- `i64`: 64 Bit integer type
- `f32`: 32 Bit IEEE 754 floating point type
- `f64`: 64 Bit IEEE 754 floating point type

More complex types can be formed from these basic types.

### 2.6.10. Binary and Text Format

WASM code has a binary and a textual encoding. Usually WASM is encoded in binary format. It is more compact and can be transferred and parsed more efficiently. The textual encoding, called WebAssembly Text Format (WAT), is a human readable format. WAT is used for debugging, inspecting or writing WASM code by hand. Both formats can be easily transcoded to the other format via tools.

```
$ wasm-objdump -d example.wasm
example.wasm:          file format wasm 0x1

Code Disassembly:

...
000215: 41 01      | i32.const 1
000217: 41 02      | i32.const 2
000219: 6a         | i32.add
...
```

Figure 2.6.: Object dump of WASM file. Addition of two constant 32 Bit integer numbers. Hexadecimal representation of binary format on the left. WAT format on the right.

```
(module
  (func (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add)
  (export "add" (func 0)))
```

Figure 2.7.: WASM module exporting an add function for integer values, represented in WAT format. Indexes are used to reference functions and values.

```
(module
  (func $add (param $a i32) (param $b i32) (result i32)
    (i32.add
      (local.get $a)
      (local.get $b)))
  (export "add" (func $add)))
```

Figure 2.8.: WASM module exporting an add function for integer values, represented in WAT format. For better readability, folded expressions and name references are used. Semantically equivalent to Figure 2.7.

### 2.6.11. Current State

In March 2017 WASM project reached version 1.0 MVP. The MVP sets the foundation for further features, which are in active development (see Section 2.6.13). Since December 2019 the WebAssembly Core Specification is an official W3C web standard (see W3C 2019). Today all major browsers support executing WASM.

1 out of 600 websites in the Alexa 1 million ranking use WASM (see Musch et al. 2019, section 7). Alexa Internet is a web traffic analysis company. Common applications include games in browsers, ported desktop programs, libraries to speed up certain tasks, but also hidden cryptocurrency miners (see Section 4.5).

Few examples that already benefit from WASM usage are the Google Earth application in browsers (see Chromium Blog 2019), the game engine Unity (see Trivellato 2018) and the Lichess chess server (see *stockfish.wasm* n.d.).

### 2.6.12. Implementations

All major browser vendors have developed independent implementations of WASM. V8, SpiderMonkey and JavaScriptCore, the JS engines of Chrome, Firefox and WebKit reuse their optimizing JIT compilers to compile WASM modules ahead-of-time before instantiation (see Haas et al. 2017, section 7). This results in predictable high performance. The Microsoft engine Chakra lazily translates individual functions to an intermediate format and later compiles the hottest functions (see Haas et al. 2017, section 7). This method approaches faster startup. 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.

Beside browsers there are also implementations for other environments (see Section 2.6.14). Node.js is based on V8, therefore WASM modules can be included in Node.js applications (see Section 2.5). Wasmer is a WASM runtime that can be integrated in different language environments (see Section 2.7.6). Via the WASI interface, it is possible to execute WASM standalone without the necessity of another language, which includes the WASM module. In Section 2.7.6 further standalone WASM runtime environments are presented.

### 2.6.13. Planned Features

WASM is in an early stage. The current WASM version 1.0 is a MVP that is missing a lot of planned features (see *webassembly.org* n.d.a). The goal is to make WASM a more attractive compilation target for high-level languages and use the full potential of modern hardware capabilities. Prototypes already exist for some features. This section presents some of the planned features.

## Feature Testing

An important feature for future version is Feature Testing. Applications should be able to check if a specific feature is supported by the runtime. Currently if a feature is not supported by the runtime engine the module validation fails. Feature Testing will allow applications to run on engines with different features and use them if supported.

## SIMD

Single Instruction, Multiple Data (SIMD) instructions are a class of instructions where a single instruction simultaneously performs the same operation on multiple data elements. Most modern hardware supports SIMD instructions. Audio, video and image processing software benefits from the improved computing performance. By adding SIMD instructions to the WASM architecture, WASM applications can take advantage of these hardware capabilities. SIMD support is already implemented in some runtimes as an experimental feature, e.g. WAVM.

## Threads

The WASM MVP is single-threaded. Modern computers have multiple Central Processing Unit (CPU)'s. Multi-threaded applications can divide intensive tasks among several threads, which can then be processed by the system simultaneously. There is already a feature proposal for WASM to make use of this faster processing method. WASM is going to get support for shared memory and atomic operations that allow threads to communicate via a shared linear memory. Some browsers already have experimental thread support by utilizing JS' Web Workers and SharedArrayBuffers.

## Exception Handling

WASM does not support throwing nor catching exceptions. Emscripten uses a workaround letting JS throw and catch those exceptions. However it increases



code size and slows down code execution, even if no exception is thrown (see [emscripten.org](https://emscripten.org/) n.d.). Emscripten recommends disabling exception catching, which results in termination of execution on any exceptions. Exceptions are an essential part of many programming languages, such as C++. WASM has to support exception handling to fully support those languages. There are multiple proposals on how to extend WASM to support exceptions.

### **Garbage Collection**

Currently all languages which use Garbage Collection (GC) and compile to WASM have to bundle and ship their GC as a part of the binary. This results in large modules and slower loading time. A proposal to make WASM a more attractive compilation target for higher-level languages is to allow WASM applications to use the GC provided by the host environment. For example, a WASM module written in Go that is embedded into a JS application could integrate with the GC of JS.

### **Interface Types**

Interface Types is a proposal describing high-level values, like string, sequences, records and variants, without committing to a single memory representation (see *WebAssembly Interface Types Proposal* n.d.). Different languages use different representation for similar types. WASM modules should be able to interoperate seamlessly with other modules and runtimes written in all kinds of languages. Currently WASM can only import and export functions transferring number types. The user of a WASM module must somehow know how types are represented and convert them manually. Without any external documentation of the interface the user cannot know what high-level type is meant because this information is erased at compile time. Interface Types could solve this problem, by adding interface annotations to modules containing complex types.

## **wasm64**

The WASM MVP's has an 32 bit address space and is therefore limited to 4 GB linear memory. To handle larger memory sizes, WASM will support 64 bit indices in the future.

### **2.6.14. Utilization in Non-Web Environments**

#### **Mobile and Desktop Applications**

JS is already popular programming language for creating mobile and desktop applications. Using JS beyond the web allows developments to use the same tools, libraries and code on the web and mobile/desktop platform. Same applies to WASM. Cross-platform JS frameworks like Electron can already embed and utilize WASM modules. But WASM modules cannot only be embedded in JS applications. Wasmer for example, allows embedding in a wide range of languages (see Section 2.7.6). With WebAssembly System Interface (WASI) standalone WASM applications can be created without the necessity to embed the module into another language's runtime system (see Section 2.7).

#### **Server and Cloud**

Node.js is already a very popular platforms for backend development, such as web services. WASM modules can be easily embedded in a Node.js server application (see Section 2.5). 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) cloud applications. Lucet provides fast module initialization and low memory overhead (see Section 2.7.6). WASM is sandboxed. Cloud applications benefit from the security by isolating applications from the host system and other applications. Therefore WASM

is also a lightweight alternative to containers and virtualization technologies.

### 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 (see Gurdeep Singh & Scholliers 2019, section 1). There are also various high-level programming languages available for writing microcontroller programs, e.g. Python and JS. The disadvantages of high-level languages are slower execution and limited peripheral support (see Gurdeep Singh & Scholliers 2019, section 1). WASM can be a middle ground between high-level and low-level languages (see Gurdeep Singh & Scholliers 2019, section 1). It could be used as a low-level hardware abstraction to enable writing portable and performant programs for embedded systems and Internet of Things (IoT) devices. There are several projects creating lightweight WASM interpreters and compilers for microcontrollers, such as WARduino (see Gurdeep Singh & Scholliers 2019), WAMR (see Bytecode Alliance n.d.b) and Wasm3 (see *Wasm3* n.d.).

### Blockchain

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 a vast variety of 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) by a new VM called Ethereum flavored WebAssembly (ewasm), a deterministic subset of WASM (see *Ethereum flavored WebAssembly (ewasm)* 2019). WASM is closer to actual hardware instructions than EVM bytecode, resulting in a more efficient code execution.

## Polyglot Programming

WASM is designed to be embedded into JS' runtime environment. Other languages and frameworks can also interoperate with WASM. Polyglot Programming is the approach of writing a 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 a early experimental phase. GraalVM announced GraalWasm, a WASM engine for GraalVM (see Salim et al. 2019). WASM modules will be able to interoperate with already supported GraalVM languages like JS, Python, Ruby, R, Java, C, etc. There are also more lightweight embedding efforts for many language environments. One ambitious project is Wasmer (see *Wasmer* 2019). Wasmer is a WASM runtime and can be used as a library to embed WASM code in a variety of languages. Wasmer currently supports Go, Rust, Python, Ruby, PHP, C, C++, C#, etc (see Section 2.7.6).

## 2.7 WebAssembly System Interface



Figure 2.9.: WebAssembly System Interface logo  
(source *WASI* 2019)

### 2.7.1. Introduction

By default WASM cannot talk to the 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, network sockets, etc. In March 2019 Mozilla announced WASI, which is planned to become the standard system interface for WASM (see Clark 2019b). 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 (see Bytecode Alliance n.d.a). 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 (Clark 2019b). The reason not to choose an existing system interface is to

perfectly fit WASM's key principals: portability and security. WASI allows WASM to talk to the system in a secure way. With WASI, WASM application can run standalone (see Figure 2.10). 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 2.7.6).

```
$ cat hello.rs
fn main() {
    println!("Hello World!");
}
$ rustc --target=wasm32-wasi hello.rs -o hello.wasm
$ ls
hello.rs  hello.wasm
$ wasmtime hello.wasm
Hello World
```

Figure 2.10.: Command line view content of *hello.rs*, a Hello World Rust program.

Code is compiled to the WASM-WASI target. Output file is listed in the current file system directory. Wasmtime runtime executes the standalone WASM module *hello.wasm*.

### 2.7.2. System Calls

The WASI Core API is much inspired by CloudABI and POSIX (see Bytecode Alliance 2019b). The WASI Core API is not yet finalized and there will be changes. WASI system calls include accessing command line arguments, accessing the system clock, reading environment variables, accessing the file system, exiting execution with an error code, getting random numbers and accessing network sockets.

### 2.7.3. Portability

WASM is a portable binary format. Once compiled WASM code should run on different kinds of machines. This is also a design goal for WASI. But different operating systems have different system calls, e.g. Windows uses the Windows

API and Mac or Linux uses POSIX. That is why WASI defines an abstract system interface for a conceptual operating system. The WASM runtime environment has to translate the WASI calls to the real system calls. WASI is designed to be similar to existing system interfaces, to keep translation simple. If the runtime environment itself has no access to system resources, resources can be simulated. For example, browsers usually can not access files from the host system, but JS can provide an in-memory file system for the WASM instance to operate on. Section 3.3 showcases how WASM code can be utilized in different environment.

#### **2.7.4. Security**

The web has very strict security policies. Untrusted code should not be able to do any harm to the user or host system. That is the reason why the web is sandboxed. The browser can limit what an application can do. Most system interfaces other than WASI interacts with the system on behalf of the user. A regular system program executed by a user can do anything the user can. This only make sense if you trust your software, but modern systems use a lot of third party code of unknown trustworthiness. Third-party libraries talking directly to your system can be dangerous. For this reason WASI uses capability-based security. It is designed to limit what a program can do on a program and even module basis, not on user basis. A program should get the minimal permissions necessary for its task to perform. For example a text editor only needs permissions for the file to edit. Accessing other files, opening network connections, etc. must be prevented by the WASM runtime. Section 3.4 shows how WASM's security is evaluated for this thesis.

#### **2.7.5. Current State**

WASI is not yet finalized and it still evolves. It should eventually become the standard API for all types of systems. WASI already supports a list of system calls (see Section 2.7.2). Although WASI is currently capable of receiving data, sending

data and closing network sockets, network connections cannot be opened via WASI. Also files cannot be locked, or monitored for changes. These are some of the features that might be added to the interface (see Clark 2019b). The current WASI interface defines the core features every system can provide. More interfaces will be added to support more specific environment features. Even though WASI is still in an early stage, there are already some standalone and embedded implementations (see Section 2.7.6).

### 2.7.6. Implementations

#### **Wasmtime**

Wasmtime is a standalone non-web runtime for WASM-WASI (see *Wasmtime* 2019). This is a project of the Bytecode Alliance. Wasmtime can be used as a command-line utility or as 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**

WAVM is a WASM VM, designed for use in non-web applications (see *WAVM* 2019). 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 further 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**

Wasmer is a standalone WASM runtime outside browsers (see *Wasmer* 2019). 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 in browsers. Wasmer can also be used as an extension for PostgreSQL databases. WASM functions can be called within the PL/pgSQL language.

### **Lucet**

Lucet is a platform for executing WASM-WASI modules in a FaaS cloud infrastructure (see *Lucet* 2019). A major 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 (2019) claims Lucet can instantiate WASM modules in under 50 microseconds and handle tens of thousands of requests per second in a single system process. Lucet is currently use in an experimental platform of the cloud service provider Fastly.

### **Wasm3**

Wasm3 is a high performance WASM interpreter (see *Wasm3* n.d.). The runtime supports the WASI interface. Wasm3 can be used on a great variety of system architectures, OSs, Single-Board Computers (SBCs), microcontrollers and browsers. It runs as a standalone runtime and can be used as a library for C, C++, Go and Rust. 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.

# Chapter 3

## Method

All code and data used for this thesis is available at Spies (2020). All measurements done in this section depend on the used hardware and software versions. A listing of all specifications can be found in Appendix B. All tests are performed in similar conditions on the same hardware. Different hardware and versions might lead to slightly different results. Nevertheless the following measurements should represent the technologies quite well.

### 3.1 Benchmarking WebAssembly

This section benchmarks WASM code. Asm.js and native x86-64 code are used as reference points for comparison. In some real-world scenarios it might make sense to replace native binaries by WASM binaries in order to achieve benefits in portability and security. WASM can also be a higher-performing alternative or addition for highly dynamic, memory safe and sandboxed languages such as JS. Instead of using standard JS, the following benchmarks use the better performing asm.js subset. This section benchmarks code size, startup and execution time of the different technologies. Obviously the used system hardware, OS, compilers, runtime environments, algorithms and its versions define the benchmark results (see

Appendix B). Furthermore WASM is a new technology especially as a standalone WASI executable. It is most likely that the benchmark score alter as the technology consolidates. Therefore the following benchmarks are an evaluation of a variety of currently available and commonly used tools.

### 3.1.1. Choosing Benchmark Suite

PolyBenchC is the used benchmark suite. PolyBenchC is a benchmark suite of 30 numerical computations from various application domains like, linear algebra, image processing, physics simulation, dynamic programming and statistics (see Pouchet et al. 2016). PolyBenchC was chosen because it contained common algorithms which are used in many software applications. PolyBenchC benchmarks the single thread computational performance, which is what the WASM MVP is designed for.

### 3.1.2. Choosing Runtime Environments

Native binaries are executed in a Linux environment (see Appendix B). Asm.js is executed on the Node.js platform in V8, the most used JS engine. V8 is also the JS engine of Google Chrome. The second asm.js execution environment is SpiderMonkey, the engine from Firefox browser. Node.js and SpiderMonkey are additionally used to run WASM code. To execute WASM standalone, three WASI supporting runtimes are chosen: Wasmtime, Wasmer and WAVM.

### 3.1.3. Compiling Benchmarks

WASM code for JS embedding and asm.js code is generated by Emscripten. Native x86-64 code and WASM-WASI code is generated by clang. Used compilers are called with similar arguments to keep them as comparable as possible. Figure 3.1 shows what exact commands are used for compiling. Appendix B shows the used compiler versions.

```
$ clang -O3 -s -flto --target=x86_64-pc-linux-gnu \  
-DPOLYBENCH_TIME -lm <source>
```

(a) Compiling with clang to x86-64

```
$ emcc -O3 -flto -DPOLYBENCH_TIME -s WASM=0 \  
-s ALLOW_MEMORY_GROWTH=1 --memory-init-file 0 <source>
```

(b) Compiling with emcc to asm.js

```
$ emcc -O3 -flto -DPOLYBENCH_TIME -s WASM=1 \  
-s ALLOW_MEMORY_GROWTH=1 <source>
```

(c) Compiling with emcc to WASM. The command emits a WASM file and also a JS file that loads and executes the WASM code.

```
$ clang -O3 -s -flto --target=wasm32-unknown-wasi \  
--sysroot /opt/wasi-libc -DPOLYBENCH_TIME <source>
```

(d) Compiling with clang to WASM with WASI support

Figure 3.1.: Compiling PolyBenchC benchmark sources to different targets

### 3.1.4. Binary Size Measurement

Executable size is not the most important for non-web applications. Nevertheless it may make some difference if fast deployment, e.g. to a server, is a requirement or little storage is available, like on microcontrollers. But binary size is especially important for web applications. Fast file transfers and fast file validation is important for a snappy web experience. On the server-side, e.g. Node.js, size is not nearly as essential as in browsers. 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 to WASM. It produces two files, one WASM file

and a JS file to load and execute the WASM code. For the Emscripten WASM version, the sum of the two file sizes is considered for comparison.

All modern browsers support HTTP Content-Encoding. Files are not transferred in plaintext. The HTTP content is compressed to speedup transfer and save bandwidth. A common compression format used by browsers is gzip. Therefore this thesis also compares the compressed file size and the compression rate of the code formats. Gzip's compression level can be adjusted. For our test compression level 9 is used, which is the highest compression level. 9 is a common default value and is used for example by Apache HTTP servers.

### 3.1.5. Execution Time Measurement

This section describes the execution time measurements of x86-64, asm.js and WASM code. One design goal of WASM is being a faster and more predictable alternative to JS. Reason number one to use native code is computing performance. Compiling and optimizing system languages code like C, C++ and Rust directly for the desired architecture should produce the highest performing code. This measurements are used to evaluate the performance costs of security and portability compared to native code. And they are used to determine the performance gain compared to JS code. To execute WASM code, 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. Startup time measurement is described in Section 3.1.6. All PolyBenchC benchmarks are compiled with the *POLYBENCH\_TIME* macro definition (see Section 3.1.3). This option makes every PolyBenchC executable output its execution time before exit. Figure 3.2 shows what commands are used to run the benchmarks in the different runtime environments. All PolyBenchC benchmarks are repeated 15 times to achieve sufficient accuracy. This

thesis also compares the benchmark results with Haas et al. (2017), which also uses PolyBenchC for benchmarking WASM.

```
$ <benchmark>
```

(a) x86-64

```
$ node <benchmark>.js
```

(b) Node.js

```
$ spidermonkey <benchmark>.js
```

(c) SpiderMonkey

```
$ wasmtime --disable-cache <benchmark>.wasm
```

(d) Wasmtime

```
$ wavm run --nocache <benchmark>.wasm
```

(e) WAVM

```
$ wasmer run --disable-cache <benchmark>.wasm
```

(f) Wasmer

Figure 3.2.: Running PolyBenchC benchmarks in different runtime environments

### 3.1.6. Startup Time Measurement

Besides execution time, startup time is the main reason to use native binary applications. Web applications also demand fast instantiation times to provide good user experience. Like all other non-native code execution, WASM requires additional startup steps, e.g. VM startup, code validation, code compilation and

code optimization. This section describes the startup time measurements of x86-64, asm.js and WASM code in different runtime environments.

For JS all major browsers use a combination of 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 longer startup phase compared to asm.js.

To measure the startup time the PolyBenchC benchmarks are modified to stop the process directly after the main function entry. The startup time is determined by measuring the process time. The startup time should vary between benchmarks, because more code needs more time for validation and optimization. And it should vary between runtimes because they perform different task at startup. All examined WASI runtimes can cache already compiled and optimized native code. For this tests caching is disabled to observe the behavior of a first execution. The time measurement is repeated 15 times, until sufficient accuracy is achieved to make a meaningful statement. This test uses the same commands from Figure 3.2 with the additional argument *initonly* (see Figure 3.3).

```
$ <runtime> <benchmark> -- initonly
```

Figure 3.3.: Running benchmark with argument *initonly* to exit process immediately after main function entry

## 3.2 Determining Portability

WASM is designed to run on a variety of environments, on different OSs, ISAs and embedded in different programming languages. This thesis analyses the portability of WASM by showing the requirement for an execution environment to be able to run WASM code. In addition, the currently available portability options are determined by reviewing WASM runtime environments available on the internet. The portability of a WASM module using the WASI interface is showcased in Section 3.3.



### 3.3 Utilizing the Same WebAssembly Binary in Different Environments

This test aims to showcase the portability of WASM code by creating one WASM binary that can be deployed to different environments. The module should also access files through the WASI interface. WASI is also necessary for standalone execution. The following implementations and environments are chosen for the showcase:

- standalone WASM command line application, executable by WASI runtimes
- browser application with offline data processing, WASM embedded in the JS environment
- Node.js web service with server-side data processing, WASM embedded in the JS environment

There are already a few standalone WASI runtimes that can be used for the command line application (see Section 2.7.6). To embed a WASI module in another languages runtime, the host environment has to provide a WASM runtime and an implementation for the WASI interface. The WASI system call implementations have to be imported in order to instantiate the WASM module. The Wasmer-JS package provides a WASI implementation for Node.js and browsers. Embedding into more web-distant environments would not be very convenient at the moment, because the availability of WASI libraries is not that great at its young stage.

#### 3.3.1. Choosing an Algorithm and Tools

Gzip is a software application for file compression and decompression. Gzip consists of two algorithms, deflate for compression and inflate for decompression. Gzip was chosen because it is a common application in many areas. Gzip mainly relies on numerical calculations, which is an ideal use case for WASM.

Embedded in JS applications benefit from WASM's faster execution times. A command line application benefit from WASM's portability, easy distribution and secure sandbox. The implementation provides file system access through WASI. This gzip implementation is more secure than a typical command line application because it can only access the files necessary for the task.

As a language Rust is chosen. Besides C and C++ it can use the full potential of current WASM, by having a minimal runtime. Rust's standard tools Cargo and rustup pave the way for implementing and compiling.

The implementation created for this thesis is called wzip, a gzip implementation that can be compiled to WASM-WASI. See Figure 3.4 how wzip can be used.

```
$ wzip --help
wzip 1.0.0
Benedikt Spies
gzip implemented in Rust, able to be utilized in various
environments

If neither -d nor -c flag is set, wzip evaluates the MIME type of
the input file.

USAGE:
    wzip [FLAGS] [OPTIONS]

FLAGS:
    -c, --compress      Compress
    -d, --decompress    Decompress
    -h, --help          Prints help information
    -V, --version       Prints version information

OPTIONS:
    -i, --input <FILE>    Input file. Default stdin.
    -o, --output <FILE>  Output file. Default stdout.
```

Figure 3.4.: Usage of wzip, a gzip Rust implementation

### 3.3.2. Compiling Implementation to WebAssembly

Rust's standard tool `rustup` is used to install the latest `wasm32-wasi` cross-compiler (see Figure 3.5). The build system Cargo can now target the `wasm32-wasi` platform. With the command in Figure 3.6, Cargo creates a single WASM binary file named `wzip.wasm`. The `wzip.wasm` file contains the whole application. This exact file should be used in all environments mentioned in Section 3.3.

```
$ rustup target add wasm32-wasi
```

Figure 3.5.: Using `rustup` to install target platform for WASM-WASI

```
$ cargo build --target wasm32-wasi
```

Figure 3.6.: Using Cargo to compile code to WASM-WASI target platform

### 3.3.3. Running on Command Line

The WASI interface allows WASM modules to be executed as a standalone application. There are multiple runtimes supporting WASI (see Section 2.7.6). On the command line, the application can read directly from `stdin` and write to `stdout` of the system (see Figure 3.7). `Wzip` can also directly read from files and write files. By default WASI runtimes prevent access to the host file system. In order for it to work, `Wasmtime` must be instructed by the `--dir` option to grant the WASM application access to the parent directory (see Figure 3.8). `Wzip` can now only access files from the this directory.

```
$ wasmtime wzip.wasm < hello.gz  
Hello World!
```

Figure 3.7.: Running `wzip` on command line using the `Wasmtime` runtime, reading from `stdin` and writing to `stdout`

```
$ wasmtime --dir=./ wzip.wasm -- -i ./hello.gz -o ./hello.txt
$ cat ./hello.txt
Hello World!
```

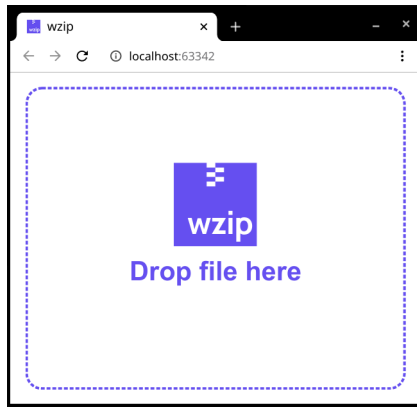
Figure 3.8.: Running wzip on command line using the Wasmtime runtime, reading from file and writing to file

### 3.3.4. Running in Browser

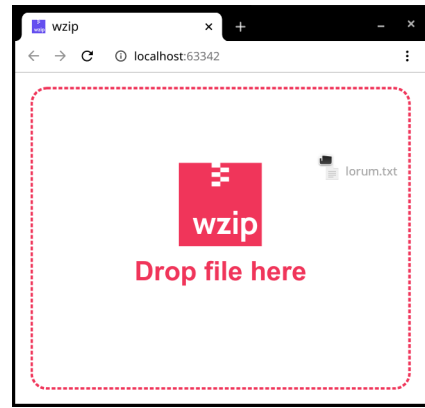
This section shows how the exact same binary *wzip.wasm* from Section 3.3.2 is used in the browser. WASM modules have to be loaded by JS. By default browsers do not support WASI. To instantiate wzip, all used WASI API imports has to be satisfied (see WASI API at Bytecode Alliance (2019b)). Wasmer-JS (see Section 2.7.6) is used to provide a WASI implementation. The file system cannot be accessed from browser's JS sandbox. Wzip requires a file system to operate on. Wasmer-JS provides a library called wasmfs which allows to create an in-memory file system. Input and output is temporarily stored as files in the in-memory file system. See full code at Spies (2020). A simple user interface is created with HTML and JS. This browser application can now compress and decompress files by dragging and dropping files in the browser window (see Figure 3.10). This application does not require a server, except for transferring the application itself. After downloading the HTML, JS and WASM files, the application can be used offline.

```
...
let module = await WebAssembly.compileStreaming(fetch('wzip.wasm'));
let fs = new WasmFs().fs;
let wasi = new WASI({
  args: ['wzip'].concat(args),
  bindings: {
    ...WASI.defaultBindings,
    fs: fs
  }
});
let instance = await WebAssembly.instantiate(module, {
  wasi_unstable: wasi.wasiImport
});
...
```

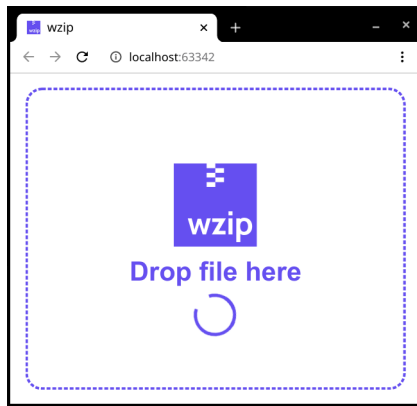
Figure 3.9.: Instantiation of wzip in browser’s JS using Wasmer-JS. Full source code at Spies (2020).



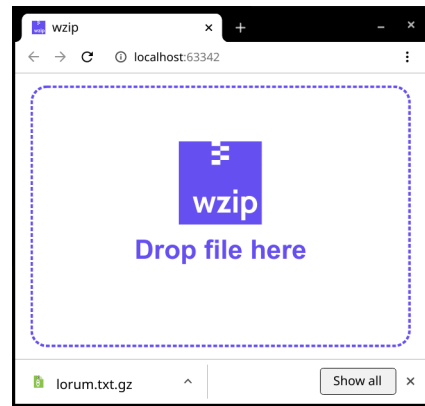
(a) Idle state



(b) Drag and drop file



(c) Compressing file



(d) Downloading compressed file

Figure 3.10.: Compressing file with wzip running in browser. User can drag and drop files to automatically inflate or deflate them.

### 3.3.5. Running in Node.js

This section describes how wzip can be used on the server-side with Node.js. Like browsers, Node.js does not implement WASI by default. Wasmer-JS is also a suitable WASI implementation here. The in-memory file system wasmfs temporarily stores the HTTP request body and output of wzip. Alternatively, it could also be implemented so that Node.js stores the temporary data in the real host file system. The instantiation of the WASM module is similar to the browser approach in Section 3.3.4. See full code at Spies (2020). After compression

or decompression, the output is sent back to the client in the HTTP response body.

```
$ node index.js  
Listening on port 3000
```

Figure 3.11.: Starting Node.js wzip web service from command line

```
$ curl --data-binary "@hello.gz" -X POST http://localhost:3000  
Hello World!
```

Figure 3.12.: Calling the wzip web service from command line to decompress a file

## 3.4 Evaluating Security

Security is assessed through the analysis of these publications: Disselkoen et al. (2019), McFadden et al. (2018), Haas et al. (2017), McFadden et al. (2018) and WebAssembly Working Group (2019). This analysis should identify benefits and possible vulnerabilities, resulting from the use of WASM. The security is compared to other code formats and languages, primarily native code and JS. In addition, this thesis takes a look at the security concepts used in WASI and compares them to the native and JS platform.



# Chapter 4

## Analysis

### 4.1 Binary Size

#### 4.1.1. Uncompressed

Binary size is measured as described in Section 3.1.4. The size of every PolyBenchC benchmark measurement can be found in Table A.3 and Figure 4.2. For a general overview Figure 4.1 shows the geometric mean of all 30 benchmarks per compilation target. All 30 benchmarks from PolyBenchC are compiled to the four target platforms: x86-64, asm.js, standalone WASM-WASI, and WASM from emcc to be embedded in the JS context.

If one takes a look at Table A.3 and Figure 4.2, the first thing that stands out is that all benchmarks of a compilation target have almost the same file size. This is because the algorithms that performs the heavy calculation consists of only a few lines of code. The major part is included libraries used by all benchmarks like *stdio.h*, *unistd.h*, *math.h*, or the PolyBenchC specific *polybench.h*. Therefore it is sufficient to consider the geometric mean of each compilation target.

The average x86-64 binary is 14 KB. The clang WASM binary is 36 KB. The emcc

WASM binary is 39 KB (including JS glue code). And the average asm.js code is 66 KB.

Comparing the x86-64 and the clang WASM binary versions, the WASM files are on average 246 % of the native file. Comparing the minified emcc asm.js and the emcc WASM code, the WASM version is on average 58 % of the asm.js file. The measurements show that the WASM is roughly in the middle between x86-64 and asm.js code. A better statement could be derived with more diverse source codes.

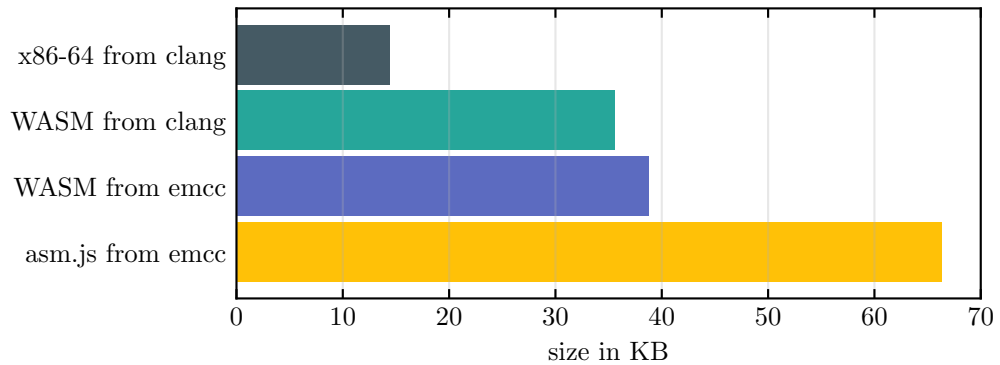


Figure 4.1.: Binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Geometric mean of 30 benchmark binaries per compile target. Data from Table A.3.



Figure 4.2.: Binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Data from Table A.3.

### 4.1.2. Compressed

The gzip compressed size of every PolyBenchC benchmark can be found in Table A.4 and Figure 4.4. For a general overview Figure 4.3 shows the geometric mean of all 30 benchmarks per compilation target.

The average gzip compressed version of asm.js is 23 KB. The compressed x86-64 code is 3.7 KB. The compressed clang WASM is 14 KB. And the compressed emcc WASM is 16 KB.

The best average compression ratio has x86-64 with a ratio of 3.96 : 1. The compression ratio of asm.js benchmarks is 2.87 : 1. The compression ratio of clang WASM files is 2.59:1. And the compression ratio of emcc WASM files 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.

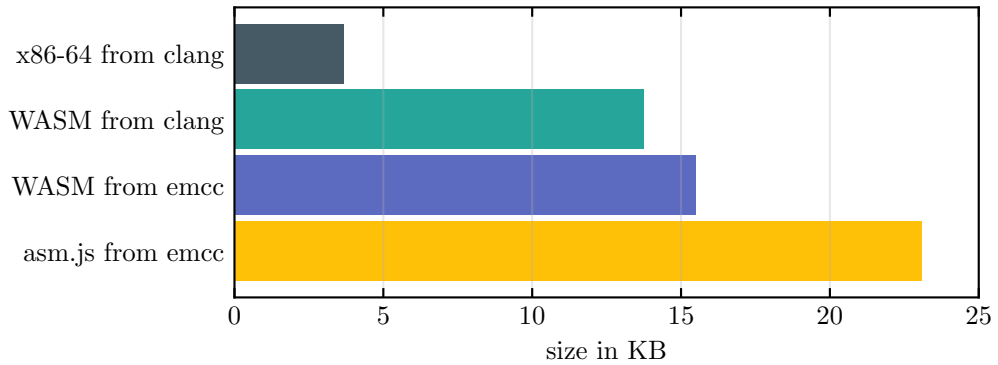


Figure 4.3.: Compressed binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Gzip is used for compression. Geometric mean of 30 benchmark binaries per compile target. Data from Table A.4.

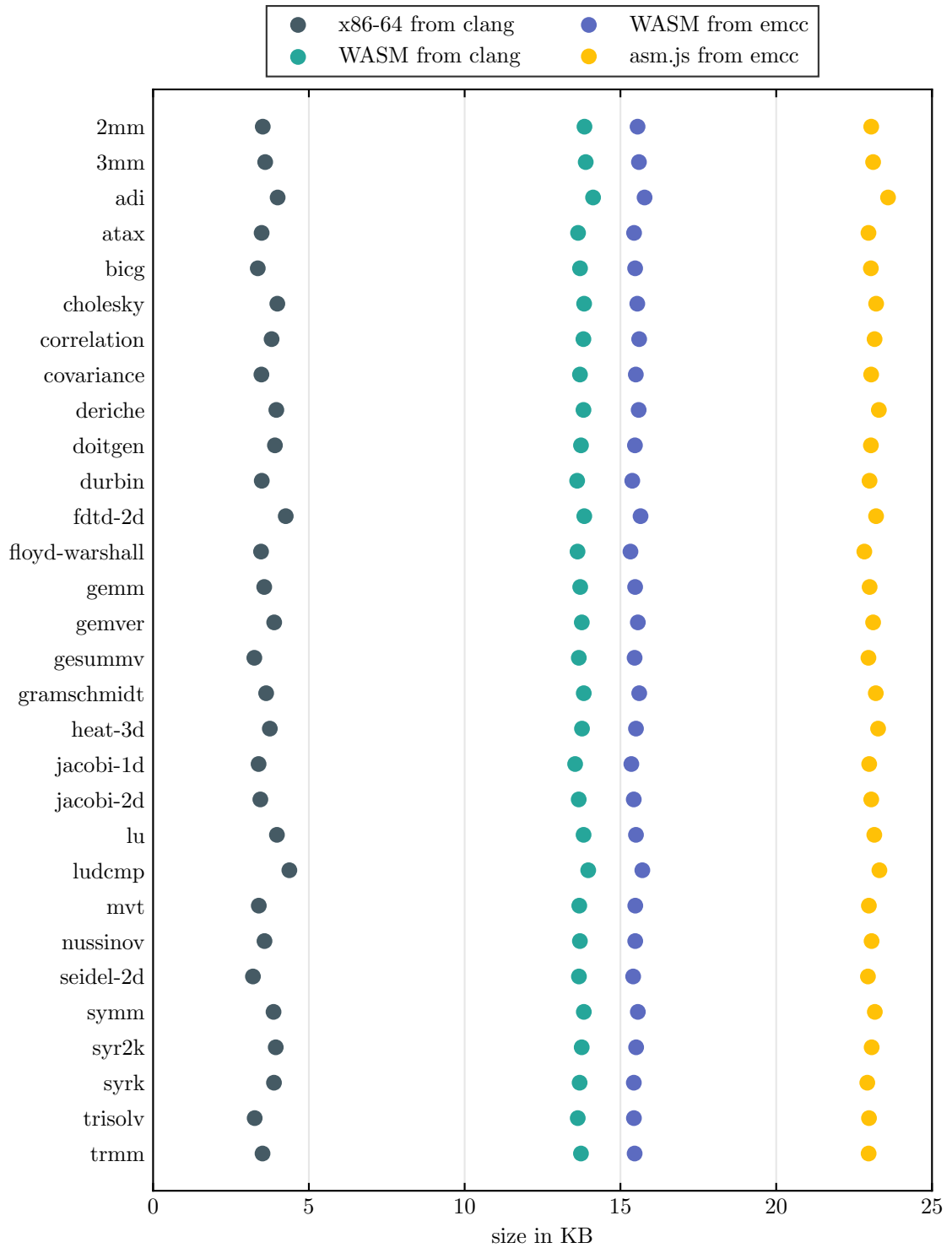


Figure 4.4.: Compressed binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Gzip is used for compression. Data from Table A.4.

## 4.2 Execution Time

Execution time is measured as described in Section 3.1.5. The times of every individual measurement can be found at Spies (2020). The arithmetic mean of 15 runs per benchmark can be found in Table A.1. For a general overview Figure 4.5 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, Figure 4.5 shows that execution time can vary greatly for different WASM runtimes. The use of WASM code does not necessarily mean similar performance. It heavily depends 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 values. The Wasmtime execution time is on average 345 % of the native x86-64 time. This difference correlates with the startup time in Section 4.3. 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 (see Section 4.3). 18 of 30 WAVM benchmarks are within 110 % of native, 25 are within 150 % and almost all are within  $2 \times$  x86-64 execution time (see Figure 4.6). The only exception is the *jacobi-1d* benchmark, with 237 % of x86-64 execution time (see Figure 4.6).

From Figure 4.8 WASM in some benchmarks appears faster than the x86 code, but due to inaccuracy of the measurements and execution time variance. The 95 % confidence intervals overlap and therefore the values are not significant enough to draw any conclusion. Only two benchmarks, *gesummv* and *trisolv*, can achieve better performance with WASM (see Figure 4.8). *Gesummv* and *trisolv* are one of the shortest benchmarks and the difference in speed is less than 1 ms (see Table A.1). Although this is not the most significant proof, WASM runtimes can

perform better than Ahead-of-Time (AOT) compiled native code by using better JIT optimizations and generate code for the exact machine that executes the code, not just for a generic x86-64 machine.

Node.js (asm.js) on average is 195 % of native time and SpiderMonkey (asm.js) on average is 264 % of native time.

After WAVM, Node.js and SpiderMonkey achieve the next best WASM performance. On average Node.js (WASM) is 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 (see Figure 4.7). Exceptions are *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. This is a significant performance improvement. Haas et al. (2017) comes to the conclusion that WASM is 33.7 % faster than asm.js. The conclusion from our tests is that on geometric mean WASM is 45.0 % faster than asm.js. The percentages are not directly comparable because Haas et al. (2017) uses different methods and also includes startup time. The most extreme improvement can be observed in SpiderMonkey with the *heat-3d* algorithm, it takes on average 30.0 s as asm.js and only 4.97 s as WASM code. This is a  $6.06 \times$  improvement.

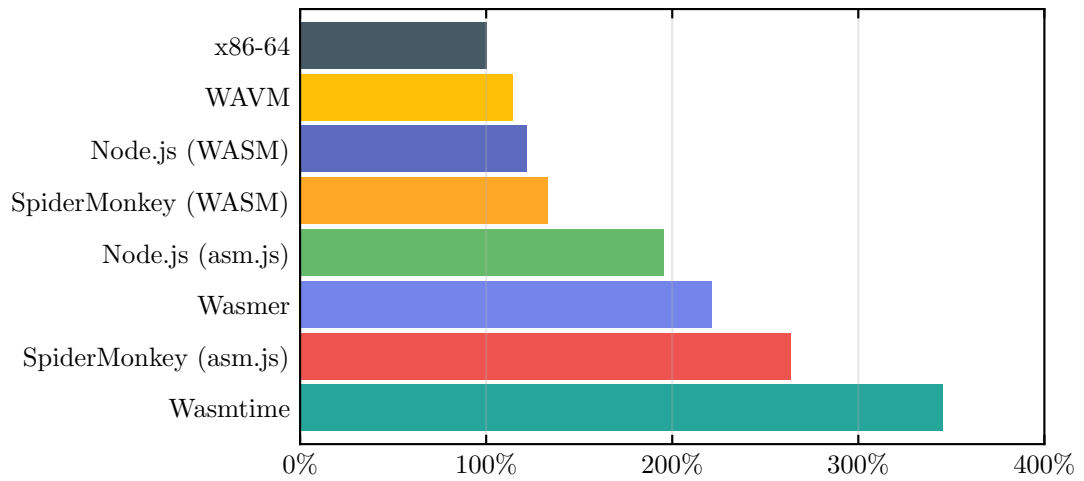


Figure 4.5.: Execution time of PolyBenchC benchmarks on different runtime environments. Values are normalized to native x86-64 execution time. Arithmetic means of 15 runs per benchmark. Geometric mean of all 30 benchmarks per runtime environment. Low values are better. Data from Table A.1.



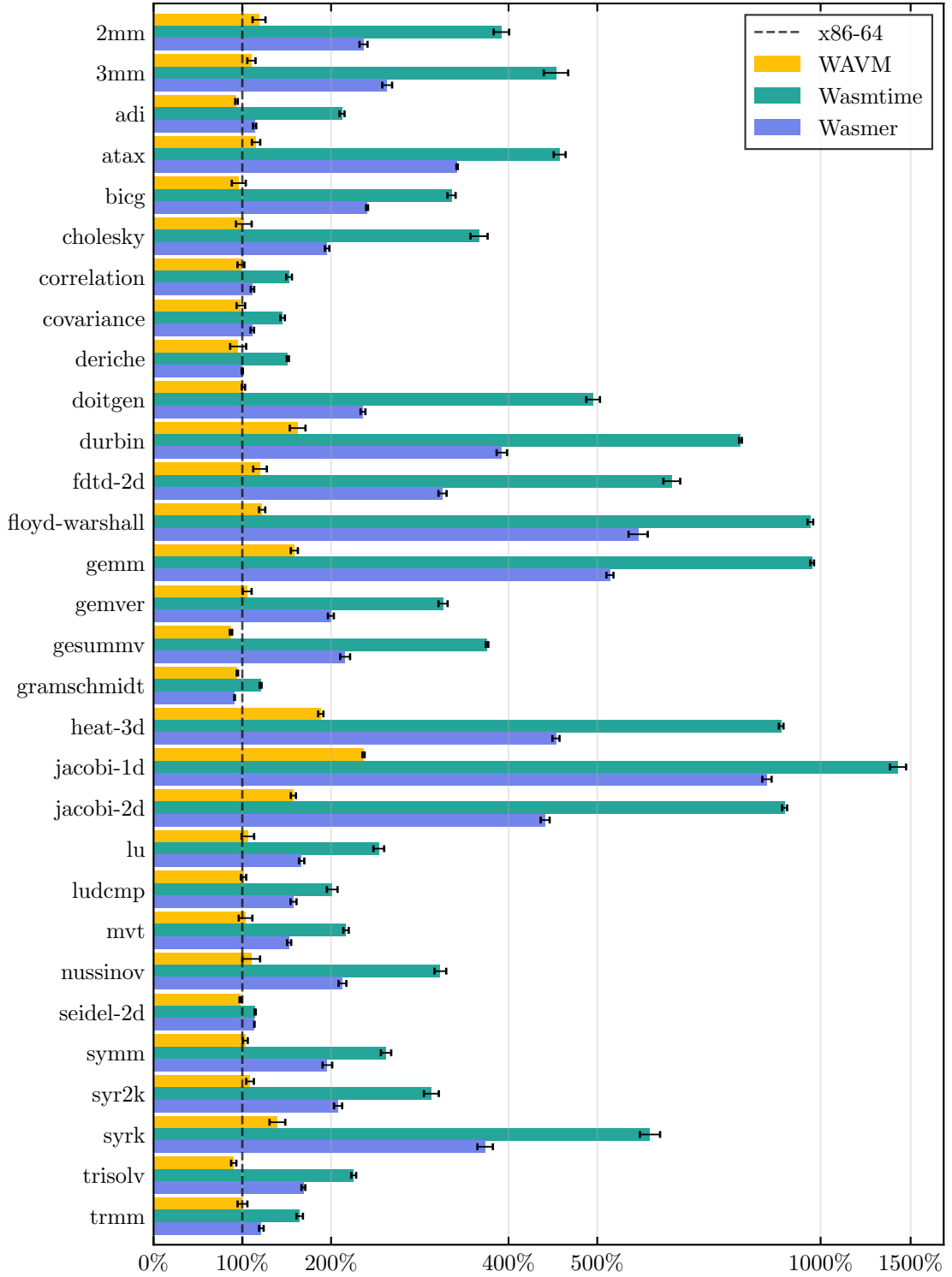


Figure 4.6.: Execution time of PolyBenchC benchmarks on WASI runtimes. Values are normalized to native x86-64 execution time and plotted on a logarithmic scale. Arithmetic means of 15 runs and 95 % confidence intervals. Low values are better. Data from Table A.1.

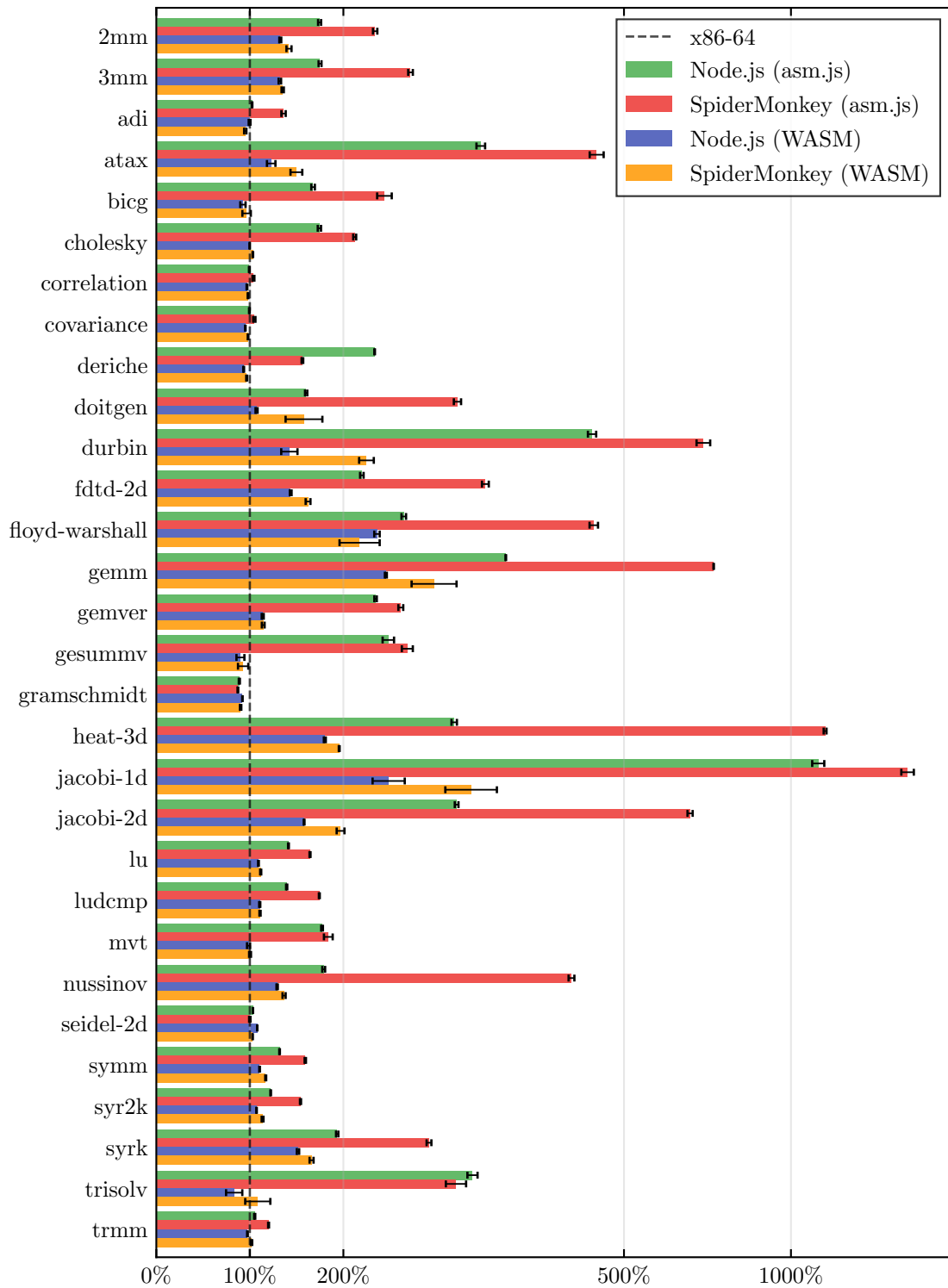


Figure 4.7.: Execution time of PolyBenchC benchmarks compiled to asm.js and WASM code. Values are normalized to native execution time and plotted on a logarithmic scale. Arithmetic means of 15 runs and 95% confidence intervals. Low values are better. Data from Table A.1.

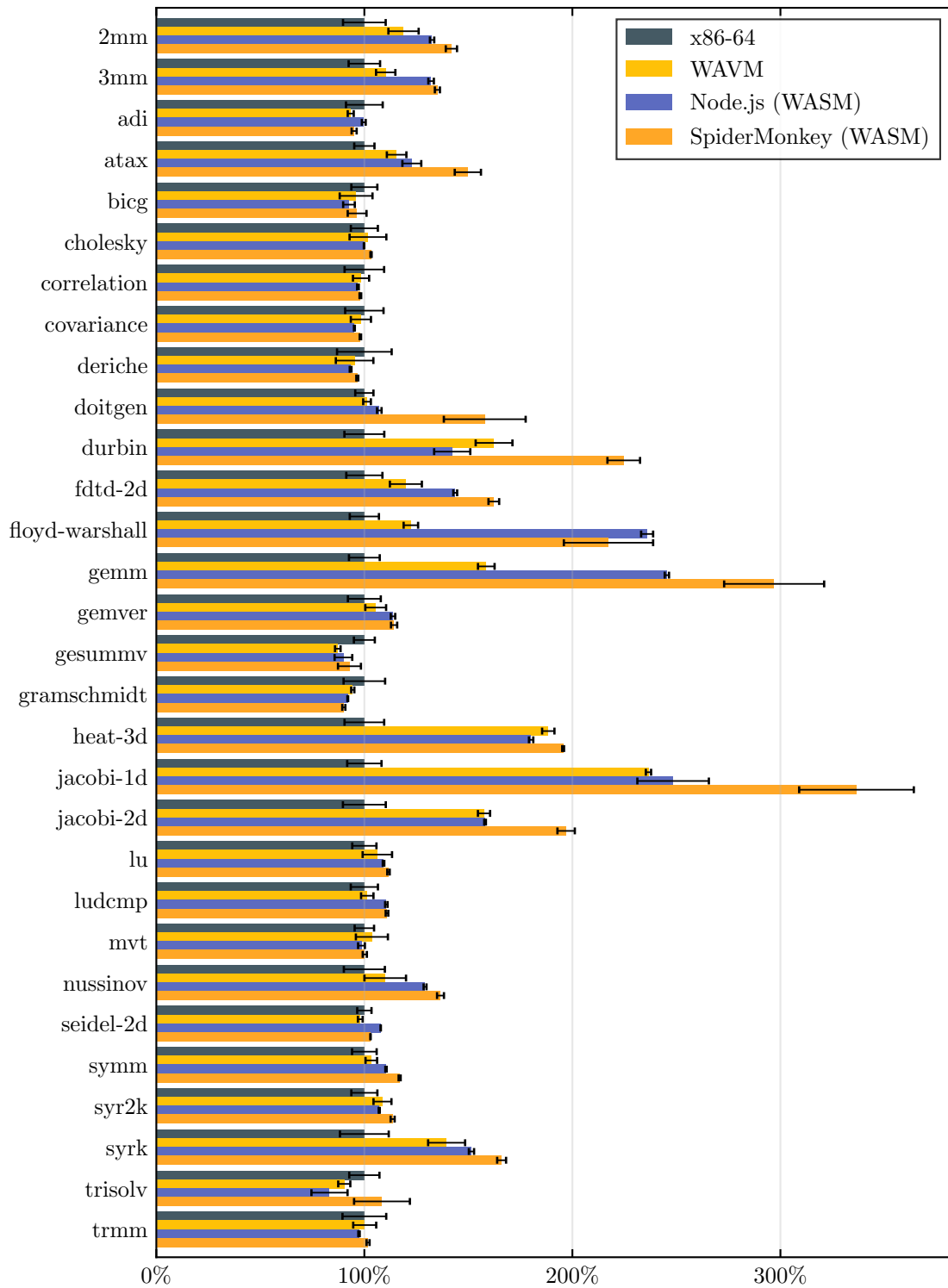


Figure 4.8.: Comparison of the top 4 runtimes with the best PolyBenchC execution times. Values are normalized to native x86-64 execution time. Arithmetic means of 15 runs and 95 % confidence intervals. Small values are better. Data from Table A.1.

### 4.3 Startup Time

Startup time is measured as described in Section 3.1.6. The times of every individual measurement can be found at Spies (2020). The arithmetic mean of 15 runs per benchmark can be found in Table A.2. For a general overview Figure 4.9 shows the geometric mean of all benchmark measurements per runtime environment.

An average x86-64 PolyBenchC startup takes 4.2 ms. The compared runtimes are at least  $7 \times$  slower. Starting up JS and WASM code requires additional steps. First of all, the runtime itself has to be started and the benchmark code has to be loaded, including steps such as validation, interpretation, JIT compilation and optimization.

Figure 4.10 and Figure 4.11 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 what is faster (see Figure 4.11). On geometric mean SpiderMonkey (WASM) starts up in 33.3 ms and Node.js (asm.js) in 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. In SpiderMonkey the average asm.js startup is 2.8 ms (8.4 %) faster than WASM.

All three benchmarked WASI standalone runtimes start slower than the JS embedding runtime. The average Wasmer startup time is 62.5 ms. 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 (see Figure 4.10). But WAVM is the fastest WASM runtime in terms of execution time (see Section 4.2). WAVM takes more time for compiling and optimizing and therefore reaches the best execution

time.

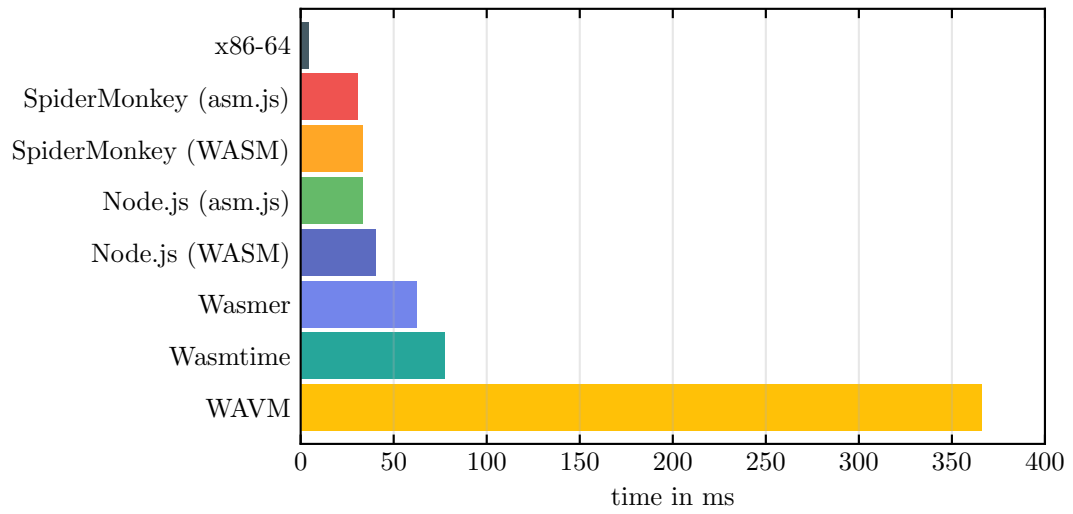


Figure 4.9.: Startup time (in milliseconds) of PolyBenchC benchmarks on different runtime environments. Arithmetic means of 15 runs per benchmark. Geometric mean of all 30 benchmarks per runtime environment. Low values are better. Data from Table A.2.

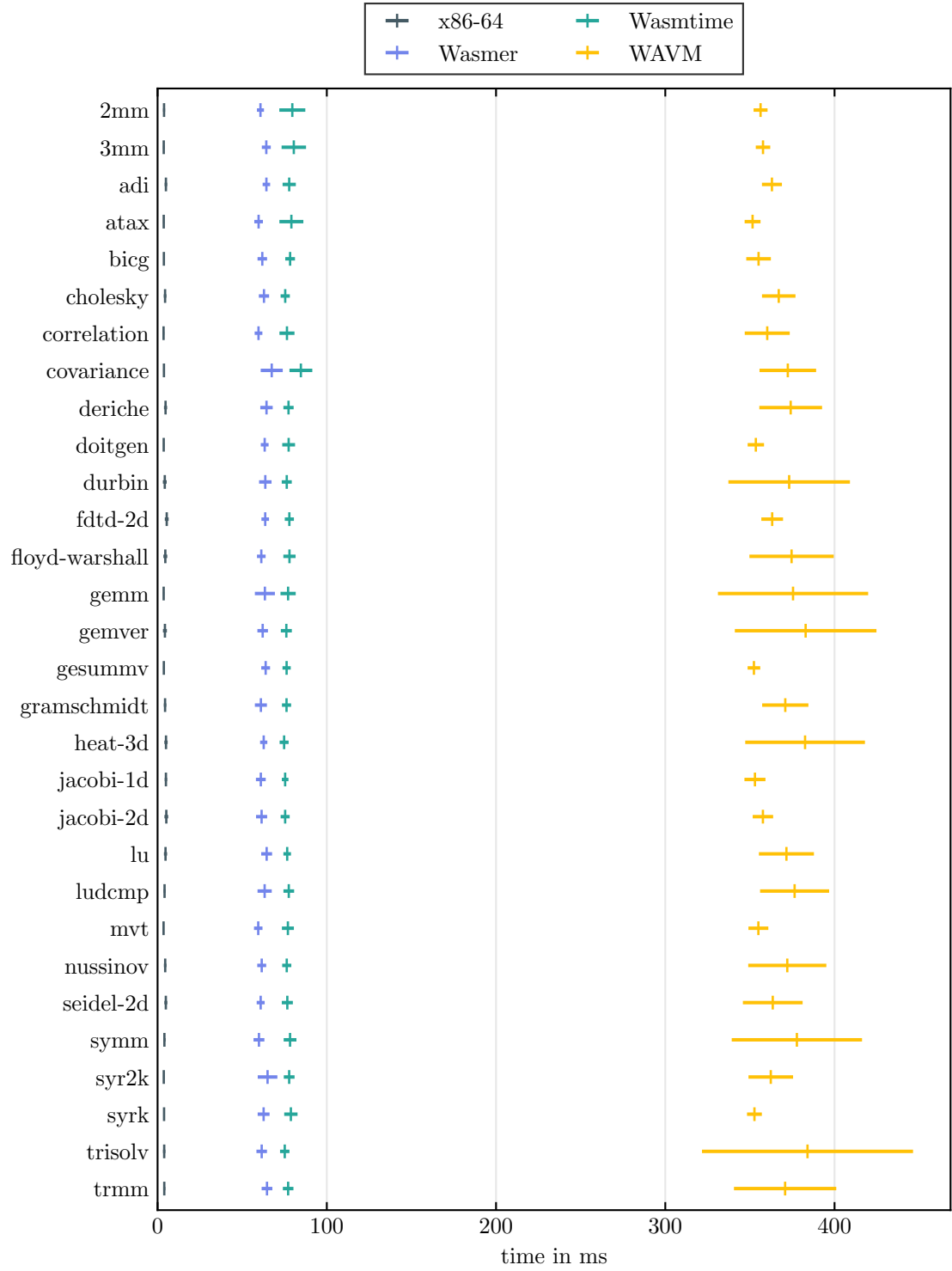


Figure 4.10.: Comparison of PolyBenchC benchmark startup time of native x86-64 and WASI standalone runtimes. Arithmetic means of 15 startups and 95 % confidence intervals. Low values are better. Data from Table A.2



Figure 4.11.: Comparison of PolyBenchC benchmark startup time of native x86-64, asm.js and WASM embedded in JS environments. Arithmetic means of 15 startups and 95 % confidence intervals. Low values are better. Data from Table A.2

## 4.4 Portability

In order to run WASM modules the runtime has to offer deterministic behavior as specified in WebAssembly Working Group (2019). The WASM specification allows limited nondeterminism for some operations as a compromise, e.g. to achieve native performance (see [webassembly.org](https://webassembly.org) n.d.b). 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 (see [webassembly.org](https://webassembly.org) n.d.c). 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 (see [webassembly.org](https://webassembly.org) n.d.c). This might lead to poor performance (see [webassembly.org](https://webassembly.org) n.d.c).

WASM can already be executed on a variety of ISAs, OSs and application environments. WASM runs in all major browser, e.g. Firefox, Chrome, Safari and Edge (see [webassembly.org](https://webassembly.org) n.d.d). WASM runs on the most common processor architectures, e.g. x86, ARM, MIPS and RISC-V (see *Wasm3* n.d.). WASM runs on most system, e.g. Linux, Windows, macOS, Android and iOS (see *Wasm3* n.d.). WASM runs on SBCs and Microcontroller Units (MCUs), e.g. Raspberry Pi, Orange Pi, Arduino and ESP8266 (see *Wasm3* n.d.). WASM runs embedded in different language environments, e.g. Rust, C, C++, C#, Python, Go, PHP and Ruby (see *Wasmer* 2019).

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 of embedding the module into another language environment. Existing runtimes are e.g. Lucet, Wasmer, Wasmtime, WAVM and Wasm3 (see Section 2.7.6).

Wzip (see Section 3.3.1) showcases a WASI module that can run as a standalone



---

Command-Line Interface (CLI) application or be embedded in the JS context of Node.js and browsers. The standard Rust tooling has already great WASM and WASI support and therefore makes the compilation a straightforward process. Nevertheless WASI is in an experimental stage (see Section 2.7.5). It has limitations like opening network sockets and asynchronous I/O (see Section 2.7.5). And there are currently hardly any implementations of the WASI interface for other programming languages, except JS.

## 4.5 Security

Native machine code is not designed to be fetched from untrusted sources. On most desktop and server Operating Systems native executables act in behalf of the user, malicious or faulty software can compromise the system. Most modern operating systems use some techniques to protect the system. Common techniques are sandboxing, code signing, access control and program checking (see Loureiro et al. 2003, Protection of a host from a mobile code). Code signing was used by ActiveX, but never part of JS' or WASM's philosophy.

WASM's sandbox is more strict than JS' default browser sandbox. WASM provides no ambient access (see WebAssembly Working Group 2019, section 1.1.3). WASM can only perform numerical operations and access its linear memory. Communication with the environment is possible via import and export of functions or memory. It is the embedder's responsibility to establish suitable security policies by controlling the access through the provided imports (see WebAssembly Working Group 2019, section 1.1.3).

Before a WASM module can be executed safely, it must be validated (see Haas et al. 2017, section 4). Validation is a simple task which can be performed in a single pass and while streaming compilation (see Section 2.6.3). Validation prove the absence of undefined behavior in the execution semantics (see Haas et al. 2017, section 4.2). This implies the absence of type safety violations such as invalid calls or illegal accesses to locals and it ensures the inaccessibility of code addresses or the call stack (see Haas et al. 2017, section 4.2). WASM code is isolated from the host, keeping the environment safe from malicious code. Many independent WASM instances can exist in the same system process without violating memory safety (see Haas et al. 2017, section 2.2). WASM also uses traps to immediately terminate execution on illegal code behavior, such as accessing addresses outside its linear memory.

Because WASM memory model is an untyped array of bytes, memory safety vulnerabilities, like buffer overflows and use-after-frees, remain a problem (see Disselkoen et al. 2019, section 1). WASM does not prevent attackers from exploiting memory-safety bugs to compromise the WASM code and any data it handles (see Disselkoen et al. 2019, section 1). Bytecodes like Java and .NET enforce strong memory safety by default. But this cannot be the default for WASM. WASM code is intended to be as performant as possible and should be an efficient compilation target for C and C++. Strong memory safety would definitely come along with performance trade-offs. Disselkoen et al. (2019) suggests an extension to WASM that allows developers to optionally enable certain provided memory safety features.

If the compiler toolchain Emscripten is used, one has to be extra careful. Emscripten defines a function collection for JS interoperability which can compromise security with Cross-Site Scripting (XSS) (see McFadden et al. 2018, section 4.2). The function `emscripten_run_script()` can be used to execute arbitrary JS code outside the WASM sandbox.

WASM can also be used for malicious code. WASM's performance makes it attractive for hidden cryptocurrency mining. In addition, WASM can be used to hide JS malware code inside WASM modules, to prevent detection by analysis tools (see Musch et al. 2019, section 4.5). Musch et al. (2019) shows that 56 % of all WASM usage in the Alexa top 1 million websites is for malicious purposes (see Musch et al. 2019, section 4.1).

WASI is designed with a capability based security model (see Section 2.7.4). Potentially dangerous resource accesses must be permitted by the user. Developers can compose applications from multiple modules with individually granted capabilities. Even when native or Node.js are somehow isolated from system resources, permission generally apply to all the modules of a program.

In conclusion, WASM offers advantages over native and JS code. It is still necessary

---

to carefully manage the modules imports and exports. In addition, there is room for improvement in order to optionally enforce stronger memory safety in WASM instances. Modern application are composed for the most part of third party dependencies. This untrusted sources can contain vulnerable or malicious code. WASM can be a tool to isolate software and offer more security.

# Chapter 5

## Discussion

### 5.1 Near-Native Performance

It is not generally true to say that WASM code runs with near-native performance. As one can see in Figure 4.5, it strongly depends on the used runtime. The tested runtimes of this thesis used JIT compilers. The performance of WASM interpreters is certainly even worse. Nevertheless WASM can provide better performance than its alternative asm.js. The runtimes V8, SpiderMonkey and WAVM achieve comparable results to native applications in many tests (see Figure 4.8).

### 5.2 Usage of WebAssembly in Production

A relevant question to ask is whether WASM is major enough to be used in production software yet. WASM is definitely ready to be used in the web environment. All major browsers have stable support for running WASM code. Also several mature tools exist, which are essential for software development with WASM. Compilers, command line tools and debugging support in browsers are paving the way for productive use. Examples are the Emscripten compiler, the WABT tools (see *WABT: The WebAssembly Binary Toolkit* n.d.) and debugging in Google Chrome's Dev-

Tools (see Stepanyan 2019).

Beyond the web, the use of WASM is rather experimental and many features are missing. It might fit some applications very well, but beforehand one should definitely check the capabilities for the required use case. The WASI API is not yet finalized and therefore WASI is more suitable for prototyping and small projects.

### 5.3 When to Prefer WebAssembly

WASM performs better than JS and even asm.js. WASM enables new kinds of applications on the web platform, e.g. image, audio, video processing apps, games, cryptography and science applications. WASM is in any case a better compilation target than JS for low-level languages like C and Rust. Developers should consider WASM for any part of the web applications that requires high computing performance.

### 5.4 When Not to Prefer WebAssembly

Most web apps currently won't benefit from WASM. JS is often fast enough and the work done by these apps mostly consists of network data transfer and user interface related tasks. Due to the lack of standard interfaces, e.g. to access the Document Object Model (DOM) and lack of features, e.g. interface types (see Section 2.6.13), all data has to be provided by the host environment and transferred in and out of the WASM linear memory. For such an application WASM MVP cannot boost performance, it has rather the opposite effect. WASM is also not ideal for asynchronous or multithreading applications. WASM MVP is not yet a good compilation target for dynamic, high-level languages with GC. Therefore in many scenarios native binaries and JS remain unbeaten.

## 5.5 Future Work

### 5.5.1. Comparing WebAssembly to Other Bytecodes

This thesis mainly compares WASM to native and JS code. But WASM probably has more in common with other bytecodes. Some established formats are Java, .NET and LLVM bytecode. The question arises, why no existing bytecode could be established on the web. And what are the technical differences? Future work could also compare performance, portability, security and compactness.

### 5.5.2. Performance Benchmarking with Added Features

Some upcoming WASM features that are currently being proposed or are in an experimental state require new performance benchmarks. Proposed features such as SIMD support, threads, bulk memory operations, GC and exception handling can be benchmarked against native code. But PolyBenchC is not suitable for these benchmarks. More feature specific tests are necessary.

### 5.5.3. Benchmarking Further Runtimes

Benchmarking more runtimes is beyond the scope of this thesis. But there are further interesting WASM runtimes designed for various environments. Also the runtimes used in this thesis can often be started with different optimization levels and features. This thesis is focussed on WASM runtimes using JIT compilers, because it is more meaningful to compare them to popular JS engines like V8 and SpiderMonkey. In some scenarios pure WASM interpreters might be more suitable. The comparison of execution time, startup time and runtime size of further runtimes provides more transparency in choosing the right runtime.

#### 5.5.4. Security of WASI Sockets

Currently WASI is only capable of receiving data from sockets, sending data to sockets and closing sockets (see Bytecode Alliance 2019b). Network connections cannot be initiated through WASI. Therefore network connections has to be pre-opened by the host runtime. Future WASI might get a capability describing a set of possible sockets that can be created. Capabilities could be permitted by a set of ports, addresses, or protocols (see Bytecode Alliance 2019a). Such a feature would be especially interesting for secure FaaS cloud services. Future work can compare WASI networking security to other cloud technologies.

#### 5.5.5. Evaluating Benefits and Costs on Microcontrollers

Section 2.6.14 introduces the utilization of WASM on microcontrollers. But this thesis does not focus much on microcontroller specific technologies. Microcontrollers are more likely to use WASM AOT compilers and WASM interpreters. Microcontrollers also prefer more lightweight WASM runtimes. A direct comparison of different microcontroller software development technologies could show appropriate areas of application for WASM on microcontrollers. Besides native C implementations, WASM could be compared to alternatives such as TinyGo and MicroPython.

#### 5.5.6. Comparing ewasm to EVM

As shown in Section 2.6.14, Ethereum might replace its current Ethereum Virtual Machine (EVM) with ewasm. That raises the question, what exactly are the differences of ewasm, EVM and other smart contract VMs. Is WASM suitable for smart contracts in the Blockchain and what advantages can it offer?



# Chapter 6

## Conclusion

As shown by this thesis, WASM can be solution for common software requirements such as performance, portability and security. WASM is as portable as JS. WASM is faster than asm.js. And WASM can reinforce security by isolating code from its environment.

But WASM is a young technology and many features are missing, so the alternatives might currently be the better choice. Nevertheless many missing features and tools are in active development and will make WASM suitable for further areas.

WASI is a great example, showing that a standardized interface can open up further areas of use, without adding any environment specific features to the WASM core. Besides interfaces, WASM can be the foundation for other technologies. Technologies can build on parts of WASM or extend WASM. One example is ewasm which is defined as a subset of WASM used for smart contracts on the Ethereum platform (see Section 2.6.14).

It remains to be seen until WASI is more mature and eventually become a standard API for all kinds of environments, from IoT devices to scalable cloud applications.

---

WASM is more than just a high performance extension to JS for the web. WASM could be an universal bytecode that runs everywhere. WASM is a technology that has the potential to change the way web and non-web software is developed.

# Bibliography

Bytecode Alliance (2019a), ‘Additional background on Capabilities’, GitHub. Accessed: February 29, 2020.

**URL:** *<https://github.com/bytecodealliance/wasmtime/blob/master/docs/WASI-capabilities.md>*

Bytecode Alliance (2019b), ‘WASI Core API’, GitHub. Accessed: December 21, 2019.

**URL:** *<https://github.com/bytecodealliance/wasmtime/blob/master/docs/WASI-api.md>*

Bytecode Alliance (n.d.a), ‘About the Bytecode Alliance’, Bytecode Alliance. Accessed: February 27, 2020.

**URL:** *<https://bytecodealliance.org/>*

Bytecode Alliance (n.d.b), ‘WebAssembly Micro Runtime’, GitHub. Accessed: February 15, 2020.

**URL:** *<https://github.com/bytecodealliance/wasm-micro-runtime>*

Cantelon, M., Harter, M., Holowaychuk, T. & Rajlich, N. (2013), *Node.js in Action*, 1st edn, Manning Publications Co., Greenwich, CT, USA.

chrome.com (2017), ‘WebAssembly migration guide’. Accessed: January 16, 2020.

**URL:** *<https://developer.chrome.com/native-client/migration>*

Chromium Blog (2019), ‘WebAssembly brings Google Earth to more browsers’. Accessed: March 2, 2020.

**URL:** <https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html>

Clark, L. (2019a), ‘Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly’, Bytecode Alliance. Accessed: January 23, 2020.

**URL:** <https://bytecodealliance.org/articles/announcing-the-bytecode-alliance>

Clark, L. (2019b), ‘Standardizing WASI: A system interface to run WebAssembly outside the web’, Mozilla Hacks – the Web developer blog. Accessed: January 4, 2020.

**URL:** <https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/>

Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A. & Stefan, D. (2019), Position paper: Progressive memory safety for WebAssembly, *in* ‘Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy’, HASP ’19, ACM, New York, NY, USA, pp. 4:1–4:8.

**URL:** <http://doi.acm.org/10.1145/3337167.3337171>

emscripten.org (2019), ‘Release notes’, Emscripten documentation. Accessed: January 16, 2020.

**URL:** [https://emscripten.org/docs/introducing\\_emscripten/release\\_notes.html](https://emscripten.org/docs/introducing_emscripten/release_notes.html)

emscripten.org (n.d.), ‘Optimizing code’, Emscripten documentation. Accessed: March 1, 2020.

**URL:** <https://emscripten.org/docs/optimizing/Optimizing-Code.html>

*Ethereum flavored WebAssembly (ewasm)* (2019), GitHub. Accessed: January 6, 2020.

**URL:** <https://github.com/ewasm/design>

Gallant, G. (2019), *WebAssembly in Action*, 1st edn, Manning Publications Co., Greenwich, CT, USA.

Gurdeep Singh, R. & Scholliers, C. (2019), WARDuino: A dynamic WebAssembly virtual machine for programming microcontrollers, *in* ‘Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes’, MPLR 2019, Association for Computing Machinery, New York, NY, USA, p. 27–36.

**URL:** <https://doi.org/10.1145/3357390.3361029>

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. & Bastien, J. (2017), ‘Bringing the web up to speed with WebAssembly’, **52**(6), 185–200.

**URL:** <http://doi.acm.org/10.1145/3140587.3062363>

Hickey, P. (2019), ‘Announcing Lucet: Fastly’s native WebAssembly compiler and runtime’, Fastly. Accessed: February 15, 2020.

**URL:** <https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime>

*JS Logo By The Community* (n.d.), Github. Accessed: January 15, 2020.

**URL:** <https://github.com/voodootikigod/logo.js>

Loureiro, S., Molva, R. & Roudier, Y. (2003), ‘Mobile code security’.

*Lucet* (2019), GitHub. Accessed: December 28, 2019.

**URL:** <https://github.com/bytecodealliance/lucet>

McConnell, J. (2017), ‘WebAssembly support now shipping in all major browsers’. Accessed: December 27, 2019.

**URL:** <https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers/>

McFadden, B., Lukasiewicz, T., Dileo, J. & Engler, J. (2018), ‘Security chasms of WASM’. Accessed: January 4, 2020.

**URL:** <https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native-Exploits-On-The-Web-wp.pdf>

Mozilla (n.d.), ‘Mozilla Firefox release notes’, Mozilla. Accessed: February 9, 2020.

**URL:** <https://www.mozilla.org/en-US/firefox/releases/>

Musch, M., Wressnegger, C., Johns, M. & Rieck, K. (2019), New kid on the web: A study on the prevalence of WebAssembly in the wild, *in* R. Perdisci, C. Maurice, G. Giacinto & M. Almgren, eds, ‘Detection of Intrusions and Malware, and Vulnerability Assessment’, Springer International Publishing, Cham, pp. 23–42.

nodejs.org (n.d.), ‘Node.js’. Accessed: January 15, 2020.

**URL:** <https://nodejs.org/>

Pouchet, L.-N., Bondugula, U. & Yuki, T. (2016), ‘PolyBench/C’, GitHub. Accessed: January 4, 2020.

**URL:** <https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1>

Salim, S. S., Nisbet, A. & Luján, M. (2019), 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’, SPLASH Companion 2019, ACM, New York, NY, USA, pp. 15–16.

**URL:** <http://doi.acm.org/10.1145/3359061.3362780>

Spies, B. (2020), ‘Utilization of WebAssembly in non-web environments’, GitHub. Bachelor thesis. Under supervision of Prof. Dr. Markus Mock.

**URL:** <https://github.com/HAWMobileSystems/BSSpiess>

Stack Overflow (2019), ‘Stack Overflow developer survey 2019’, Stack Overflow. Accessed: January 21, 2020.

**URL:** <https://insights.stackoverflow.com/survey/2019>

Stepanyan, I. (2019), ‘Improved WebAssembly debugging in Chrome DevTools’, Google Developers. Accessed: February 24, 2020.

**URL:** <https://developers.google.com/web/updates/2019/12/webassembly>

*stockfish.wasm* (n.d.), GitHub. Accessed: February 24, 2020.

**URL:** <https://github.com/niklasf/stockfish.wasm>

Trivellato, M. (2018), ‘WebAssembly is here!’, Unity Blog. Accessed: March 4, 2020.

**URL:** <https://blogs.unity3d.com/2018/08/15/webassembly-is-here/>

W3C (2019), ‘World Wide Web Consortium (W3C) brings a new language to the web as WebAssembly becomes a W3C recommendation’, W3C. Accessed: December 27, 2019.

**URL:** <https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en>

*WABT: The WebAssembly Binary Toolkit* (n.d.), GitHub. Accessed: February 25, 2020.

**URL:** <https://github.com/WebAssembly/wabt>

*WASI* (2019). Accessed: January 4, 2020.

**URL:** <https://wasi.dev/>

*Wasm3* (n.d.), GitHub. Accessed: January 6, 2020.

**URL:** <https://github.com/wasm3/wasm3>

*Wasmer* (2019). Accessed: December 29, 2019.

**URL:** <https://wasmer.io/>

*Wasmtime* (2019), GitHub. Accessed: December 29, 2019.

**URL:** <https://github.com/bytecodealliance/wasmtime>

*WAVM* (2019), GitHub. Accessed: December 27, 2019.

**URL:** <https://github.com/WAVM/WAVM>

WebAssembly Working Group (2019), ‘WebAssembly Core Specification’, W3C.  
Accessed: December 27, 2019.

**URL:** *<https://www.w3.org/TR/wasm-core-1/>*

*WebAssembly Interface Types Proposal* (n.d.), GitHub. Accessed: February 29, 2020.

**URL:** *<https://github.com/WebAssembly/interface-types/blob/master/proposals/interface-types/Explainer.md>*

webassembly.org (n.d.a), ‘Features to add after the MVP’. Accessed: December 23, 2019.

**URL:** *<https://webassembly.org/docs/future-features/>*

webassembly.org (n.d.b), ‘Nondeterminism in WebAssembly’. Accessed: March 2, 2020.

**URL:** *<https://webassembly.org/docs/nondeterminism/>*

webassembly.org (n.d.c), ‘Portability’. Accessed: March 2, 2020.

**URL:** *<https://webassembly.org/docs/portability/>*

webassembly.org (n.d.d), ‘WebAssembly’. Accessed: January 4, 2020.

**URL:** *<https://webassembly.org/>*



# List of Figures

2.1. JavaScript (JS) logo . . . . .	5
2.2. Integer add function written in C, compiled to asm.js. The bitwise OR suffix <code>  0</code> does not modify the value, but is a type hint for the JS compiler to treat <i>a</i> and <i>b</i> as 32 bit signed integers. . . . .	7
2.3. Command line showing content of <i>hello.c</i> , a Hello World C program. Code is compiled with emcc to JS and additional WASM file. Output files are listed in the current file system directory. Node.js executes the <i>hello.js</i> . . . . .	8
2.4. Node.js logo . . . . .	9
2.5. WebAssembly (WASM) logo . . . . .	10
2.6. Object dump of WASM file. Addition of two constant 32 Bit integer numbers. Hexadecimal representation of binary format on the left. WAT format on the right. . . . .	14
2.7. WASM module exporting an add function for integer values, represented in WAT format. Indexes are used to reference functions and values. . . . .	15
2.8. WASM module exporting an add function for integer values, represented in WAT format. For better readability, folded expressions and name references are used. Semantically equivalent to Figure 2.7. . . . .	15
2.9. WebAssembly System Interface logo . . . . .	22

2.10. Command line view content of <i>hello.rs</i> , a Hello World Rust program. Code is compiled to the WASM-WASI target. Output file is listed in the current file system directory. Wasmtime runtime executes the standalone WASM module <i>hello.wasm</i> . . . . .	23
3.1. Compiling PolyBenchC benchmark sources to different targets . . .	29
3.2. Running PolyBenchC benchmarks in different runtime environments	31
3.3. Running benchmark with argument <i>initonly</i> to exit process imme- diately after main function entry . . . . .	32
3.4. Usage of wzip, a gzip Rust implementation . . . . .	35
3.5. Using rustup to install target platform for WASM-WASI . . . . .	36
3.6. Using Cargo to compile code to WASM-WASI target platform . . .	36
3.7. Running wzip on command line using the Wasmtime runtime, read- ing from stdin and writing to stdout . . . . .	36
3.8. Running wzip on command line using the Wasmtime runtime, read- ing from file and writing to file . . . . .	37
3.9. Instantiation of wzip in browser's JS using Wasmer-JS. Full source code at Spies (2020). . . . .	38
3.10. Compressing file with wzip running in browser. User can drag and drop files to automatically inflate or deflate them. . . . .	39
3.11. Starting Node.js wzip web service from command line . . . . .	40
3.12. Calling the wzip web service from command line to decompress a file	40
4.1. Binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Geometric mean of 30 benchmark binaries per compile target. Data from Table A.3. . . . .	43
4.2. Binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Data from Table A.3. . . . .	44

4.3. 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. Data from Table A.4. . . . .	45
4.4. Compressed binary sizes of the PolyBenchC benchmarks compiled to different target platforms. Gzip is used for compression. Data from Table A.4. . . . .	46
4.5. 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. Low values are better. Data from Table A.1. . . . .	49
4.6. 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. Low values are better. Data from Table A.1. . . . .	50
4.7. 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. Low values are better. Data from Table A.1. . . . .	51
4.8. 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. Data from Table A.1. . . . .	52
4.9. 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. Low values are better. Data from Table A.2. . . . .	54

---

4.10. Comparison of PolyBenchC benchmark startup time of native x86-64 and WASI standalone runtimes. Arithmetic means of 15 startups and 95 % confidence intervals. Low values are better. Data from Table A.2 . . . . .	55
4.11. Comparison of PolyBenchC benchmark startup time of native x86-64, asm.js and WASM embedded in JS environments. Arithmetic means of 15 startups and 95 % confidence intervals. Low values are better. Data from Table A.2 . . . . .	56

# List of Tables

A.1. Execution time of PolyBenchC benchmarks in seconds on different runtime environments. Excluding VM startup, validation and JIT compilation. Arithmetic mean of 15 runs. . . . .	87
A.2. Startup time of PolyBenchC benchmarks in seconds on different runtime environments. Arithmetic mean of 15 runs. . . . .	88
A.3. File sizes (in Byte) of the PolyBenchC benchmarks compiled to different target platforms . . . . .	89
A.4. Compressed file sizes (in Byte) of the PolyBenchC benchmarks compiled to different target platforms. Files are compressed with gzip. . . . .	90
B.1. Specification of system used for executing PolyBenchC benchmarks	91
B.2. Software versions used for building, executing and analysis of wzip (see Section 3.3.1) and PolyBenchC benchmarks . . . . .	92

# Glossary

**.NET** software framework developed by Microsoft

**ActiveX** framework from Microsoft for downloading and running native machine code in browsers

**Apache** HTTP server software

**Arduino** software and hardware microcontroller platform

**ARM** Reduced Instruction Set Computer (RISC) ISA

**asm.js** subset of JS that is highly optimizable

**Blockchain** decentralized, distributed, digital record of transactions

**Bytecode Alliance** community creating software on standards such as WASM and WASI

**C** procedural programming language with manual, low-level memory management

**C++** extension of C by object-oriented, generic, and functional features

**C#** strongly typed, high-level programming language

**Cargo** build system and package manager for Rust

**Chakra** JS engine developed by Microsoft

**Chrome** web browser developed by Google

**clang** compiler frontend for C and C++ using LLVM

**CloudABI** system interface with capability-based security

**Cranelf** low-level code generator

**cross-compiler** Compiler creating code for a platform other than the one on which the compiler is running

**ECMAScript** specification of Ecma International to standardize JS

**Edge** web browser developed by Microsoft

**Electron** framework to develop desktop GUI applications using web technologies

**emcc** Emscripten Compiler Frontend

**Emscripten** LLVM-to-web compiler that produces JS and WASM

**ESP8266** low-cost microcontroller

**Ethereum** blockchain-based, decentralized software platform featuring smart contracts

**Firefox** web browser developed by the Mozilla Foundation

**Go** statically typed programming language

**GraalVM** universal virtual machine for running applications that can be written in many different languages

**GraalWasm** WASM engine implemented in the GraalVM

**gzip** file format and software used for file compression and decompression

**HTML** markup language for creating webpages

**HTTP** protocol for data communication on the web

**Java** high-level programming language developed by Sun Microsystems

**JavaScript** dynamically typed programming language, one of the core web technologies

**JavaScriptCore** JS engine of WebKit

**JSON** human-readable data interchange format

**LLVM** collection of modern compiler and toolchain technologies

**Lucet** WASM compiler and runtime, designed for the cloud

**MicroPython** version of Python optimized to run on microcontrollers

**MIPS** RISC ISA

**NaCl** Google Native Client, sandbox for running a subset of native code in web browsers

**NativeScript** JS framework for cross-platform mobile applications

**Node.js** platform for executing JS server-side

**NoSQL** not only SQL, alternative to traditional relational databases

**Orange Pi** open-source SBC

**PHP** programming language designed for web development

**PL/pgSQL** procedural programming language for PostgreSQL

**PNaCl** Portable Native Client, architecture independent version of NaCl

**PolyBenchC** benchmark suite of numerical computations from operations in various application domains, implemented in C



**Polyfill** implementation of a feature that is not supported or not yet supported by the runtime

**Polyglot Programming** practice of writing a single program or software in several languages in order to use the most appropriate functionality for the task to be solved

**POSIX** Portable Operating System Interface, standard for software compatibility with Unix and similar operating systems

**PostgreSQL** open-source relational database management system (DBMS)

**Python** dynamically typed programming language

**R** programming language for statistical computing

**Raspberry Pi** SBC

**React Native** JS framework for cross-platform mobile applications

**RISC-V** RISC ISA

**Ruby** dynamically typed programming language

**Rust** programming language focused on safety and high performance

**rustup** installer for Rust

**Safari** web browser developed by Apple

**SpiderMonkey** first JS engine, formerly used in Netscape Navigator and today in Firefox

**TinyGo** Go compiler for places such as microcontrollers, WASM and command-line tools

**V8** JS engine of Google Chrome

**W3C** consortium for the standardization of technologies on the web

**WABT** WebAssembly Binary Toolkit, suite of tools for WASM

**WAMR** WebAssembly Micro Runtime, WASM runtime with small footprint

**WARDuino** WebAssembly VM for microcontrollers

**Wasm3** WebAssembly interpreter

**Wasmer** standalone runtime for running WebAssembly outside the browser

**Wasmer-JS** JS packages enabling easy use of WASM in Node.js and the browser

**Wasmtime** runtime for WASM with WASI support

**WAVM** WebAssembly VM, designed for use in non-web applications

**WebAssembly** portable bytecode for high performance applications on the web  
and other environments

**WebAssembly System Interface** system interface for WebAssembly

**WebKit** browser engine developed by Apple

**x86** ISA initially developed by Intel

**x86-64** 64 Bit version of x86

# Acronyms

**AOT** Ahead-of-Time

**API** Application Programming Interface

**CLI** Command-Line Interface

**CPU** Central Processing Unit

**DBMS** database management system

**DOM** Document Object Model

**EVM** Ethereum Virtual Machine

**ewasm** Ethereum flavored WebAssembly

**FaaS** Function as a Service

**GC** Garbage Collection

**GUI** graphical user interface

**I/O** input/output

**IoT** Internet of Things

**ISA** Instruction Set Architecture

**JIT** Just-in-Time

**JRE** Java Runtime Environment

**JS** JavaScript

**MCU** Microcontroller Unit

**MVP** Minimum Viable Product

**OS** Operating System

**RISC** Reduced Instruction Set Computer

**SBC** Single-Board Computer

**SIMD** Single Instruction, Multiple Data

**VM** Virtual Machine

**WASI** WebAssembly System Interface

**WASM** WebAssembly

**WAT** WebAssembly Text Format

**XSS** Cross-Site Scripting

# **Appendix A**

## **PolyBenchC Measurements**

Algorithm	x86-64	Node.js (asm.js)	SpiderMonkey (asm.js)	Node.js (WASM)	SpiderMonkey (WASM)	Wasmtime	Wasmer	WAVM
2mm	3.2826	5.7301	7.6801	4.3513	4.6561	12.8659	7.7651	3.9008
3mm	5.9062	10.3393	16.0525	7.8015	7.9798	26.7803	15.5476	6.5123
adi	8.9630	9.1747	12.1929	8.9378	8.5227	19.0338	10.2067	8.3761
atax	0.0062	0.0215	0.0291	0.0076	0.0093	0.0283	0.0212	0.0071
bicg	0.0102	0.0171	0.0249	0.0095	0.0099	0.0343	0.0246	0.0098
cholesky	1.5279	2.6639	3.2409	1.5257	1.5769	5.6036	2.9875	1.5544
correlation	6.0929	6.0520	6.3278	5.9051	5.9761	9.3021	6.7889	5.9960
covariance	6.1102	6.0818	6.4029	5.8177	5.9938	8.8869	6.7957	6.0088
deriche	0.2548	0.5947	0.3979	0.2379	0.2461	0.3855	0.2547	0.2428
doitgen	0.5373	0.8607	1.7298	0.5759	0.8482	2.6610	1.2679	0.5442
durbin	0.0032	0.0151	0.0209	0.0046	0.0073	0.0225	0.0127	0.0053
fdtd-2d	2.2073	4.8507	7.7657	3.1715	3.5811	12.8904	7.1871	2.6478
floyd-warshall	15.1710	40.1407	70.9601	35.7923	32.9747	144.8849	82.8400	18.5631
gemm	0.7334	2.7402	4.9928	1.7998	2.1779	7.0585	3.7715	1.1632
gemver	0.0252	0.0591	0.0659	0.0287	0.0288	0.0822	0.0503	0.0266
gesummv	0.0046	0.0114	0.0123	0.0041	0.0043	0.0173	0.0099	0.0040
gramschmidt	10.9002	9.6741	9.5065	10.0317	9.8235	13.1455	9.9517	10.2935
heat-3d	2.5398	8.0893	30.0997	4.5759	4.9665	21.2657	11.5163	4.7854
jacobi-1d	0.0008	0.0095	0.0149	0.0021	0.0028	0.0118	0.0065	0.0020
jacobi-2d	1.8416	5.9132	11.1526	2.9104	3.6279	15.6576	8.1262	2.9008
lu	8.1222	11.4756	13.3471	8.8713	9.0645	20.6126	13.5509	8.6294
ludcmp	7.9360	11.0612	13.8311	8.7758	8.7991	15.9745	12.5160	8.0475
mvt	0.0210	0.0371	0.0385	0.0207	0.0210	0.0454	0.0320	0.0217
nussinov	6.3340	11.3355	28.1155	8.1824	8.6527	20.4684	13.4834	6.9698
seidel-2d	19.0384	19.6615	19.0295	20.5391	19.6019	21.7994	21.6238	18.6651
symm	3.3321	4.3916	5.3075	3.6787	3.8973	8.7290	6.5253	3.4438
syr2k	4.8236	5.8977	7.4363	5.1691	5.4795	15.0998	10.0304	5.2436
syrk	1.0024	1.9384	2.9223	1.5185	1.6637	5.6072	3.7450	1.3987
trisolv	0.0026	0.0089	0.0085	0.0022	0.0029	0.0060	0.0045	0.0024
trmm	2.3875	2.5109	2.8656	2.3263	2.4301	3.9324	2.8994	2.3917

Table A.1.: Execution time of PolyBenchC benchmarks in seconds on different runtime environments. Excluding VM startup, validation and JIT compilation. Arithmetic mean of 15 runs.

Algorithm	x86-64	Node.js (asm.js)	SpiderMonkey (asm.js)	Node.js (WASM)	SpiderMonkey (WASM)	Wasmtime	Wasmer	WAVM
2mm	0.003 84	0.034 27	0.029 58	0.040 93	0.034 43	0.079 62	0.060 80	0.356 30
3mm	0.003 65	0.035 00	0.032 70	0.040 77	0.032 29	0.080 51	0.064 25	0.357 74
adi	0.004 94	0.033 48	0.031 34	0.040 95	0.032 61	0.077 78	0.064 30	0.363 02
atax	0.003 65	0.035 71	0.032 26	0.040 86	0.033 41	0.079 06	0.059 71	0.351 58
bicg	0.003 73	0.032 09	0.027 89	0.039 77	0.033 56	0.078 32	0.061 88	0.355 12
cholesky	0.004 45	0.034 88	0.030 60	0.040 49	0.033 70	0.075 43	0.062 88	0.367 03
correlation	0.003 55	0.031 65	0.027 61	0.041 09	0.033 74	0.076 46	0.059 65	0.360 24
covariance	0.003 77	0.031 34	0.028 02	0.041 92	0.034 58	0.084 68	0.067 45	0.372 39
deriche	0.004 72	0.033 84	0.030 81	0.040 12	0.033 35	0.077 38	0.064 37	0.374 10
doitgen	0.003 64	0.032 61	0.028 62	0.039 90	0.034 36	0.077 46	0.063 30	0.353 51
durbin	0.004 27	0.032 50	0.030 35	0.041 03	0.033 16	0.076 34	0.063 66	0.373 20
fdtd-2d	0.005 42	0.033 08	0.032 33	0.039 98	0.032 39	0.077 87	0.063 61	0.363 16
floyd-warshall	0.004 58	0.032 57	0.032 84	0.040 71	0.031 97	0.077 94	0.061 29	0.374 61
gemm	0.003 61	0.032 74	0.030 18	0.041 19	0.033 54	0.077 13	0.063 41	0.375 49
gemver	0.004 35	0.035 20	0.031 63	0.040 22	0.033 28	0.076 09	0.062 11	0.382 93
gesummv	0.003 71	0.033 47	0.031 95	0.039 68	0.032 10	0.076 25	0.063 87	0.352 37
gramschmidt	0.004 44	0.035 17	0.029 54	0.040 71	0.033 49	0.076 20	0.061 05	0.370 90
heat-3d	0.004 99	0.034 14	0.030 23	0.040 85	0.033 68	0.074 78	0.062 72	0.382 60
jacobi-1d	0.004 94	0.033 52	0.032 23	0.039 84	0.033 23	0.075 39	0.061 02	0.352 98
jacobi-2d	0.005 17	0.033 41	0.029 59	0.041 37	0.033 43	0.075 39	0.061 47	0.357 68
lu	0.004 70	0.033 72	0.031 53	0.040 85	0.032 74	0.076 59	0.064 45	0.371 58
ludcmp	0.004 13	0.034 23	0.029 92	0.040 81	0.033 49	0.077 54	0.063 27	0.376 41
mvt	0.003 55	0.032 07	0.027 66	0.040 48	0.032 73	0.077 01	0.059 49	0.355 01
nussinov	0.004 51	0.032 54	0.031 98	0.040 11	0.032 50	0.076 33	0.061 55	0.372 08
seidel-2d	0.004 89	0.034 72	0.029 10	0.040 66	0.033 54	0.076 67	0.060 93	0.363 50
symm	0.004 04	0.033 93	0.032 30	0.040 04	0.033 42	0.078 28	0.059 91	0.377 75
syr2k	0.003 67	0.034 25	0.029 45	0.041 52	0.033 70	0.077 77	0.064 99	0.362 34
syrk	0.003 85	0.032 63	0.030 60	0.040 16	0.032 38	0.078 77	0.062 67	0.352 68
trisolv	0.003 93	0.032 69	0.031 72	0.040 00	0.033 21	0.075 15	0.061 53	0.384 03
trmm	0.003 97	0.034 91	0.030 84	0.041 12	0.033 80	0.077 16	0.064 66	0.370 80

Table A.2.: Startup time of PolyBenchC benchmarks in seconds on different run-time environments. Arithmetic mean of 15 runs.

Algorithm	x86-64 from clang	WASM from clang	WASM from emcc	asm.js from emcc
2mm	14480	35886	38933	66535
3mm	14480	36205	39087	66884
adi	14480	36526	39514	67815
atax	14488	35369	38587	66056
bicg	14488	35587	38692	66361
cholesky	14496	35653	38864	66522
correlation	14488	35795	38956	66561
covariance	14480	35495	38739	66113
deriche	14480	36026	39071	67150
doitgen	14480	35488	38723	66193
durbin	14488	35145	38467	65509
fdtd-2d	14480	36213	39188	67086
floyd-warshall	14480	35130	38383	65737
gemm	14480	35479	38720	66089
gemver	14480	36074	38926	66360
gesummv	14480	35454	38608	65789
gramschmidt	14496	35819	38960	66617
heat-3d	14480	35671	38952	67184
jacobi-1d	14480	35096	38415	65398
jacobi-2d	14480	35387	38663	65941
lu	14488	35659	38848	66505
ludcmp	14488	36206	39215	67411
mvt	14480	35678	38785	66124
nussinov	14488	35282	38614	65881
seidel-2d	14480	35189	38531	65998
symm	14480	35724	38899	66335
syr2k	14480	35495	38727	66081
syrk	14480	35311	38630	65900
trisolv	14480	35228	38527	65690
trmm	14480	35401	38674	65953

Table A.3.: File sizes (in Byte) of the PolyBenchC benchmarks compiled to different target platforms



Algorithm	x86-64 from clang	WASM from clang	WASM from emcc	asm.js from emcc
2mm	3520	13846	15550	23049
3mm	3598	13886	15594	23110
adi	3998	14122	15775	23589
atax	3484	13640	15438	22961
bicg	3360	13703	15472	23038
cholesky	3988	13835	15541	23207
correlation	3804	13812	15601	23157
covariance	3477	13701	15494	23047
deriche	3956	13814	15585	23294
doitgen	3911	13734	15466	23039
durbin	3488	13611	15380	22995
fdtd-2d	4258	13840	15646	23205
floyd-warshall	3465	13623	15322	22828
gemm	3567	13709	15472	22998
gemver	3887	13760	15559	23110
gesummv	3250	13664	15455	22960
gramschmidt	3629	13823	15603	23196
heat-3d	3747	13766	15499	23270
jacobi-1d	3386	13546	15350	22984
jacobi-2d	3443	13661	15428	23050
lu	3975	13817	15499	23148
ludcmp	4374	13966	15703	23312
mvt	3393	13679	15477	22974
nussinov	3576	13699	15475	23061
seidel-2d	3205	13668	15408	22944
symm	3866	13825	15559	23165
syr2k	3937	13757	15504	23063
syrk	3880	13690	15430	22923
trisolv	3261	13630	15433	22978
trmm	3513	13733	15456	22966

Table A.4.: Compressed file sizes (in Byte) of the PolyBenchC benchmarks compiled to different target platforms. Files are compressed with gzip.

# Appendix B

## System Specification and Software Versions

---

System Manufacturer	LENOVO
System Product Name	20QES01L00
Kernel Name	Linux
Kernel Release	5.3.0-26-generic
Kernel Version	#28-Ubuntu SMP Wed Dec 18 05:37:46 UTC 2019
Hardware Architecture	x86_64
CPU	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
RAM	16 GB

---

Table B.1.: Specification of system used for executing PolyBenchC benchmarks

---

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/2019-11-04 (0cfad6a)

---

Table B.2.: Software versions used for building, executing and analysis of wzip (see Section 3.3.1) and PolyBenchC benchmarks