Benchmarking Runtime Scripting Performance in WebAssembly

Devon Hockley
devon.hockley@ucalgary.ca
University of Calgary
Calgary, Alberta, Canada

Carey Williamson
carey@cpsc.ucalgary.ca
University of Calgary
Calgary, Alberta, Canada

ABSTRACT

In this paper, we explore the use of WebAssembly (WASM) as a sandboxed environment for general-purpose runtime scripting. Our work differs from prior research focusing on browser-based performance or SPEC benchmarks. In particular, we use micro-benchmarks and a macro-benchmark (both written in Rust) to compare execution times between WASM and native mode. We first measure which elements of script execution have the largest performance impact, using simple micro-benchmarks. Then we consider a Web proxy caching simulator, with different cache replacement policies, as a macro-benchmark. Using this simulator, we demonstrate a 5-10x performance penalty for WASM compared to native execution.

KEYWORDS

WebAssembly, benchmarking, scripting, performance, caching

ACM Reference Format:
Devon Hockley and Carey Williamson. 2022. Benchmarking Runtime Scripting Performance in WebAssembly. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

WebAssembly [23] (WASM) is a sandboxed low-level virtual machine originally designed to be used alongside the JavaScript virtual machine in Web browsers. It is often a compilation target for languages like C, Rust [7] and Go [9], which can run the compiled modules safely in a sandbox on a client machine.

Despite the name WebAssembly, there are no aspects of the design that constrain its use solely to the Web context. In fact, there are multiple browser-independent implementations of WASM, including Wasmer, WasmTime, and the runtime embedded in Node.js. The WASM specification allows for implementations to provide methods to the runtime environment to facilitate interacting with the outside world, but defines very few itself. This means that even though it originated as a feature for browsers, WASM can be used in other environments.

One problem that hampers development of complex applications in WebAssembly is that WASM does not support functions with arguments other than integers and floats. Because of this limitation, it is difficult to implement a general scripting environment.

One possible solution involves memory sharing for arbitrary data types, similar to the use of shared-memory [4] or message-passing to communicate between processes. In this approach, each module first defines a static array of memory. The host environment writes data to that array, and then calls the module using a function that specifies the size and location of the memory to read (effectively a pointer). The module then reads that memory to create its own internal version of the struct (i.e., data structure).

The goal of this paper is to evaluate the impact of this approach on performance, especially in programs that involve many invocations of simple scripts. This will be done by creating a set of benchmarks to measure the performance impact of this memory sharing technique in WASM.

The WebAssembly standard is also designed for creating secure sandboxed environments. The level of security provided ultimately depends on the implementation of the host runtime. Prior work has studied the basic level of security in the browser [10]. The binary security of a WASM module has a variety of flaws, including vulnerabilities to several known exploits [12]. An even more serious flaw in the host security could allow a malicious module to break out of the sandbox, allowing arbitrary file writes from WASM scripts [12]. One key limitation is that the current WASM standard does not generate code with stack canaries, which are a method of leaving extra markers on the stack to detect malicious writes and prevent the execution of malicious code.

Our work focuses on the functionality and performance of WebAssembly, rather than its security aspects. Specifically, we devise a generalized solution for function calls from a host program into a WASM script, with arbitrary data types. We develop and evaluate our solution using the Wasmer runtime. One caveat is that we cannot make any assertions about the performance of our solution in a fully-secure WebAssembly environment.

Our benchmarking programs are implemented in Rust [7], which is a low-level systems programming language that prioritizes program correctness. It was originally released in 2012 by Mozilla Foundation as an alternative to C++ that sought to prevent entire classes of bugs (e.g., segmentation faults) at compile time. It has changed a lot since 2012, and reached v1.0 in 2015. At this point, the language tracks memory ownership and memory lifetimes using a system called the Borrow Checker, obviating the need for a garbage collector. The correctness of this system has not been formally proven, but the overall algorithm has been studied, for example in 2015 with a proxy of the language called Patina [22] and in 2021 with a calculus core [21]. This memory safety property makes it easy to write fast and lightweight modules for WebAssembly that prevent memory bugs from happening. Furthermore, the modules do not require bundling with a large runtime system, such

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

as with Blazor/C# [19]. These advantages make Rust and WASM an attractive environment for general-purpose runtime scripting, which we demonstrate and evaluate in our paper.

The rest of this paper is organized as follows. Section 2 summarizes prior related work on WASM benchmarking. Section 3 describes the experimental methodology used for our benchmarking study. Section 4 presents results from the micro-benchmarks, while Section 5 focuses on the macro-benchmark results. Finally, Section 6 concludes the paper.

2 PREVIOUS WORK

2.1 Benchmarking Efforts

To the best of our knowledge, the most substantial academic effort for benchmarking WASM performance appeared at the 2019 USENIX conference [11]. The authors tested the SPEC benchmark suite in various browsers, using an environment simulating an OS kernel in the browser. They found up to a 2.5x performance penalty when running SPEC benchmarks (compiled to WebAssembly code) in the browser, depending on the workload and browser chosen.

The focus of our work is to test *non-browser* environments, using micro-benchmarks to identify performance bottlenecks. Relative performance comparisons could be made to the prior work [11], but the vast differences in technology between the two solutions means such comparisons might not be very useful. Additionally, our research focuses on measuring and comparing the base cost of calling a function hosted in WebAssembly, not on the overall execution time of the function that is called.

2.2 Runtimes

There are two standalone WebAssembly runtimes: WasmTime and Wasmer. WasmTime [2] is written by the Bytecode Alliance group, who are responsible for defining WebAssembly. Wasmer [26] is made by an independent group. Wasmer claims to be superior to WasmTime due to supporting multiple JIT backends, and being capable of faster runtime performance as a result.

Wasmer has three different options for its JIT backend: LLVM [13], Cranelift [1], and Singlepass. These all support different features [24] for the compiled code, such as enabling threads or multi-value function return. For this paper, we are interested in the speed of the compiled code. According to Wasmer [25], LLVM is the fastest, although it takes the longest to compile the bytecode. Singlepass is a simple compiler designed to compile code quickly, although the generated code isn't optimized. Cranelift is an independent compiler written in Rust, which falls between LLVM and Singlepass for both compile time and code optimization.

2.3 Lunatic

Lunatic [15] is a multi-language runtime built using WebAssembly. While it has somewhat similar goals to our work, it is designed as a language-agnostic alternative to the Erlang virtual machine [5], also known as BEAM. This means it prioritizes running many small processes in parallel and passing events between them, and prevents process crashes from affecting the entire program. Our work targets a lower-level benchmark. Furthermore, Lunatic introduces additional code that wraps the underlying Wasmer or WasmTime backend. In doing so, it introduces steps that are not features of WebAssembly itself, but instead features of Lunatic.

3 EXPERIMENTAL METHODOLOGY

The goal of this paper is to show that WebAssembly can be used as a scripting runtime to create useful applications. We do so by developing micro- and macro-benchmarks, and evaluating execution-time performance of WASM versus native mode.

Our benchmarking experiments involve five components: (1) the WebAssembly runtime itself; (2) a benchmarking program using that runtime module; (3) a set of basic WASM modules for testing that runtime with the benchmarking program; (4) a Web proxy caching simulator program built on that WebAssembly runtime as a macro-benchmark; and (5) a set of WASM modules to run in the simulator. All code is written in Rust, because it is the implementation language for the runtimes being used, and it has a mature WASM backend for its compiler. This also allowed us to reuse the exact same module implementations when comparing WASM performance to native execution.

3.1 Runtime Environment

All of our benchmarking experiments were done using Wasmer. The limited documentation available for WasmTime made it difficult to create an apples-to-apples comparison, so WasmTime was not used. Our Wasmer runtime module contains the methods used for copying arbitrary data into memory for a WASM module to use in any potential applications.

3.2 Micro-Benchmark Design

Table 1 provides a tabular summary of the experimental design for our micro-benchmarking experiments. We created a custom micro-benchmarking program to measure the performance impacts of individual techniques for optimizing WebAssembly. Overall, the benchmark tests three main factors: the optimization level of the compiler, the ABI (Application Binary Interface) of the module, and the caching of Wasmer references.

The optimization level determines how carefully the compiler optimizes the generated code. We tested the host code using both Debug and Release modes, which are defined in the Rust toolchain [6]. These modes change several things about the compilation, but the main variable is the opt-level, which is similar to C-style optimization levels. Debug mode is Level 0, with no optimizations. Release mode is Level 3, which applies all implemented optimizations.

For ABIs, there were three different levels tested: Pair, Bincode, and Bytemuck. The Pair ABI passes the two numerical arguments (integers or floats) directly to the multiplication function used in our micro-benchmark. Bincode uses a binary encoding to store the data, and then passes pointers to that stored data. Finally, we used the Bytemuck library to store a standard C struct encoding, which is similar to Bincode. All three of these approaches are listed under ABI in Table 1.

Reference Caching is used to cache the memory references in the compiled code returned by Wasmer. Caching was applied to all possible references. It was tested with either all applicable values cached, or none of them being cached. In addition, the Pair and Bytemuck ABIs had additional factors specific to them, in the form

of Preload versus Hotload for the Pair ABI, and static versus dynamic memory for Bytemuck. These are the columns Loading and Memory in Table 1, respectively. Pair also tested a different form of caching, in the form of Self-Referential Structs (SRS in Table 1).

All the factors, except for compiler optimization level, are tested within one run of the program. The program measures the time required to call the multiplication function 100,000 times, and then replicates this test 100 times to compute mean, standard deviation, and 95% confidence intervals on results. It performs this test for each combination of factors, printing and graphing the average times in seconds, as measured using Rust std::time::Instant [20]. This is a monotonically increasing timer [8] with sub-microsecond precision [20]. It calls the Win32 QueryPerformanceCounter on the Windows 11 machine used for the experiments.

Table 1: Experimental Factors for Benchmarking Tests

Test	ABI	Cached	Loading	SRS	Memory
1	Pair	No	Hotload		
2	Pair	No	Preload		
3	Pair	Yes	Hotload	No	
4	Pair	Yes	Preload	No	
5	Pair	Yes	Hotload	Yes	
6	Pair	Yes	Preload	Yes	
7	Bincode	No			Dynamic
8	Bincode	Yes			Dynamic
9	Bytemuck	No			Dynamic
10	Bytemuck	Yes			Dynamic
11	Bytemuck	No			Static
12	Bytemuck	Yes			Static

Only the Cranelift backend was tested due to technical limitations using the LLVM backend on Windows. In particular, LLVM requires a static binary that we were unable to create using the instructions provided by LLVM. Furthermore, Wasmer states that Singlepass is not designed for production use, and is meant only as a fast compilation option when developing quick code iterations.

3.2.1 Benchmark Modules. Figure 1 shows the software architecture of our micro-benchmarks. A set of WASM modules was developed to benchmark basic WASM operations, so that we can compare the impact of different operations. These modules all perform basic multiplication, using different ABIs for passing the parameters. These modules each have their own linear memory segment, which is used for both the heap and stack.

3.2.2 Application Binary Interface (ABI). In the benchmark, a series of options were tested for calling a very basic multiplication function that is in the WASM virtual machine. Three main ABIs were tested for these modules. Pair passes the arguments directly to the function in the virtual machine, while Bincode and Bytemuck

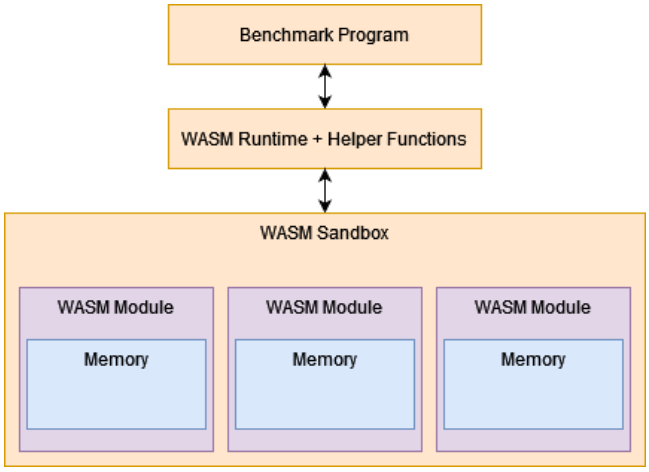


Figure 1: Software Architecture of Micro-Benchmarks

use shared memory to pass the variables. Bincode [17] uses an unspecified binary encoding to serialize and deserialize the memory. Bytemuck copies raw C-style structs to and from the module, while providing a thin wrapper for alignment checking [14]. Both are invoked by passing a pointer and length as direct arguments to the function, which are then used to fetch the relevant memory containing the parameters.

Bytemuck was tested with both static and dynamic memory allocation. Static allocation allows the buffer to be reused on subsequent calls, while dynamic allocation requires a new buffer from the module on every call. Due to the overhead of this extra call, dynamic allocation is about twice as slow as static memory allocation, which can cache and reuse the pointer.

3.2.3 Reference Caching. The final optimization tested was caching the WASM structs returned by the Wasmer library. These structs each represent what is essentially a function pointer. That is, they act as a reference to a function that we can invoke in a WASM module. In the micro-benchmark, we have two settings for Cached (Yes/No), and two more (Yes/No) for Self-Referential Struct (explained in the next subsection). Both of these cache the function reference when the module is first loaded, instead of each time it is called. Since this call involves at least one string comparison to find the name of the matching function, there is a small performance penalty, even if the Wasmer library caches those functions. The library does not specify how it stores and indexes the compiled WASM bytecode. A more in-depth analysis of the library's source code could allow for improvements, such as better caching if it's not already present. Since this paper approaches the problem from the perspective of a library user, this detail is out of scope.

3.2.4 Self-Referential Structs. The final version that was tested used self-referential structs, which are more representative of an object-oriented approach. Each loaded module is represented as a struct, with methods on the struct exposing the methods of the underlying WASM. This seemingly obscure factor has a noticeable impact on the performance of the solution because of how the

library works and how Rust manages its memory. Notably, Wasmer does not return a concrete `struct(Function)`, but rather a reference to a `struct (&Function)`. The actual Function struct is owned by the Instance struct, so its ownership cannot be transferred. In essence, our benchmark borrows the struct as a reference.

Borrow tracking is a core element of Rust's compile-time memory management, and is why it doesn't need a garbage collector. This creates a problem, though, because these references are like C++ references, and point to a location in memory. This means the reference is invalidated if the Instance is ever moved (i.e., pointing to where the struct used to be, not where it currently is). This means that creating the naively-designed struct in Figure 2 is impossible, since the module must be moved inside the struct to create the struct, thus invalidating the &Function and &Memory references.

```
pub struct WasmCachedBincodePolicyModule{
    module : Instance,
    mem: &Memory,
    alloc: &Function,
    send: &Function,
    init: &Function,
    stats: &Function,
}
```

Figure 2: Naive Self-Referencing Struct

Figure 3 shows a better solution, using *late initialization*. Specifically, we create the &Functions after the struct has been made. Since Rust does not allow us to just assign null arbitrarily, the struct must be changed to use `Option<T>`, which we can set to `None` initially, and then update later with `Some<T>`, so that the module isn't moved after the references are created.

```
pub struct WasmCachedBincodePolicyModule{
    module : Instance,
    mem: Option<&Memory>,
    alloc: Option<&Function>,
    send: Option<&Function>,
    init: Option<&Function>,
    stats: Option<&Function>,
}
```

Figure 3: Self-Referencing Struct with Late Initialization

This delayed initialization approach works fine until we try to move the struct, such as storing it in a list of modules for a benchmarking test run. At that point, the same problem arises again, because it attempts to move the struct into the new data structure, which invalidates the references again.

The next solution, illustrated in Figure 4, is to store the Instance on the heap, and pin the memory so that it cannot move. In this approach, `Box<T>` is a type that represents a heap allocation, and automatically de-allocates the memory when the Box leaves the

```
pub struct WasmCachedBincodePolicyModule{
    module : Pin<Box<Instance>>,
    mem: Option<&Memory>,
    alloc: Option<&Function>,
    send: Option<&Function>,
    init: Option<&Function>,
    stats: Option<&Function>,
}
```

Figure 4: Self-Referencing Struct with Pinned Instance

stack. Since a Box itself does not forbid movement, we also use `Pin<T>` to prevent the contained data from moving.

Combined, these two techniques allow a struct to contain references to itself, facilitating a general-purpose runtime scripting environment. Since the generated code is verbose, and highly sensitive to the order in which data is allocated and de-allocated for the struct, we used a Rust library called Ouroboros [16] to generate the boilerplate code (see Figure 5).

```
#[self_referencing]
pub struct WasmCachedBincodePolicyModule{
    module : Instance,
    #[borrows(module)]
    mem: &'this Memory,
    #[borrows(module)]
    alloc: &'this Function,
    #[borrows(module)]
    send: &'this Function,
    #[borrows(module)]
    init: &'this Function,
    #[borrows(module)]
    stats: &'this Function,
}
```

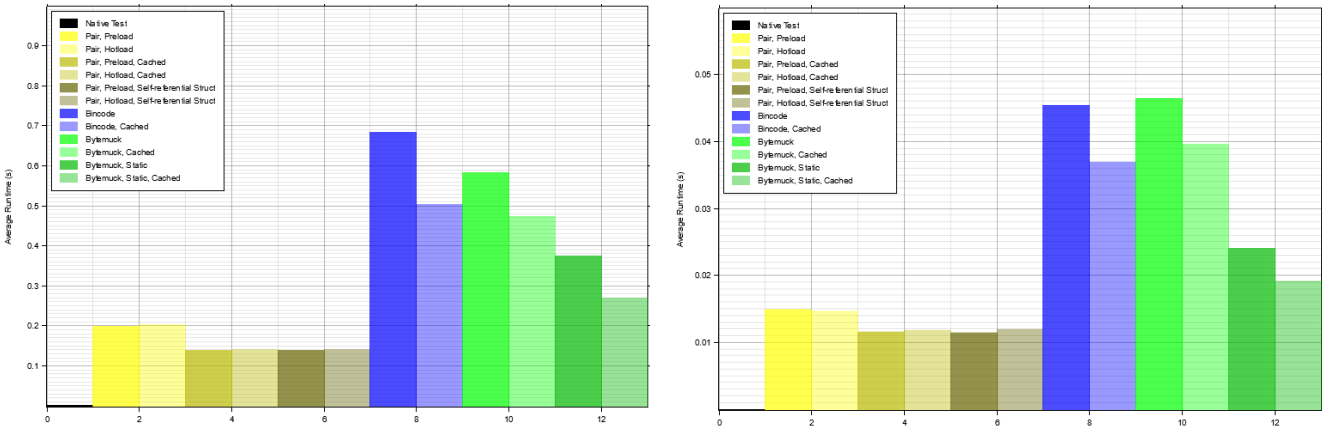
Figure 5: Self-Referencing Struct with Ouroboros

In this code, the #[borrows()] and #[self_referencing] are macros that instruct the library to generate code. The addition of *'this* to the various fields informs the compiler that the references must have the same lifetime as the struct, so that they don't become dangling pointers. The compiler verifies that they are destroyed at the same time; if not, the compiler will report an error.

4 MICRO-BENCHMARK RESULTS

We used our micro-benchmark program to determine which factors affected WASM performance. As would be expected, Debug mode in Figure 6(a) was a lot slower than Release mode in Figure 6(b). Specifically, Release mode improved execution time by about an order of magnitude (note the different vertical scales on the graphs).

The results for the rest of the micro-benchmark configurations are structurally similar in Figure 6(a) and Figure 6(b), though there are a few small differences. Most notably, the relative performance



(a) Debug Mode (b) Release Mode

Figure 6: Micro-benchmarking Results: Execution Time (in seconds) for Each Configuration

of Preload and Hotload changes. In Debug mode, there is a large performance difference between these two settings, but in Release mode they perform almost equally. This is most likely due to the compiler aggressively optimizing the Hotload version to match the Preload version.

All WebAssembly modules for the remaining benchmarking experiments were compiled using Release mode. This mode better represents modules provided by third parties. It also allows us to focus the benchmark on fine-grain interactions with the virtual machine, rather than measuring how fast the module code is.

4.1 ABI

The results show that Pair is the fastest ABI version. This makes sense since it avoids the overhead of passing the arguments as a struct. However, this method doesn't work for types that cannot be represented as an integer (i32, i64) or floating point (f32,f64). Types such as strings would have to be passed through shared memory using one of the other two methods. While it's possible to represent more complex types as raw integers passed this way, the resulting ABI would be difficult to implement for third parties, such as in the case of a plugin system, and would be incapable of passing data of unknown or arbitrary length.

The two other ABIs pass the data using the memory sharing technique. Among these approaches, the Bincode and the dynamic Bytemuck version were the slowest. This is because both versions require two calls to be made: one to allocate the memory, and the other to actually call the function. Notably, even though Bincode is a parsed binary encoding, it isn't significantly slower than Bytemuck, which just uses a raw C struct format. Since Bincode isn't currently formally defined, there is no way to know for certain how Bincode serializes the data. However, based on additional experiments (not reported here), we believe that Bincode's binary representation results in a C struct for the case of two 32-bit integers (i32). This observation provides some insight into why they perform similarly.

Our results show that there is negligible performance difference between Bincode and Bytemuck in the simple case. Bytemuck's performance improves when static memory is used, making it faster than Bincode. Note that static allocation is not possible with Bincode, since Bincode is unable to determine serialized size at compile time (a strict requirement for static allocation).

The static Bytemuck solution is faster than Bincode but slower than Pair. This Bytemuck library just casts a given C-style struct to an array of raw bytes, which can be copied and cast back, either using Bytemuck or any other mechanism to treat this set of bytes as a given C-style struct. Bytemuck only checks for memory alignment, so there is very little overhead other than copying the data into the shared memory.

Despite their performance differences, all three ABIs have their potential use cases. For simple data types, such as integers, Pair is the obvious solution, since it is the fastest, and the easiest to implement. For more complex data, the static Bytemuck option is faster, but the Bincode version could still have value when a C-struct is insufficient. However, a formal specification of Bincode format [18] is needed to facilitate its implementation in other languages. While Bincode itself currently only works with Rust, a similar protocol such as ProtoBuff could be leveraged to use this technique with multiple languages.

4.2 Memory Allocation

Using static memory allocation in Bytemuck substantially improved performance. Doing so reduces the number of calls into the VM from $2n$ to $n+1$, where n is the number of times we call the multiplication method. That is, it makes only a single call to get the memory pointer, rather than doing it every time. Since the WASM virtual machine defaults to a 32-bit machine, we can return this to the host program using only a single 64-bit integer, with the first 32 bits for the pointer, and the remaining 32 bits for the allocated size. This pointer is then reused for every subsequent call.

4.3 Reference Caching

Caching Wasmer references improved execution time by up to 15%. This was the case for both the simple caching, where the reference is held in a local variable, and the more complex self-referential struct option. This is good, because the self-referential version

better represents realistic programming requirements, such as the simulator scenario discussed next.

5 MACRO-BENCHMARK RESULTS

For our macro-benchmark, we developed a Web proxy caching simulator as an example of an application program (see Figure 7). We implemented the simulator in Rust, closely following the code of an existing simulator in C++ [3]. The simulator models the movement of different Web objects into and out of a Web proxy cache. Simulation parameters specify the size of the cache, as well as the replacement policy used to manage the cache space. The input to the simulator is a sequence of Web object requests, each expressed as a tuple of object ID and size, using two i32 values. The experiments used a synthetically generated request stream with 1 million requests generated to 30,000 objects. The macro-benchmark experiments generate a workload containing many small executions of different scripts, for different cache configurations, with the input trace passed using the memory sharing technique.

5.1 Web Caching Simulator

We implemented four different cache replacement policies in our simulator. These were FIFO (time-based), LRU (recency-based), LFU (frequency-based), and GD-SIZE (size-based) policies to produce different CPU and memory demands on the virtual machine. FIFO is relatively simple, just using a queue, while the other three use priority queues. LRU and LFU need their priorities updated dynamically, making them more demanding computationally. Each algorithm was created as its own Rust file, which was then imported into four separate Rust projects: one for the simulator itself for native execution, and three different WASM modules, with one for each of the ABIs in the benchmark. All modules for the macro-benchmark were compiled using Release mode.

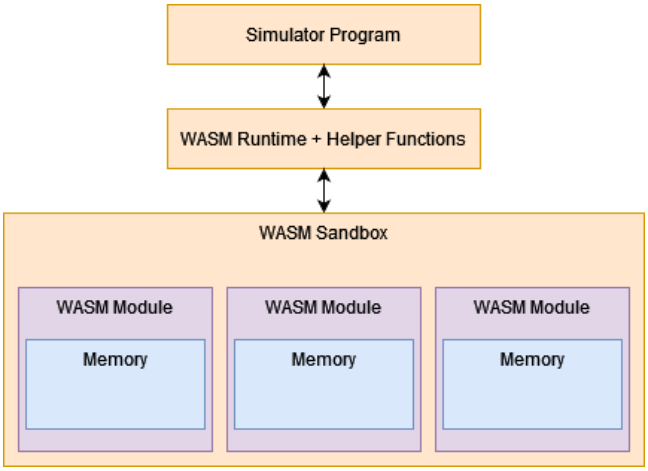


Figure 7: Software Architecture of Macro-Benchmark

5.2 Experimental Design

To reduce the number of WASM experiments, a few factors were omitted, based on the micro-benchmarking results. For example, static memory allocation performed strictly better in the micro-benchmarks, so dynamic allocation was not considered. Similarly, Preloading was always superior to Hotloading. Furthermore, we restricted these experiments to self-referential structs, since this is required for general-purpose scripting. This resulted in seven different scenarios for each algorithm. First, the native one serves as a baseline for comparison; it is embedded directly in the program and uses no WASM at all. Then each of the three ABI configurations gets two tests: one with reference caching, and one without.

5.3 Simulator Correctness

As a sanity check, we verified the simulator's correctness by comparing its object hit rate (and byte hit rate) results for different policies and cache sizes to those from a C++ version of the same simulator [3]. Each version of each policy matched the expected hit rate. The hit rates improve as the cache gets larger, until reaching a plateau as shown in Figure 8. The remaining experiments focus on execution-time performance of the simulator, rather than its application-level Web caching results.

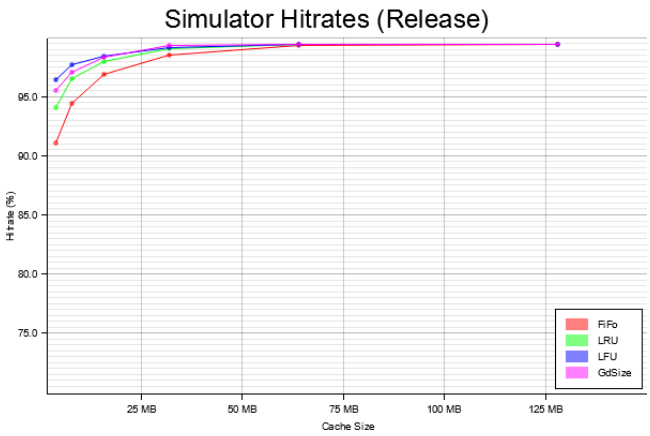


Figure 8: Simulator Hit Rates

5.4 Reference Caching Results

In the simulator, the performance increase from reference caching was not as pronounced as it was in the micro-benchmarks (see Figure 9 results for a 4 MB proxy cache size). This is likely because the cost of calling into WASM is a fixed overhead, and this overhead gets amortized when the called function does some meaningful computational work.

5.5 ABI Results

Once again, Pair was the fastest ABI, since it doesn't need to copy and read any memory. We also see that the Bytemuck solution is now faster than the Bincode one, instead of being the same. The reason is that Bytemuck is using static memory allocation, while Bincode has to rely on dynamic allocation. In these experiments, Bytemuck was only 20% slower than Pair. For each of these, caching Wasmer references improved performance, but not by as much as in the micro-benchmark. Again, this is because more computational

work is happening, instead of a basic multiplication. So in a more realistic problem such as this, reference caching is still worthwhile, but the performance gains aren't as pronounced.

5.6 Relative Performance

Our results indicate that there is a fixed cost to calling a WebAssembly function from outside the sandbox. This cost doesn't depend on how long the actual WASM code takes to execute. In Figure 9, for example, the Native version of FIFO is more than twice as fast as the Native version of LFU, but the WASM versions don't show nearly the same relative difference. This is further demonstrated by the fact that all six WebAssembly versions run the exact same source code on the exact same data, but have drastically different execution-time results. This fixed penalty for making a function call into WASM explains the 5-10x difference between the Native and WASM versions, as seen in Figure 9. This is far worse than the 2.5x difference reported in prior work [11].

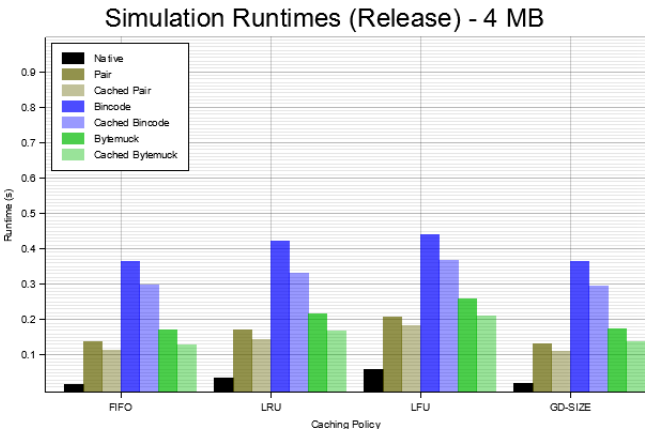


Figure 9: Execution-Time Performance of Simulator

6 CONCLUSION

In this paper, we have explored a memory sharing technique to allow general-purpose runtime scripting of WASM in non-browser environments. Our benchmarking experiments (micro and macro) show that WebAssembly suffers a large performance slowdown, which is attributable to the fixed cost overhead of making calls into the VM. To reduce this overhead, caching the Wasmer references helps in all cases, so it should be done whenever possible. Among the three ABIs tested, they all have potential use cases. Pair is the fastest, but only works for data that can be represented as a fixed-length tuple of integers or floats. Bytemuck with static memory allocation is the next fastest for passing arbitrary data, however this method might face challenges in the case of complex pointers or references within the data. Finally, Bincode is the slowest, but if it is possible to remove the extra call for dynamic memory allocation, it could be as fast as Bytemuck, while offering many more features.

There are several potential directions for future work. These include exploring the differences between Bincode and Bytemuck on more complex data types. Additionally, these methods could be tested and verified using other languages that support WebAssembly, such as C, C++ and C#. Another avenue is investigating the performance of these methods with variable-length data, and batching the data to reduce the number of calls in the sandbox. Finally, there is a forthcoming feature of WASM called WebAssembly Interface Types. Once available, its performance should be compared to these other ABI techniques, to see if a built-in solution in the runtime is better.

REFERENCES

[1] Bytecode Alliance. [n.d.]. Cranelift. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift
[2] Bytecode Alliance. [n.d.]. wasmtime: Standalone JIT-style runtime for WebAssembly, using Cranelift. https://github.com/bytecodealliance/wasmtime
[3] Mudashiru Busari and Carey Williamson. 2002. ProWGen: A Synthetic Workload Generation Tool for the Simulation Evaluation of Web Proxy Caches. *Computer Networks* 38, 6 (June 2002), 779–794.
[4] Xiao-hui Cheng and Liang Zhang. 2011. "A Research of inter-process communication based on shared memory and address-mapping". In *Proceedings of International Conference on Computer Science and Network Technology* (San Jose CA USA), Vol. 1. 111–114. https://doi.org/10.1109/ICCSNT.2011.6181920
[5] Erlang. [n.d.]. Erlang – Implementations and Ports of Erlang. https://erlang.org/faq/implementations.html
[6] Rust Foundation. [n.d.]. Rust Optimization Levels. https://doc.rust-lang.org/cargo/reference/profiles.html#default-profiles
[7] Rust Foundation. [n.d.]. Rust Programming Language. https://www.rust-lang.org/
[8] Rust Foundation. [n.d.]. Rust Time. https://doc.rust-lang.org/std/time/struct.Instant.html
[9] Google. [n.d.]. Go Programming Language. https://cloud.google.com/go
[10] 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. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. https://doi.org/10.1145/3140587.3062363
[11] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. (July 2019), 107–120. https://www.usenix.org/conference/atc19/presentation/jangda
[12] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann
[13] LLVM. [n.d.]. LLVM. https://llvm.org/
[14] Lokathor. [n.d.]. Bytemuck. https://docs.rs/bytemuck/latest/bytemuck/
[15] lunatic.solutions. [n.d.]. Lunatic, Erlang inspired WASM runtime. https://lunatic.solutions/
[16] Joshua Maros. [n.d.]. Ouroboros. https://github.com/joshua-maros/ouroboros
[17] Nathan McCarty. [n.d.]. Bincode. https://github.com/bincode-org/bincode
[18] Nathan McCarty. [n.d.]. Bincode Specification. https://github.com/bincode-org/bincode#specification
[19] Microsoft. [n.d.]. ASP.NET Core Blazor hosting models. https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0#blazor-webassembly
[20] Microsoft. [n.d.]. Query Performance Counter. https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter. https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter
[21] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 1 (April 2021), 1–73.
[22] Eric C. Reed. 2015. Patina : A Formalization of the Rust Programming Language.
[23] W3C and Bytecode Alliance. [n.d.]. WASM (WebAssembly). https://webassembly.org/
[24] Wasmer. [n.d.]. Wasmer Backend Features. https://docs.wasmer.io/ecosystem/wasmer/wasmer-features
[25] Wasmer. [n.d.]. Wasmer Backend Performance. https://medium.com/wasmer/a-webassembly-compiler-tale-9ef37aa3b537
[26] Wasmer. [n.d.]. Wasmer, The Universal WebAssembly Runtime. https://wasmer.io/