WebAssembly as a Runtime Scripting Environment

Devon Hockley

December 2021

Abstract

In this project, we explore the use of WebAssembly as a general-purpose sandboxed scripting environment. This was demonstrated using a Web Cache Replacement Policy Simulator, comparing the time it took to execute the simulation between WASM and native. We first focused on trying to measure which parts of executing a script hosted in this sandbox have the largest performance impact, using a simpler benchmark program. The Simulator was built using the data gathered from this simple benchmark. Using this Simulator, we demonstrated the performance penalties caused by using WebAssembly to host parts of a desktop program, when compared to the native implementation. This differs from prior work on the same topic, as most prior benchmarks focus on browser-based performance or on benchmarks such as SPEC [22].

1 Introduction

WebAssembly [23] (WASM) is a sandboxed low-level virtual machine originally designed to be used alongside the JavaScript virtual machine in various browsers. It is designed to be a compilation target for languages like C, Rust [6] and Go [8], that can run the compiled modules on a client machine safely in a sandbox.

There are multiple implementations that are independent of the browser, including Wasmer, WasmTime and the runtime embedded in Node.js. This is because despite the name WebAssembly, there are no aspects of the design bound to the web except for its sandboxed nature. 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 even though it originated as a feature for browsers, it can be used in other environments.

One of the common problems that hampers development of complex applications with WebAssembly is that it does not support functions with arguments other than integers and floats, so it is difficult to implement a rich scripting environment within it. One possible workaround is for each module to define a static array of memory, and for the host environment to write data to that array, then call the module using a function that takes in the location of the memory to read (effectively a pointer) and having the module read that memory to create its version of the struct. This is similar in concept to shared memory [3] and message passing being used to communicate between two or more processes.

The goal of this project is to evaluate the impact this workaround has 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 impact of this workaround, and finally creating a demonstration cache replacement policy simulator based on WebAssembly.

The WebAssembly standard is also designed for creating secure sandboxed environments, but this security is ultimately dependent on the implementation of the host runtime. There are no comprehensive papers focused on the runtimes we are using, but there is a basic level of security in the browser runtimes as described in a paper [9] published by employees of Apple, Microsoft, Google and Mozilla. The binary security of a WASM module has a variety of flaws, detailed in another paper [11]. These include several previously known exploits, such as one example given where the current standard does not contain stack canaries in the generated code, which are a method of leaving extra markers on the stack to detect malicious writes and prevent the execution of malicious code.

A security flaw more relevant to this project is a flaw in the host security, which would allow a malicious module to break out of the sandbox. In this [11] case, it allows for arbitrary file writes from WASM scripts, which will be a flaw present in our system. However, this project is not focused on the security aspect of WebAssembly and is instead focused on providing a reasonable workaround for the lack of functionality in making complex function calls from a host program into a WASM script. Due to the potential for large performance degradation caused by possible mitigations for these security flaws, we cannot make any assertions about the performance of our solution in a secure WebAssembly environment.

The demonstration programs were implemented in Rust [6], which is a low level systems programming language that prioritizes speed and program correctness. It was originally released in 2012 by Mozilla Foundation as a low level alternative to C++ that sought to prevent entire classes of bugs at compile time, such as segfaults. It has changed a lot since 2012, and reached version 1.0 in 2015. At this point, the language has evolved to no longer use a garbage collector of any kind, and instead tracks memory ownership and memory lifetimes with a system called the Borrow Checker. The correctness of this system does not seem to have been rigorously proved directly, but the overall algorithm has been reviewed a few times, for example in 2015 with a proxy of the language called Patina [21] and in 2021 with a calculus core [20]. This memory safety and lack of runtime makes it easier to write lightweight modules for WebAssembly by not requiring a large runtime to be bundled in the module such as with Blazor/C# [18] and still preventing memory bugs from happening, reducing the overall number of bugs in the system.

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 for the Web Caching Simulator. 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 WebAssembly performance is a report [10] from the University of Massachusetts testing the SPEC benchmark suite in various browser environments, using an environment simulating an OS kernel in the browser. They found up to a 2.5x performance penalty when running SPEC compiled to WebAssembly in the browser, depending on the workload and browser chosen.

The focus of this project is to test non-browser environments using a custom simpler benchmark to try and pinpoint where performance struggles. Broad relative performance could be compared between their project [10] and this one, but the vast differences in technology between the two solutions means such comparisons might not be very useful. Additionally, our project focuses on measuring and comparing the base cost of calling a function hosted in WebAssembly, not on the overall runtime of the function that is called.

2.2 Runtimes

There are two standalone WebAssembly runtimes: Wasmtime [2] written by the Bytecode Alliance group who is responsible for defining WebAssembly, and Wasmer [26] which 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 3 different options for its JIT backend: LLVM [12], Cranelift [1] and Singlepass. These all support different features [24] for the compiled code, such as enabling threads or multi-value return. For this paper, we are interested in the speed of the compiled code. According to Wasmer [25], LLVM should be the fastest in this regard, although takes the longest to compile the bytecode. Singlepass is a simple compiler made by Wasmer that is designed to compile code quickly, although the generated code isn't optimized. Finally, Cranelift is an independent compiler written in Rust, which should land in between Singlepass and LLVM for both compile speed and optimization.

2.3 Lunatic

Lunatic [14] is a multi-language runtime built using WebAssembly. While it has somewhat similar goals to what I am trying to achieve, it is designed as a language-agnostic alternative to the Erlang virtual machine [4], also known as BEAM. This means it prioritizes running many small processes in parallel and passing events between them, and prevents process crashes from taking down the entire program. My project is targeting a lower level benchmark, and Lunatic introduces more code that wraps the underlying Wasmer or Wasmtime backend, so it simply introduces steps that are not features of WebAssembly itself, but instead features of Lunatic.

3 Experimental Methodology

The goal of this project is to demonstrate that WebAssembly can be used as a scripting runtime to create useful applications. We used a cache replacement policy simulation as an example of a useful workload comprised of many small executions of different scripts. The experiment is comprised of five components: the WebAssembly runtime itself, a benchmarking program using that runtime module, a set of basic WASM modules for testing and benchmarking that runtime with the benchmarking program, a caching simulator program built on that WebAssembly runtime, and finally a set of WASM modules to run in that simulator. This was all written in Rust because this is the language used to implement the runtimes being used, and it has a mature WASM backend for its compiler. It also allowed us to reuse the module implementations to better compare native and WASM performance.

3.1 Runtime

Only Wasmer was tested due to lack of features and documentation in WasmTime. This lack of information prevented us from creating a fair comparison, so WasmTime was skipped. This runtime module contains the methods used for copying arbitrary data into a WASM modules memory, to reuse between any potential applications.

3.2 Benchmark

Using the runtime, a benchmark program was created to measure the performance impact of individual techniques for using WebAssembly using a series of micro benchmarks. Overall, the benchmark tests 3 main factors; the ABI (Application Binary Interface) of the module, the caching of Wasmer References, and the optimization level of the compiler.

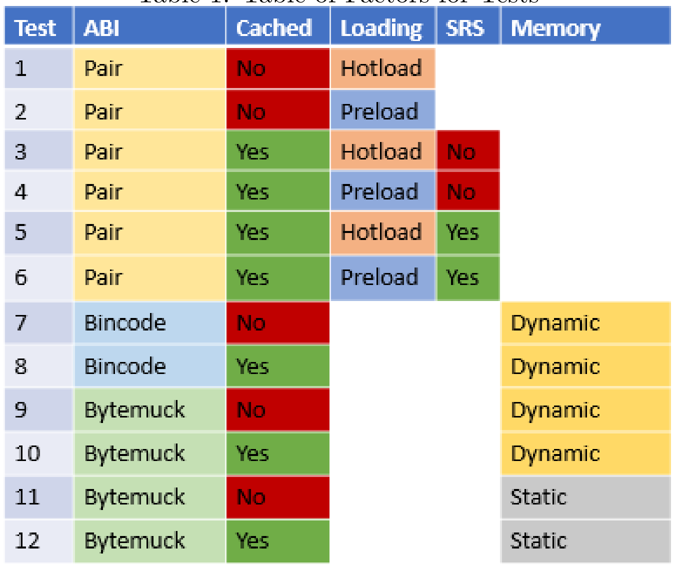
For ABIs, there were 3 different levels tested, one that passed arguments directly as a Pair, one that used a binary encoding called Bincode to store the data, then passing in pointers to that stored data, and finally one that used a similar mechanism to Bincode but used a standard C struct encoding using a library called Bytemuck. These are listed under ABI in Table 1.

Reference Caching caches the various references to compiled code that Wasmer returns and 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 vs Hotload for the Pair ABI, and static vs 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).

The optimization level tells the compiler what level of optimizations to apply. The host code was tested using both Debug and Release modes, which are defined by the Rust toolchain [5]. These modes change a few things about the compilation, but the main variable is the opt-level, which is very similar to C-style optimization levels. Debug uses 0, applying no optimizations, and Release uses 3, which applies all optimizations.

All the factors are tested within one run of the program, except for the optimization level. The program measures the time required to call the multiplication 100,000 times, and replicates this test 100 times, taking the average of the 100 runs. It performs this test for each combination of factors, printing and graphing the average times in seconds, as measured using Rust std::time::Instant [19]. This is a monotonically increasing time [7] that calls the Win32 QueryPerformanceCounter under the hood on the Windows 11 machine we ran the benchmark on, which claims sub-microsecond precision [19].

Table 1: Table of Factors for 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, as it requires a static LLVM binary I was unable to reproduce using the instructions provided by the LLVM project. Wasmer claims that Singlepass is not designed for deployment, and is meant as a compilation speed optimized development option for quick iterations.

3.2.1 Benchmark Modules

A set of WASM modules was developed, designed 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 in. These modules each have their own linear memory segment, which is used for both the heap and stack. These can be seen inside each module in Figure 1.

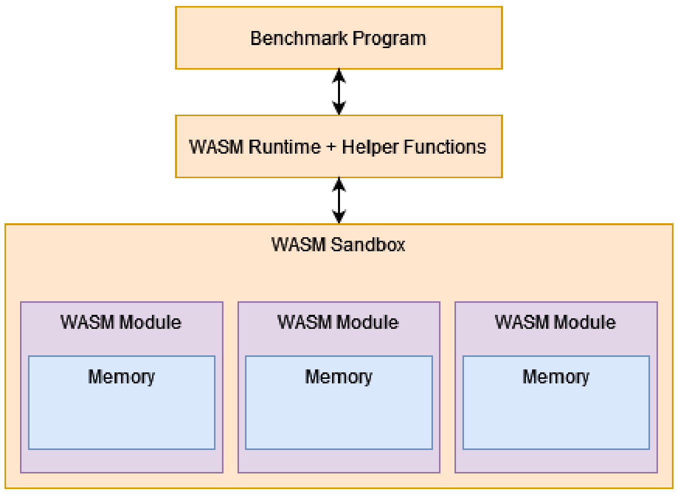


Figure 1: Software Architecture of Test Program

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, one that passes the arguments directly as a function of the virtual machine, referred to as Pair in the benchmark, and two that use shared memory to pass the variables across. Of these, one uses a binary encoding called Bincode [16] to serialize and deserialize the memory, and another copies raw C-style structs to and from the module, using a thin wrapper for alignment checking called Bytemuck [13]. Both are invoked by passing a pointer and length as direct arguments to the function, which are used to fetch the relevant memory with the parameters in it.

Finally, the Bytemuck version was tested using statically allocated memory allowing the buffer to be reused, as well as dynamically allocated memory, which needed us to request a new buffer from the module on every call. Due to the overhead of making a second call to the module for every actual call, this option was a bit over twice as slow as the static memory version that cached and reused the pointer.

3.2.3 Reference Caching

The final optimization tested was caching the WASM structs that the Wasmer library gave us.These structs each represent what is essentially a function pointer. They act as a reference to a function in a WASM module, that we can invoke. In the micro benchmark, we have two results that are Cached, and two more that use a Self-referential Struct. Both of these cache this function reference when the module is loaded, instead of each time we try to call it, like the other runs. Since this call involves at least one string comparison to find the matching function of that name, this carries a performance penalty, even if the Wasmer library caches those functions. Since the library does not specify how it stores and indexes the compiled WASM bytecode, it's hard to tell what causes this. A more in depth analysis of the library's source code could allow for improvements to the library, such as better caching if it's not already present. Since this paper approaches the problem from the perspective of a library consumer, this is out of scope.

3.2.4 Self-Referential Structs

The final version that was tested was using self-referential structs, which is more representative of an object-oriented approach. This is where each loaded module is represented as a struct, with methods on the struct exposing the methods of the underlying WASM. The reason this is a notable factor, and impact the performance of the solution, is because of how the library works and how Rust manages its memory. First, Wasmer does not return a concrete struct(Function), but a reference to a struct (&Function). This is because the actual Function struct is owned by the Instance struct, so its ownership cannot be transferred. We are borrowing the struct as a reference.

This 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, which makes the pointer invalid by pointing to where the struct used to be, not where it currently is. This means that attempting to construct the struct in Figure 2 is impossible, because the module must be moved into the struct to create the struct, invalidating all the &Function and the &Memory.

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

Figure 2: Naive Self-Referencing Struct

This means we have to 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 can be either Some<T> or None, which we can set to None initially then update with Some<T> after the fact, so that module isn't moved after the references are created, as can be seen in the code in Figure 3.

```
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 works fine, until we try to move the struct, such as storing it in a list of modules to run a test on. Then, we have the same problem, because it attempts to move the struct into the new data structure, which would invalidate the references again. The solution is to store the Instance on the heap and pin that memory, so that it never moves in Figure 4.

```
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

Box<T> is a type that represents a heap allocation, and automatically de-allocates the memory when the Box leaves the stack. Since a Box doesn't guarantee it won't move, we also use a Pin<T> which forces the contained data to never move. Together, this lets us create a struct that contains references to itself. Since this code is very verbose, and contains some issues with the order that data is allocated and de-allocated based on the order of the struct, I used a Rust library called Ouroboros [15] that can generate this boilerplate for us.

```
#[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 is informing the compiler that the references must have the same lifetime as the struct, so that they don't turn into dangling pointers. The compiler will verify that they are destroyed at the same time, otherwise the compiler will fail with an error.

3.3 Web Caching Simulator

Based on the benchmark, a caching policy simulator program was created implementing four cache policies. We implemented FIFO, GDSIZE, LFU, and LRU to try and model different ways the virtual machine could be stressed. FIFO is relatively simple, just using a queue, while the other three use priority queues. LFU and LRU must have priorities updated, meaning these are the most complex computationally. Each algorithm was created as its own Rust file, which were each imported into four seperate Rust projects: The simulator itself to test the native performance, and three different WASM modules, one for each of the ABIs described in the benchmark.

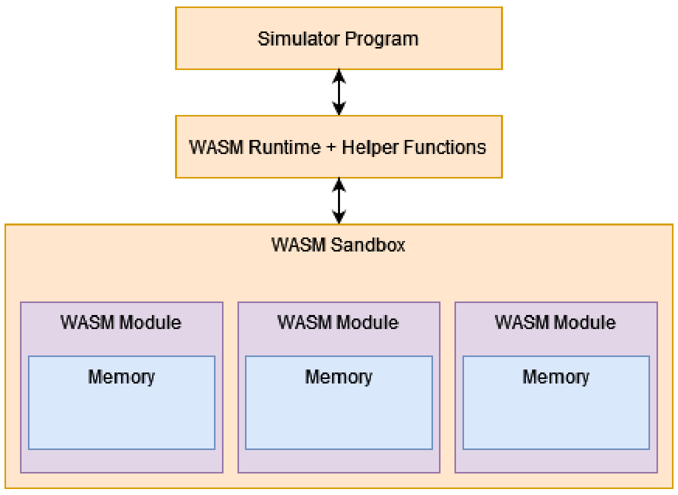


Figure 6: Software Architecture of Simulator

4 Micro-Benchmark Results

First, we gathered results from the benchmark program to determine which factors made meaningful differences to performance. As would be expected, the Release mode in Figure 8 is substantially faster than the Debug mode in Figure 7. It improved performance by slightly over an order of magnitude, which can be seen on the vertical axis of Figures 8 and 7. Interestingly, the relative performance varies, most notably the relative difference between Preload and Hotload. In debug, there is a large performance difference, but in Release 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 micro benchmark were compiled using Release mode, to represent optimal modules provided by third parties, to isolate the benchmark to how we interact with the virtual machine, not measuring how fast the module code itself is.

4.1 ABI

Among the results presented, the fastest ABI was the Pair version, because it removed the overhead of collecting and passing the arguments as a struct entirely. This is to be expected because it simply skips steps, but 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 methods. While it's possible more complex types could be broken down into raw integers and 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 arbitrary length data.

The other ABIs pass the data as shared memory. Out of these, the Bincode and dynamic Bytemuck version were the slowest. This is because both versions require 2 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 without a deep analysis of Bincode. However, a bug in my benchmark code bug resulted in the Bytemuck module being invoked using the dynamic Bincode benchmark unintentionally. Notably, this did not cause any error and worked perfectly fine, so it would seem that Bincode's binary representation results in a C struct for a pair of i32. Since this is not a guarantee of the library though, this result was disregarded and the bug fixed for final benchmarks, although it does provide some insight into why they perform similarly.

Based on the performance demonstrated, it would seem that there is no performance difference between these two levels in isolation. We do see a performance improvement with Bytemuck when the memory is static, which means that it is indirectly faster, as this option is not available to Bincode. This is cause Bincode is unable to determine serialized size at compile time, which is a requirement of the method.

This leaves the Static Bytemuck solution as the compromise, because it is faster than the 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 tell your code 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.

Overall, all three ABIs have their use cases. For basic data, such as this simulator when id's are integers, the Pair is the obvious solution because it is the fastest, and the easiest to implement. For more complex data, while the Static Bytemuck option is faster, the Bincode version could still have value when a C-struct is insufficient, or if Bincodes format is ever codified into a specification [17], because otherwise it cannot be easily implemented 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.

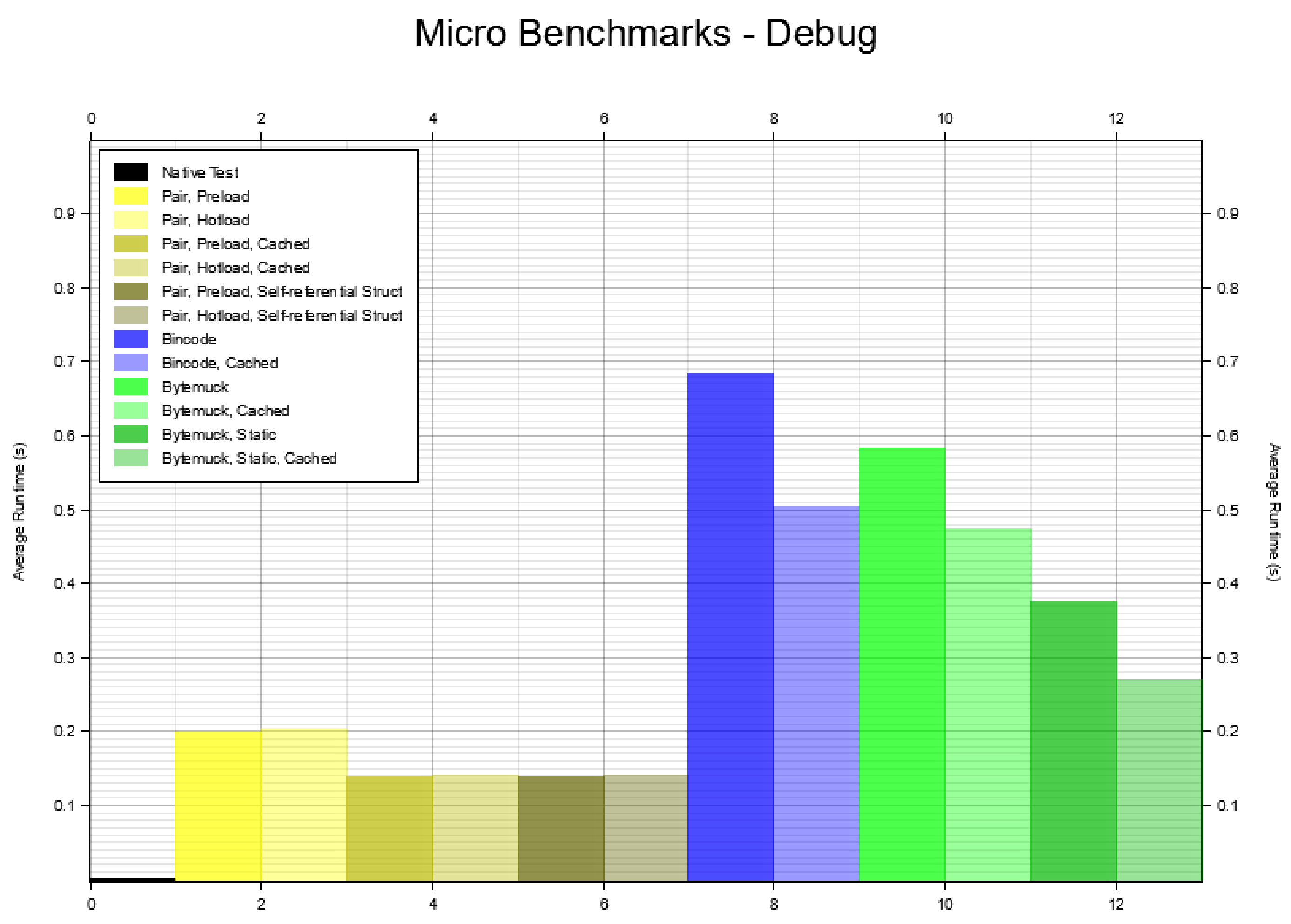


Figure 7: Debug Mode Performance of Benchmark

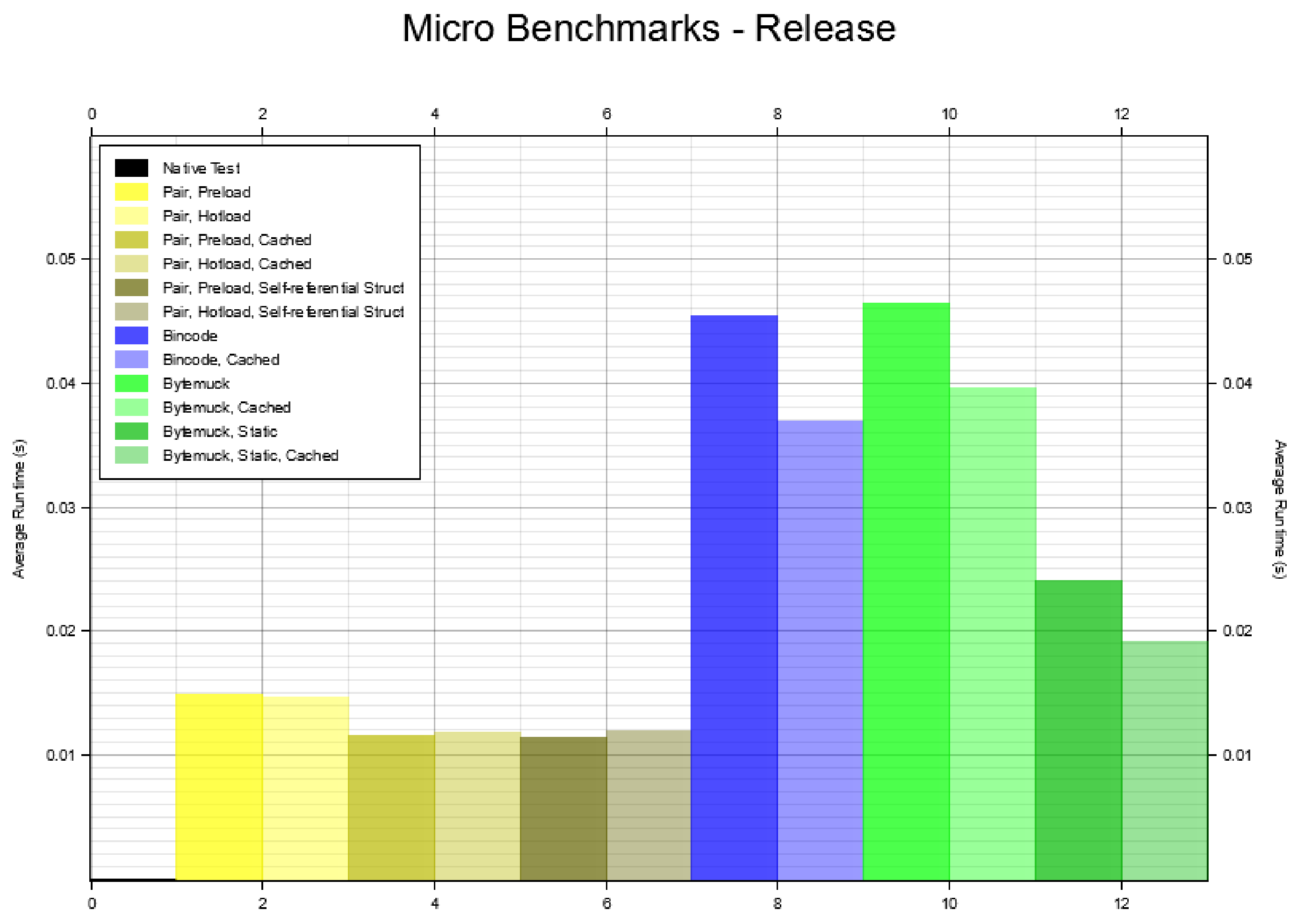


Figure 8: Release Mode Performance of Benchmark

4.2 Memory Allocation

Making the memory Static in Bytemuck immensely improved the performance, as it changed the benchmark from making $2*n$ calls into the VM to making $1+n$, where n is the number of times we call the multiplication method, by making only a single call to get the memory pointer. 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, using the first 32 bits for the pointer, and the later 32 bits for the length that was allocated. This pointer is then reused for every subsequent call.

4.3 Reference Caching

By caching these references, there was up to a 15.3% improvement in execution time in the benchmark 8. 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 while the local variable option works, the self-referential version better represents a real world scenario like the simulator.

5 Macro-Benchmark Results

All modules for the macro benchmark were also compiled using Release mode, to represent optimal policy modules provided by third parties. Results were captured using a trace file of 1 million requests.

5.1 Web Caching Simulator

A few factors were omitted, such as the non-self referential reference caching, because it was not feasible to implement, and the dynamic allocation version of Bytemuck, because the static one performed strictly better in the benchmark. This leaves seven different versions of each algorithm. First, the native one to act as a baseline for comparison, which is embedded directly in the program and uses no WASM at all. Then there are tests with each ABI, one with reference caching and the other without. The factors Loading method and Memory management were locked in as preloading and static memory respectively, because these versions performed strictly better in the micro benchmarks.

5.2 Simulator Correctness

The simulator program's correctness was verified by capturing the hit rates of the different algorithms at different sizes. Each version of each policy reported the exact same number of hits, and same number of requests. The hit rates get better as the cache gets larger, increasing until they plateau as expected in Figure 9.

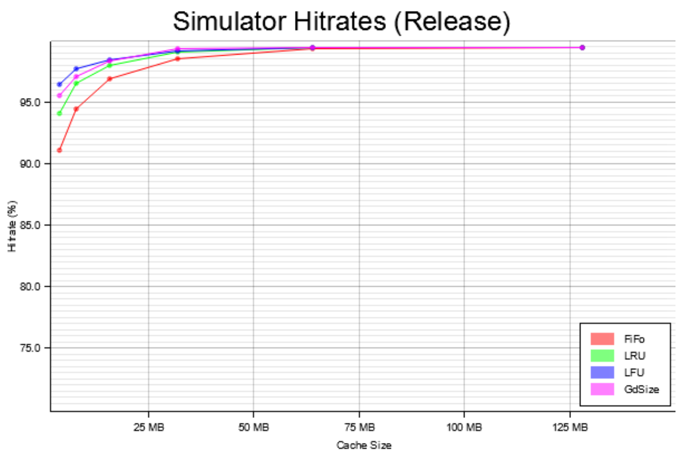


Figure 9: Simulator Hit Rates

5.3 Reference Caching Results

In the simulator, the performance increase from caching was not as pronounced across all the different policies in comparison to the Micro Benchmark, which can be seen for the 4MB cache size in Figure 10. This is likely because the cost of calling into WASM is fixed, and because we are doing real work now instead of a simple operation, this fixed cost looks smaller by comparison.

5.4 ABI Results

Overall, the Pair was the fastest ABI, by bypassing the need to copy and read any memory. When we examine the results from the simulator, we can see that the Bytemuck solution was faster than the Bincode one. Based on the micro benchmarks, this is caused by the change in memory allocation, not the change in ABI directly. The Bytemuck solution was only around 20% slower than the direct pair solution. For each, caching the Wasmer References resulted in performance gains, but less than the gains seen in the micro benchmark. This is likely because a lot more real work is happening, instead of a basic multiplication. So in a more realistic problem such as this, caching is still worth the effort, but the performance gains aren't as pronounced.

5.5 Relative Performance

There appears to be a fixed cost to calling into a WebAssembly function from outside the sandbox. This cost doesn't seem to scale with how long the actual WASM code takes to execute, because for example in Figure 10 the Native version of FIFO is over twice as fast as the Native version of LFU, but the WASM versions don't have nearly the same relative difference. This is further demonstrated by the fact that all 6 WebAssembly versions run the exact same source code, on the exact same data, but have drastically different results. This flat penalty for making a function call into WASM explains the five to ten times difference between the Native and WASM versions, as seen in Figure 10. This is opposed to the two to three times difference reported by the University of Massachusetts [10].

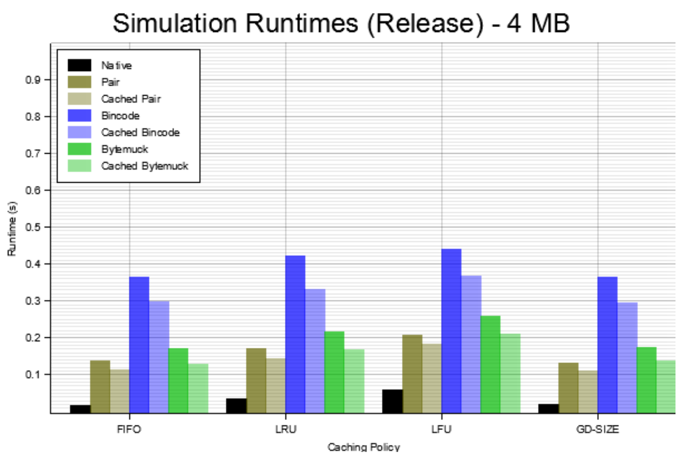


Figure 10: Sample Performance of Simulator

6 Conclusion

Overall, WebAssembly seems to suffer significant fixed cost slowdowns from calling into it. To minimize these costs, based on the results collected here, caching the Wasmer References is a performance improvement in all cases, so it should be done whenever possible. Between the three ABIs tested, they all have a potential use case. Pair is the fastest, but only works for data we can represent as a fixed length tuple of integers or floats. Next, Bytemuck with static memory is fastest for passing arbitrary data, however this method will struggle in the case of complex pointers or references within the data. Finally, Bincode is the slowest but if it is possible to remove the second call to do the dynamic allocation, could be as fast as Bytemuck, but with more features.

An unexpected result I encountered was the lack of difference in performance between the Bincode and Bytemuck versions in the Release version in Figure 10. Since Bincode is a parsed format and Bytemuck just checks alignment on a struct, I would expect the Bincode version to be slower. This is what happened in the Debug version, but it seems to have been optimized away in release mode.

It's possible this is caused by the underlying representation being the same for both in this circumstance, so the code to deserialize the Bincode version is the same as the code used to copy the struct using Bytemuck. I discovered this when the modules were mixed up by accident, invoking the Bytemuck module with Bincode host code, and it worked as expected. So because the memory layout is the same, a highly optimized Bincode version could result in similar code to the Bytemuck, just performing a simple copy.

Potential options for future work include exploring the unexpected results between Bincode and Bytemuck. It's possible this discrepancy could disappear if the data were more complex. Additionally, these methods could be tested and verified using other languages that support WebAssembly, such as C, C++ and C#. Another avenue to check would be investigating the performance of these methods with variable length data, and batching the data to reduce the number of calls in the sandbox. Finally, in the future, there will be a feature of WebAssembly called WebAssembly Interface Types. Once available, this should be compared to the performance of these techniques, to see if a solution built into the runtime is superior.

References

[1] Bytecode Alliance. Cranelift. `https://github.com/bytecodealliance/wasmtime/tree/main/cranelift`.

[2] Bytecode Alliance. wasmtime: Sandalone JIT-style runtime for WebAssembly, using Cranelift. `https://github.com/bytecodealliance/wasmtime`.

[3] X. Cheng and L. Zhang. "A Research of inter-process communication based on shared memory and address-mapping". In *Proceedings of International Conference on Computer Science and Network Technology*, volume 1, pages 111–114, July 2011.

[4] Erlang. Erlang – Implementations and Ports of Erlang. `https://erlang.org/faq/implementations.html`.

[5] Rust Foundation. Rust Optimization Levels. `https://doc.rust-lang.org/cargo/reference/profiles.html#default-profiles`.

[6] Rust Foundation. Rust programming language. `https://www.rust-lang.org/`.

[7] Rust Foundation. Rust Time. `https://doc.rust-lang.org/std/time/struct.Instant.html`.

[8] Google. Go programming language. `https://cloud.google.com/go`.

[9] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[10] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not so fast: Analyzing the performance of webassembly vs. native code. pages 107–120, July 2019.

[11] D. Lehmann, J. Kinder, and M. Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020.

[12] LLVM. Llvm. `https://llvm.org/`.

[13] Lokathor. Bytemuck. `https://docs.rs/bytemuck/latest/bytemuck/`.

[14] lunatic.solutions. Lunatic, erlang inspired wasm runtime. `https://lunatic.solutions/`.

[15] J. Maros. Ouroboros. `https://github.com/joshua-maros/ouroboros`.

[16] N. McCarty. Bincode. `https://github.com/bincode-org/bincode`.

[17] N. McCarty. Bincode specification. `https://github.com/bincode-org/bincode#specification`.

[18] Microsoft. ASP.NET Core Blazor hosting models. `https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0#blazor-webassembly`.

[19] Microsoft. Query Performance Counter. `https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter`.

[20] D. Pearce. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(1):1–73, April 2021.

[21] Eric C. Reed. Patina : A formalization of the rust programming language. 2015.

[22] SPEC. SPEC (Standard Performance Evaluation Corporation). `https://www.spec.org/benchmarks.html`.

[23] W3C and Bytecode Alliance. WASM (WebAssembly). `https://webassembly.org/`.

[24] Wasmer. Wasmer Backend Features. `https://docs.wasmer.io/ecosystem/wasmer/wasmer-features`.

[25] Wasmer. Wasmer Backend Performance. `https://medium.com/wasmer/a-webassembly-compiler-tale-9ef37aa3b537`.

[26] Wasmer. Wasmer, the universal webassembly runtime. `https://wasmer.io/`.