

# Let It Unthread: The Good, The Bad and The Ugly within WebAssembly Portable Multithreading

Marc Sánchez-Artigas\*, Julen Bohoyo-Bengoetxea\*

\*Computer Science and Maths, Universitat Rovira i Virgili

Tarragona, Catalonia, Spain

{marc.sanchez, julen.bohoyo}@urv.cat

**Abstract**—With a well-defined low-level virtual instruction set, low memory footprint and fast start-up times, WebAssembly has been strongly positioned as a lightweight alternative to containers. For a long time, one missing piece from WebAssembly standalone runtimes has been the incapacity to run shared-memory parallel code and benefit from multicore execution. With the recent WASI threads proposal, the tantalizing promise of cross-platform high-performance execution is more real than ever. In this article, we explore to what extent this promise is fulfilled by investigating the translation of POSIX threads applications to WebAssembly, and how their execution compares to native code. Using standardized benchmarks and a deep analysis of a popular standalone runtime, we reveal interesting findings on the lack of performance in multithreaded code cross-compiled to WebAssembly. We elaborate on the difficulty of correcting these inefficiencies, and even provide a mitigation to excessive thread locking caused by the default WASI libc memory allocator. Overall, we see WASI threads as a good starting point for the efficient execution of multithreaded code.

**Index Terms**—WebAssembly, WASI threads, Cloud computing

## I. INTRODUCTION

In the last few years, WebAssembly has become as popular bytecode language that claims rapid execution, and a low-level memory model [1]. Originally developed for web browsers, it has gained significant traction in the server space over the past years as an alternative to Linux containers [2]. For this reason, WebAssembly code can be natively run in many cloud services (e.g., Microsoft AKS [3]), not to mention all the big providers (e.g., Google, AWS), thanks to its wide support in Kubernetes cloud distributions [4], backed by the Cloud Native Computing Foundation (CNCF)<sup>1</sup>. In parallel, a number of research works have investigated the benefits of WebAssembly over containers [5]–[8], concluding that programs compiled to WebAssembly promise fast startup times, while running at near-native speeds.

Despite achieving great success in numerous contexts, one major limitation of server-side WebAssembly has been the lack of standard multithreading support for CPU-bound tasks. They can be from a parallel `gzip`<sup>2</sup> up to the DGEMM kernel from `ParRes` [9], as run in the `Faabric` [10] WebAssembly runtime. To fill this gap, the Bytecode Alliance<sup>3</sup>, founded by Microsoft, Mozilla, Intel, etc., released the WASI threads [11] proposal in 2023. In a few words, WASI threads standardizes the form of

spawning threads in WebAssembly runtimes, while building on the older WebAssembly threads proposal [12] to define shared memories to be accessed by multiple threads. Altogether, these two proposals allow WebAssembly virtual machines to execute `pthread`s applications compiled to WebAssembly.

Spurred by the advent of WASI threads, this work performs the first investigation of WebAssembly for multithreaded code. Despite the documented advantages in terms of cross-platform portability and application packaging [13], the simple fact that WebAssembly was not originally conceived for multithreaded programs raises a key question: *To what extent is WebAssembly ready to run multithreaded code?*

To answer this question, this paper examines the brand new WASI threads proposal in depth. Our analysis spans numerous dimensions: from memory management to synchronization, in order to best portray the real proficiency of WebAssembly for shared-memory multi-threaded programs. For each aspect, we identify the major issues and then quantify their impact using specific benchmarks. The objective of this paper is not only to give quantitative numbers but to spell out the elements hurting performance, so that they can be re-formulated in a near future. To achieve fine control over evaluation, we used C/C++ POSIX threads applications of several flavors: memory-intensive, with synchronization primitives, and even three HPC benchmarks.

It is worth to mention here that this research evaluates WASI threads, but the gained insights are universally applicable to all previous work, since all WASI systems suffer at least from the “instance-per-thread” model, the only existing way to execute WebAssembly code to the date [14].

After our benchmarking and detailed analysis, we provide a concise discussion on the complexities of resolving the major sources of overhead. As a general conclusion, we observe that most of the performance limiting factors are relatively easy to fix. To support this claim, we show the feasibility to clear one of the bottlenecks in threaded WebAssembly: heap contention. To this end, we replace the default WASI libc memory allocator by `mimalloc` [15], an open-source memory allocator invented by Microsoft. By using `mimalloc`, each WASI thread can allocate memory from its own heap and thus mitigate heap contention. Finally, our results certify that the `Wasmtime` [16] runtime delivers near native performance compared to `musl-libc`, a lighter alternative C library to GNU `glibc`.

**Contributions.** In summary, this paper contributes:

<sup>1</sup><https://www.cncf.io/>

<sup>2</sup><https://github.com/bytecodealliance/wasm-parallel-gzip>

<sup>3</sup><https://bytecodealliance.org/>

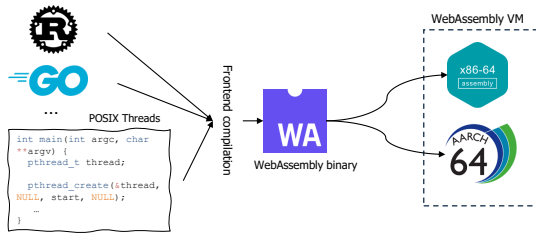


Fig. 1: A holistic view of Wasm compilation and execution.

- A profound performance analysis of WebAssembly multi-threading support for server-side execution through WASI threads. Our goal is not only to comprehend the effects of the different architectural decisions taken in this proposal, but also in the WASI C library, to help users establish an informed view of WebAssembly for multithreaded code.
- An thorough evaluation of WebAssembly overheads using both standard HPC benchmarks and handcrafted code, the latter to assess the specific contribution of WebAssembly virtual machines to the parallel execution overheads.
- Practical evidence that existing multithreading support in WebAssembly bears room for improvement, along with a discussion on the complexity of the enhancements within the WebAssembly ecosystem. For example, we showcase how the replacement of the default WASI libc allocator by an allocator based on per-thread heaps can alleviate one of the major sources of overhead.

**Artifacts.** <https://doi.org/10.5281/zenodo.14899517>.

## II. BACKGROUND ON WEBASSEMBLY

**Overview.** In Fig. 1, we provide an holistic view of the typical WebAssembly **compilation** and **execution** pipeline. When an application, written in a high-level language, such as C, C++, Rust, etc., is compiled into WebAssembly, the resulting binary is called a “module”. After compilation, the module can be run on any of the supported platforms (e.g., x86) via a standalone WebAssembly VM [1] such as Wasmtime [16].

The WebAssembly VM is a stack machine without registers. This means that the instructions pop their inputs from and push their output results to the implicit stack. There are 4 primitive data types: 32-bit and 64-bit integers and floats. Complex data types such as C/C++ structs do not exist. Their source-level types are encoded as raw bytes and saved to the linear memory.

Besides traditional control flow instructions, e.g., `if-else`, WebAssembly adds specific ones. For instance, `br_table` can define multiple target branches to jump to. Further, it must be noted that WebAssembly features Control Flow Integrity (CFI) to prevent exploits that attempt to take control of program flow.

**Memory.** WebAssembly does not provide “managed” memory or garbage collection by design. Instead, the standard proposes a byte-addressable linear memory region, where load and store instructions can access arbitrary addresses within this area. To this end, WebAssembly uses 32-bit pointers, where the “`i32`” type serves as the pointer type. This imposes a boundary on the total memory of a module of 4 GBs, but also enables ultra-fast

bound memory checks at runtime [17], which is basic to grant Software Fault Isolation (SFI) [18] capabilities to modules. For effective dynamic memory allocation, each frontend compiler embeds a memory allocator of its choice.

The compiler is also responsible for arranging the memory section by dividing the linear memory into a call stack, a heap, and the static data area. For instance, the Clang [19] compiler includes a call stack with its top pointed by the global variable `__$stack_pointer`. Above the stack, Clang places the heap, such that it can freely grow towards higher memory addresses, and request extra memory when needed. Below the stack is the static data section [20].

**WebAssembly System Interface (WASI) [21].** In a nutshell, WASI is a standard API that enables WebAssembly modules to interact with the underpinning host OS, e.g., to implement I/O operations outside the browser. To facilitate the compilation of C/C++ applications with WASI support, developers can make use of the WASI-SDK [22], which is equipped with Clang [19] and its own C library based upon `musl libc`: the `wasi-libc`. This library replaces most of the syscalls to the underlying OS by the respective WASI hostcalls imported from the standalone VM (or runtime). Recall that a “hostcall” is a function called within the module that is not defined in it, but provided by the host. Due to the ubiquity of `glibc` [23] on Linux systems, some applications may depend on `glibc`-specific functions or certain semantics, requiring code rewriting prior to their compilation to WASI. This occurred in some of our benchmarks.

## III. ENABLING THREADS OFF-BROWSER: WASI THREADS

### A. Overview

One vital requirement for HPC applications is the ability to spawn new threads within a process. This feature was officially missing in WebAssembly since its inception until the release of the WASI threads [11] specification last year. Interestingly, this proposal consists of a single hostcall that must be implemented by the WebAssembly VM. In pseudo-code:

```
status wasi_thread_spawn(thread_start_arg* start_arg);
```

where `status` is the non-negative thread ID (TID) of the new thread, or a negative number if the Wasm runtime failed to create the new thread. The role of the `start_arg` parameter is to allow programmers to pass start arguments to the newly created thread as defined in the POSIX standard.

As expected, a `wasi-threads` module initially executes a single thread — the main thread. As `wasi_thread_spawn` is called, more and more threads begin to execute. Consequently, WASI threads can be seen as a 1:1 thread library, where each thread created via `wasi_thread_spawn` corresponds one-to-one with a user thread. How user threads map into schedulable entities in the OS kernel (e.g., lightweight processes in Linux) depends on each VM. To illustrate, Wasmtime [16], built upon top of Rust, uses the standard library, `std::thread`, to create threads. The Rust standard library adopts a 1:1 model of thread implementation, where each user thread corresponds to one OS thread. In this sense, WASI threads does not pre-establish any

relationship between a WebAssembly-level thread and a kernel thread, opening the door to other models —e.g., green threads.

**Execution model.** The thread execution mechanism in WASI threads abides by the “instance-per-thread” approach. To put it baldly, each thread executes an independent cloned instance of the module. To facilitate inter-thread interaction, all the cloned instances share a common linear memory. In this way, threads can exchange data by accessing the same memory locations in the shared memory. As standard multithreaded programming, unprotected memory access can lead to harmful data races that yield unpredictable program state.

Upon a call to `wasi_thread_spawn`, the WASI host must:

- instantiate the module again; the child instance will run in the new thread;
- in the child instance, import all the same Wasm objects as the parent (shared memory, host functions, etc.);
- spawn a new host-level thread;
- get a positive, non-duplicate thread ID (TID) for the new thread and return it to the parent; and finally
- in the new thread, call the exported entry function in the child instance: `wasi_thread_start(TID, start_arg)` to start running parallel code.

**Thread termination.** Termination can be caused by:

- returning from the thread start function; the other threads continue to execute;
- a trap or a `proc_exit` call in any thread, which provokes the rest of threads to immediately terminate.

### B. Major overheads

We identified three major sources of overhead:

**Issue 1. Memory management.** WASI threads does not revisit memory management to support multithreaded code. The bad consequence is that memory-intensive parallel applications can suffer a severe slowdown due to the impossibility to undertake concurrent heap management activities. We analyze this in §V.

**Issue 2. Synchronization support.** Like above, WASI threads does not cover synchronization primitives. They have just been borrowed from the WebAssembly-level threads proposal [12]. This API was devised for the execution model of browsers, so its server-side performances is uncertain. We study this in §VI.

**Issue 3. Execution model.** As noted above, the WASI threads spawning mechanism adopts the “instance-per-thread” model. This approach strongly differs from the native execution model of C/C++ `pthread`s code, where a single process can contain multiple threads. This atypical way of executing multithreaded code raises some efficiency concerns, as shown in §VII.

## IV. METHODOLOGY

In this section, we describe the methodology of our study.

### A. WebAssembly runtimes

Currently, three runtimes already support the WASI threads proposal: Wasmtime [16], Wasmer [24], and WAMR [25]. To not favor one runtime over the others, we will use the same execution model in all runtimes: JIT compilation. Specifically,

TABLE I: Specific versions of runtimes and C libraries.

Software:	Wasmtime	Wasmer	WAMR	glibc	musl-libc
Version:	16.0.0	4.2.3	1.3.1	2.35	1.2.4

for Wasmtime and Wasmer, we will leverage Cranelift [26] as the compilation backend. For WAMR, we will use LLVM [19]. As reported in [27], the speed of machine code generated by Cranelift is 14% slower than LLVM. Its compilation speed is, however, an order of magnitude faster than LLVM. Cranelift is thus interesting when faster compilation is really important. To compile source code to Wasm, we used WASI-SDK 21.

Furthermore, we will compare the performance of the three Wasm runtimes against GCC + glibc and GCC + musl-libc. The main reason to use musl-libc is because wasi-libc has been built from musl-libc by means of conditional compilation (i.e., `#ifdef` blocks), so it is a ideal baseline for performance comparison. In Table I, we list the specific version of the three runtimes and the distinct C libraries used throughout the paper.

### B. Benchmarks

To investigate each of the three category of issues that WASI threads encounters, we make use of a large variety of standard benchmarks to better capture the idiosyncrasies of each issue. To wit, we use several real world programs from the mimalloc-benchmark [15] such as `LarsonN` to measure thread memory contention. Also, we combine these issue-specific benchmarks with more general benchmarks such as the PARSEC suite [28], which features state-of-the-art, CPU-intensive algorithms and very diverse workloads from different areas of computing. In each section, we will indicate the specific benchmarks used.

### C. Setup

All experiments were performed on a server equipped with AMD EPYC 7502 32-Core Processor CPU (2 sockets, 32 cores per socket, Simultaneous Multi-Threading (SMT), 2.5 to 3.35 GHz, 128MB L3 cache) and 128 GB of RAM. The operating system is Ubuntu 22.04.3 LTS 64-bit.

## V. MEMORY MANAGEMENT

### A. Memory layout

In Fig. 2, we illustrate the memory layout for WASI threads. Perhaps non-surprising, because `wasi-libc` is based on musl, we found that the per-thread data structures are arranged in the linear memory in the order defined by musl: the thread stack is in first place, followed by thread-local storage (TLS), then the thread control block (TCB), leaving thread-specific data (TSD) at the end. We also noticed that the TCB is the same as musl, with the exception of the field that points to the dynamic thread vector (DTV), which is not available in WebAssembly, because all TLS allocations are static<sup>4</sup>. Moreover, the TSD region can hold a maximum of 128 keys, which is the minimum allowed by POSIX. This implies that the TSD can occupy at most 512 bytes per thread with 32-bit pointers.

<sup>4</sup>In this way, the compiler can calculate the thread pointer-relative offsets to TLS variables statically.

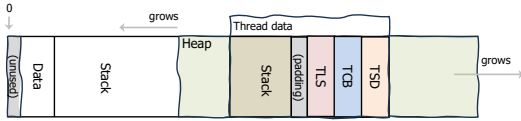


Fig. 2: Example of memory layout for one thread.

Very interestingly, we found that all the per-thread memory structures are dynamically allocated with `malloc`, which has important consequences. First off, it prevents the exploitation of virtual memory to install guard pages between threads with the help of `mmap` and `mprotect` syscalls. Consequently, stack overflows in a given thread can surreptitiously corrupt data in its adjacent thread, and for instance, tamper with values in the TSD region, or even the TCB. Second, it implies that the per-thread memory structures are deterministically arranged in the linear memory, i.e., the thread stack, TLS and TSD positions are predictable from the compiler and program. So for instance two consecutive calls to `pthread_create` will place the two thread memory structures contiguously in the linear memory. Due to this memory management, WASI threads programs can be more vulnerable to crashes or attack than native code.

### B. Memory Setup

Another interesting point is how thread memory is set up in WASI threads applications, that is, when `pthread_create` is called to spawn child threads. Compared to native `pthread` code, the process is a little intriguing. The reason is that WASI threads do not use the syscall `clone`, unavailable in WASI, but rather the so-called hostcall `wasi_thread_spawn`.

**How does it work?** Essentially, the process seeks to initialize two pointers: the stack pointer (i.e., `$_stack_pointer`) and the pointer to the base address of TLS (i.e., `$_tls_base`). To make it work, these two pointers are treated as module globals by the compiler. The “trick” here is that since each instance has its own set of globals, these two pointers effectively operate as thread locals. The bad news is that the correct values for these two pointers are calculated in the parent instance and must be passed to the child thread. This parent-to-child communication can only be done via the WASI host. More specifically, WASI threads adds both pointers as members of an internal structure that `pthread_create` uses to store the start argument:

```
struct start_args {
    void *stack;
    void *tls_base;
    void *(*start_func)(void *);
    void *start_arg;
};
```

In this way, both pointers can be opaquely passed to the WASI-native runtime via the `wasi_thread_spawn` hostcall. To copy the pointers back to the child instance, WASI threads specifies an entry function exported as “`wasi_thread_start`” for that purpose. This function is called by the WASI runtime, and acts as a trampoline function that sets the `$_stack_pointer` and `$_tls_base` globals with the addresses in `start_args`, and finally executes the thread entry function: `start_func`.

Compared to native `pthread`s, the memory setup for WASI threads is expensive due to communication between parent-to-child Wasm instances, which is slow (see §VII).

**TLS initialization.** The Clang compiler places all thread local variables into a single passive `.tdata` segment. As expected, the address of a thread local variable consists of `$_tls_base` + the offset from the start of the `.tdata` segment. In this case, the parent instance initializes the thread local storage for the child instance without assistance from the WASI runtime. To correctly allocate memory for threads, Clang introduces a new intrinsic function called `__builtin_wasm_tls_size`, which determines the size of the thread-local storage. To complete the initialization, the parent instance calls a new linker-synthesized function, `__wasm_init_tls`, which takes the `$_tls_base` address and the size of the thread local storage, and copies the `.tdata` segment into the linear memory via the `memory.init` instruction.

### C. Heap Management

Another important aspect is heap management. Particularly, how the memory allocator shipped with the Wasm binaries can affect the performance of `pthread`s programs. We recall that the official WebAssembly specification does not overly impose a concrete allocator. Consequently, compilers statically include a memory allocator as part of the Wasm binary. For this reason, allocators for WebAssembly programs tend to be simpler and lightweight (e.g., `wee_alloc`<sup>5</sup> for Rust programs).

`wasi-libc`, however, comes along with an allocator based on `dlmalloc` [29], which is competitive with native code for single-threaded programs and is relatively small in code size. It roughly occupies 7.67 KBs, and produces small fragmentation. Compared to `dlmalloc`, the `wasi-libc` allocator inherits and amplifies its multithreading inefficiencies and raises new ones:

The first important difference occurs when the `wasi-libc` allocator cannot find a suitable chunk for an allocation request. In that case, the allocator invokes a custom version of the `sbrk` syscall to extend the heap. This custom `sbrk` is nothing but a wrapper of the `memory.grow` instruction with some additional checks (e.g., to validate that the heap increment is page-sized). Albeit simple, this approach has important consequences. The first is that the heap can only be extended in units of page size, which is of 64 KiB. The second is that WebAssembly does not define any instruction to shrink the linear memory. Therefore, there is no way to `unmap` and release unused memory back to the runtime. As a result, the `wasi-libc` allocator can lead to significant amplification and high fragmentation [30].

The second issue with the `wasi-libc` allocator is that it is not thread-safe out of the box. This behavior is inherited from `dlmalloc`, which simply surrounds every call to `malloc` and `free` with a custom spinlock. As spinlocks become wasteful if held for long duration, the implementation calls `sched_yield` after some loop iterations to relinquish CPU time. In the WASI world, `sched_yield` is not a POSIX syscall but a hostcall that

<sup>5</sup>[https://github.com/rustwasm/wee\\_alloc](https://github.com/rustwasm/wee_alloc)

each runtime has to provide to be fully WASI-compliant. For instance, this hostcall has been implemented in the Rust-based Wasmtime runtime by calling `thread::yield_now` from the `std` module. This design has two main downsides:

First, the global spinlock is a major bottleneck for memory-intensive programs. Irrespective of this, its implementation varies from runtime to runtime due to hostcalls (see §VI), which it introduces more non-determinism in the execution of multithreaded code compiled to WebAssembly.

#### D. Evaluation

We evaluate the performance of the `wasi-libc` allocator in this section. As noted above, it is the major memory bottleneck for multithreaded programs. Other memory structures such as the thread stack depend on the architecture and the conventions adopted by the JIT compiler, making no big difference whether they are native or not. To grasp the efficiency of the WASI libc allocator, we picked the `mimalloc-benchmark` [15], a standard benchmark designed to stress memory allocators. We split the test applications into two categories: multithreaded and single-threaded, in order to better reveal the performance degradation in multithreaded code. We also utilized two `glibc` benchmarks: `bench-malloc-simple` and `bench-malloc-threads` from `glibc` 2.35 benchmarks [31]; and `t-test1` from `ptmalloc` [32].

Results for single-threaded and multithreaded test programs are depicted in Fig. 3 and Fig. 4, resp., all normalized to native `glibc`, and with degradation percentages displayed above bars. For single-threaded test programs, the degradation in execution time is tolerable, generally staying within 100 percent (i.e., less than double). However, the degradation becomes very apparent in multithreaded code, typically being at least  $19\times$  higher (e.g., `mstress`), but exhibiting a severe slowdown of  $\sim 1,700\times$  for `lardon`. Overall, Wasmtime performs the best among all the WebAssembly runtimes with values on par with `musl-libc`.

Regarding memory usage, taken as the maximum resident set size (max. RSS), we see that degradation is not too severe, excepting Wasmer with RSS values  $\sim 10\times$  larger than `glibc` in some programs. This result is rather encouraging and indicates that both Wasmtime and WAMR do a good job of exploiting virtual memory to allocate new WebAssembly memory pages. We recall that the Wasm page size is of 64KiB (i.e.,  $16\times$  larger than in Linux). Precisely, because of Wasm memory pages are larger, Wasmtime and WAMR perform better than native code in the `malloc-large` benchmark (see Fig. 3a).

**Main takeaway:** The WASI libc memory allocator is a major limiting factor for memory-bound multithreaded code compiled to WebAssembly. The “good” is that the overhead in memory usage is small in general.

## VI. SYNCHRONIZATION

### A. Overview

As said before, WebAssembly uses WASI to operate outside the browser and ensure portability across platforms. Following suit, the WASI threads proposal emerged to support threads in WASI. Despite several years of standardization efforts, WASI

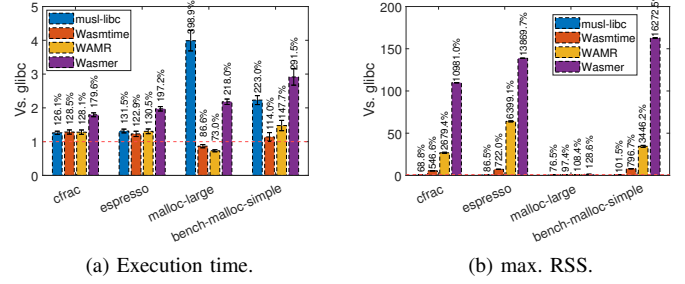


Fig. 3: wasi-libc allocator for non-threaded benchmarks.

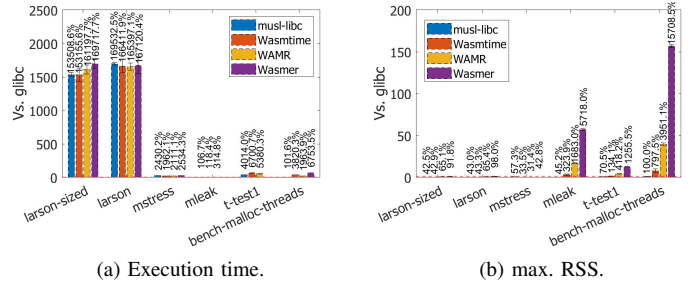


Fig. 4: wasi-libc allocator for multithreaded benchmarks.

still remains as an overly simplified OS interface, which makes it hard to execute many applications. WASI does not only lack common OS features such as memory-mapping, process `fork`, and signals, but also syscalls such as `futex`, which are used to implement synchronization primitives such as mutexes in user-space thread libraries. In its stead, the approach taken by WASI threads has been to borrow the instructions defined in another proposal, the WebAssembly threads proposal [12], to program `pthread`s synchronization primitives. This path was followed to encourage rapid adoption since the operations defined in this proposal are supported by all the major WASI-native runtimes.

All in all, this proposal contributes new instructions to allow atomic memory accesses in the linear memory. These operators can be grouped into three classes: load and store; read-modify-write; and compare-exchange. As of today, the memory model for all these atomics is “sequential consistency” [14], which is the only memory ordering supported in WebAssembly. This is a strong memory model that requires preserving a total order of atomics across all threads, which incurs high penalties. Weaker memory models such as “release-acquire” will be of great help to raise performance by allowing certain reordering of memory accesses. WebAssembly support for “release-acquire” will for instance make it possible to create efficient critical sections on weakly-ordered systems (e.g., ARM [33]), which today cannot be unlocked to their full potential due to lack of WebAssembly support. This situation is not that critical in our benchmarking experiments since x86 TSO memory model is a relaxed variant of sequential consistency [34].

### B. Atomics Implementation

The major problem with the current approach is that atomics must be reimplemented in each WASI runtime. Fortunately, for most of the runtimes, the embedded backend compilers do that



job, and convert these atomic operations straight into machine instructions for the most common architectures (e.g., x86\_64). For example, Cranelift maps the `i32.atomic.rmw.cmpxchg` instruction to the machine-independent `atomic_cas` operator and then down to the x86 lock-prefixed instruction `cmpxchg`. However, other instructions such as `memory.atomic.wait32` and `memory.atomic.notify` do not get that luck and need to be handled by the runtime. This occurs because `wait32` allows to suspend a thread if the `i32` value on a given memory address matches the expected value. This can only be realized with the help of the WASI-native runtime, which is the only authorized process that can leverage OS features such as the Linux syscall `futex` for that purpose.

To illustrate, Wasmtime implements this instruction by first using the Rust atomic `AtomicU32` to atomically validate if the value in memory equals to the expected one. If not, the thread is put to sleep by invoking the `park_timeout` function in the `std::thread` Rust API. In Linux, `park_timeout` triggers a call to `futex wait`. Also, the memory ordering for the atomic is set to ‘SeqCst’ to ensure sequential consistency as dictated by the threads proposal [12], though Rust atomics admit more relaxed memory orderings.

The “ugly” consequence of this model is that the same synchronization functionality has to be re-implemented in each runtime.

### C. Case Study: Mutex

As a representative case study, we evaluate the `wasi-libc` implementation of `pthread_mutex` to create critical sections. This choice is not casual. The `wasi-libc` implementation of `mutex` is almost identical to that in the `musl` C library, thereby allowing a one-to-one comparison between WASI threads and a native `pthread`s library. More specifically, the `wasi-libc` (`musl`) implementation of `mutex_lock` consists of a fast path and a slow path:

- A thread is on the fast track when the value of the `mutex` lock is 0 (i.e., “unlocked”). The lock can then be acquired with one atomic operation: `i32.atomic.rmw.cmpxchg`, which atomically sets the lock value to `EBUSY` (“locked”) if successful.
- Otherwise, the thread goes to the slow path. In first place, it enters a busy-waiting loop of at most `E=100` iterations, where it busy-waits for the lock holder to release the lock provided there are no waiters. A “waiter” is nothing but a suspended thread waiting to acquire the `mutex` lock. This is done to avoid the burden of suspending and wakening threads in low-contention situations, e.g., when a critical section is short.
- After the busy-waiting loop, the `mutex` lock is assumed to be under congestion. The thread then enters an acquisition loop. In the loop condition, the thread attempts to acquire the lock. If that fails, the thread is finally put to sleep with `memory.atomic.wait32` and becomes a waiter.

Function `mutex_unlock` is straightforward. At a high level, it follows the reverse path. It first unlocks the `mutex` lock with

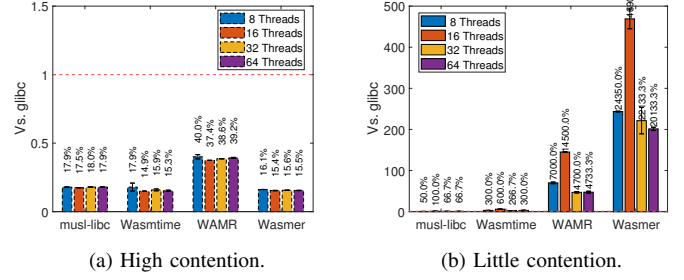


Fig. 5: Latency slowdown of `wasi-libc` mutex w.r.t. `glibc`.

`i32.atomic.rmw.cmpxchg` and if there are waiters, it calls `memory.atomic.notify` to wake up one suspended thread.

**Micro-benchmark.** To assess the performance of `mutex`, we have implemented a simple microbenchmark. The benchmark spawns `N` threads that try to access 100 times a critical section guarded by the `mutex`. To get a complete view, we adapted the code in the critical section to emulate two opposite scenarios: one with little contention, and another with high contention by performing longer math calculations. Fig. 5 depicts the results. We make two relevant observations here. First, the `wasi-libc` implementation of `mutex` performs better than `glibc` in highly-contended situations because its based upon the more efficient `musl-libc` implementation. However, this situation reverses for lightly-contended executions, where the time slowdown ranges between  $2.6\times$  (Wasmtime) up to  $500\times$  (Wasmer). This verifies that the performance of WebAssembly synchronization largely depends on the `musl-libc` base code.

The second key observation is that despite building upon the same base code (`musl-libc`), the way each runtime handles the involved memory atomics and hostcalls makes a big difference in terms of synchronization as reflected in Fig. 5.

### D. Evaluation

To evaluate synchronization, we make use of a set of widely-used benchmarks, written in C/C++ for Linux/`pthread`s. The benchmark suite includes 17 programs from SCTBench [35], which use the main synchronization objects of the `pthread`s library. For all programs, Fig. 6 reports a significant slowdown in WAMR and Wasmer. Conversely, Wasmtime performs fairly well with a geometric mean slowdown of  $1.9\times$ , which signals that efficient thread synchronization can be realized despite the overheads of WebAssembly sandboxing. It is worth to note that the SCTBench programs are worst case for WebAssembly. In general, they are short running and lightly contended, meaning that the slower creation of WebAssembly threads has a higher impact than native execution (see §VII). For instance, 79% of the total time in `phase01` was due to thread creation, and only 21% was due to synchronization in the Wasmtime VM.

**Main takeaway:** The performance of synchronization primitives highly varies across Wasm VMs. This occurs because the native OS syscalls in the `pthread`s API are reimplemented as hostcalls in every runtime, as well as the way atomics are handled by the backend compilers. The difference is more visible under low contention.

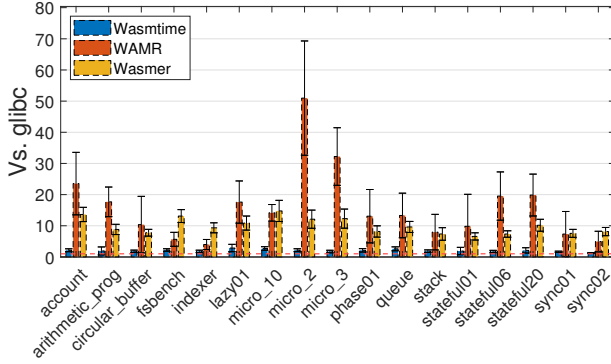


Fig. 6: Average execution time slowdown relative to glibc in the SCTBench [35] benchmark.

## VII. EXECUTION MODEL

### A. Overview

As aforementioned, a non-ordinary trait of WASI threads is the so-called instance-per-thread model. This execution model has profound roots in the early attempts to bring threading to WebAssembly. Before WASI threads, several WASI runtimes (e.g., WAMR, WAVM [36], Faasm/Faabric) already developed their own multithreading libraries following the instance-per-thread model. For this reason, the observations made here can be generalized to a large extent to all the runtimes.

**Memory consumption.** The first obvious observation to report is the higher memory consumption of the “instance-per-thread” model. Since each thread starts an independent instance of the module, the internal data structures (e.g., to represent “tables”) are duplicated, which may cause memory bloating. To confirm this, we ran a simple microbenchmark that creates a number of threads that lock into a barrier until all threads have reached it, and recorded the maximum resident set size (max. RSS). Two observations ensue from Fig. 7a. First, the memory consumed by the runtimes is at least an order of magnitude larger than native execution (glibc, musl-libc). And second, the increase in memory consumption per thread is significant. For instance, Wasmer consumes 189 KB of memory per thread despite that threads do nothing but waiting in a barrier. For more complex code, e.g., with many hostcalls, this cost may be even higher.

**Thread creation time.** The second key observation is that the thread creation time becomes significantly longer compared to native code. Although there are some runtime structures in the WebAssembly abstract machine that are “thread-safe”, module instantiation is a heavy process, even if the same module is re-instantiated multiple times. To better see this, Fig. 7b plots the cumulative time to spawn ten thousands threads in the different runtimes. As shown in the figure, Wasmer is by far the slowest VM, with a slowdown factor relative to glibc greater than 200. Fortunately, we were positively surprised to see that Wasmtime took a little more than double of the time of native execution in glibc, and 70% in musl-libc, to create 10,000 threads, which are excellent news. The major reason is that Wasmtime rebuilt

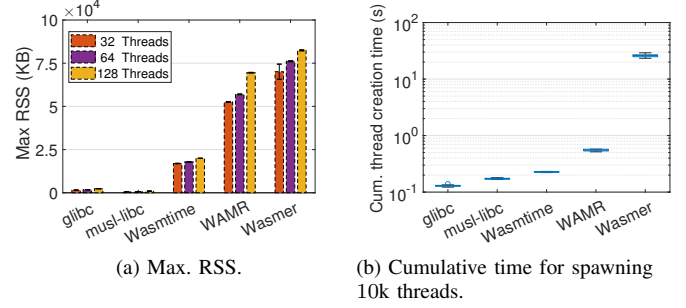


Fig. 7: Efficiency of the WebAssembly runtimes.

TABLE II: Top 5 benchmarks with most imported functions.

Benchmark:	fluidanimate	blackscholes	expresso	larson	larson-sized
# imports:	18	16	13	13	13

their instantiation model to avoid repetitive checks of the same WebAssembly binary; see next section for further details.

### B. Case Study: Wasmtime

Since the implementation of the instance-per-thread model varies significantly from one runtime to another, we decided to focus our efforts on Wasmtime, because it is the fastest runtime as shown in Fig. 7b. Due to space constraints, we omit a deeper analysis on how a WebAssembly binary is loaded and JITted to native code. To simplify exposition, we consider that a module has already been compiled and serialized into an ELF image. Internally, a module is represented as an atomically-reference-counted object. This means that all child thread instances share the same compiled code, but very importantly, that the meat of validation, e.g., to validate the signature of functions, happens just once for all instances.

If the WASI threads flag is set, Wasmtime then leverages a WebAssembly-level linker of its own to define WASI imports. This process includes the `wasi_thread_spawn` function from WASI threads. If successful, Wasmtime instantiates the WASI threads context (WTctx) whose role is to spawn a child thread upon a call to `wasi_thread_spawn`. The instantiation of WTctx creates a special structure of type `InstancePre`. The purpose of this object is to frontload import type-checking, as well as to generate the trampoline code for function imports prior to the instantiation event itself (see “lazy initialization” in [37]). The result of this design is that the instantiation process becomes lighter for threads since pre-instantiation occurs only once.

The `InstancePre` is also an atomically-reference-counted object. Consequently, it can be cheaply cloned. Upon each call to `wasi_thread_spawn`, the WASI threads context clones the `InstancePre` and creates a new OS thread. The new thread finally instantiates the module from the cloned preinstance and calls the entry function `wasi_thread_start` to start running useful thread code.

**Impact of imports.** Despite pre-instantiation proves effective, we wished to see whether the impact of imported functions is

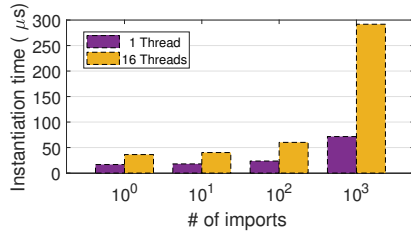


Fig. 8: Time to create a child instance in Wasmtime as function of the number of imports.

important at thread instantiation time. To this aim, we recorded the average per-thread instantiation time for a growing number of imports. The results given in Fig. 8 show that the per-thread instantiation time starts to become significant before a hundred of imported functions. For a hundred imports, it takes between 23 microseconds (1 thread) to 64 microseconds (16 threads) to spawn a child thread. At first glance, this overhead may seem large if one considers that the time to create a native thread in our system was 12 microseconds. In practice, however, the number of imports tends to be small. Table II reports the top 5 benchmarking programs with the highest number of imported functions. None of them declared more than 18 imports. Then it follows that imports will typically have little adverse effect on most WASI threads programs.

**Trampolines.** While most of the CPU time during execution is typically spent in the WebAssembly application itself, there are moments that the WebAssembly code has to perform hostcalls, or the other way round, the runtime has to call a WebAssembly function. For example, this occurs to `pthread_create` where `wasi_thread_spawn` is a Wasm-to-host call, and expectedly, `wasi_thread_start` is a host-to-Wasm call. Since calls that cross the boundary between WebAssembly and the runtime are common, fast trampolines are vital. For this reason, we decided to measure the overheads of Wasmtime trampolines.

In the past, the assembly trampolines were hand-written and did a tail call for increased efficiency. But in the newer versions of Wasmtime, the novel Cranelift-built trampolines maintain a call frame on the runtime stack and perform a register-indirect call (e.g., `call *%r13` in the `x86_64` architecture). This was realized to improve stack unwinding [37] at the cost of a minor regression in performance. We carefully analyzed the `x86_64` trampolines and found that they follow `x86_64`’s conventions with minor changes to keep a linked list of frame pointers.

Table III reports the time median type for Wasm-to-host and host-to-Wasm calls. For completeness, we include the time for Wasm-to-Wasm calls. Recall that these calls do not need to go through a trampoline because they do not cross the “boundary” between the host and Wasm. This table shows that inter-host-to-Wasm calls are time-consuming. Thus, abusing of hostcalls may introduce a high overhead to parallel execution.

### C. Evaluation

To conclude this paper, we ran several real-world programs in order to gain a wider perspective of Wasm performance for multithreading. We chose 3 programs from PARSEC 3.0 [28]:

TABLE III: Trampoline performance in Wasmtime. Data given in parentheses is the slowdown relative to a native call (glibc).

Call type:	Wasm-to-host	Host-to-Wasm	Wasm-to-Wasm
Median time:	7.93 ns (13.5× slowdown)	19.65 ns (33.5× slowdown)	1.59 ns (2.71× slowdown)

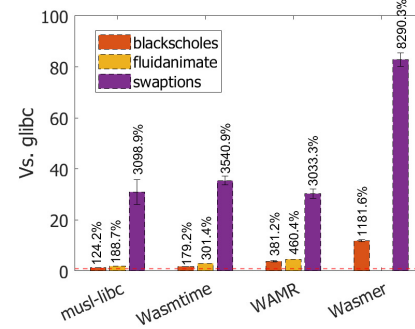


Fig. 9: Average execution time slowdown relative to glibc in three PARSEC [28] applications.

`blackscholes`, `fluidanimate` and `swaptions`, executing each with 64 threads, so that each thread was pinned to a core. We executed each application one hundred times and as before, we measured its slowdown relative to glibc. Results are plotted in Fig. 9. We notice that all executions of `swaptions` hang forever in Wasmer with no way to identify the cause. As seen in the figure, `blackscholes` and `fluidanimate` presented a geometric mean slowdown of roughly  $2.3\times$  in the fastest VM (Wasmtime), with Wasmer being the slowest VM in all cases. For `swaptions`, we found a large degradation in performance across all runtimes, as well as in `musl-libc`. Interestingly, the degree of performance loss of WAMR and Wasmtime was “on par” with the native execution of this application in `musl-libc`. After some research, it became clear that the problem was the memory allocators of both `musl` and WASI libc, which abused of `futex` waits.

The above similarity suggested us an insightful exercise: the performance comparison of multithreaded Wasm against native execution in `musl-libc`. In this situation, the geometric average slowdown reduced to 33% in Wasmtime, which can be seen as an encouraging result given the intrinsic overheads of SFI (e.g., **trampolines** to transition into and out of sandboxes), CFI, and JIT compilation [38].

**Main takeaway:** The “instance-per-thread” model hinders both thread creation and execution (e.g., because signaling other threads that some shared state has changed requires Wasm-to-host call). Nevertheless, with a thoughtful design like that of Wasmtime, the performance gap is not so huge to outweigh the benefits of portability.

## VIII. DISCUSSION

As a general conclusion, we see that there exists a significant performance gap for multi-threaded code compiled to Wasm. Only the fastest VM (Wasmtime) can be rated as ‘competitive’ today, with slowdowns  $< 33\%$  compared to native execution in `musl-libc`. This result agrees well with the literature, where



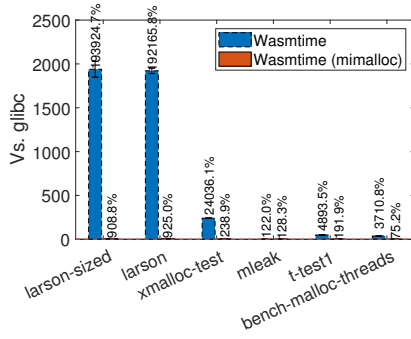


Fig. 10: Comparison of `wasi-libc` allocator vs. `mimalloc` for multithreaded benchmarks on Wasmtime for 64 threads.

other analysis [39] reported average slowdowns between 45% and  $2.5\times$  for sequential Wasm code in browsers.

The other question worth asking is if the performance issues identified here are fundamental. We believe that the first two of the identified issues are not. That is, they could be ameliorated by improved implementations. Unfortunately, the third issue is a foundational problem. We discuss one by one.

**Issue 1. Memory allocation.** One major bottleneck for parallel execution is that the default `wasi-libc` allocator uses a single global lock to protect each (de)allocation, hurting performance for memory-intensive programs. To demonstrate the feasibility of breaking this bottleneck, we managed to replace the default allocator by the lock-free `mimalloc` allocator [15], which uses thread-private heaps, eliminating the need for costly locking if allocation requests can be serviced by the local heap. We reran most of the multithreaded benchmarks of §V with and without `mimalloc` on Wasmtime for 64 threads. Results are plotted in Fig. 10. As shown in this figure, the use of a lock-free allocator has reduced the allocation times dramatically. For instance, the latency reduction in the `larsen-sized` program has been of  $213\times$ , which represents two orders of magnitude improvement over the WASI libc allocator. For `bench-malloc-threads`, WebAssembly was even faster than native execution by a 25%. In conclusion, improving memory allocation in multithreaded WebAssembly is within reach with modest effort.

**Issue 2. Synchronization.** Similar to Issue 1, synchronization is not a fundamental issue, as it would be increasingly resolved with improved implementations at different levels (WASI libc, VM, and compilers). At the WASI libc level, by reengineering the slow primitives to make their performance similar to glibc. At the VM level, by providing optimized implementations of `wait32` and `notify` hostcalls to suspend and resume threads. Of course, all of these complemented by a proper translation of WebAssembly memory atomics to CPU atomics.

**Issue 3. Execution model.** Conversely to the above two issues, the instance-per-thread model is a WebAssembly foundational principle. Since its inception, the workaround to enable multithreaded programs in web browsers has been to spawn multiple Web Workers, each instantiating the same module and sharing memory with a `SharedArrayBuffer` object [40]. Replacing this model is thus not the competence of WASI threads, but a

major change to the standard, requiring community consensus. The reason is that supporting the “many-threads-per-instance” model would require WebAssembly objects such as tables and globals to be “shareable”, as well as to add new instructions to dynamically start threads within a function such as `fork` [14].

## IX. RELATED WORK

Since the breakthrough of WebAssembly, numerous studies have been investigating the performance of WebAssembly [1], [39], [41]–[43]. For instance, [39] studied the performance gap between WebAssembly and C programs, while [41] presented an empirical study of 8,461 real-world WebAssembly binaries and examined their security properties, source languages, and use cases. To the best of our knowledge, our work conducts the first comprehensive study on the performance of multithreaded code compiled to WebAssembly.

Three years before the WASI threads proposal, WAMR [25] provided its own implementation of the `pthread` library. As WASI threads, its implementation followed the “instance-per-thread” model. And hence, the insights developed in this paper can be extended to WAMR as well. The major difference with WASI threads is that WAMR statically pre-allocates the stack memory for each thread before execution. Faasm [5] adopted the same strategy to enable the execution of OpenMP programs compiled to WebAssembly. Perhaps the most interesting aspect to highlight from Faasm is the design of the first “instance-per-thread” incarnation some years before it was the multithreaded model chosen for WASI threads. So we believe that most of the insights gained here can also be valuable to Faasm.

In addition to shared-memory multithreading, Faasm and MPIWasm [13] have taken the first steps towards enabling high performance execution of MPI programs over WebAssembly. Compared to multithreaded code, MPI code is more amenable to WebAssembly because the “instance-per-process” model fits well the concept of MPI process. Actually, the results of these papers show competitive native application performance.

## X. CONCLUSION

Recent progress in WebAssembly promises a cross-platform runtime for multithreaded code compiled from C/C++, as well as other languages to achieve near-native speeds. This research presents the first in-depth investigation of WebAssembly multithreaded support to debate to what extent this promise has been fulfilled. Our analysis identifies the major sources of overhead and quantifies their impact on performance. Our findings show that the overheads are high but also prove that the slowdown of multithreaded WebAssembly is by and large inferior to  $2\times$  for Wasmtime, which is on a par with that found in web browsers. We conclude this work by solving one of the performance gaps and giving actionable guidance for future optimization effort.

## ACKNOWLEDGMENT

This research has been partially funded by the EU under the Horizon Europe programme (Grants 101092646, 101093110), the Spanish MICIU/AEI Grant no. PID2023-148202OB-C21, and NextGenerationEU (CLOUDLESS UNICO I+D CLOUD 2022 project). Marc Sánchez-Artigas is a Serra Hùnter Fellow.

## REFERENCES

- [1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with webassembly,” *SIGPLAN Not.*, vol. 52, no. 6, pp. 185–200, jun 2017. [Online]. Available: <https://doi.org/10.1145/3140587.3062363>
- [2] S. Hykes, “Wasm+wasi: An alternative to linux containers,” 2021, accessed: January 17, 2024. [Online]. Available: <https://twitter.com/solomonstre/status/111100491322234225?lang=en>
- [3] M. G. et al., “Create webassembly system interface (wasi) node pools in azure kubernetes service (aks) to run your webassembly (wasm) workload (preview),” 2023, accessed: February 3, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/aks/use-wasi-node-pools>
- [4] “Kwasm – kubernetes operator for webassembly,” 2023, accessed: February 5, 2024. [Online]. Available: <https://kwasm.sh/>
- [5] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in *2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC’20)*. USA: USENIX Association, 2020.
- [6] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, “Sledge: a serverless-first, light-weight wasm runtime for the edge,” in *21st International Middleware Conference (Middleware’20)*, 2020, pp. 265–279. [Online]. Available: <https://doi.org/10.1145/3423211.3425680>
- [7] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, “Pushing serverless to the edge with webassembly runtimes,” in *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid’22)*, 2022, pp. 140–149.
- [8] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, “Webassembly as a common layer for the cloud-edge continuum,” in *2nd Workshop on Flexible Resource and Application Management on the Edge (FRAME’22)*, 2022, pp. 3–8. [Online]. Available: <https://doi.org/10.1145/3526059.3533618>
- [9] P. Team, “Parallel research kernels,” 2021. [Online]. Available: <https://github.com/ParRes/Kernels>
- [10] S. Shillaker, C. Segarra, E. Mappoura, M. Fournial, L. Vilanova, and P. Pietzuch, “Faabric: Fine-grained distribution of scientific workloads in the cloud,” 2023.
- [11] “Wasi threads – the webassembly system interface api to add native thread support,” 2023. [Online]. Available: <https://github.com/WebAssembly/wasi-threads>
- [12] “Threads proposal for webassembly,” 2024. [Online]. Available: <https://github.com/WebAssembly/threads>
- [13] M. Chadha, N. Krueger, J. John, A. Jindal, M. Gerndt, and S. Benedict, “Exploring the use of webassembly in hpc,” in *28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP’23)*, 2023, pp. 92–106. [Online]. Available: <https://doi.org/10.1145/3572848.3577436>
- [14] C. Watt, A. Rossberg, and J. Pichon-Pharabod, “Weakening webassembly,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: <https://doi.org/10.1145/3360559>
- [15] D. Leijen and M. Research, “mimalloc,” 2024, accessed: May 21, 2024. [Online]. Available: <https://github.com/microsoft/mimalloc>
- [16] B. Alliance, “Wasmtime – a standalone runtime for webassembly,” 2023. [Online]. Available: <https://github.com/bytecodealliance/wasmtime>
- [17] F. Denis, “Memory management in webassembly: guide for c and rust programmers,” 2019. [Online]. Available: <https://www.fastly.com/blog/webassembly-memory-management-guide-for-c-rust-programmers>
- [18] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 203–216, dec 1993. [Online]. Available: <https://doi.org/10.1145/173668.168635>
- [19] “The llvm compiler infrastructure,” 2023. [Online]. Available: <https://github.com/llvm/llvm-project>
- [20] D. Lehmann, J. Kinder, and M. Pradel, “Everything old is new again: Binary security of webassembly,” in *29th USENIX Conference on Security Symposium (SEC’20)*, 2020.
- [21] “Wasi – the webassembly system interface,” 2023. [Online]. Available: <https://wasi.dev/>
- [22] “Wasi-sdk,” 2024. [Online]. Available: <https://github.com/WebAssembly/wasi-sdk>
- [23] “The gnu c library (glibc),” 2024. [Online]. Available: <https://www.gnu.org/software/libc/>
- [24] “Wasmer: The universal webassembly runtime,” 2024. [Online]. Available: <https://github.com/wasmerio/wasmer>
- [25] “Wamr – webassembly micro runtime,” 2024. [Online]. Available: <https://github.com/bytecodealliance/wasm-micro-runtime>
- [26] B. Alliance, “Cranelft code generator,” 2023. [Online]. Available: <https://github.com/bytecodealliance/wasmtime/tree/main/cranelft>
- [27] H. Xu and F. Kjolstad, “Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: <https://doi.org/10.1145/3485513>
- [28] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in *17th International Conference on Parallel Architectures and Compilation Techniques (PACT’08)*. ACM, 2008, pp. 72–81. [Online]. Available: <https://doi.org/10.1145/1454115.1454128>
- [29] D. Lea, “A memory allocator,” 2000. [Online]. Available: <http://gee.cs.oswego.edu/dl/html/malloc.html>
- [30] A. Hunter, C. Kennelly, P. Turner, D. Gove, T. Moseley, and P. Ranganathan, “Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI’21)*. USENIX Association, Jul. 2021, pp. 257–273. [Online]. Available: <https://www.usenix.org/conference/osdi21/presentation/hunter>
- [31] I. Free Software Foundation, “The gnu libc benchtests,” 2022. [Online]. Available: <http://alpha.gnu.org/gnu/glibc/>
- [32] W. Gloger, “ptmalloc: a memory allocator,” 2006. [Online]. Available: <http://www.malloc.de/en/>
- [33] S. Parker-Haynes, “Sequential consistency in armv8,” 2022. [Online]. Available: <https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/armv8-sequential-consistency>
- [34] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, “x86-tso: a rigorous and usable programmer’s model for x86 multiprocessors,” *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010. [Online]. Available: <https://doi.org/10.1145/1785414.1785443>
- [35] P. Thomson and A. Donaldson, “Sctbench: a set of c/c++ pthread benchmarks for evaluating concurrency testing techniques,” 2016, retrieved: January 13, 2024. [Online]. Available: <https://github.com/mc-imperial/sctbench>
- [36] “Wavm – webassembly virtual machine,” 2024. [Online]. Available: <https://wavm.github.io/>
- [37] C. Fallin, “Wasmtime 1.0: A look at performance,” 2022, accessed: March 21, 2024. [Online]. Available: <https://bytecodealliance.org/articles/wasmtime-10-performance>
- [38] A. Khrabrov, M. Pirvu, V. Sundaresan, and E. de Lara, “JITServer: Disaggregated caching JIT compiler for the JVM in the cloud,” in *2022 USENIX Annual Technical Conference (USENIX ATC’22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 869–884. [Online]. Available: <https://www.usenix.org/conference/atc22/presentation/khrabrov>
- [39] A. Jangda, B. Powers, E. D. Berger, and A. Guha, “Not so fast: Analyzing the performance of WebAssembly vs. native code,” in *2019 USENIX Annual Technical Conference (USENIX ATC’19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 107–120. [Online]. Available: <https://www.usenix.org/conference/atc19/presentation/jangda>
- [40] A. Crichton, “Multithreading rust and wasm,” 2018, retrieved: August 23, 2024. [Online]. Available: <https://rustwasm.github.io/2018/10/24/multithreading-rust-and-wasm.html>
- [41] A. Hilbig, D. Lehmann, and M. Pradel, “An empirical study of real-world webassembly binaries: Security, languages, use cases,” in *Web Conference 2021 (WWW’21)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 2696–2708. [Online]. Available: <https://doi.org/10.1145/3442381.3450138>
- [42] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, “Understanding the performance of webassembly applications,” in *21st ACM Internet Measurement Conference (IMC’21)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 533–549. [Online]. Available: <https://doi.org/10.1145/3487552.3487827>
- [43] Z. Liu, D. Xiao, Z. Li, S. Wang, and W. Meng, “Exploring missed optimizations in webassembly optimizers,” in *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’23)*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 436–448. [Online]. Available: <https://doi.org/10.1145/3597926.3598068>