WASMine

A WebAssembly Runtime with AOT, JIT and Interpreter Backends

Enrico Fazzi
Lukas Döllerer
Technical University of Munich
{enrico.fazzi,lukas.doellerer}@cit.tum.de

Abstract

WebAssembly is the modern, high-performance bytecode for execution in web browser sandboxes. Because of its high portability, isolation, and low overhead, it has also gained popularity as a program encoding for execution in standalone runtimes. They replicate the sandboxing techniques of browsers while enhancing the execution environment through additional APIs like WASI.

This paper explores the architecture of such a standalone WebAssembly runtime, WASMine. We introduce components for each of the lifetime stages of WebAssembly execution: Loading, parsing, linking, and execution. Depending on the specific use case, the latter can be performed by a compiling backend using the LLVM compiler toolchain or an interpreter. We also establish an extensive test and benchmark system to ensure specification-compliant behavior and adherence to performance requirements.

Keywords WebAssembly, LLVM, Interpreter

1 Introduction

WebAssembly (Wasm) is an open binary instruction format for a stack-based virtual machine, designed as a portable compilation target for programming languages, enabling deployment on the web for both client and server applications. Its primary goals are to provide high performance, security, and portability across different platforms as a compilation target for many popular languages.

WebAssembly (Wasm) has gained significant traction in recent years due to its ability to run code at near-native speed and its compatibility with existing web technologies. This has led to its adoption in various domains beyond web browsers, including server-side applications, standalone runtime environments, and even IoT devices.

We introduce WASMine, a standalone WebAssembly runtime that supports ahead-of-time (AOT) compilation, just-in-time (JIT) compilation, and interpretation. We discuss the architecture of WASMine, detailing its components for loading, parsing, linking, and executing Wasm modules. Additionally, we present our testing and benchmarking setup to ensure compliance with the Wasm specification and to measure performance in comparison with existing runtimes. This paper is structured as follows: Section 2 provides an overview of the Wasm bytecode format and its features. Section 3 details the implementation of WASMine, including its architecture and components. Section 4 presents our testing and benchmarking setup and discusses the results. Finally, Section 5 concludes the paper and outlines future work.

2 Background

The following section provides an overview of the WebAssembly (Wasm) bytecode format.

2.1 WebAssembly

Web browsers employ sandboxing to ensure safety and containment while executing arbitrary source code for the rendering and interactivity of web pages. For a long time, the only language available for execution in a browser was JavaScript. Modern browsers feature very complex just-in-time (JIT) compiling JavaScript engines to increase execution performance of the dynamically typed scripting language, like Mozilla's SpiderMonkey [18], Google's V8 [14], or Apple's JavaScriptCore [4]. However, optimizations that can be efficiently applied to dynamic languages like JavaScript are limited, meaning its execution performance lacks behind the performance of natively compiled languages.

To improve execution performance, browser engineers from the major browser vendors Google, Microsoft, Mozilla, and Apple invented a new bytecode

format specifically built for low overhead execution in web browsers, WebAssembly [15].

2.2 Specification & Features

The code format is described in the *WebAssembly Core Specification*, which at the time of writing is available as an editor's draft for Wasm version 2.0. It specifies the structure of Wasm modules, their validation and execution, and describes their representation in both binary and text forms.

2.2.1 File Formats

The Wasm specification employs an abstract syntax to describe the components of a Wasm module, such as instructions and data types, in a highly abstract manner. It also defines two concrete formats for storing Wasm modules. The binary format is a compact version of this abstract syntax, designed to optimize for size and parsing efficiency, and is stored with the `.wasm` file extension. In contrast, the text format uses S-Expressions [22] to represent the same abstract syntax in a human-readable form, typically stored with the `.wast` file extension.

The `.wast` file format is an unofficial extension of the Wasm text S-Expression format. It introduces "commands" that allow for structuring a Wasm text file to create test scripts for Wasm.[1] The Wasm project provides a test suite for runtimes to verify conformance with the specification, which is written using the `.wast` format.

2.3 Design

2.3.1 Modules

Each Wasm file corresponds to a single Wasm module, which is the biggest organizational unit for Wasm programs. It groups Wasm function code together with all the metadata required for linking, run-time environment setup and its execution. A textual representation of such a module is shown in Figure 1. In the binary representation, modules are organized in sections of which most contain lists of objects that are referred to by index in the section.

2.3.2 Function Types

The Wasm language is a statically typed language with the following seven basic types:

[1] https://github.com/WebAssembly/spec/tree/master/interpreter#scripts (accessed 2024-10-01)

```
1  (import "math" "i2f"
2    (func $i2f (param i32) (result f32)))
3
4  (type $main-type
5    (func (param i32) (result i32 f32)))
6
7  (global $offset i32 (i32.const -2))
8
9  (table $t0 2 4 funcref)
10 (elem
11   (table $t0)
12   (i32.const 2)
13   func $main $i2f)
14
15 (memory 1 2)
16 (data (i32.const 1) "WASM ROCKS")
17
18 (func $start
19   (i32.const 42) (drop))
20
21 (export "best-main" (func $main))
22
23 (start $start)
```

Figure 1. Example Wasm module code in the Wasm text format (Section 2.2.1).

- `i32`: 32-bit uninterpreted integers.
- `i64`: 64-bit uninterpreted integers.
- `f32`: 32-bit IEEE-754-2019 floats [3].
- `f64`: 64-bit IEEE-754-2019 doubles [3].
- `funcref`: A reference to a Wasm function of the same module.
- `externref`: A reference to a not further specified host function.
- `v128`: 128-bit vector type.

These basic types are complemented by function types, representing the signature of function objects. They are used for ensuring the type safety of function implementations, imported functions, function calls, and signatures in structured control flow. Such function types consist of two tuples: a tuple of input types, which we call *arguments*, and a tuple of output types, which we call *results*. Both tuples may be of arbitrary length, where empty tuples are similar to the `void` type in C-like programming languages and represent no parameters or return values. All function types that occur in a module are stored in the function types section of the module and are at later points in the program only referenced by their index in that section. An example of a function type declaration can be seen in Figure 1, Line 4.

2.3.3 Globals

Wasm global variables, shown in Figure 1, Line 7, are either mutable or immutable global state that is initialized on startup of the Wasm runtime. Their type and initializer are stored in the module's globals section. They are only referred to via their index in that section.

Initialization is performed via a constant expression, i.e. a sequence of constant instructions terminated by an `end` marker. The specification defines constant instructions as instructions that either push an immediate integer or an internal function reference to the stack, or access an imported constant global.

2.3.4 Tables

Indirect function calls in Wasm use tables for dynamic function selection based on a selector variable. Therefore, a Wasm table serves as a statically populated jump table with dynamic function signature verification. An example is shown in Figure 1, Line 9. *Elements* statically specify function references that are loaded into a table at run-time either at the startup of the runtime or on-demand via the `table.init` instruction. They are stored in an elements segment of the module or inserted as element declarations in the text format, shown in Figure 1, Line 10. Specifications of tables are likewise stored in the tables segment of the module. Both, elements and tables, are only referred to by their index in their respective segments.

2.3.5 Memories

Wasm memory (Figure 1, Line 15) is abstracted via *Wasm memories*. Each Wasm memory represents an array of linear memory, addressable via a 32-bit unsigned integer address, starting at 0. A Wasm memory is divided into Wasm pages with a size of 64 kiB. A Wasm memories definition contains a minimum size, which the runtime allocates for the memory on startup, and an optional maximum size. The size of a memory can be dynamically increased using the `memory.grow n` instruction, which increases its size by `n` pages if possible. Wasm datas specify initial content of a memory and can, like elements, be either loaded on startup of the runtime or on-demand via the `memory.init` instruction. An example data declaration of the text format is shown in Figure 1, Line 16.

The current Wasm specification [23] allows two instruction types to access a Wasm memory: `load` and `store` instructions. Both pop a 32-bit integer from the stack and interpret it as the address inside the module's Wasm memory. The instructions themselves additionally encode a 32-bit integer offset in their *memarg*, which is added to the loaded address. When accessing a value of type t at the resulting 33-bit address a in memory, the actually accessed bytes are within the range $[a, a + \text{sizeof}(t) - 1]$, and all bytes need to be within the bounds of the allocated memory region. The largest type defined by the current specification is the 128-bit wide vector type, meaning $\text{sizeof}(t) \leq 16$. It suffices to check whether address $a' = a + \text{sizeof}(t) - 1$ is in bounds, which also covers all other accessed addresses [15]. Following references to the "address" for bounds checking refer to this end address a'.

2.3.6 Functions

A Wasm module's function section (Figure 1, Line 18) contains the actual code in the form of instructions. Instructions are defined to interact with a stack machine; however, because of the strict type system, the stack layout is statically computable so that actual implementations never have to materialize the stack. Even though instructions may consume operands from the current stack, they can also encode operands within themselves.

Some instructions can deviate from normal control flow by resulting in a trap. This is similar to an exception and causes the execution of the instruction's Wasm thread to halt. Wasm programs cannot handle traps themselves. The specification requires the outside environment to catch and handle the failure. Without the changes proposed in the Wasm threads proposal [1], this always means the full termination of the program execution and destruction of volatile state. The threads proposal builds on the concept of ECMAScript agents [10], called Wasm agents, to represent a single Wasm execution thread. Agents can interact with each other using shared Wasm memories, atomic- and synchronizing instructions. In this context, a trap only leads to the termination of its respective agent.

One notable source for traps are memory accesses outside of the bounds of the allocated memory.[23]

2.3.7 Linking of Modules: Exports & Imports

A Wasm module may expose functions, tables, memories or globals to be linked, i.e. made accessible, by other Wasm modules. These exposed objects are registered through export declarations in the export section of a module. Most notably, an export declaration assigns a name to the exported object which, combined with

name under which the Wasm module was registered with the linker, is used for unique object identification across multiple Wasm modules.

These exported objects can be imported as resources for a Wasm module through import declarations which are stored in the import section of a Wasm module. Imported objects are referred to by index as if they were originally declared in their respective object's section.

A Wasm runtime linker is responsible for managing and providing imported resources at run-time. An imported object is not copied into the importing module's execution context, but rather referred to, meaning changes to the object are reflected in its source module's execution context. Therefore, every exported resource must be persisted for the lifespan of the linking context to enable later referral to the resource through an import declaration of a lazy-loaded Wasm module.

An import and export declaration in the text format is shown in Figure 1, Line 1 and Line 21.

2.4 Start Section

The entrypoint of a Wasm module is specified in the start section (Figure 1, Line 23). It simply contains a reference to one of the module's functions. The runtime executes the start function as the last step of the module initialization procedure, after all other objects have been properly set up.

2.5 WASI

Wasm was originally created as a bytecode for web browser execution [15]. As such, it includes a robust JavaScript Application Programming Interface (API) that enables the instantiation of Wasm modules, invocation of exported functions, and interaction with Wasm memory. Due to the sandboxed nature of JavaScript within browsers, Wasm can only communicate with the external environment through the runtime-provided APIs. These are limited to web APIs, which are also accessible from standard JavaScript.

To enable Wasm execution outside of the browser, in environments such as Wasmtime [6] or Wasmer [2], the WebAssembly System Interface (WASI) API was introduced by the Wasmtime project. WASI allows Wasm to interact with the underlying system in a platform-independent way, removing the need for a browser environment. The leading implementation of WASI is the wasi-sdk[2] project, which includes wasi-libc, a partial implementation of the C standard library [7], and a modified LLVM compiler infrastructure [17] with Wasm support [13]. The example in Figure 2 was compiled using the clang compiler from the wasi-sdk.

While the Emscripten project provides partial support for WASI APIs, its primary focus is on in-browser execution and JavaScript APIs [27]. Therefore, we focused on utilizing the wasi-sdk.

```c
#include <unistd.h>

int main() {
    write(0, "Hello, World!\n", 14);
}
```

```
(func $main (type 3) (result i32)
    i32.const 0
    i32.const 1024
    i32.const 14
    call $write
    drop
    i32.const 0)

(func $write (type 4)
    (param i32 i32 i32)
    (result i32)
    ;; Omitted wrapper code...
    call $__wasi_fd_write
    ;; Omitted wrapper code...
)

(data $.rodata (i32.const 1024)
    "Hello, World!\0a\00")
```

Figure 2. Comparison between C and wasi-sdk Wasm code for writing a string to the standard output file descriptor using the WASI API.

2.6 Standalone WebAssembly Runtime

As mentioned in Section 2.5, standalone execution of Wasm builts on top of the portability and simplicity of Wasm while enhancing it with APIs that, among others, provide access to the underlying operating system. Therefore, a standalone Wasm runtime can execute a Wasm module with capabilities equal to native system binaries.

2.6.1 Competitors

The increasing popularity of Wasm standalone execution created an excellent breeding ground for open

[2]https://github.com/WebAssembly/wasi-sdk (accessed 2024-01-30)

source Wasm runtimes. We want to highlight some of them in the following paragraphs.

Wasmtime[6] is written in Rust and offers several features that make it the gold standard for many standalone Wasm runtime use cases:

- **AOT Compilation** via Cranelift.
- **JIT Compilation** via Cranelift.
- **Security**: Wasmtime design decisions are made via a formal RFC process. It undergoes 24/7 fuzzing, and there are efforts to formally verify parts of the runtime and compiler.
- **Cross-Platform Support**: Wasmtime offers precompiled binaries for Linux, Windows, and macOS. Written in Rust, it should support most Rust targets as well, and can be embedded in other programs.

WasmEdge [8] is a standalone Wasm runtime that is popular in the blockchain community. It is written in C++.

- **AOT Compilation** via an LLVM-based compiler.
- **Interpretation**: WasmEdge offers an interpreter for Wasm code.
- **Cross-Platform Support**: WasmEdge supports various OSes, microkernels, and hardware architectures.
- **Cloud-native Extensions**: WasmEdge offers a plugin system for cloud-native extensions.

Wasm3 [24] is a WebAssembly (Wasm) interpreter written in C. It is known for its small size and portability. Unfortunately, development has been greatly slowed due to the Russian invasion of Ukraine.

- **Interpretation**: Wasm3 is an interpreter for Wasm code.
- **Small Size**: Wasm3 requires very few resources to run: 64Kb of storage and 10Kb RAM
- **Portability**: Wasm3 is written in C and can be compiled for many platforms, including embedded systems such as microcontrollers.
- **Implementation of proposals**: Wasm3 supports many Wasm proposals, with more planned.

aWsm [11], formerly known as Silverfish, is an LLVM backend and runtime for Wasm written in C. It is optimized for cloud function execution. Notably, aWsm does not include its own compilation times in its benchmarks.

- **AOT Compilation** via LLVM.
- **Cloud**: aWsm is optimized for cloud function execution.
- **Cross-Platform Support**: aWsm can target many platforms, including embedded systems.

3 Implementation

The following section elaborates on our concrete implementation of a fully featured WebAssembly (Wasm) runtime. The actual runtime program contains a high amount of engineering effort and complexity, this report can only scratch the surface of our implementation. Therefore, it is highly recommended to refer to the reference implementation [9] for specific implementation details.

3.1 Loading

Loading is done by simply reading the Wasm binary's bytes, whether from a file or from memory, into a memory buffer. This is then passed to the parser for further processing.

3.2 3-Stage Parsing

Our implementation completes the following three steps in one pass:

3.2.1 Parsing

Wasm's LL(1) [23] grammar lends itself perfectly to table-based parsing. Jump tables are a very fast way of mapping certain types of keys to values: if the keys can be used as (mostly compact) offsets, the corresponding values can simply be stored at that offset. Value retrieval is then as fast as any other array access. Indeed our implementation makes use of such tables for parsing, reading one or a few bytes at a time to represent the module, its data, and its instructions.

3.2.2 Validation

The Wasm specification also specifies how Wasm code can be statically checked to be well-formed and valid before execution [23]. In our implementation, this is done during the parsing step by asserting invariants and simulating the stack types.

3.2.3 SSA Form

The SSA is a type of Intermediate Representation (IR). As the name suggests, this representation assigns to variables only once. Our parser converts Wasm's stack-based instructions into ones which operate on single-assignment variables. This is necessary for the generation of LLVM IR used for compilation as this is also in SSA form.

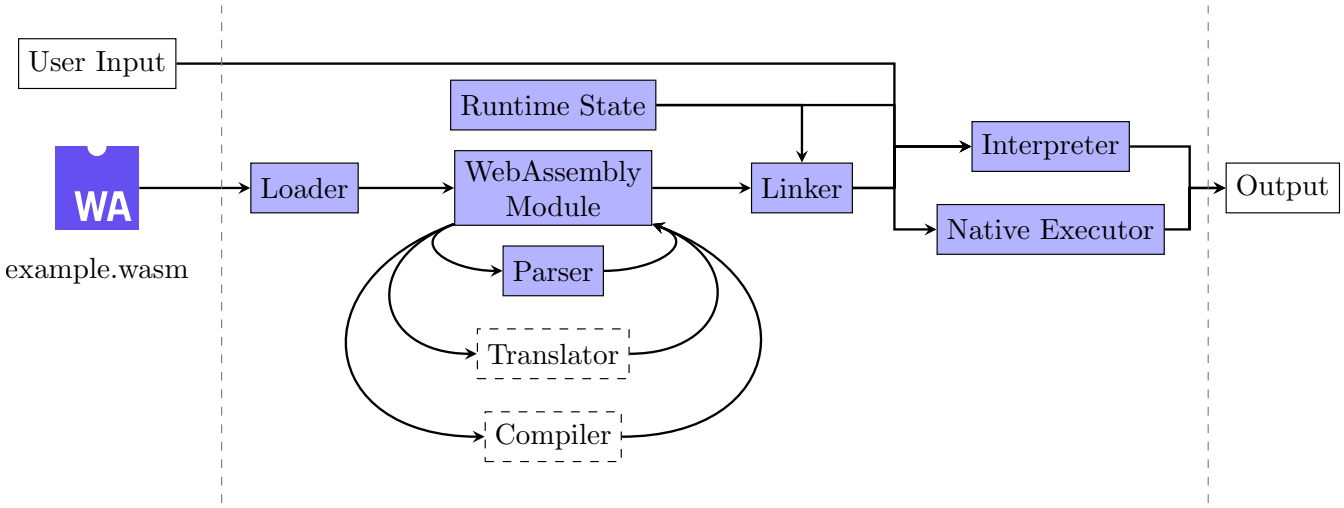

Figure 3. Wasm module execution workflow.

3.3 Implementation of Efficient Function Type Compression

Wasm function types represent the signature of function objects. They are used for ensuring the type safety of function implementations, imported function objects, and indirect function calls. They consist of a tuple of input types, which we call *arguments*, and output types, which we call *results*. Both tuples may be of arbitrary length, where empty tuples are similar to the `void` type in C-like programming languages and represent no parameters or return values.

Wasm code execution requires handling many function type instances because they are used not only for function signatures but also for block arguments and return signatures. During the execution of Wasm code, function types have to be compared frequently, for example, once for every indirect function call.

Wasm function types directly correlate to function signatures of higher-level programming languages. Therefore, the distribution of lengths of the parameter and result tuples in a typical Wasm program is very dense and centered around 0. This allows us to define an efficient encoding for a subset of the infinite set of possible function types.

3.3.1 Requirements

We call the limited set of Wasm types T with $|T| = n$ and $n = 7$ for the current Wasm specification. Subsequently, we define a preliminary maximum number of argument or result types k. However, we want to be able to model function signatures with a variable number of arguments or results. Therefore, we introduce an additional type, $\perp$, which symbolizes no type (*void*).

We start with the encoding of the arguments tuple. The process for the results tuple is identical. A tuple of k types now describes every possible arguments signature, each being either one of the Wasm types or $\perp$, resulting in $(n+1)^m$ possible types. However, we require the mapping function to be bijective to guarantee a deterministic mapping of function signatures to tuples. Therefore, we need to exclude similar tuples. With $k = 3$, $(I32, \perp, I32)$, $(\perp, \perp, I32)$ and $(I32, \perp, \perp)$ all represent a valid encoding for a signature with a signal argument of type `I32`. In order to remove this redundancy, we introduce an invariant, stating that $\perp$ may only appear at the end of a tuple, such that no Wasm type appears after a $\perp$. This invariant only allows the last variant, $(I32, \perp, \perp)$.

3.3.2 A Simple Encoding Approach

One could now come up with the following straightforward approach. Encode each type entry of such a tuple with a certain number of bits. For $T = 7$, we require 8 different configurations of bits to represent every Wasm type and the empty slot $\perp$. We also require a clear boundary between the encoding of different types of the tuple, meaning we may only assign a whole-number amount of bits to each type encoding. For the 8 states, we assign 4 bits per type.

This encoding has two severe limitations. The first limitation is the encoding's poor information density of only 53%. The second limitation is that the invariant

for tuples containing at least one $\perp$ symbol has to be manually enforced, leading to the possibility of invalid encodings.

3.3.3 A Refined Approach

To fix the limitations of the simple encoding, we introduce a refined encoding based on the efficient packing of type encodings. Again, we encode a tuple as an integer of p. Therefore, we assign each Wasm type an ID through the function ID, starting at 0. $ID(\perp) = undef$. We construct the integer p in such a way that it represents the index into the array of all possible tuples (following the previously introduced invariant), sorted by contained IDs, ascending.

For a given tuple of types $TT = (t_0, t_1, \ldots, t_{k-1}) \in T^k$, we calculate the integer p as follows.

$$p(TT) = \sum_{0}^{k} \begin{cases} ID(t_i) \cdot s_{k-i} + 1 & \text{if } t_i \neq \perp \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Where s_i is the i-th partial sum of the geometric series with the common ration $r = |T|$. In particular,

$$s_n = 1 + r + r^2 + \ldots + r^{n-1} \overset{r=|T|\neq 1}{=} \frac{1 - r^n}{1 - r} \tag{2}$$

. [21]

Decoding of $p(TT)$ into the encoded $TT = (t_0, t_1, \ldots, t_{k-1})$ is performed via the inverse of the operation shown in Equation (1).

$$t_i = \begin{cases} \perp & p(TT)_i = 0 \\ ID^{-1}(\frac{p(TT)_i}{s_{k-i}}) & \text{otherwise} \end{cases} \tag{3}$$

$$p(TT)_{i+1} = p(TT)_i - (ID(t_i) \cdot s_{k-i}) \tag{4}$$

For a function type of the form $TT_{param} \rightarrow TT_{return}$, we can now calculate the two numbers $p_{TT_{param}}$ and $p_{TT_{return}}$, and store them together to allow a unique identification of the function type.

In our implementation, we set

$$T = \{\text{I32}, \text{I64}, \text{F32}, \text{F64}, \text{FuncRef}, \text{ExternRef}, \text{Vector}\}$$

with $|T| = 7$ and $k \leq 20$ for encoding of parameters and $k \leq 2$ for result types. This results in

$$\lceil log_2(max(p_{params})) \rceil = \lceil log_2(p(\text{Vector}^{20})) \rceil = 57$$

and

$$\lceil log_2(max(p_{results})) \rceil = 6$$

, so that we can divide our 64-bit function type integer into 1-bit for marking the type value as a pointer into overflow storage, 6 bits for the results index and 57 bits for the parameter type index. The size of the parameter and result types were chosen based on an analysis of benchmark files that showed that all C and C++ programs only use single-value return values while having a maximum of 22 parameters, with most programs having at most 15 parameters.

Therefore, $log_2(max(p)) = log_2(p(\text{Vector}^{10}))$. 10 is the highest amount of types, which still allows for two of the indices to fit into a single 64-bit integer, with the highest bit used to indicate whether the integer stores the index or a pointer to an overflow type storage that holds all types that have more than 10 input or output parameters.

A feature that is possible through our encoding approach is the incremental encoding and decoding of a function type. By performing the calculation presented in Equation (1) in a lazy manner, we can create a function type encoding API that features a builder pattern and a decoding API that works via an iterator over the parameter and results types. This eliminates the need to ever store the encoded types in any collection type, reducing stack and heap allocations.

3.4 Interpretation

Our interpreter operates on the same SSA form as the compiler backends. Instead of generating LLVM IR and then machine code, the interpreter executes the SSA instructions with minor indirection (serialisation and deserialisation), and currently without optimisation. This approach has a much lower start-up time at the cost of a much longer execution time for heavier workloads and is advantageous when running never-executed code [25].

4 Results

In this section, we present the results of our runtime implementation. We evaluate the correctness of our implementation by running the official Wasm specification tests. We also compare the performance of our runtime in interpreter, JIT compilation, and AOT compilation execution modes with other leading standalone Wasm runtimes that support the same execution modes. We use two widely recognized benchmark suites: SPEC CPU[5] and Polybench[20].

4.1 Wasm Specification Tests

The WebAssembly (Wasm) specification repository [26] graciously includes a comprehensive test suite. Using these tests, we can verify the correctness of our runtime implementation for all backends.

Execution Backend	Percentage of Tests Passed
Interpreter	100%
LLVM JIT	100%
LLVM AOT	100%

Figure 4. Specification test compliance of the three offered execution backends.

4.2 LLVM JIT and AOT Compilation

To assess the performance of our runtime in both JIT and AOT execution modes, we compare its end-to-end execution time with other leading standalone Wasm runtimes that also support JIT and AOT execution. In the following benchmarks, these runtimes are Wasmtime [6] and WasmEdge [8]. Wasmtime uses a custom compiler backend called Cranelift for its compilation process, whereas WasmEdge, like our runtime, is based on LLVM.

We evaluate the runtimes using two widely recognized benchmark suites: SPEC CPU®,2017 [5] and Polybench [20]. SPEC CPU®,2017, created by the Standard Performance Evaluation Corporation (SPEC), includes computationally intensive, real-world programs written in C, C++, or Fortran [5]. Although these benchmarks are widely used and well-known, they are designed for native code execution and compilation with fully featured compilers. As a result, compiling the SPEC CPU®,2017 benchmarks to Wasm presents challenges due to the limited feature set of the wasi-sdk C and C++ compiler and constraints imposed by the Wasm specification. Despite these challenges, we were able to successfully compile and execute four different benchmarks of the suite.

- `505.mcf_r`, a C-program for scheduling public mass transport.
- `508.namd_r`, part of a biomolecular system simulator.
- `519.lbm_r`, a fluid simulator.
- `557.xz_r`, a compression utility.

These build and execution challenges of the SPEC CPU® 2017 benchmarks for Wasm lead many other runtimes to resort to simpler, open source benchmarks. A commonly utilized benchmark suite is the Polybench suite, which is comprised of 30 numerical computations, extracted from real-world applications. Its popularity is predominantly based on its simplicity, as its benchmark codes do not interact with the operating system and do not require any third-party dependencies or inputs. [20]

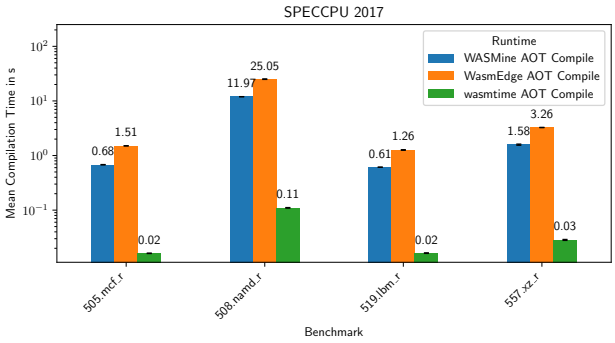

Figure 5. SPEC CPU® 2017 benchmark compilation times.

We recorded the compilation times of all tested AOT runtimes and present them in Figure 8 and Figure 5. For both benchmark suites it is evident that the wasmtime runtime is about 1-2 orders of magnitude faster than both WASMine and WasmEdge. They feature comparable compilation times as they are both based on the LLVM compiler framework, even though it is notable that WASMine consistently outperforms WasmEdge.

The execution times are shown in Figure 9 and Figure 10 for the Polybench benchmark suite, and in Figure 6 for the SPEC CPU® 2017 suite. For the latter, the WasmEdge runtime's JIT mode did not execute any benchmark within the cutoff time of 1 minute and, therefore, has no exact execution time measurements.

For the SPEC CPU® 2017 suite, we can see that AOT consistently outperforms JIT, which was to be expected, because we do not include the required compilation time for AOT in the shown execution times. WASMine JIT is consistently faster than the WasmEdge JIT execution, but significantly slower than wasmtime JIT. Meanwhile WASMine AOT execution, besides its comparably long compilation times, features either a slightly higher execution time than wasmtime for longer-running benchmarks like `505.mcf_r`, or has a similar performance to wasmtime and WasmEdge, even outperforming both on the `519.lbm_r` benchmark.

The observations for the SPEC CPU® 2017 benchmark suite also apply to the Polybench benchmark suite. To highlight the execution time comparison of short-running benchmarks, we extracted all benchmarks that have an execution time shorter than 2 seconds for all runtimes (except WasmEdge) into Figure 10. For some benchmarks, the WasmEdge runtime's JIT mode did not finish execution until the cutoff time of 1 minute, symbolized through a missing bar.

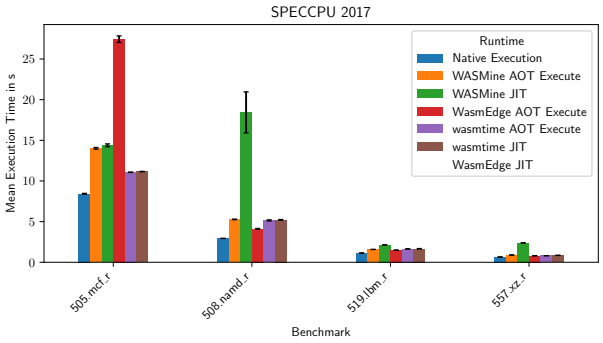

Figure 6. SPEC CPU® 2017 benchmark execution times of AOT and JIT compiled code.

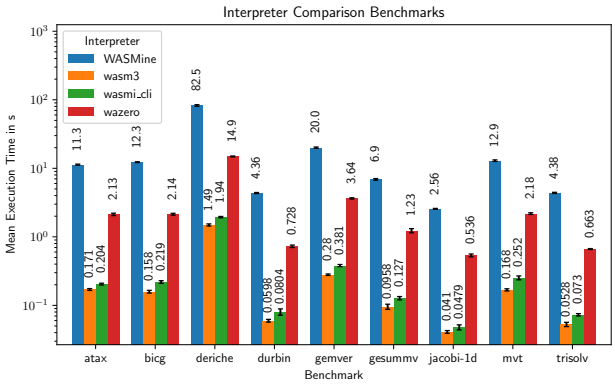

Figure 7. Selected polybench benchmark execution times of interpreters.

4.3 Interpreter

To assess the performance of our interpreter, we compare its end-to-end execution time with other leading standalone Wasm runtimes that also support interpretation. In the following benchmarks, these runtimes are Wasm3 [24], Wasmi [12], and Wazero [16]. Wasm3 is a Wasm interpreter written in C, known for its small size, speed, and portability. Wasmi is a Wasm interpreter written in Rust, while Wazero is a Wasm interpreter written in Go and prides itself on its lack of dependencies. For these interpreters, we chose only a subset of the Polybench [20] benchmark suite, as the SPEC CPU[5] benchmarks and many of the PolyBench suite are too computationally intensive for an interpreter to execute within a reasonable time frame. The benchmarks were executed using Hyperfine [19] on each interpreter's CLI tool with 3 warm-up runs and a minimum of 10 benchmark runs. As such, they include the time required for parsing, any transformation, and interpretation of the Wasm code.

This comparison shows that our interpreter is not competitive with other implementations as it is significantly slower, consistently being outperformed by even the next slowest interpreter, Wazero. Wasm3 is consistently the fastest interpreter in these benchmarks, always outperforming our implementation by over an order of magnitude. Wasmi follows Wasm3 closely in performance.

5 Summary & Outlook

In the course of this project, we have implemented a fully-featured Wasm runtime. We developed a parser that can parse Wasm modules, validate them, and convert them into an SSA form and a linker to link them. We implemented an interpreter that can execute Wasm code in this form. We also implemented a JIT compiler that can compile Wasm code into native machine code at runtime. Furthermore, we implemented an AOT compiler that can compile Wasm code into native machine code ahead of time. We implemented a WASI API that allows Wasm code to interact with the underlying operating system. In order to efficiency encode and decode function types at runtime, we implemented a function type compression algorithm. To tie it all together, we implemented a runtime library the backends make use of. We evaluated the correctness of our implementation by running the official Wasm specification tests and compared the performance of our runtime and all its execution modes with that of competing runtimes.

We learnt about the Wasm specification and language, the WASI API, and the different execution modes of Wasm code. By implementing this runtime, we learnt everything from parsing and validating Wasm modules, through LLVM and its API, interpreter design, and JIT compilation.

5.1 Future Work

Possible future work includes improvements to the performance of the interpreter through meta-instructions for commonly found instruction sequences and patterns.

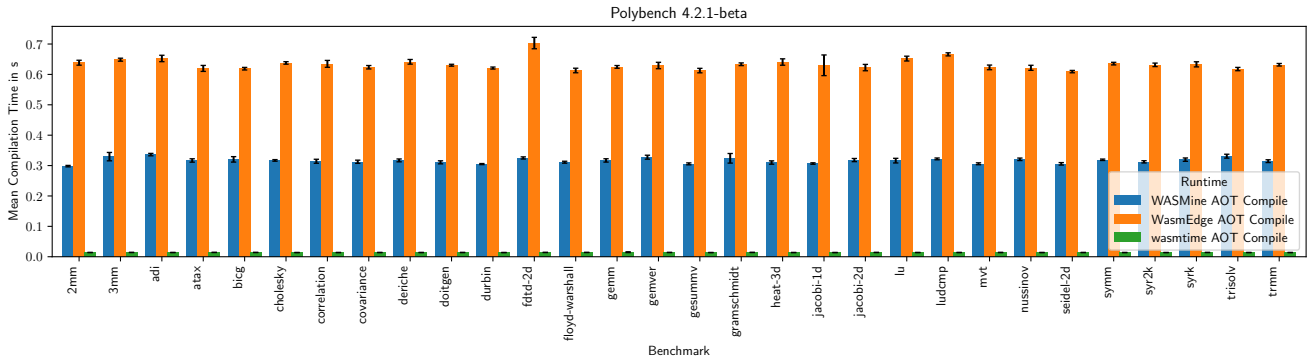

Figure 8. Polybench benchmark compilation times.

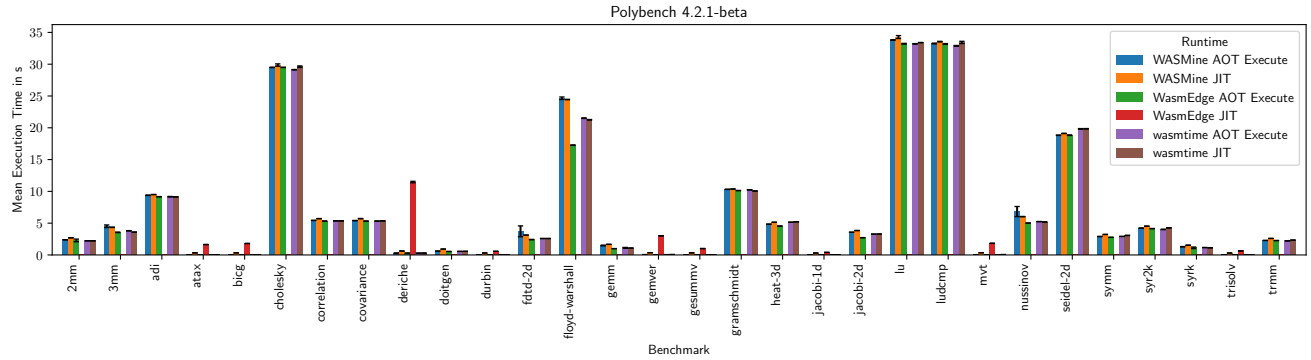

Figure 9. Polybench benchmark execution times of AOT and JIT compiled code.

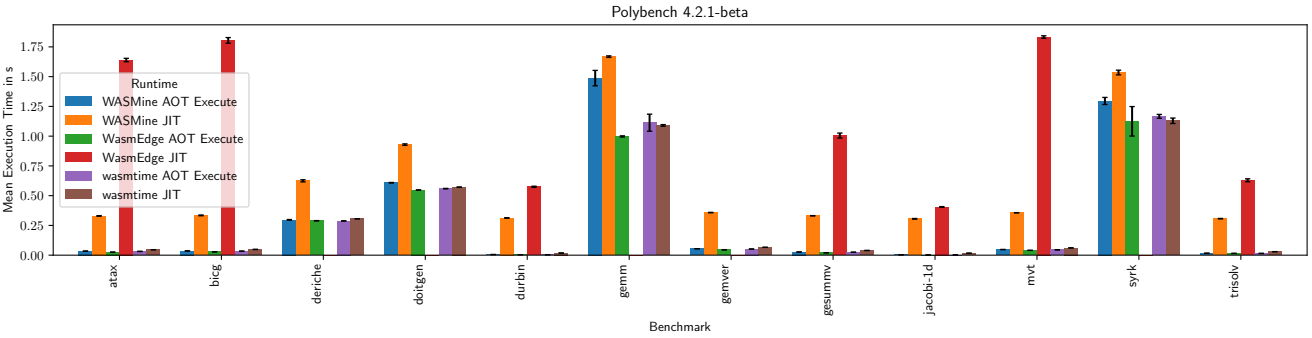

Figure 10. Polybench benchmark execution times of AOT and JIT compiled code, focused on benchmarks running for less than 2 seconds on all runtimes.

References

[1] [n. d.]. threads/proposals/threads/Overview.md at master · WebAssembly/threads. https://github.com/WebAssembly/threads/blob/master/proposals/threads/Overview.md

[2] [n. d.]. Wasmer. Wasmer Inc.. https://wasmer.io

[3] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[4] Apple Inc. [n. d.]. JavaScriptCore. https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html

[5] James Bucek, Klaus-Dieter Lange, and Jóakim V. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, Berlin Germany, 41–42.

https://doi.org/10.1145/3185768.3185771
[6] Bytecode Alliance. [n. d.]. Wasmtime. https://wasmtime.dev/
[7] C Standards Committee - ISO/IEC JTC1/SC22/WG14. 2018. *International Standard ISO/IEC 9899:2018(E) — Programming Language C*. Standard. International Organization for Standardization, Geneva, CH.
[8] Cloud Native Computing Foundation. [n. d.]. WasmEdge. https://wasmedge.org/
[9] Lukas Doellerer and Enrico Fazzi. 2024. WASMine — An AOT & JIT Compiling and Interpreting WebAssembly Runtime. https://gitlab.db.in.tum.de/epd24s/wasm-rt/
[10] ECMA. 2025. ECMAScript® 2025 Language Specification. https://tc39.es/ecma262/
[11] The Embedded and Operating Systems group at GWU. [n. d.]. aWsm. https://github.com/gwsystems/aWsm
[12] Robin Freyler. [n. d.]. Wasmi. https://github.com/wasmi-labs/wasmi
[13] Dan Gohman. 2019. *WASI: WebAssembly System Interface*. Technical Report. bytecodealliance. https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md
[14] Google. [n. d.]. V8 JavaScript Engine. https://v8.dev/
[15] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363
[16] Tetrate Labs. [n. d.]. Wazero. https://github.com/tetratelabs/wazero
[17] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665
[18] Mozilla Corporation. [n. d.]. SpiderMonkey JavaScript/WebAssembly Engine. https://spidermonkey.dev/
[19] David Peter. [n. d.]. Hyperfine. https://github.com/sharkdp/hyperfine
[20] Louis-Noel Pouchet and Tomofumi Yuki. [n. d.]. Polybench 4.2.1-beta. https://sourceforge.net/projects/polybench/
[21] Murray H. Protter and Charles B. Morrey. 1985. *Intermediate Calculus*. Springer New York. https://doi.org/10.1007/978-1-4612-1086-3
[22] Ronald Rivest and Donald Eastlake. 2023. *S-expressions*. Internet-draft draft-rivest-sexp-03. Internet Engineering Task Force / Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-rivest-sexp/03/
[23] Andreas Rossberg. 2022. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-2/
[24] Volodymyr Shymanskyy and Steven Massey. [n. d.]. wasm3. Wasm3 Labs. https://github.com/wasm3/wasm3
[25] Ben L. Titzer. 2022. A Fast In-Place Interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 646–672. https://doi.org/10.1145/3563311
[26] WebAssembly Community Group. [n. d.]. WebAssembly Specification Repository. https://github.com/WebAssembly/spec
[27] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, Portland Oregon USA, 301–312. https://doi.org/10.1145/2048147.2048224