TruffleWasm: A WebAssembly Interpreter on GraalVM

DOI:
10.1145/3381052.3381325

TruffleWasm: A WebAssembly Interpreter on GraalVM

Salim S. Salim
University of Manchester
Manchester, UK
salim.salim@manchester.ac.uk

Andy Nisbet
University of Manchester
Manchester, UK
andy.nisbet@manchester.ac.uk

Mikel Luján
University of Manchester
Manchester, UK
mikel.lujan@manchester.ac.uk

Abstract

WebAssembly is a binary format originally designed for web-based deployment and execution combined with JavaScript. WebAssembly can also be used for standalone programs provided a WebAssembly runtime environment is available.

This paper describes the design and implementation of *TruffleWasm*, a *guest* language implementation of a WebAssembly hosted on Truffle and GraalVM. Truffle is a Java framework capable of constructing and interpreting an *Abstract Syntax Tree (AST)* representing a program on standard JVMs. GraalVM is a JVM with a JIT compiler which optimises the execution of ASTs from Truffle.

Our work is motivated by trying to understand the advantages and disadvantages of using GraalVM, and its support for multiple programming languages, to build a standalone WebAssembly runtime. This contrast with developing a new runtime, as Wasmtime and other projects are undertaking. TruffleWasm can execute standalone WebAssembly modules, while offering also interoperability with other GraalVM hosted languages, such as Java, JavaScript, R, Python and Ruby.

The experimental results compare the peak performance of TruffleWasm to the standalone Wasmtime runtime for the Shootout, C benchmarks in JetStream, and the PolyBenchC benchmarks. The results show the geo-mean peak performance of TruffleWasm is 4% slower than Wasmtime for Shootout/JetStream, and 4% faster for PolyBenchC.

CCS Concepts • **Software and its engineering → Interpreters**; **Runtime environments**;

Keywords WebAssembly, Wasm, JVM, GraalVM, Just In Time compilation

ACM Reference Format:
Salim S. Salim, Andy Nisbet, and Mikel Luján. 2020. TruffleWasm: A WebAssembly Interpreter on GraalVM. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3381052.3381325

VEE '20, March 17, 2020, Lausanne, Switzerland

1 Introduction

WebAssembly, sometimes abbreviated as Wasm, is a compact, portable, statically typed stack-based binary format. WebAssembly is easy to parse, specialise, and optimise compared to equivalent code fragments in JavaScript [8, 19, 20]. Performance critical components of web applications can be expressed in languages such as C, C++ or Rust. These components are compiled into WebAssembly which typically executes faster than the equivalent JavaScript. Thus, initial WebAssembly implementations have focused on client-side language execution on the browser to support tight integration with JavaScript and its APIs. Nonetheless, recent development efforts have included WebAssembly targets for standalone execution, such as on IoT, embedded, mobile devices, and servers.

Standalone WebAssembly environments require a runtime to provide access to IO and external resources, leading to the *WebAssembly System Interface (WASI)* that defines a *POSIX*-like interface. Standalone environments can provide their own developed optimising compiler, but typically, they reuse existing back-ends such as LLVM [11]. For example, Wasmer (see Section 7) provides an option to choose between different deployment specific compiler back-ends balancing compilation time vs. quality of generated code. Browser engines typically use less aggressive optimisations with *Ahead of Time* (AoT) compilation because of the relatively short execution times that are expected for client-side scenarios.

An alternative approach to support WebAssembly could be based on a Java Virtual Machine (JVM). Modern JVMs provide a highly optimised managed runtime execution environment that has been ported to many different target platforms. In addition, they support the JVM compiler interface (JVMCI) which, for example, enable the Graal JIT compiler to be integrated with the JVMs part of the OpenJDK community. The Truffle framework [25] enables *guest* language interpreters with competitive runtime performance to be hosted on a JVM with relatively low development effort in comparison to native implementations. GraalVM[1], is a JVM distribution packaged with the Graal JIT compiler which

[1] GraalVM website https://www.graalvm.org/

provides an aggressive general set of optimisations, and specific transformations for Truffle-hosted guest languages [24] as well as cross programming language interoperability [7] [6] [3]. Using a JVM provides access to a wide range of the Java ecosystem support tools, such as debugging and instrumentation [18], including those designed specifically for GraalVM. Cross-language interoperability in GraalVM could allow efficient embedding of WebAssembly with other Truffle-hosted languages; such as GraalJS (JavaScript), FastR (R), GraalPython (Python), and TruffleRuby (Ruby).

We present TruffleWasm, the first implementation of a WebAssembly language interpreter using Truffle and GraalVM with support for WASI, and compare its performance to the standalone *Wasmtime* implementation. TruffleWasm implements version 1.0 (*Minimal Viable Product*) of the specification, and passes all core WebAssembly spec-tests provided by the specification. In summary, the main contributions of this paper are:

- A WebAssembly interpreter implemented using Truffle on the JVM that exploits partial evaluation and JIT compilation (See Section 4).
- A peak-performance comparison between WebAssembly using TruffleWasm and Wasmtime (See Section 6).
- An evaluation of WebAssembly features and how they map to Truffle language implementation framework.
- An addition to the GraalVM ecosystem which allows other Truffle languages to reuse existing WebAssembly modules and libraries using Truffle interoperability (See Section 4).

We select Wasmtime as the main comparison point as it is the main standalone WebAssembly runtime with support for WASI required by the benchmarks. An anecdotal inspection of relative complexity using Lines of Code as a rough and ready metric shows that Wasmtime requires more than double the number of lines of code than TruffleWasm. The next sections of the paper are organised as follows. Section 2 introduces background material on WebAssembly, its runtimes and execution use-cases. Section 3 presents a short high-level overview of how Truffle and Graal support the implementation of *Abstract Syntax Tree* (AST) interpreters for hosting *guest* language execution on JVMs. Section 4 discusses the design and implementation of TruffleWasm. Section 5 describes the experimental methodology and benchmarks used to compare TruffleWasm against Wasmtime, while Section 6 discusses the results. Section 7 presents relevant related work. Conclusions are presented in Section 8.

2 WebAssembly Overview

The main concepts and features of WebAssembly, and the important aspects of browser engine execution support are presented. WebAssembly code is compiled in a single scope called a *module* where different components are defined that specify the functionality of a whole program as shown in Figure 1. In the C/C++ ecosystem, there are currently three ways for C/C++ front end compilers to generate WebAssembly modules. By compiling core execution source code to WebAssembly and wrapping I/O and other necessary initialisation in JavaScript, using standardised WASI API functions, or by providing a standalone[2] module where standard library functions are added into a module as imports. Standalone runtimes must implement any imported functionality using their own (library code) mechanisms.

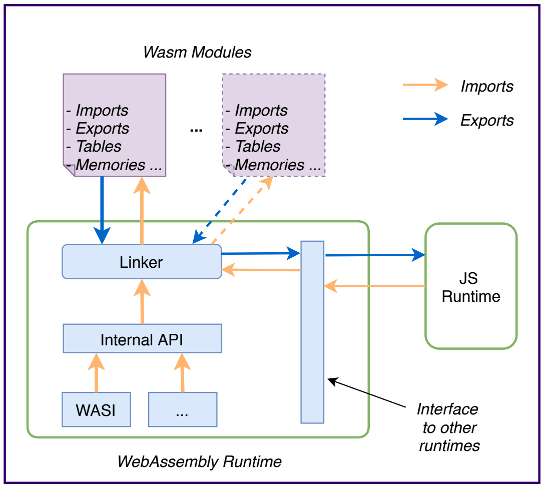


Figure 1. WebAssembly modules on an abstract runtime.

2.1 Imports and Exports

Imports and exports are key features for inter-module and language interoperability. Imported components are expected to be supplied by a host runtime, such as a JavaScript runtime, or a standalone runtime defining its own built-ins or exports from another module. Clearly, imports such as JavaScript built-ins, and memory (that describe how storage is assigned to a module) or table definitions are likely to influence overall performance. For example, if a module requires significant interactions with its host runtime, then it will be heavily dependent on the host's implementation of the imported functions.

For instance, if a JavaScript code creates an `ArrayBuffer` data structure that is sent to WebAssembly as an import, then the WebAssembly code can use this as its linear memory storage, and any writes to this storage are visible to both JavaScript and WebAssembly.

2.2 WASI API and Standalone Imports

The WebAssembly System Interface[3] official specification outlines a modular standard API. The core module of WASI provides an API that covers aspects such as file system and

[2] https://github.com/kripken/emscripten/wiki/WebAssembly-Standalone
[3] https://wasi.dev/

```
(import (func $__wasi_fd_prestat_get (type $t2)))
(import (func $__wasi_fd_prestat_dir_name (type $t0)))
(import (func $__wasi_environ_sizes_get (type $t2)))
(import (func $__wasi_environ_get (type $t2)))
(import (func $__wasi_args_sizes_get (type $t2)))
(import (func $__wasi_args_get (type $t2)))
(import (func $__wasi_proc_exit (type $t3)))
(import (func $__wasi_clock_time_get (type $t10)))
(import (func $__wasi_fd_fdstat_get (type $t2)))
(import (func $__wasi_fd_close (type $t4)))
(import (func $__wasi_fd_seek (type $t5)))
(import (func $__wasi_fd_write (type $t6)))
```

Listing 1. WASI functions required by a typical Shootout benchmark.

networking based interactions. WASI provides a standardised, POSIX interface with a CloudABI [16] capability-based access that enables WebAssembly modules to interact with a conceptual system. Import functions (See Listing 1 for an example) are defined from the `wasi` module. Standalone WebAssembly runtimes that support `wasi` based modules must provide implementations for the APIs as outlined by the standard.

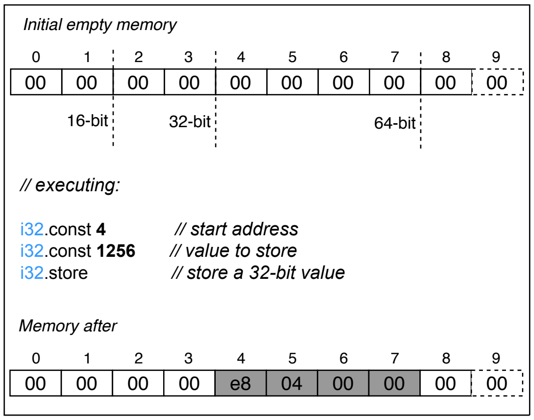
`Emscripten`, `WasmExplorer`, `LLVM` and other languages tools provide an option to generate standalone modules, where no JavaScript code is generated. Standard library functions are added as imported functions and it is the runtime's responsibility to provide their implementation.

Many WebAssembly runtimes (See Section 7) support only one deployment option. For example, browser engines typically only support modules that import from JavaScript. Wasmtime supports WASI targeted modules, whereas Wasmer can execute both Emscripten standalone and WASI modules. To execute a WebAssembly module on a different deployment target runtime may require the original source to be recompiled with appropriate flags.

The specific compilation target for WebAssembly modules and any associated dependencies can have different effects on the WebAssembly module performance when executed on different implementations. Even though the same original source code is used, a compiled module with a JavaScript wrapper will perform differently compared to a WASI or a standalone module. In summary, the environment where a WebAssembly module obtains its required imports influences how it performs and behaves. Note, as WebAssembly runtimes are being embedded in many other languages, the need for high-performance optimised language interoperability functionality is essential because the cross-language performance wall associated with embedding multiple runtimes will be visible in the overall performance.

2.3 Linear Memory and Tables

Many WebAssembly modules require interactions with linear memory that provides a raw byte array addressed using an index. Using special memory access operations, a specific 32/64 bit integer or floating point type (`i32`, `i64`, `f32` or `f64`) can be read/stored from/to an array of bytes in memory.



0	1	2	3	4	5	6	7	8	9
00	00	00	00	00	00	00	00	00	00

```
// executing:

i32.const 4        // start address
i32.const 1256     // value to store
i32.store          // store a 32-bit value
```

0	1	2	3	4	5	6	7	8	9
00	00	00	00	e8	04	00	00	00	00

Figure 2. A linear memory operation example.

External APIs, such as WASI and other JavaScript imports, may use linear memory to communicate with user functions by reading and writing into a pre-defined offset in linear memory. Memory `load` and `store` instructions also provide an alignment value to ensure that the memory accessed by a load/store operation is n-byte aligned. This value can be used by runtimes as an optimisation hint to provide aligned memory access where beneficial.

Figure 2 shows a linear memory operation which takes a value `1256` and stores it into memory at address 4. The value is stored in `little-endian` and 4 bytes in the memory are modified. Tables, another global element of a module, store function references (`anyfunc`, see Listing 4) in an array structure. They are used by an indirect call operation to implement function pointer calls available in languages such as C/C++.

2.4 Inside a Function Body

To illustrate different features of WebAssembly, we demonstrate how a simple C code snippet from Listing 2 compiles to WebAssembly code in Listing 3. Please note, in this illustration, all C-code structures and arrays are stored in a linear memory, and WebAssembly functions will access such structures using a 32-bit integer value as a pointer to the beginning of such data in a linear memory. Local variables and function arguments are indexed from zero. So `local.get 0` will get the first local variable (and for functions with arguments, that will be the first argument).

WebAssembly code follows a structured control flow. That is `br*` operations can only jump to one of the enclosing blocks. The values of the `br*` instruction (Lines 7 and 23), specifies how many blocks outside to break to, with zero specifying the inner most block relative to the instruction. To maintain structured control, a compiler targeting WebAssembly would convert unstructured flow to a structured

```
typedef struct tn {
   struct tn*    left;
   struct tn*    right;
} treeNode;

long count(treeNode* tr) {
   if (!tr->left)
     return 1;
   else
     return 1 + count(tr->left) + count(tr->right);
}
```

Listing 2. A simple code snippet in C.

```
1   func count ;; (i32) -> i32
2       block
3           local.get       0
4           i32.load        0
5           local.tee       1
6           i32.eqz
7           br_if           0   # 0: down to label0
8           i32.const       1
9           local.set       2
10          loop            # label1:
11              local.get       2
12              local.get       1
13              i32.call        count@FUNCTION
14              i32.add
15              i32.const       1
16              i32.add
17              local.set       2
18              local.get       0
19              i32.load        4
20              local.tee       0
21              i32.load        0
22              local.tee       1
23              br_if           0   # 0: up to label1
24          end_loop
25          local.get       2
26          return
27      end_block       # label0:
28      i32.const       1
29  end_function
```

Listing 3. WebAssembly code for the example in Listing 2, compiled with `clang 8.0` using Compiler Explorer.

one by introducing multiple (nested) blocks. For a code segment with a complex logic, the nesting depth can be large. For example, `printf_core`[4] function generated by clang as part of the WebAssembly module, contains around 300 blocks + loops with the nested depth of up to 72. A loop block has its target label at the beginning of that block. That is a `br*` instruction targeting a loop block will cause a block to run another iteration, and if a block finishes without any jump back to the beginning, then the loop stops and execution continues to the next instruction.

2.5 Using WebAssembly from JavaScript

WebAssembly modules can be called from JavaScript using the `WebAssembly.*` JavaScript API, which is supported by the four major browsers. Listing 4 illustrates a simple example of how a WebAssembly module is loaded and instantiated from JavaScript. It shows (simple and cut-down) JavaScript code that initiates a WebAssembly module and runs a function called `hello() (see Line 23)`. Other than JavaScript, a recent published proposal provides *Interface Types* to allow WebAssembly to interoperate with arbitrary languages using a single interface[5].

```
1   const fs = require('fs');
2   async function run() {
3     function createWebAssembly(bytes) {
4       const memory = new WebAssembly.Memory({initial: 256,
5                                              maximum: 256 });
6       const table = new WebAssembly.Table({
7                   initial: 0, maximum: 0, element: 'anyfunc' });
8       const env = {
9         table, __table_base: 0,
10        memory, __memory_base: 1024,
11        STACKTOP: 0, STACK_MAX: memory.buffer.byteLength,
12      };
13    return WebAssembly.instantiate(bytes, { env });
14  }
15    const result =  await createWebAssembly(
16    new Uint8Array(fs.readFileSync('test.wasm')));
17    console.log(result.instance.exports.hello());
18  }
19  run();
```

Listing 4. A simple JavaScript WebAssembly API call in Node.js. First memory is created plus any other required imports by a WebAssembly module. The module is then instantiated and its exported members can be accessed from JavaScript.

Front-end tools such as `Emscripten` generate both WebAssembly and JavaScript code. JavaScript APIs are used to instantiate and interact with WebAssembly modules. The JavaScript APIs include functions to handle features such as I/O operations and exception handling, and to provides stubs for accessing the C/C++ standard library. The WebAssembly code can call JavaScript functions as imported functions. Such calls between WebAssembly and JavaScript (or any other language) create a language wall or barrier that may influence how the overall module performance is measured or interpreted.

2.6 Tiered Compilation in Browsers

The JavaScript engines for Google Chrome's V8, Mozilla Firefox's SpiderMonkey, and Webkit's JavaScriptCore all initially started to support WebAssembly by AoT compiling modules on arrival, and then later with streaming compilation. The engines typically reused their existing JavaScript JIT compilers for AoT compilation. Chakra (a former JavaScript engine in Microsoft Edge), used lazy function interpretation followed by JIT compilation of hot functions [8]. The AoT approach targets predictable peak performance and reduces unpredictability associated with JavaScript JIT warm-up times. This helped achieve faster start-up and lower memory consumption for Microsoft Edge using Chakra.

Nevertheless, compilation times were still significant for larger modules [9]. Thus, both V8 and SpiderMonkey now

[4] From Wasi libc: https://github.com/CraneStation/wasi-libc/blob/master/libc-top-half/musl/src/stdio/vfprintf.c

[5] https://hacks.mozilla.org/2019/08/webassembly-interface-types/

provide a tiered compilation approach where modules are first compiled using a baseline compiler (called Baseline in SpiderMonkey and Liftoff in V8) as modules arrive. The baseline provides a fast and efficient first tier compiler that decodes the module, performs validation, and emits machine code in a single pass. Hot paths are identified and JIT compiled using an optimising compiler (IonMonkey in SpiderMonkey and TurboFan in V8) to produce more efficient machine code. The same optimising compilers are used as top tier compilers within the JavaScript compilation pipeline. Converting WebAssembly instructions into an AST for interpretation is much slower than a *decode->validate->generate* approach used by a baseline compiler. Hence, the interpreter, used as a first tier for JavaScript, is only used for WebAssembly when in debug mode.

2.7 JIT Compilation for WebAssembly Modules

Many server side applications are long running services that can amortise the overheads of lengthy compilation times with aggressive optimisation to improve performance. Similarly, we leverage Truffle and Graal, that collect profiling information, to aid efficient application of aggressive optimisations.

For environments where start-up times are important, such as embedded devices that require low memory and energy footprints, Truffle interpreters can be AoT compiled to produce smaller binaries that are appropriate for such use cases [26]. Together these two configurations cover many use cases outside of the client-side browser. The re-configurable nature of the JVM, allows the interpreter to be supplied with additional JVM flags to cater for the specific execution environment requirements. For example, a GraalVM language can be passed options to configure the underlying JVM for a specific heap size and Garbage Collection implementation, as well as a range of other flags controlling JVM features.

3 GraalVM and the Truffle Framework

The Truffle framework provides a Java API for building guest language Abstract Syntax Tree (AST) interpreters on any JVM. The framework also allows annotating AST nodes during interpretation with additional information. That information is used by the Graal JIT compiler during the optimisation phase [7] as to enable custom compilation and lowering of guest language features. Other features of the Truffle framework are:

- A *self-optimising interpreter* to support dynamic languages by specialising nodes based on run-time type information.
- *Type system and domain-specific language* utilities for type management and mapping between GraalVM languages.
- *Interoperability API* for languages to interoperate with other languages hosted by GraalVM [7] to give efficient access to code and data storage.

The interoperability (Interop 2.0) and `TruffleLibrary` features included in newer Truffle versions (since version 19.0) allow polymorphic inline caching and profiling between language-boundary calls. This has improved performance, provided a new protocol for message passing, and reduced the memory footprint required for interpreters.

3.1 AST Interpreters using the Truffle Framework

To model the stack based WebAssembly code using Truffle nodes, WebAssembly blocks (such as functions, loops and other block kinds) contain individual instructions that make up the block as their children nodes. In each WebAssembly block, each instruction which pops a value(s) from a WebAssembly stack will maintain an instruction that previously pushed a value(s) to the stack as its child(ren) node(s) in a tree form (AST). For instance, WebAssembly from Listing 3 will be converted into an AST as illustrated in Figure 3.

Each Truffle node contains an `execute` method which implements the execution logic for interpretation, including calling to its children and handling control flow. Truffle AST interpretation transfers control flow between nodes using Java exceptions. Logic for instructions such as function return or break are implemented using `ControlFlowException`. When a control flow exception is thrown, the current node stops its execution, and transfers execution to the parent node, which may catch or propagate the exception upwards to its parent node.

Truffle interpreters use profiling to provide hints for the partial evaluation phases. These hints can enable better machine code to be generated during JIT compilation. Truffle profiling of runtime execution behaviour involves each node collecting information, such as branch and value profiling, and identifying any run-time constant values. Values that can be constant during JIT compilation, but are not declared as final during runtime profiling can be annotated. Interpreters can also utilise Truffle assumptions to guide optimisation decisions. Truffle assumptions are used to let the compiler optimistically treats state of an object as unchanged. When the assumption is invalidated, the specialisation cached code guarded by that assumption is also discarded. The interpreters can also limit the number of cached copies of generated code, such as for value-based specialisations of an integer add operation.

3.2 Truffle Native Function Interface (TruffleNFI)

The Truffle framework provides an optimised mechanism for interpreters to call native code via the TruffleNFI[6]. When running on a regular JVM, this is mapped into a Java Native

[6] https://github.com/oracle/graal/blob/master/truffle/docs/NFI.md

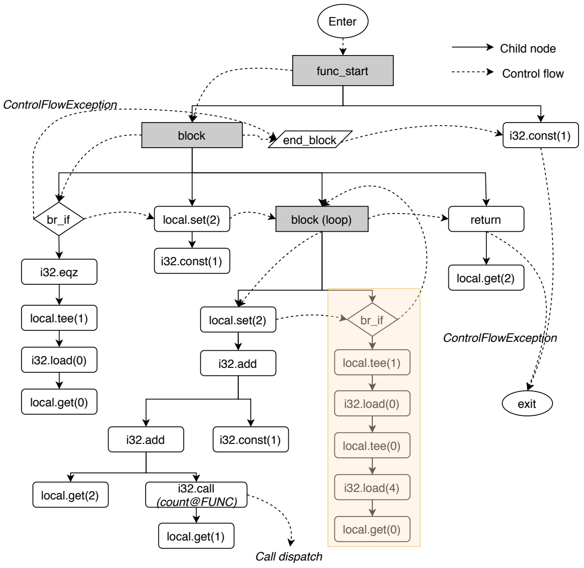


Figure 3. AST and control flow of the WebAssembly code from Listing 3. The area highlighted in yellow corresponds to Lines 18 to 23.

Interface (JNI). Using `libffi`, TruffleNFI wraps the necessary functionalities to allow a guest language to access native functions that are not available in Java and even to provide a Foreign Function Interface (FFI) for their language.

3.3 The Graal Compiler

The Graal JIT compiler applies speculative optimisations and partial evaluation to optimise its internal graph-based intermediate representation [2]. Partial evaluation allows interpreters to be specialised with respect to the current values and types associated with AST nodes. The evaluator performs aggressive optimisations such as constant propagation and method in-lining using assumptions associated with current values [12]. Aggressive optimisation of node specialisation are constrained by guards (such as Truffle assumptions), which when not valid, may trigger re-specialisation or discard the generated code and return to the interpretation mode in a process typically referred to as *deoptimisation.*

3.4 GraalVM Native Image

GraalVM native image tools allow AoT compilation of JVM-based languages into a native executable. Truffle language interpreters can be AoT compiled to produce an executable having a smaller memory footprint, lower overhead and faster start-up times. The native image can then be run standalone with no JVM overhead. Another advantage provided by the native image support is that a shared library can be used to expose a C function (C-API) that can be called from other languages, such as from Java through the Java Native Interface (JNI). This is generally relevant for our use case, as it can be used to expose the functionality of TruffleWasm as C-APIs to other languages as specified in the proposal [7].

4 Design and Implementation

Figure 4 illustrates the TruffleWasm high level design. It uses `Binaryen`[8], a compiler infrastructure tool-chain for WebAssembly, for parsing and static validation. After parsing, `Binaryen` converts a stack-based WebAssembly IR source into its tree-based internal IR.

Two goals of WebAssembly are to be efficient and safe [8]. As such, WebAssembly engines are expected to deliver relatively good performance whilst sand-boxing execution from the underlying host machine.

In the initial implementation of TruffleWasm, WebAssembly features/elements such as *globals*, *builtins* and *table* are implemented using `TruffleObject` and Truffle Nodes. This is a default way for Truffle interpreters to build language features. Linear memory on the other hand, provides a different challenge. In JavaScript, when creating a new memory with `WebAssembly.Memory` (See Listing 4, Line 5), an `ArrayBuffer` is created with the specified size and limit[9]. In GraalJS for instance, `TypedArray` and `ArrayBuffer` are backed by a Java `ByteBuffer` or a `byte[]` depending on the implementation option. GraalJS provides different optimisation and specialisation for arrays of different types. We applied the same technique and provided linear memory as a `TruffleObject` encapsulating a `ByteBuffer`. Nevertheless, accessing a byte array for reading and writing to memory did not produce good performance (See Section 6). This is also because WebAssembly modules have more loads and stores as discussed in [10].

For the above limitations, Truffle and Java provide the following solutions:

- Specialisation of linear memory read and writes using Truffle *Object Storage Model* [23].
- A linear memory implementation using `Unsafe` API and the wrapping of WASI APIs using TruffleNFI (Section 3).

The second option provides one more advantage, by managing native memory, TruffleWasm can add memory alignment optimisation in the future for `load` and `store` operations. As such, TruffleWasm currently implements linear memory using Java `Unsafe` API and bound checks all accesses to native memory. The WASI API calls are wrapped as C++ functions that control access as specified by the standard and are called by TruffleWasm using TruffleNFI. Currently a slower version using `ByteBuffer` is available through a flag `--wasm.emulated=true`. The rest of the sub-sections explain the TruffleWasm approach for implementation of

[7] https://github.com/WebAssembly/proposals

[8] https://github.com/WebAssembly/binaryen

[9] https://hacks.mozilla.org/2017/07/memory-in-webassembly-and-why-its-safer-than-you-think/

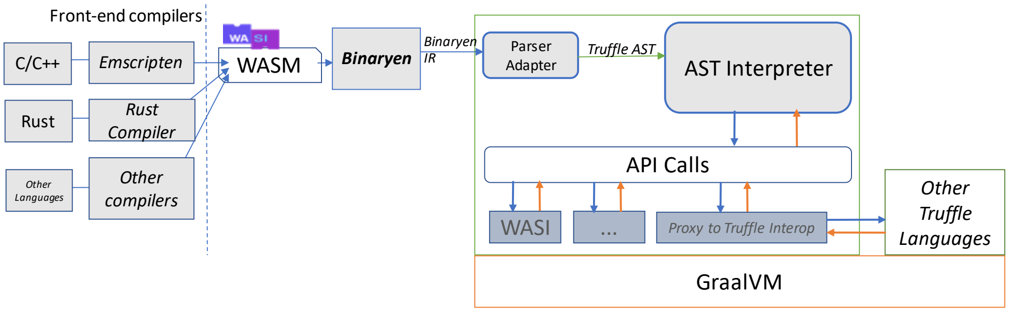


Figure 4. An overview of the TruffleWasm design.

```
usize_t fd_write(fd, iov, iovcnt, nwritten) {
    // checks ...
    // get wasi 32-bit version iovec
    wasi_iovec <- iov;
    // convert to native iovec
    iovec_c <- wasi_iovec
    // call POSIX writev
    written = writev(fd, iovec_c , iovcnt);
    if (written < 0)
        return errno; // return errorno
    // record written bytes
    *nwritten = written;
    //cleaning ...
    return 0; //ESUCCESS
}
```

Listing 5. A description of a POSIX API wrapper for WASI `fd_write`. This is called from Java through TruffleNFI and only the necessary values are passed.

various WebAssembly features that produce a runtime with competitive performance as discussed in Section 6.

4.1 WebAssembly System Interface (WASI) Implementation

To provide a standalone runtime for WebAssembly, the imported environment functions must be implemented by the runtime engine. For `wasi`, TruffleWasm provides implementations for the required functions. Typically, these functions interact heavily with linear memory (Section 4.3), and efficient access is necessary for high performance WASI execution. TruffleWasm uses C++ wrapper functions around the POSIX API to implement WASI functions. The wrappers perform all necessary checks and call an appropriate POSIX function if all requirements are satisfied. Note that all WASI APIs return an error number, and each wrapper function must check for, and return an appropriate error code. For instance, `fd_write` wrapper is implemented as outlined in Listing 5.

In `wasm32`, pointers are presented as 32-bit integers. But, when sending the `iovec` pointer to C/C++, `iov_base` uses `uintptr_t`, which on a 64-bit machine, will not map correctly. We provide an `i32 iovec` struct that is mapped to the native `iovec` before being passed to `writev`.

In an earlier version of the TruffleWasm implementation, support was present for other non standard APIs generated by compiler font-ends such as WasmExplorer and standalone Emscripten. Support was dropped for these front-ends because their APIs changed between versions and the APIs were not following any standardisation efforts. As a result, TruffleWasm now favours WASI functions and supports loading imported functions from a host embedder[10] such as a JavaScript runtime.

4.2 Instruction Interpreter and Control Flow

TruffleWasm uses Truffle Nodes to build an AST interpreter for nodes read from Binaryen. Each instruction is converted into a Truffle node, that can be further specialised at runtime, in order to provide an optimised method to execute its behaviour. Since WebAssembly is statically typed, and type information is already known during parsing, specialisation is mostly used to separate slow- and fast-path code for an operation and to potentially cache the constant and repeated appearance of specific input values to an instruction. This can separate out problematic corner cases that may require complicated control flow from fast path code. Generating code only for the specialised instances seen during profiling, and only for the code dispatch points as needed, can lead to reductions in the compiled code size, and improvement in the produced code quality.

Calls to functions, both direct and indirect (through a table lookup), are dispatched using Truffle's `InteropLibrary`. This allows exported functions to be called outside TruffleWasm by other Truffle/JVM languages.

In TruffleWasm, loop blocks are implemented using the class `RepeatingNode` of Truffle, that is designed to help profiling to identify *On-Stack Replacement* (OSR) optimisation opportunities. Nevertheless, loops in WebAssembly are controlled by break instructions that jump to the beginning of loop blocks (See Figure 3). When a break targets a loop block, the loop continues, otherwise, execution continues to the next instruction after the loop block (see Listing 3). This

[10] https://webassembly.github.io/spec/core/intro/overview.html#embedder

creates a challenge when profiling for counting loops. Other bytecode-based Truffle languages have observed challenges in modelling loop blocks in Truffle [13–15].

The deep nested nature of WebAssembly blocks (see Section 2.4) implies that using the `ControlFlowException` of Truffle from inner blocks to the outer-most block level would potentially require an expensive number of exception catches, and runtime re-throw(s). Each catch checks whether a block matches the jump target. Each throw happens when a match is not found at that level. Consider, for instance, `br n` which implies branching `n` blocks outwards. At each level, the parent block will catch the `ControlFlowException`, check whether it is the correct target, and if not, re-throw the exception again. Figure 5 shows the control flow for `br_if 1` (Listing 8, Line 36) when branching to an outer block. It first throws an exception which is caught by the enclosing block (block 0). The enclosing block then checks for the target, and re-throws the exception if it did not match its label. The appendix includes the C code snippet (Listing 7) and associated WebAssembly with nested blocks (Listing 8) for the control flow shown in Figure 5.

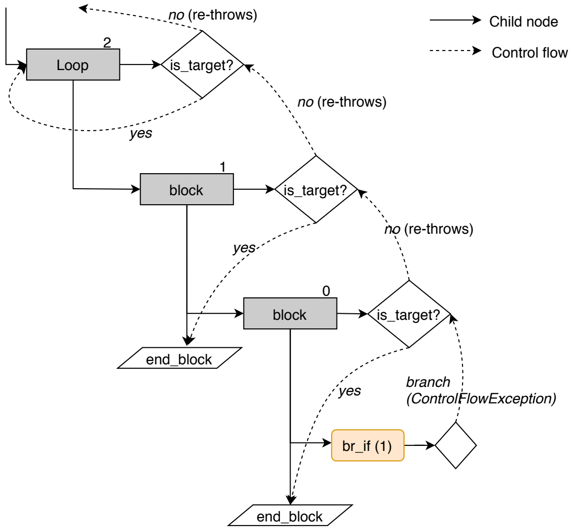


Figure 5. A control flow graph showing `br_if` (highlighted in yellow) branching out from a double nested block.

4.3 Linear Memory

WebAssembly linear memory is implemented using the Java `Unsafe` API. In TruffleWasm it is encapsulated as a `TruffleObject` to facilitate easy interoperability with other Truffle languages. When a module is instantiated, TruffleWasm allocates memory according to the specified page size. Instructions can then access memory through a specific interface by providing an offset location, this is converted to a native pointer and then bounds checked before a read or write occurs. Linear memory can be grown using the `memory.grow` instruction. Here, TruffleWasm reallocates the native memory with the new page size. Memory reallocation can be expensive, and some implementations chose to allocate a bigger than specified size at the beginning and then bounds check over the specified size, rather than the actual size of the allocated region [21]. In this way, some expensive memory re-allocations may be avoided at runtime.

In TruffleWasm, at each memory access site (`load`, `store`), `start_address`, `offset` and the `native_pointer` of the native memory are used to compute an `effective_address` that is then bounds checked. Using Truffle specialisation, the effective address can be cached for subsequent calls as long as the `start_address` and `offset` have remained invariant. This is controlled by a Truffle assumption which tracks that no memory growth has occurred. When a `memory.grow` is called, the assumption is invalidated and the access code will be deoptimised to default specialisation which re-computes the effective address. This is currently enough as TruffleWasm does not yet support the multi-threading proposal. When this is supported by TruffleWasm, approaches such as those discussed in [21] will need to be implemented.

4.4 Interoperability with Other GraalVM Languages

For other GraalVM-hosted languages (such as TruffleRuby and FastR), a WebAssembly interpreter can be initialised by accessing the WebAssembly GraalVM component. GraalVM hosted languages, such as Python, Ruby and JavaScript, can access WebAssembly modules/libraries compiled from languages such as C/C++ and Rust. These languages can also delegate other functionalities to the WASI API already supported in TruffleWasm. Listing 6 shows a Java program calling into WebAssembly.

For instance, in GraalJS, exported WebAssembly functions are called by accessing their definitions from the `export` property of a created WebAssembly instance (see Listing 4, line 23). When GraalJS instantiates a TruffleWasm module, the module’s imported functions are sent as `Proxy` objects to TruffleWasm. The `Proxy` objects can then be used to make callbacks to JavaScript functions. This ensures calls from TruffleWasm follow the ECMAScript specification (i.e. `[[Call]]`). In TruffleWasm, this proxy relation (see Figure 4) to GraalJS is implemented using `ProxyExecutable`[11] API provided by the Truffle framework. By sending a Proxy to TruffleWasm, a JavaScript function object remains in the JavaScript realm and can be modified without the need to refer changes back to TruffleWasm. This also reduces the need for TruffleWasm to maintain other JavaScript specific details related to the function object such as the `thisObject`, current context, and enclosure frame, needed to make a call in JavaScript. The proxy then gets the required arguments

[11] `ProxyExecutable` interface allows one Truffle guest language to mimic execution of another different guest language’s objects.

```
String w = "wasm";
source = Source.newBuilder(w, module).build();
ctx = Context.newBuilder(w).build();
ctx.eval(source);
export = ctx.getBindings(w).getMember(funcName);
result = export.execute(args);
```

Listing 6. Executing a WebAssembly module from Java using TruffleWasm.

from the WebAssembly current scope frame and proceeds to do the actual call in the JavaScript side.

In summary, the TruffleWasm interpreter reuses GraalVM through the Truffle framework, and it records profiling information that is used for partial evaluation, and to generate better optimised code. With this approach, most of the effort is being spent into building the interpreter, identifying fast- and slow-paths for operations and identifying profiling information that can be useful to the JIT for code generation.

5 Experimental Methodology

5.1 Experiments Goals and Benchmarks

We aim to investigate peak performance of TruffleWasm and other WebAssembly standalone runtimes running WASI modules. In the experiments, we compare a standalone execution of WebAssembly modules with Wasmtime 0.2[12] (commit 6def6de)[13] which has support for the WASI API.

We use the Shootout benchmarks [5], the C benchmarks from the JetStream 2.0[14] suite (hereafter referred to as c-JetStream), and PolyBenchC 4.2 [17]. We present the performance comparison and discuss issues in benchmarks where TruffleWasm is performing poorly. We also illustrate the performance gap between using Unsafe API and `ByteBuffer` (See Section 4) for implementing the WebAssembly linear memory.

5.2 Experimental Setup

To measure the peak performance of the JIT generated code, we execute each benchmark to run long enough for JIT compilation to be triggered for hot methods so that the runtimes reach a steady state of performance. Determining when a runtime reaches a steady state is difficult [1]. We apply a methodology presented in [15] and wrap the benchmark in a harness and record the last `30` iterations for evaluation. For TruffleWasm, we set the compilation threshold to 1000 using JVM's `-XX:CompileThreshold=1000` flag. We compile the original C programs to WebAssembly using clang 8.0 with `-O3` optimisation, and `--target=wasm32-wasi` and `--sysroot /wasi-libc` flags which sets target output to WebAssembly using WASI API and points to the WASI version of `libc` respectively. The Wasm+JavaScript modules are compiled using Emscripten 1.39.3.

PolyBenchC benchmark suite provides different compile-time options (through macros) to record execution time and other metrics. We added an option `-DPOLYBENCH_HARNESS`, which harnesses the main computation kernel of each benchmark and executes it for specified iterations and reports each iteration's execution time. We use the large data-set for all experiments (compiled with `-DEXTRALARGE_DATASET`) so that all "hot" functions are JIT compiled, and hence a steady state performance is reached when recording execution times for the last 30 iterations.

We run the experiments on a machine with Ubuntu 18.04.2 LTS, 16 GB memory, an Intel i7-6700 chip with Turbo boost and Hyper-Threading disabled; DVFS fixed to a frequency of 3.20GHz using a userspace governor. TruffleWasm runs on the GraalVM Enterprise Edition version 19.3.0.

6 Evaluation

This evaluation compares TruffleWasm and Wasmtime using peak performance execution times. Figure 6 shows TruffleWasm has a geo-mean slowdown of 4% compared to Wasmtime. TruffleWasm demonstrates comparable performance in many of the benchmarks except in *Quicksort*, *Float-mm* and *richards* showing the highest slowdowns of *82%, 68% and 33%*, respectively.

For *binarytrees*, TruffleWasm performed better than Wasmtime. *Binarytrees* is known for its memory-intensive nature. By default, when clang compiles the module to WebAssembly, it sets the initial linear memory page size to 2. This is then increased (by one page each time, as of clang 8.0) at runtime with `memory.grow` instruction every-time it gets full (until it reaches a maximum page limit). We instrumented `memory.grow` instruction to observe how it is called and by how many pages the linear memory is increased in each growth. We observed that WebAssembly runtimes incur a substantial amount of linear memory reallocation and resizing for memory intensive programs, typically adding one page at each time. For instance, *nbody* only grows its memory twice when run with 100 iterations in this experiment, while *binarytrees* program of depth 10 reallocates 52 times and that of depth 11 reallocates 102 times. For a *binarytree* of depth 17 (used in this evaluation), around 6.5K `memory.grow` operations are performed in just 100 iterations.

We also wanted to observe how the initial memory, if any, influences the overall execution time of each run. Figure 7 illustrates the results of changing the initial memory of the *binarytrees* program. For an initial 2 pages linear memory, the execution time of TruffleWasm for the first iteration is 4.7s and finishes with 6.7s in its last iteration. On the other hand, Wasmtime starts with 2.6s execution time and ends with 16.4s execution time in the 100th iteration. For larger

[12] We are using an older version of Wasmtime for comparison here, as we observed a performance regression (slowdown) in latter releases.

[13] https://github.com/bytecodealliance/wasmtime

[14] https://browserbench.org/JetStream/in-depth.html

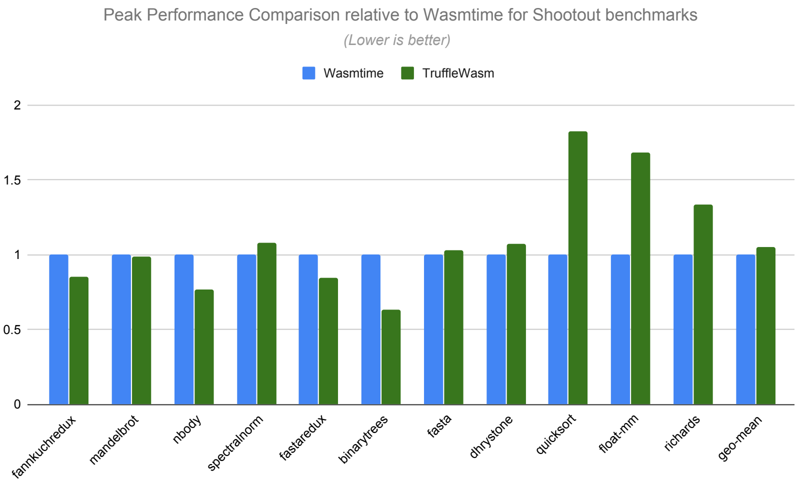


Figure 6. TruffleWasm peak performance comparison relative to Wasmtime executing the Shootout and C-JetStream benchmark suites. TruffleWasm executes with a geo-mean of 4% slower compared to Wasmtime.

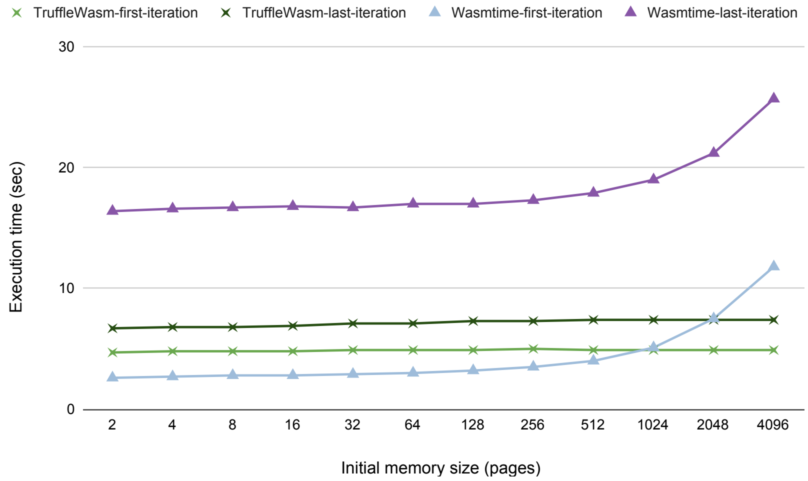


Figure 7. The benchmark *BinaryTrees* running with input depth of 17 showing the effect of changing the initial linear memory size from the 2 pages default to larger values (but less than what is required).

initial memory size, such as 4096 pages, first iteration execution time is 4.9 and 11.8 seconds for TruffleWasm and Wasmtime, respectively, and the 100th iteration execution time is 7.3 and 25.7 seconds for TruffleWasm and Wasmtime, respectively. The observation suggests that execution time of WebAssembly code with linear memory accesses increases with larger memory sizes and in some implementations, initial page size of the linear memory may influence execution times. This relationship is observable in both Wasmtime and TruffleWasm at varying degrees and it is also observed in other runtimes, such as V8 and WAVM [4].

In the initial implementation of TruffleWasm where *ByteBuffer* is used for linear memory, the slowdown is larger. This is due to the fact that linear memory is accessed continually by different operations. In each memory read, multiple bytes are read at a time and converted to a specific type such as `i64` or `f64`. This led to multiple reads just to get an int or long from memory. The same applies for a write operation where a value is converted into an array of bytes and stored back into memory. For the Java Unsafe API, reading or writing to memory is done in a single operation. Table 1 shows geo-mean slowdowns relative to Wasmtime achieved by TruffleWasm when Java `Unsafe` vs `ByteBuffer` are used for linear memory for some of the benchmarks presented in Figure 6. Table 1 demonstrates a clear and significant performance advantage for using the Unsafe API in preference to a *ByteBuffer* for linear memory support.

Figure 8 presents the evaluation using *PolyBenchC*, a benchmark containing scientific numerical computations, used to evaluate WebAssembly execution performance in browsers

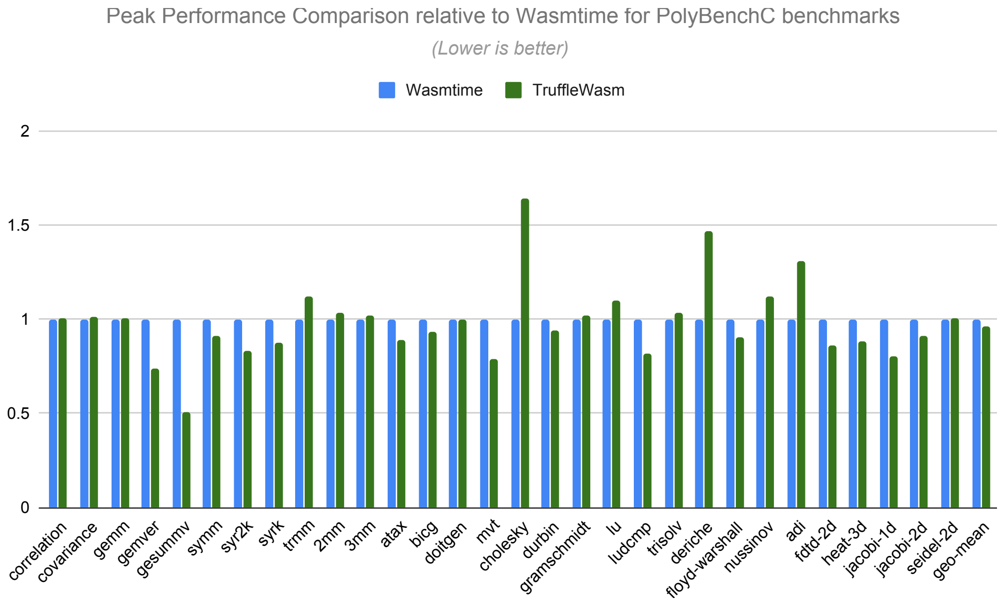


Figure 8. TruffleWasm peak performance for *PolyBenchC* benchmarks (lower is better).

Table 1. TruffleWasm normalised to Wasmtime when using Unsafe API and ByteBuffer for implementing linear memory. Values below 1 depict faster execution by TruffleWasm, while values above 1 depict faster execution by Wasmtime.

Benchmark	Unsafe API	ByteBuffer
fannkuchredux	0.85	5.58
nbody	0.76	5.76
spectralnorm	1.08	4.45
fastaredux	0.84	2.62
binarytrees	0.63	2.71
fasta	1.03	3.61
dhrystone	1.07	6.48
float-mm	1.68	4.02
richards	1.34	5.07

in [8]. TruffleWasm achieves near Wasmtime speed in most of the benchmarks, with the exceptions of *cholesky (65% slower)*, *deriche (47% slower)* and *adi (30% slower)*, and reaches a geomean of *0.96*; *4%* faster. Since *PolyBenchC* kernels measure computation and not system calls, TruffleWasm performed better and the harness reached a stable state after a short number of iterations, and these benchmarks tended to reach consistent peak performance execution times with very small deviations.

7 Related Work

Other WebAssembly standalone runtimes exist that target different deployment scenarios and platforms. Beyond browser engines, the following WebAssembly projects have influenced and are relevant to TruffleWasm:

Wasmtime is a Bytecode alliance[15] standalone runtime for WASI targeted modules. Wasmtime currently has full support for the WASI API, provides a C-API and is backed by two JIT compilers, Cranelift and Lightbeam, providing tiered code generation. Our TruffleWasm implementation closely follows Wasmtime for the WASI API, and provides similar configuration options in order that we can make the fairest possible comparisons between Wasmtime and TruffleWasm.

Wasmer is another WebAssembly standalone JIT runtime written in Rust. Wasmer provides different back-ends for generating JITed code including LLVM, Cranelift and Single-pass compilers. Wasmer also provides an API for languages such as Go and PHP, and provides support for WASI and Emscripten standalone modules.

GraalWasm[16] was recently announced as an open-source project aiming to support WebAssembly on GraalVM. However, GraalWasm does not support the WebAssembly System Interface, and has only been tested with micro-benchmarks. At the moment, GraalWasm cannot execute the benchmarks used in the evaluation of this paper.

Sulong is part of the GraalVM and executes LLVM IR, a compilation target for languages such as C, C++ and Swift, on JVMs [15]. By working with IR, Sulong manages to support multiple languages on the same implementation and it has been used to investigate techniques for the safe execution of native libraries on the JVM. In contrast, the research work in this paper investigates WebAssembly bytecode that has a similar abstraction level to LLVM IR. As such the same implementation approaches are followed with a focus on

[15] https://bytecodealliance.org/

[16] December 2019 – https://medium.com/graalvm/announcing-graalwasm-a-webassembly-engine-in-graalvm-25cd0400a7f2

supporting WebAssembly interoperability for its external imports.

To start understanding other WebAssembly runtimes, we have started experiments using the *PolyBenchC* benchmarks with Node.js v12.13.1 which uses V8. Currently, the geomean of TruffleWasm running WebAssembly with WASI is 55% slower relative to Node.js.

8 Conclusions

We have presented TruffleWasm, the first WebAssembly implementation on top of a JVM that can execute standalone WebAssembly modules and interoperate with JavaScript. TruffleWasm provides a platform for investigating WebAssembly core features, their performance, and how to provide interoperability with other Truffle-hosted languages.

The experimental results have compared the peak performance of TruffleWasm to the standalone Wasmtime runtime for the Shootout, C-JetStream and the PolyBenchC benchmarks. These results show that the geo-mean peak performance of TruffleWasm is competitive, and is only 4% slower than Wasmtime for the Shootout/C-JetStream, and 4% faster for PolyBenchC.

Considering the complexity of the implementation of TruffleWasm and Wasmtime is also an interesting question, although difficult to quantify. An indirect (and not perfect) but crude metric is the Lines of Code (LoC), TruffleWasm contains less than 50K LoC in Java, while Wasmtime contains more than 100K LoC including files in Rust, C++ and C. Wasmtime only has a JIT back-end for Intel/AMD processors, while TruffleWasm benefits from a wider range of back-ends available in the JVM ecosystem.

Future improvements will focus on adding support for new WebAssembly features such as multi-threading and SIMD that are now supported by some web browsers. We will also investigate performance improvement opportunities for peak performance, memory, and startup times; e.g. harnessing the SubstrateVM AoT [22].

Acknowledgments

This work is partially supported by the EU H2020 ACTiCLOUD 732366, and EPSRC Rain Hub EP/R026084/1 projects. Mikel Luján is funded by an Arm/RAEng Research Chair Award and a Royal Society Wolfson Fellowship.

References

[1] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133876

[2] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop.*

[3] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. 2018. Performance analysis for languages hosted on the truffle framework. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes.* ACM, 5.

[4] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting.

[5] Isaac Gouy. [n. d.]. The Computer Language Benchmarks Game. Retrieved 2020-02-18 from https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[6] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Mikel Luján. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Trans. Program. Lang. Syst.* 40, 2 (2018), 8:1–8:43. https://doi.org/10.1145/3201898

[7] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015).*

[8] 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).*

[9] Clemens Hammacher. 2019. Liftoff: a new baseline compiler for WebAssembly in V8. Retrieved 2019-06-07 from https://v8.dev/blog/liftoff

[10] Abhinav Jangda, Bobby Powers, Arjun Guha, and Emery Berger. 2019. Mind the gap: Analyzing the performance of webassembly vs. native code. *arXiv preprint arXiv:1901.09056* (2019).

[11] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04).* IEEE Computer Society, USA, 75.

[12] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).* ACM, New York, NY, USA, 821–839. https://doi.org/10.1145/2814270.2814275

[13] Raphael Mosaner, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. 2019. Supporting On-stack Replacement in Unstructured Languages by Loop Reconstruction and Extraction. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2019).*

[14] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '18).*

[15] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016).*

[16] Ed Schouten. 2015. CloudABI: safe, testable and maintainable software for UNIX. Retrieved 2020-01-04 from https://www.bsdcan.org/2015/schedule/attachments/330_2015-06-13%20CloudABI%20at%20BSDCan.pdf

[17] PolyBench: the polyhedral benchmark suite. [n. d.]. *Website.* Retrieved 2019-08-07 from http://web.cs.ucla.edu/~pouchet/software/polybench/

[18] Michael L Van De Vanter. 2015. Building debuggers and other tools: we can have it all. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems.* ACM, 2.

[19] W3C. 2018. WebAssembly Core Specification. Retrieved 2018-02-15 from https://www.w3.org/TR/wasm-core-1/

[20] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018).*

[21] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 133 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360559

[22] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. (2019).

[23] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14).*

[24] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).*

[25] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software.* ACM, 187–204.

[26] Oleg Šelajev. 2019. Lightweight cloud-native Java applications. Retrieved 2019-06-27 from https://medium.com/graalvm/lightweight-cloud-native-java-applications-35d56bc45673

A Appendix - C Snipet with Nested Blocks

```
#define sortelements 5000
long    seed = 74755L;
int sortlist[sortelements+1], biggest, littlest;
void Initarr() {
  int i; /* temp */
  long temp;
  biggest = 0; littlest = 0;
  for ( i = 1; i <= sortelements; i++ ) {
    temp = Rand();
    sortlist[i] = (int)(temp - (temp/100000L) * 100000L - 50000L);
   if ( sortlist[i] > biggest )
      biggest = sortlist[i];
   else if ( sortlist[i] < littlest )
      littlest = sortlist[i];
   }
}
```

Listing 7. Nested blocks example code snippet in C.

```
Initarr:            # @Initarr
    i32.const       0
    i32.const       0
    i32.store       littlest
    i32.const       0
    i32.const       0
    i32.store       biggest
    i32.const       4
    local.set       0
    loop                            # label0:
        local.get       0
        i32.const       sortlist
        i32.add
        i32.call        Rand@FUNCTION
        i32.const       100000
        i32.rem_s
        i32.const       -50000
        i32.add
        local.tee       1
        i32.store       0
        i32.const       biggest
        local.set       2
        block
            block
                local.get       1
                i32.const       0
                i32.load        biggest
                i32.gt_s
                br_if           0       # 0: down to label2
                i32.const       littlest
                local.set       2
                local.get       1
                i32.const       0
                i32.load        littlest
                i32.ge_s
                br_if           1   # 1: down to label1
            end_block                # label2:
            local.get       2
            local.get       1
            i32.store       0
        end_block                   # label1:
        local.get       0
        i32.const       4
        i32.add
        local.tee       0
        i32.const       20004
        i32.ne
        br_if           0    # 0: up to label0
    end_loop
end_function
```

Listing 8. WebAssembly code for the example in Listing 7, compiled with `clang 8.0` using *Compiler Explorer.*