Benchmarking Runtime Scripting Performance in Wasmer

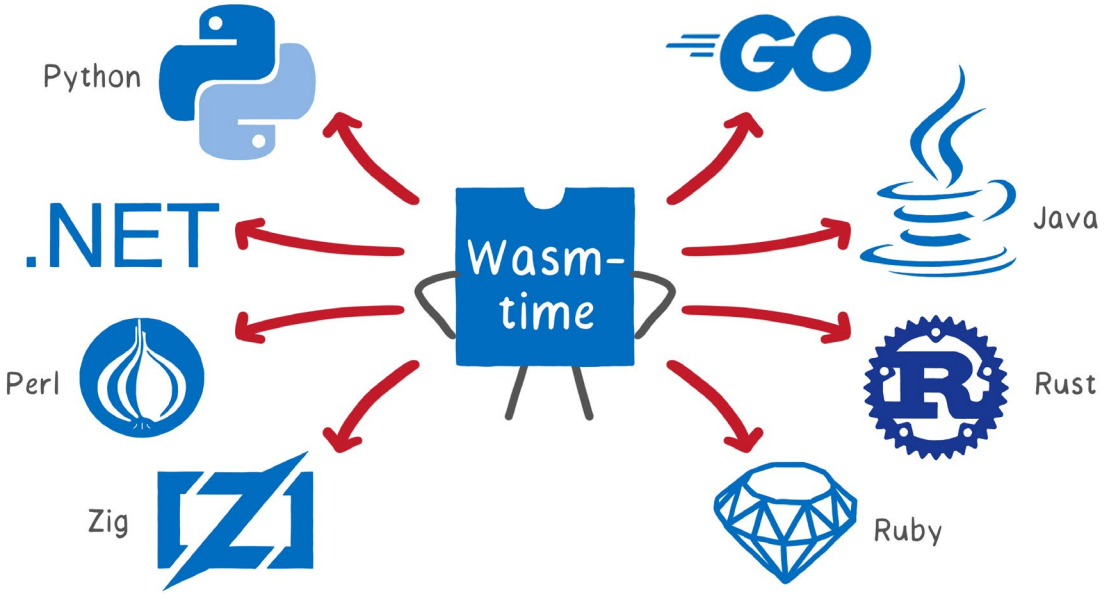
Devon Hockley (University of Calgary)

Carey Williamson (University of Calgary)

April 2022

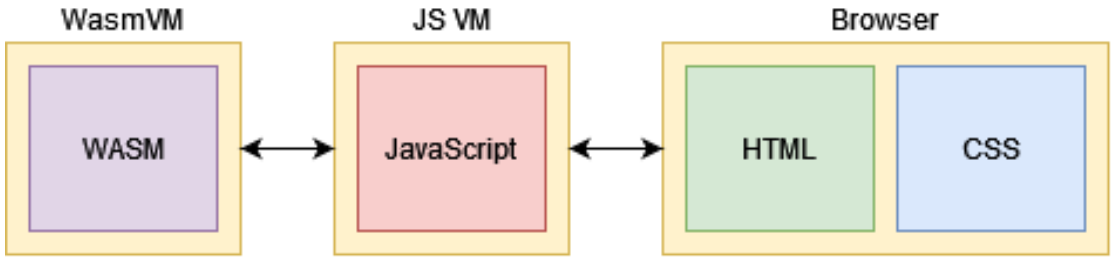
Overview

- Demonstrate that Wasmer[1] can be used to create an embedded scripting environment.
- Allow users to expand a program using any language of their choice, safely.
- A set of micro benchmarks to measure the performance of different techniques was created.
- A Web Caching Policy simulator was created as a realistic benchmark.



What Is Web Assembly?

- Created by W3C and the ByteCode Alliance
- Designed to provide a sandboxed, polyglot environment in web browsers for high performance code.
- Can interoperate with JavaScript to access DOM and perform browser operations.

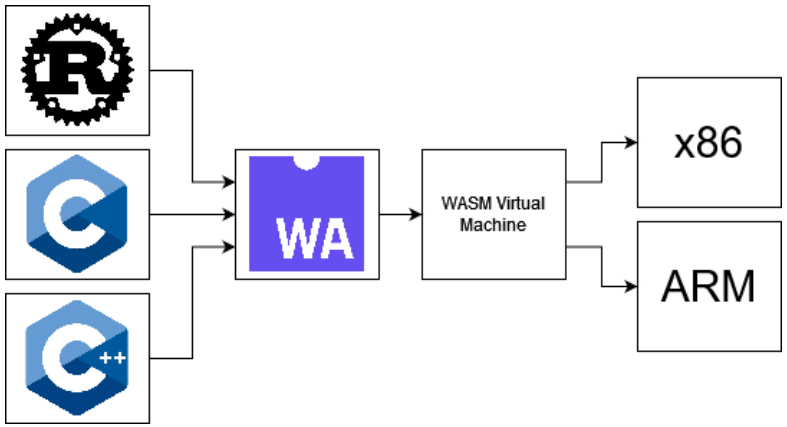


How Does WebAssembly Work?

- WASM Bytecode is a low level language based on stack machines.
- No garbage collection or classes.
- Modules that contain Functions.
- 4 primitive types: i32, i64, f32, f64
- Languages compile to an intermediary WASM bytecode.
- Bytecode is JIT-compiled to native by a WASM Virtual Machine.

```
(module
  (func $getAnswer (result i32)
    i32.const 42)
  (func (export "getAnswerPlus1") (result i32)
    call $getAnswer
    i32.const 1
    i32.add))
```

WebAssembly Text Format



Existing WASM Implementations

- Wasmtime[2]
  - Reference implementation created by BytecodeAlliance.
  - Cranelift compiler backend, created as an alternative to LLVM written in Rust that currently displays much faster compilation speeds, but slower final code.
  - Superseded older Lucet runtime by the same developers.
- Wasmer [3]
  - Independent Implementation created by Wasmer Inc.
  - Supports multiple JIT backends, including Cranelift and LLVM.

WASM outside the browser

- Lunatic [4]
  - Runtime built on top of Wasmtime or Wasmer.
  - Similar functionality to Erlang[5]/BEAM Virtual Machine.
  - Optimized for high concurrency, high reliability tasks.
  - Many small virtual processes that are isolated, so if one fails or crashes the rest of the program will keep going and that process can be restarted.
  - E.g. 1 virtual process per incoming HTTP request, each one mapping to a WASM call.
- Veloren [9]
  - Open source 3D game written in Rust.
  - Uses WASM to allow user written code to be added without recompiling the game.
  - WASM plugins are written using an event-driven architecture, with the host game dispatching events to handlers in the WASM plugin.

Previous Benchmarking

- Researchers from the University of Massachusetts Amherst found[6] up to a 2.5x performance penalty when comparing the SPEC benchmark between native and running through WebAssembly, but this was only tested using WASM implementations in the browser.
- This benchmark was also focused on a long-running program, not many short functions.
- To the best of our knowledge, there are no academic attempts at benchmarking WASM implementations outside of the browser.

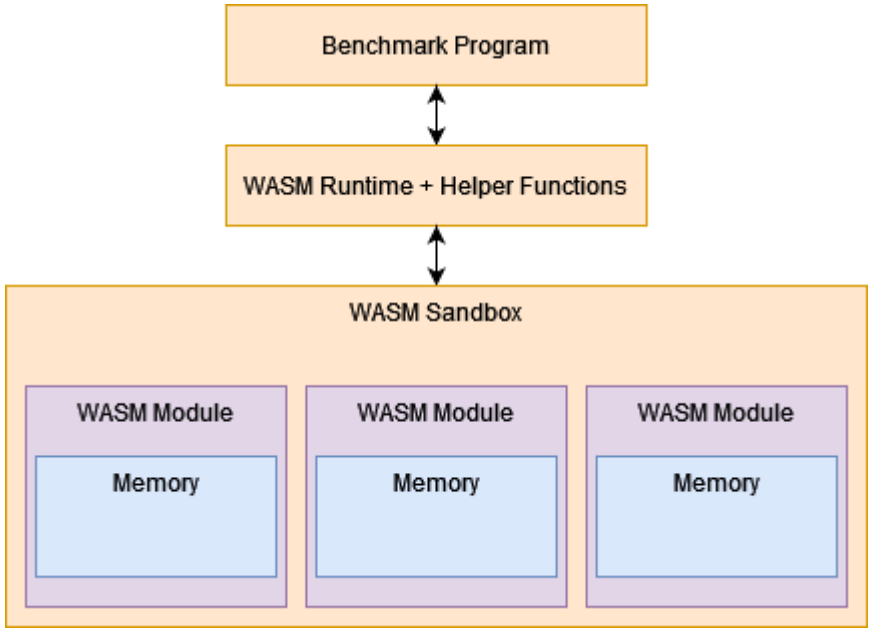
Our Benchmarks

- Previous benchmarks targeted long-running executions, evaluating the performance of a program that has been moved into WASM in its entirety.
- WASM also has applications that want to run many small functions, that are provided by the user.
- We want to explore the performance characteristics of making many small WASM function calls, instead of few long running ones, such as in event driven systems like Lunatic and Veloren.

Experimental Methodology

Micro Benchmarks

- Use Wasmer to create a Benchmark program in Rust.
- Benchmarks test various methods of calling into WebAssembly from the host program.
- Rusts correctness features, performance and low level memory control make it well suited to writing the benchmark program.
- Modules also written in Rust since it has a very mature WASM backend through LLVM.



Experimental Factors

- 3 Module Application Binary Interfaces (ABI) were created, Pair, Bytemuck and Bincode.
- For all 3 ABIs, the performance change caused by caching the references to the compiled code was measured.
- For Pair, an additional factor where we tested how the arguments were loaded was added.
- An additional caching type was tried for Pair called Self Referential Struct (SRS). This is a different style of caching, needed for object oriented style programming.
- For Bytemuck, both static and dynamic memory were tested. Bincode requires dynamic memory.
- Each of these combinations is tested in 1 run of the program, in sequence.

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

Experimental Factors

- These tests were done in both Debug and Release mode.
- These tests were also run against 3 different JIT backends for Wasmer:
  - Singlepass: Created by Wasmer, favors fast compile times over optimized output.
  - Cranelift: Alternative to LLVM written in Rust, limited optimizations available but compile code faster.
  - LLVM: Created by LLVM Project, full compiler backend with many optimizations.

Pair ABI

- Method is invoked directly, passing data as normal arguments.
- Extremely simple to implement, just requires one annotation to prevent symbol mangling.
- Limited to native types (i32,i64, f32,f64)

```
// Basic function call using WASM native types
#[no_mangle]
// Tells rustc not to mangle the symbol, required for host to find method
pub fn multiply(x : i32, y :i32 ) -> i32 {
    return x * y
}
```

Bincode ABI

- Copies data into the modules memory, then invokes the method, passing a pointer and length as arguments.
- Memory management and transmutation code are unsafe, and must be checked for soundness.
- Encoded using Bincode[7] binary encoding, similar to Protocol Buffers (ProtoBuff).
- Capable of using any type that can be represented as Bincode, not just primitives.

```rust
static mut BUFFERS : Vec<Box<[u8]>> = Vec::new();

#[no_mangle]
pub fn wasm_prepare_buffer(size: i32) -> i64 {
    let buffer :Box<[u8]> = Vec::<u8>::with_capacity( capacity: size as usize).into_boxed_slice();
    let ptr :i32 = buffer.as_ptr() as i32;
    unsafe{BUFFERS.push( value: buffer)};
    packed_i32::join_i32_to_i64( a: ptr, b: size )
}

fn internal_struct_mult(data : MultiplyParams) -> i32 { data.x * data.y }

#[no_mangle]
pub fn struct_mult(ptr: i32, buffer_size : i32) -> i32 {

    let slice :&[u8] = unsafe {
        // ptr as *const _ casts the i32 ptr to an actual pointer
        std::slice::from_raw_parts( data: ptr as *const _, len: buffer_size as usize)
        // from_raw_parts turns this data into a byte array we can use safely.
    };
    // Pop the buffer, but keep the memory allocated by assigning it to a variable.
    let _buffer_would_be_dropped  = unsafe {BUFFERS.pop()};

    // deserialize the slice.
    let x :MultiplyParams = bincode::deserialize( bytes: slice).expect( msg: "Deserialization error");

    internal_struct_mult( data: x)
}
```

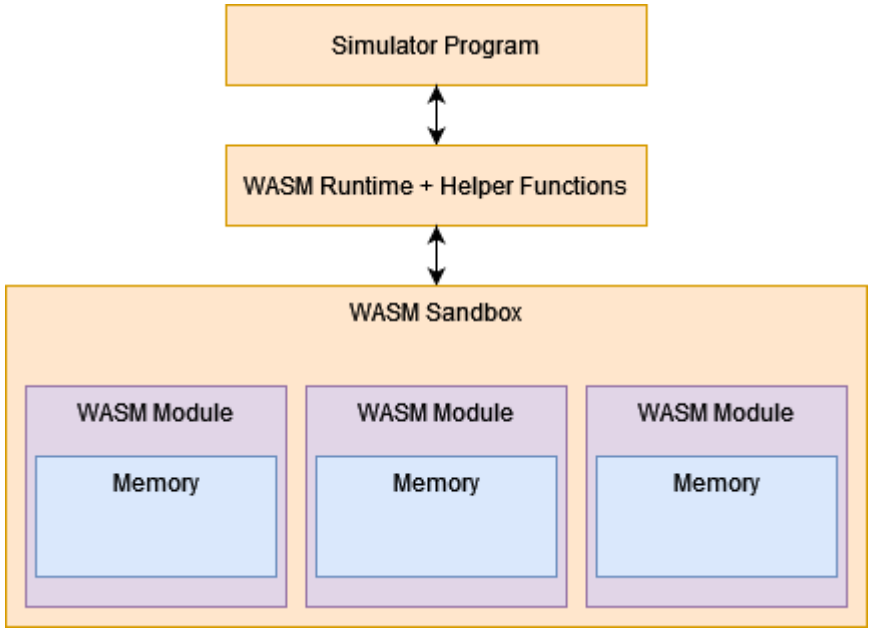
Bytemuck ABI

- Same process as Bincode, copying data into modules memory, then invoking method with pointer.
- Instead passes data as C-style struct, using Bytemuck[8] to check memory alignment.
- Static implementation slightly simpler, but still uses unsafe code.
- Can use any type that can be represented as a C-struct.

```
static mut BUFFER : [u8; std::mem::size_of::<MultiplyParams>()] = [0; std::mem::size_of::<MultiplyParams>()];

#[no_mangle]
pub fn wasm_get_buffer() -> i64 {
    let ptr :i32 = unsafe {BUFFER.as_ptr() as i32};
    packed_i32::join_i32_to_i64( a: ptr, b: std::mem::size_of::<MultiplyParams>() as i32)
}

fn internal_struct_mult(data : &MultiplyParams) -> i32 { data.x * data.y }

#[no_mangle]
pub fn struct_mult(ptr: i32, buffer_size : i32) -> i32 {

    let slice :&[u8; ?] = unsafe {
        &BUFFER
    };

    let x :&MultiplyParams = bytemuck::from_bytes::<MultiplyParams>( s: slice);

    internal_struct_mult( data: x)
}
```

Macro Benchmark

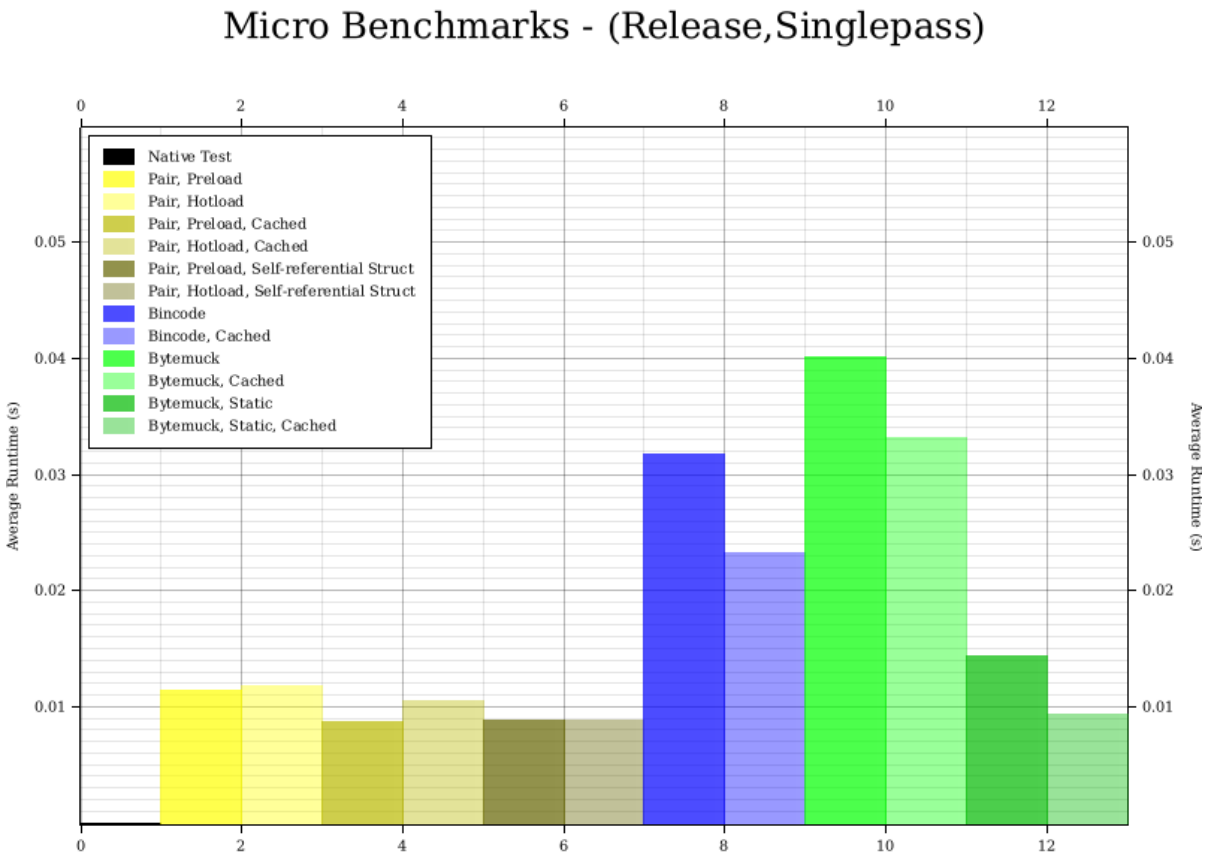
- Created a Web Caching Policy Simulator using similar architecture to the Micro Benchmark program.
- Certain factors were skipped, such as dynamic memory in the case of Bytemuck, due to performing strictly worse.
- 4 policies were tested: FIFO, LRU, LFU, and GD-SIZE



Experimental Results

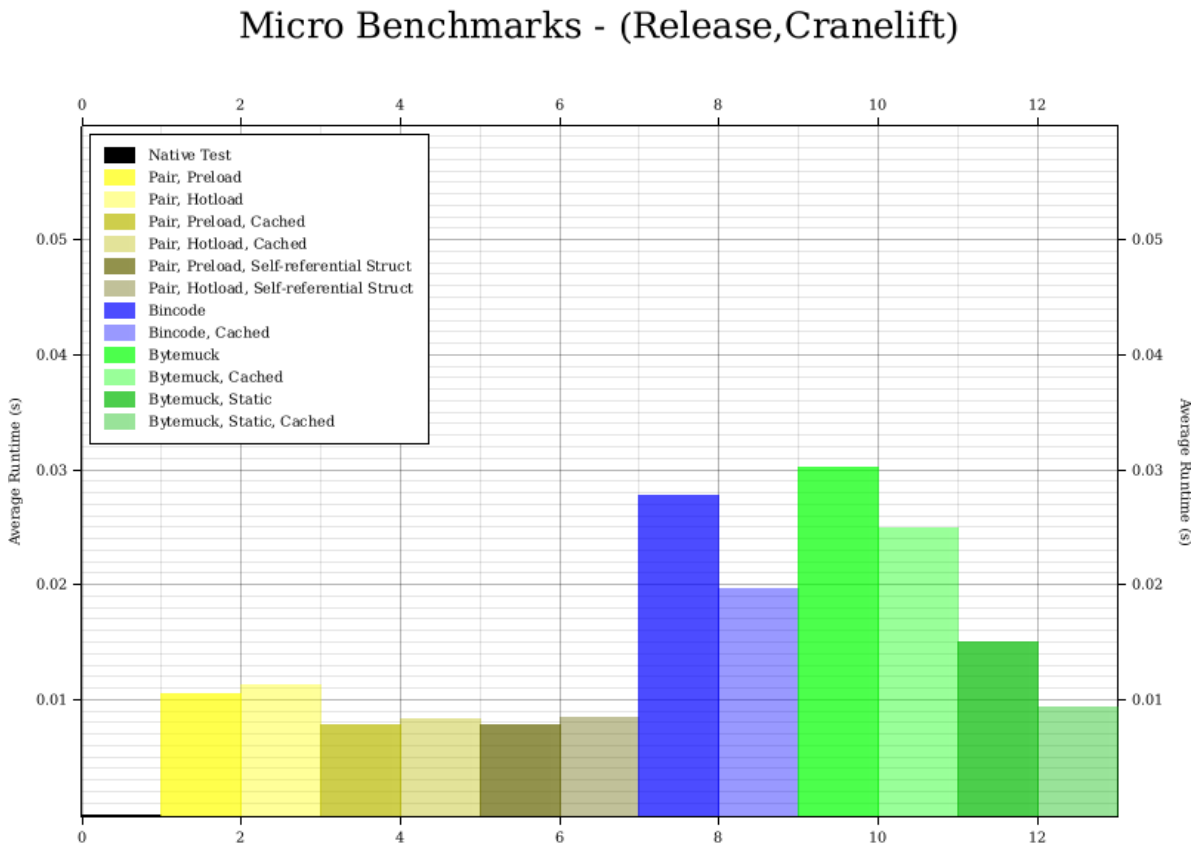
Micro Benchmark - Singlepass

- The cached variant performed better than the non-cached one for all implementations.
- Bytemuck and Bincode performed similarly, but Bytemuck has the additional optimization of static memory available for it.
- Pair performed the best across the board, although it has limited use cases due to having to pass everything as arguments. For example, no variable length data.



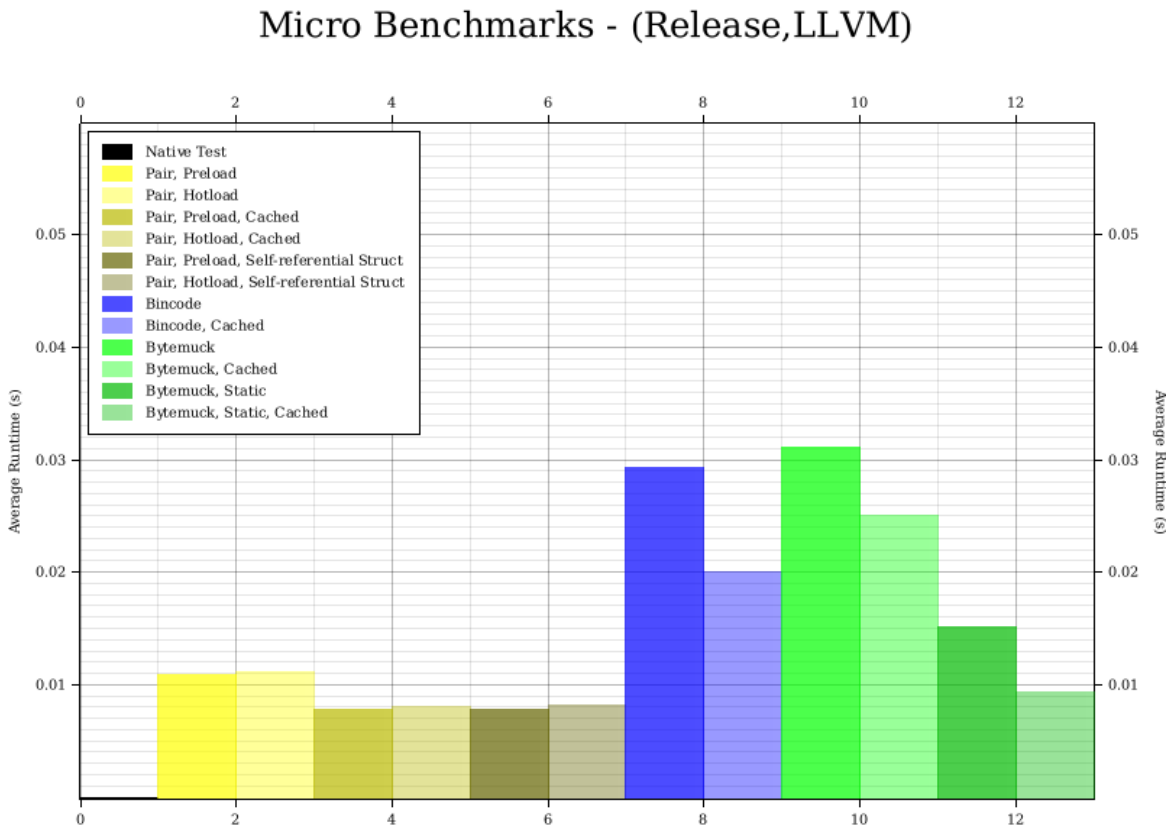
Micro Benchmark - Cranelift

- Cranelift improves performance somewhat.
- Relative performance was similar, except for dynamic Bytemuck code catching up a bit.
- Code compilation was only slightly slower.



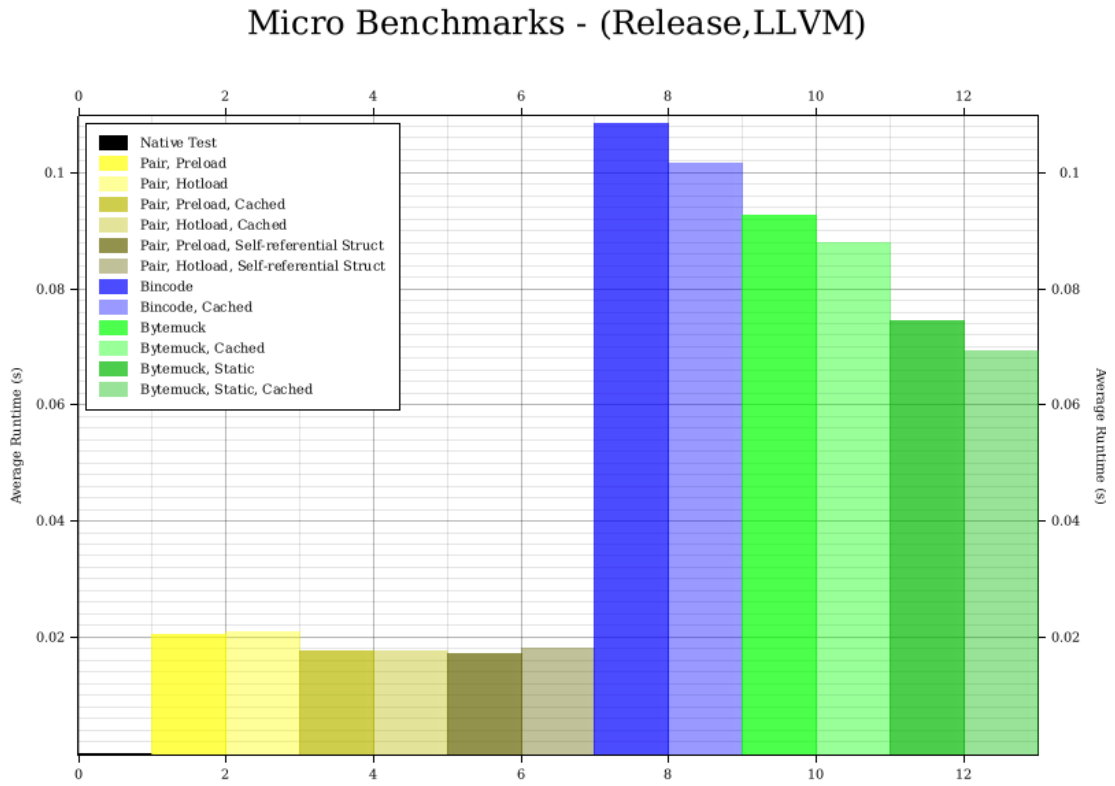
Micro Benchmark - LLVM

- LLVM backend did not improve runtime performance.
- Compilation speed was reduced drastically though.



Micro Benchmark - Compilation

- When JIT compilation times are included, Cranelift and Singlepass only slow by around 50%.
- LLVM times almost quadruple in the worst case.
- Likely because the code being compiled is very simple, so LLVM runs through many optimization runs that make no changes.



Compilation times included

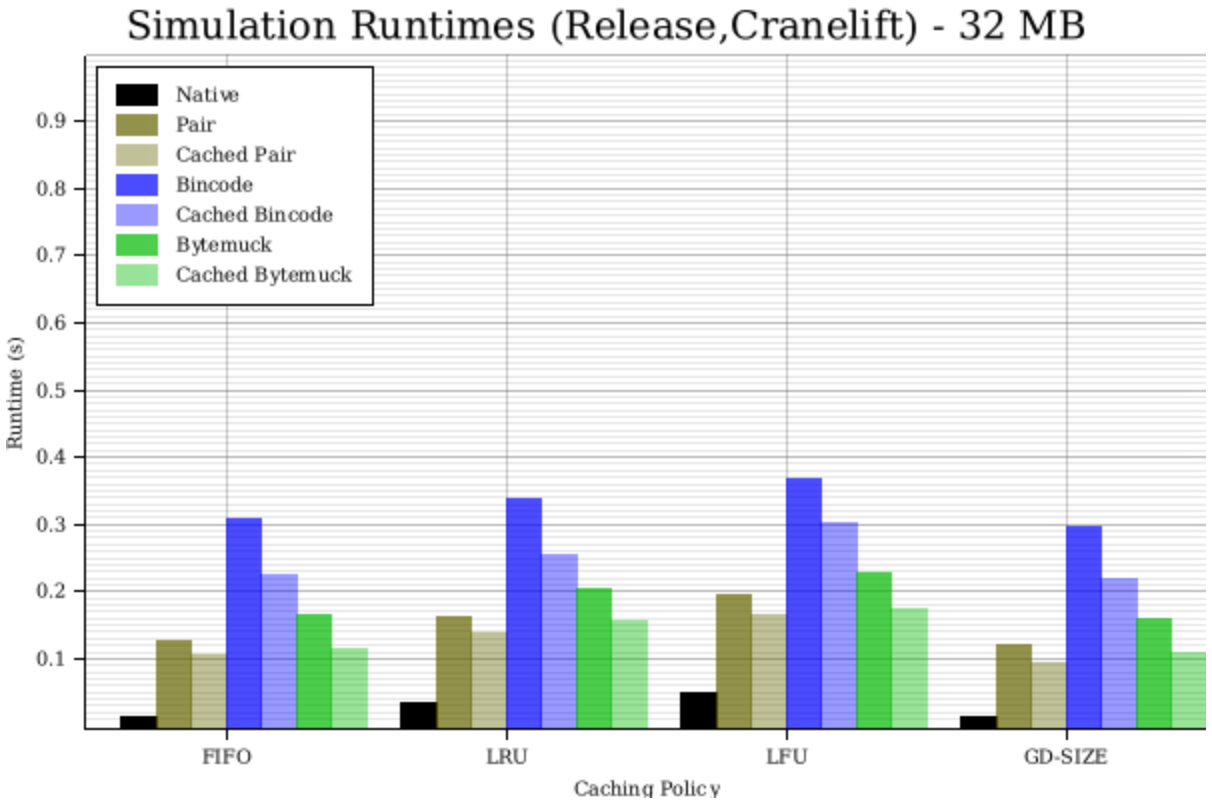
Macro Benchmark

- Certain Factors were dropped for the Macro Benchmark:
  - Hotload vs Preload – No difference in Release mode, likely optimized away.
  - Dynamic Bytemuck – Static performed strictly better.
  - Singlepass backend was dropped, as it performed strictly worse and is not intended for deployment.

- Self Referential Structs performed similarly to normal caching, so they were used for all caching instances, to better represent a real object oriented program where each policy is modelled as a struct.

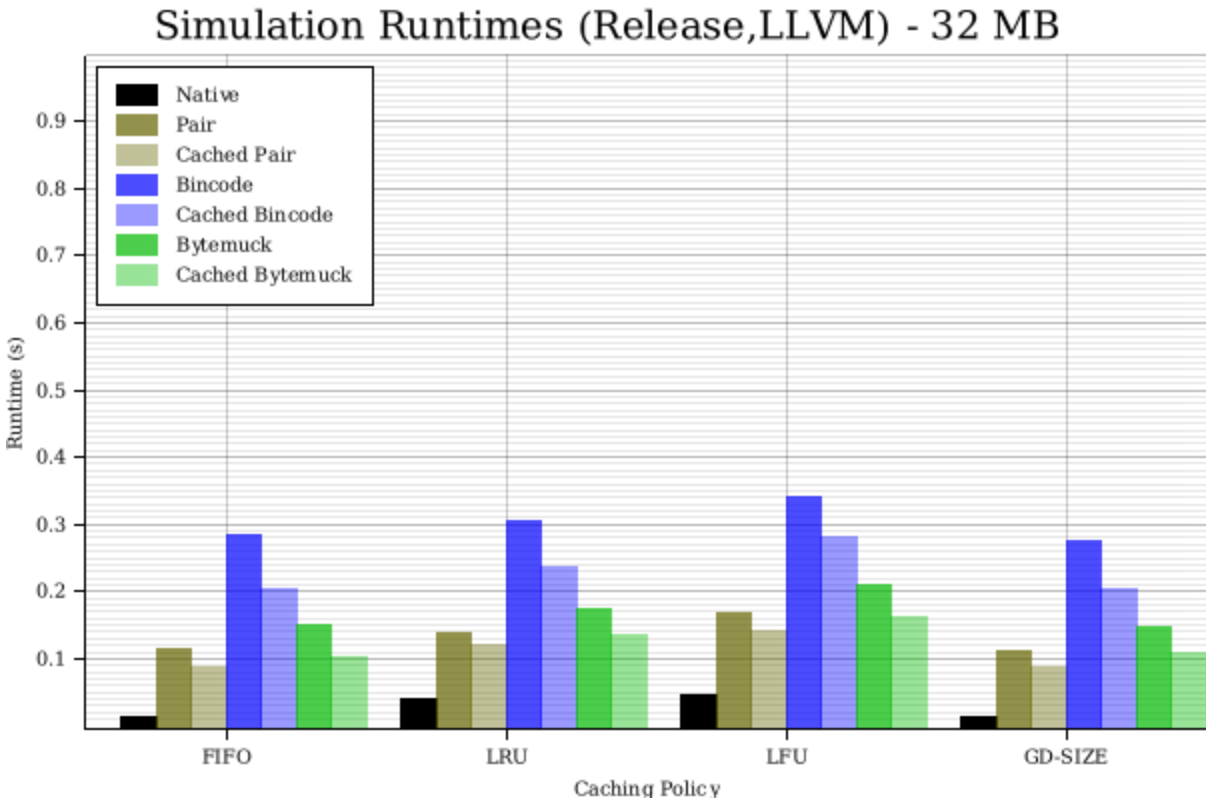
Macro Benchmark

- Caching continued to perform better than the non-cached version.
- The gap between native and WASM closed by many orders of magnitude, implying the performance degradation is primarily caused by calls into WASM.
- Relative speed of each policy simulation was the same within each ABI, so WASM seems to be a relatively stable slowdown.



Macro Benchmark

- LLVM version performed marginally faster.
- While policy code is more complex, it is still very simple, so LLVM extra optimizations do not seem to have much impact.
- Relative performance of ABIs remained the same.



Conclusions

- WASM slowdowns are within an order of magnitude for real-world scripting applications.
- There are more potential options for optimization, but when comparing the ABI it seems that direct arguments are preferred.
- If the data is more complex, then C-struct passing with static memory using tools like Bytemuck is preferred.
- If Bincode or Protobuff can be adapted to work with static memory, than its possible these could be used as a more language independent alternative to C-struct, since the primary penalty they suffer seems to be the second call, not the parsing itself.

Future Work

- Investigate implementation of Wasmer, to find potential optimizations.
- Investigate variable length data.
- WebAssembly will eventually be capable of understanding more complex types using WebAssembly Type Interface, so a future work could compare that subsystem to this methodology.
- Perform tests against other WASM runtimes such as Wasmtime and WebAssembly Micro Runtime (WAMR).
- Check performance and compatibility of these techniques with other WASM compatible languages such as C, C++, C# and AssemblyScript.

Thank you!

Source Code:

https://github.com/Sonicskater/wasm-simulator

References

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

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

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

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

[5] Erlang. Erlang – implementations and ports of erlang. https://erlang.org/faq/implementations.html

[6] 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[6]

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

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

[9] Veloren. https://gitlab.com/veloren/veloren