Leaps and bounds: Analyzing WebAssembly's performance with a focus on bounds checking

Anonymous Author(s)

ABSTRACT

WebAssembly is gaining more and more popularity, finding applications beyond the Web browser – for which it was initially designed for. However, its performance, which developers aimed at being comparable to native, requires further tuning and hasn't being studied extensively to pinpoint the cause of overheads. This paper identifies that WebAssembly unique safety mechanisms, one of the major ones being bounds-checked memory accesses, may introduce up to 650% overhead.

Therefore, we evaluate four popular WebAssembly runtimes against native compiled code. These runtimes have been enriched with modern bounds checking mechanisms and run on three different ISA CPUs, including x86-64, Armv8 and RISC-V RV64GC. We show that performance-oriented runtimes are able to achieve performance within 20% of the native performance on x86_64 platforms, 35% for Armv8. On RISC-V the V8 runtime can achieve a 17% overhead over native code for simple numeric kernels. For simple numerical kernels, we have shown that there is no significant difference in the WebAssembly performance compared to native code across different ISAs.

We have shown that in case of multithreaded scaling of the tested runtimes, which might for example be used to quickly scale up serverless instances for a single function without the overhead of spawning new processes, the default approach taken by WAVM, Wasmtime and V8 of using the mprotect syscall to resize memory can cause excessive locking in the Linux kernel and present an alternative userfaultfd-based solution to mitigate this issue.

We share our results and the tools and scripts under an open source license for other researchers to replicate our results, and monitor the progress that WebAssembly runtimes make as this technology evolves.

IISWC'22, November 6-8 2022, Austin, Texas, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/1234.5678

ACM Reference Format:
Anonymous Author(s). 2022. Leaps and bounds: Analyzing WebAssembly's performance with a focus on bounds checking. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC'22).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.475/1234.5678

1 INTRODUCTION

Language virtual machines are incredibly popular, enabling programs to be written once and run on a variety of CPUs, specifically CPUs of different Instruction Set Architectures (ISAs), without the need for recompilation, and often providing enhanced security guarantees compared to native execution. WebAssembly is a language that is steadily gaining traction and is intended to be run on a virtual stack machine. The initial goal of the WebAssembly project was to develop a portable and compact binary representation that would reduce the reliance of web applications on JavaScript and allow them to run at near-native speeds within browsers. Since then, WebAssembly has found usage in other application domains; most notably as a plugin sandbox mechanism [5] and as a Function-as-a-Service (FaaS) runtime [30]. However, despite the name, WebAssembly applications are not confined to the web. The WebAssembly System Interface (WASI) [32] provides a uniform way for WebAssembly code to communicate with the underlying system (be that the browser or the operating system), thus extending the benefits of WebAssembly far beyond the web.

However, WebAssembly is distinct from other languages that use virtual stack machines, in that it is an assembly-like language with a low-level memory model. Instead of having memory management features like a managed heap and garbage collection, it is a simple stack-based bytecode that operates on two main data structures: a linear memory, which is just a large array of bytes, and tables of function pointers, which act as a sandboxing mechanism for indirect branch instructions so that their only targets can be valid WebAssembly functions. This suggests that the bounds checking mechanism for validating linear memory (and, less frequently, function table) accesses is likely a performance overhead that is largely unique to current WebAssembly runtimes. Of course, other issues still exist, such as register allocation from the stack bytecode or limitations of the structure that WebAssembly enforces on the control flow between basic blocks, but similar concerns also exist in other native and dynamic programming languages.

In this paper, we consider a variety of WebAssembly runtimes, ranging from an interpreter to an LLVM-based AOT compiler. Our aim is to evaluate the current state of WebAssembly performance when compared to native code – without bounds checking – on three major instruction set architectures CPUs: x86-64, ArmV8 and RISC-V RV64GC. We also augment each runtime with multiple bounds checking strategies, in order to isolate the impact of the bounds checking mechanism from the rest of the code generation.

1.1 Motivation

Following [25], the first goal of WebAssembly (Wasm) is memory safety, i.e., preventing programs from compromising user data or system state, while the second goal is speed, or fast execution. Notoriously, memory safety and fast execution are conflicting goals, in fact, safety mechanisms usually negatively impact application's speed. However, as WebAssembly has now a myriad of use cases [31], and it is being widely adopted [20], achieving safety and speed is becoming increasingly important. Hence, it is crucial to identify the available memory safety mechanisms and their comparative performance. Note that herein we focus exclusively on software memory safety mechanisms, which are portable across different ISAs, at least when using the same operating system, if not different ones exposing the same API/system call.

At the same time, there are several different Wasm runtimes available to choose from, with a diverse set of designs and implementations, potentially each with a unique approach to bounds checking and code generation. Like many implementation details, the choice of bounds checking strategy can introduce significant overhead, and ultimately impact application execution time. While other issues such as register allocation and dealing with inlining are encountered by many language virtual machines, the bounds checking for all memory accesses is Wasm-specific. Despite diversity, to the best of our knowledge, most of the adopted Wasm runtimes implement bounds checking with `mprotect()` on POSIX-compliant operating systems (OS), even if several mechanisms have been made available recently and therefore need evaluation. In this paper, we will focus on POSIX OSes, specifically on Linux due to its wide adoption in data centers.

Is bounds checking the real culprit of the performance disparity with native execution? The work of Jangda et al. [12] is the first highlighting that safety checks affect performance – including stack overflow checks, indirect call checks, and reserved registers. To assess their claim we run two set of benchmarks on different Wasm runtimes and three different

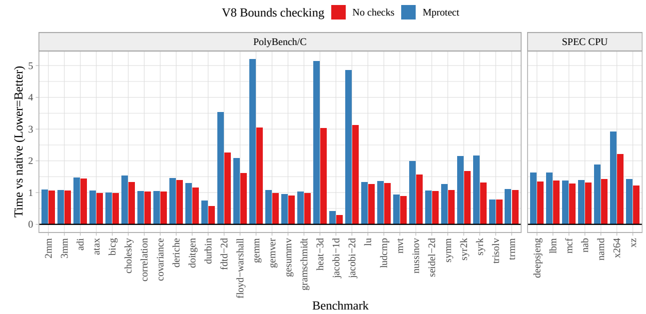


Figure 1: Cost of default bounds checking strategies in a WebAssembly runtime

ISAs (x86, Arm, and RISC-V), Section 3.3 includes details. Benchmarks have been run with and without bounds checking, Figure 1 shows the resulting execution times normalized on native execution (no bounds checks) for V8-TurboFan on x86_64. This shows that while about half of the benchmarks of PolyBench are not affected, bounds checking may introduce from 20% (Cholesky) to 220% (gemm) overhead in application execution. SPEC benchmarks show from 10% to 80% overhead. We obtained similar results on other ISAs and with different runtimes, recording overheads of up to 650% on Arm/Wasmtime, and a peak 50% overhead on RISC-V/V8. Thus, while not the only source of overhead, for many applications bounds checking negatively impacts execution time.

Driven by the above, this work is the first empirical evaluation that broadly compares different WebAssembly runtimes, specifically looking at the impact of different bounds checking strategies across diverse, modern and widely used ISAs, evaluating how well they achieve WebAssembly's principal goals.

1.2 Key Contributions

Our key contributions are:

- A comparison of the performance of prominent WebAssembly runtimes on three different ISAs.
- Isolating the impact of bounds checking mechanisms on that performance.
- Implementations of alternative bounds checking strategies for several WebAssembly runtimes.
- A reproducible benchmark suite that can be reused on new platforms – with automatic execution and data collection.
- Reproduction of past findings on WebAssembly performance, confirming and expanding upon the current knowledge.

1.3 Key Results

The key results of our investigation are:

- No difference in the relative costs of bounds checking methods across architectures: the cost of each method seems to be roughly the same on x86-64, Armv8 and RISC-V - the relative differences between architectures are within 2 percentage points of each other for the commonly used mechanisms.
- Using `mprotect()` on linux to dynamically adjust size of WebAssembly memory causes poor multithreaded scaling: lowering maximum CPU utilization by up to 25% on short-running benchmarks.
- WebAssembly is fast enough for server applications if an appropriate runtime is used, with WAVM achieving performance on par with native code for half of the tested benchmarks, and 8-20% average runtime overhead overall on x86_64.

2 BACKGROUND

2.1 WebAssembly

WebAssembly is a portable binary code format, but can also be thought of as a programming language. [25] It's a simple bytecode format for a stack-based virtual machine, that's designed to be an easy target for compilation of native programming languages such as C, C++, Go, and Rust; while itself being easy to compile to efficient native code. It grew out of a need for running safe, fast, and portable code on the Web, replacing previous attempts such as asm.js[10] and NaCl[6] with a clean-slate design. Despite its origins, WebAssembly can be used outside of the Web ecosystem. Supporting standards such as the WebAssembly System Interface (WASI) [32] were co-developed with WebAssembly and explicitly create a POSIX-like environment rather than a Web one.

In this way, WebAssembly can be compared to other programming language virtual machines, like the Java Virtual Machine (JVM) [23], which was originally advertised with the slogan "write once, run anywhere", and supports prominent programming languages such as Java, Kotlin, and Scala. Another similar example is the Common Language Runtime (CLR) [19] for languages such as C#, F#, and Visual Basic.

Despite the similarities, at a low level WebAssembly is significantly less complex in its design than the aforementioned runtimes. It currently does not have the capabilities for dynamic code generation and modification, and instead of managing a heap of objects for the programmer, it only provides a single "linear" memory buffer which can be grown in size akin to a dynamic array. Other elements of WebAssembly programs include: module(s) – an organization unit containing the definitions of other elements; functions – named containers for WebAssembly code, just like functions in most other programming languages; variables providing an infinite number of local registers within function scope; function tables – used as an security mechanism for indirect branches to avoid exposing the host's instruction pointer directly; exports – allowing providing named references to various other elements of a module for other modules or the runtime host to refer to.

There are only four value types in the language: 32 and 64-bit variants of integers and floating point numbers. Any other type has to be compiled down to instructions making use of these four primitive types before generating the final WebAssembly module.

2.2 Language Virtual Machines

Language virtual machines (VMs), also known as language runtimes, are programs that execute bytecode, such as WebAssembly. Language VMs are what allows the platform-independence and portability benefits of bytecode representations, separating the platform-specific VM implementation from the platform-agnostic bytecode specification.

There are multiple approaches to VM implementation, ranging from relatively slow, but simple interpreters, to fast, but complex Just-in-Time (JIT) and Ahead-of-Time (AOT) compiler-enabled runtimes.

Interpreters, such as Wasm3 benchmarked in this paper, follow a fetch/execute loop – they read the bytecode, and execute different native code depending on the instructions. Various implementation techniques have emerged for interpreters, the currently most prevalent one for fast execution is threaded interpreters[1] which dispatch the next instruction using a jump table with a separate indirect branch in each instruction implementation, allowing independent branch prediction of those targets for each instruction type.

Just-in-Time compilers generate native machine code during the program's execution, often inserting instrumentation and recompiling the same functions using the collected data for better performance. The V8 runtime evaluated in this paper is one of the classic examples of a JIT runtime for JavaScript and WebAssembly.

Ahead-of-Time compilers, often just called compilers, convert bytecode into machine code all at once, before the program starts executing. WAVM and Wasmtime, while using JIT frameworks to load the compiled code at runtime into the host, are in fact AOT compilers, as they never adjust the generated code after it has been compiled and loaded into the host process.

2.3 Bounds Checking Techniques

WebAssembly requires checking that each memory load and store instruction points at an address within the bounds of the active linear memory. This is similar to inserting checks for indices laying inside the bounds of an array at each array indexing operations, but here it's done for all memory accesses.

The naïve approach to ensuring instructions do not access addresses outside the confines of the virtual memory region is to simply perform a conditional branch on the address compared to the memory limit, every time a memory access occurs. However, this approach can significantly affect performance. Load and store instructions on average form 40% of x86_64 programs [9], inserting a branch instruction before every single one adds up to a significant cost, even if some proportion of them can be eliminated by an optimization pass. Therefore high-performance runtimes use operating system mechanisms to manage virtual memory themselves, to catch out-of-bounds accesses when they happen.

This is done by over-allocating a large virtual memory region, and only populating the valid memory range with read-write-allowed pages, while the rest of the region generates a CPU exception on illegal accesses, which can be subsequently caught and handled by the runtime. Because current WebAssembly limits the memory instructions to take a 32-bit integer as a base, and a 32-bit integer as an offset, the total addressable space is 8 GiB, so on 64-bit machines with virtual memory the entire 8 GiB region can be preallocated. The generated machine code mathematically cannot access the area outside of this allocation because two 32-bit numbers are added.

The downside of this approach is that managing large allocations like this can be costly in the operating system, especially on less powerful hardware. In Linux, changing the size of such an allocation requires adjusting process VMAs which are binary tree structures requiring taking an exclusive lock for modification, which can have negative scaling impact for multithreaded applications.

Various hardware-accelerated bounds checking methods have also been proposed and implemented for array accesses, some of which could be reused for WebAssembly. Some Intel processors had the MPX extension which provide bounds-checked pointer access instructions. They have been shown to have a high overhead (50% on average)[21] and therefore Intel discontinued this extension, removing it from the x86 processor manuals in 2019. An upcoming, promising approach is CHERI[34], providing capability-checked memory accesses to multiple CPU architectures with a single mechanism, however it is still in relatively early phases of development with very limited hardware availability, therefore we did not evaluate it in this work.

2.3.1 *Userspace Page Fault Handling.*

Adjusting virtual memory protection in Linux is costly in multithreaded applications due to inter-processor TLB shootdowns and locking on the process's VMA structure in the mprotect implementation[13]. One alternative mechanism for managing virtual memory is Userfaultfd[14], which lets applications reserve a region of virtual memory and handle page faults on that region in userspace, with VMAs remaining untouched and no kernel-side locking.

The page fault handler can choose to populate the faulted page, or a larger range of pages, with zero-filled pages, content copied from another range of pages, or not populate the pages at all and instead raise an exception if it determines the access is illegal. The handler can operate either as a thread polling the userfault file descriptor, being notified of events, or as a signal handler for SIGBUS signals which the kernel sends to the pagefaulting thread. The SIGBUS handler avoids back-and-forth context switches, because it gets executed on the same thread that caused the page fault, and can thus achieve lower latency.[35] This is the method we decided to use in our Userfaultfd-accelerated bounds checking implementation.

3 BENCHMARK DESIGN

In this section, we present how we analyze and compare the performance of different bounds checking techniques on selected WebAssembly runtimes, on three different ISAs.

3.1 Bounds Checking Mechanisms

We consider the following bounds checking mechanisms:

1. **none:** The entire possible memory space (8 GiB) is read-write mapped. No bounds checks are performed during execution.
2. **clamp:** All memory accesses pass through a conditional selection operator. If the given pointer is out of bounds, the memory end pointer is used instead.
3. **trap:** Another conditional selection. If an access is out of bounds, a trap to the host is generated by jumping to an invalid instruction (e.g. `ud2`), which generates a `SIGILL` to be caught by the runtime.
4. **mprotect:** The entire memory space is preallocated with no permissions. Illegal accesses during runtime generate a `SIGSEGV`, and then the handler invokes *mprotect* to modify the process' virtual memory area (VMA) to grant the necessary permissions. As implemented in Linux, this requires acquiring a lock on the VMA of the process [13], since all threads within a process share one VMA[1].
5. **uffd:** Similar to **mprotect**, but instead, the entire memory space is lazily read-write mapped and registered with the *userfaultfd* feature, so that the bounds checking can

[1]Technically, Linux treats everything as tasks, but conceptually, a process is a group of threads that share a thread group identifier (TGID) and a set of resources, and a thread is the unit of work that is scheduled.

be handled in user space. Any attempt to write a missing page generates a `SIGBUS`[2], which prompts either an `ioctl` call to copy or zero the page, or a new signal can be sent to the runtime. A lock is only acquired for the page in question rather than the entire VMA, so requests from multiple threads can be handled simultaneously (as long as they reference distinct pages).

3.2 Runtimes

We consider a total of six execution environments. Two are native environments, where the benchmarks are compiled to machine code using either GCC 11 or Clang 13, and then executed with no WebAssembly-style bounds checking to give baseline metrics. The other two are WebAssembly runtimes, where the benchmarks are first compiled to WebAssembly using Clang 13 (target wasm32-wasi) before being executed by the runtime, which just-in-time (JIT) compiles the code to native code. The WebAssembly runtimes are able to provide isolation, so the WebAssembly benchmark runner spawns one instance of the runtime for each benchmark instance, all contained within the same process in isolated threads, whereas the native benchmark runner spawns one process for each benchmark instance. The four WebAssembly runtimes are:

1. **WAVM** [26]: A virtual machine that uses the LLVM [15] compiler infrastructure (specifically the MCJIT framework [16]) to compile WebAssembly to machine code ahead of the time of execution. We modified 140 lines of code to add alternative bounds checking methods.
2. **Wasmtime** [3]: A standalone runtime that uses the Cranelift [2] code generator to compile WebAssembly to machine code. We modified 500 lines of code.
3. **Wasm3** [17]: A standalone threaded interpreter for WebAssembly bytecode. We modified 80 lines of code to integrate it with our harness.
4. **V8 TurboFan** [8] with the Node.js WASI implementation [22]: A standalone JavaScript and WebAssembly runtime, used as a part of the Chromium[7] web browser – focused on striking a balance between speed of compilation and speed of the executed code for Web applications. We modified 400 lines of code.

The WAVM, Wasmtime and V8 runtimes both use *mprotect* to implement bounds checking by default. We augmented those three runtimes with implementations for *none*, *clamp*, *trap*, and *uffd* strategies. Wasm3 effectively uses an equivalent of the *trap* mechanism, due to the way the memory instruction interpreter code is written, and because it does not generate compiled code. Since this runtime is significantly slower at executing WebAssembly we did not change this mechanism. All runtimes are also designed to be standalone; that is, able to run outside of a web browser environment. They do this by targeting the WebAssembly System Interface (WASI) [32], rather than any specific browser API. This removes the dependency on JavaScript and increases portability.

3.3 Benchmarks

We chose to use the Polybench/C benchmarks [24] in the MEDIUM configuration for evaluation, to allow us to compare with earlier results [12] [25]. We also decided to use the SPEC CPU 2017 Rate benchmark suite [27] in order to provide a more comprehensive evaluation. Due to the very long running times of the SPEC benchmarks, and very high number of tested configurations, they were run in the Train configuration rather than Ref, we estimate based on trial runs that running all of them in Ref mode in our configurations would take about a month of CPU time on each machine, possibly more in the RISC-V case if there was enough memory available on our platform to run SPEC there.

However, some of these benchmarks rely on libc and C++ functionality (e.g. signal handling, non-local exits, exceptions), and the WASI libc [33] implementation is still under development, so we were only able to compile a subset[3] of the benchmarks to WebAssembly for evaluation. As the WASI and WebAssembly standards evolves, and a Fortran to WebAssembly compiler is developed, we hope the rest of the SPEC CPU suite will run under WASI.

3.4 Hardware

The runtimes (3.2) were evaluated on the benchmarks (3.3) on three hardware configurations with different architectures:

1. **x86_64**: Intel Xeon Gold 6230R, with 16 hardware threads enabled, 768 GiB of system memory.
2. **AArch64**: Cavium ThunderX2 CN9980 v2.2 configured to have 16 hardware threads, 256 GiB of system memory.
3. **RISC-V**: Nezha D1 1GB development board, with the XuanTie C906 CPU, single core and hardware thread.

Each system was running the Ubuntu 22.04 LTS operating system, with recent kernel versions (5.16, 5.13 and 5.16 respectively). We disabled CPU vulnerability mitigations with the `mitigations=off` kernel command-line argument to better represent the architectural differences between CPUs, excluding the impact of OS-based mitigations of problems that have been and will be addressed in newer CPU models [28]. The CPU governors were set to performance mode where possible, to prefer higher operating frequency over power saving.

[2] *userfaultfd* can also use a poll-based method that listens for page faults, but this has a higher latency than the signal-based method that we use.

[3] Subset of SPEC CPU 2017 Rate suite used: 505.mcf_r, 508.namd_r, 519.lbm_r, 525.lbm_r, 531.deepsjeng_r, 544.nab_r and 557.xz_r

On each system, we ran the benchmarks with 1, 4 and 16 copies of the Rate benchmarks running pinned on separate logical cores, following how the official SPEC CPU Rate suite runner works in multithreaded configurations. The RISC-V system was only tested with the PolyBenchC suite, and only in a single-threaded mode, because the 1 GiB physical memory available made it impossible to run the SPEC suite, and the CPU only has one physical core with no simultaneous multi-threading capabilities. The WAVM and Wasmtime runtimes also do not have RISC-V backends to test – WAVM when forced to generate RISC-V code via LLVM was leading to crashes in the MCJIT framework, while Wasmtime's Cranelift backend does not have a RISC-V target implemented, leaving the RISC-V platform with the Native, Wasm3 and V8 runtimes.

3.5 Benchmarking harness

In order to make consistent measurements between all of the runtimes and native code, we implemented a custom benchmarking harness in 2000 lines of C++ code that interacts directly with the WebAssembly runtimes via their C and C++ APIs. The harness first ensures the wasm code is fully loaded into the runtime and compiled where appropriate, then a clone of that module is executed in a timed loop in each worker thread that is pinned to a CPU core. Only the module execution is timed, while the setup and tear-down between loop iterations are not a part of the reported time.

There is a warm-up phase to ensure all physical CPU threads are equally busy before the timed execution runs happen, and once each thread finished its timed workload, it continues to run the WebAssembly code for a few more iterations, until all the threads finish their measured runs, to ensure the final measurements are not affected by other CPU cores becoming less busy.

For Native code, the same overall procedure is followed, except instead of simply calling the JITted code a new process is spawned with a `vfork()` and `fexecve()` (on a pre-opened executable file descriptor) syscall combination, because loading multiple copies of native Linux executables into the same process is not achievable via any standard system interfaces. This has the downside of including the process spawning and tear-down overhead in the native code measurements, but we measured it to be on the order of a hundred microseconds once the benchmarks warm up, hence not affecting the results significantly.

We intend to release our benchmarking harness, patches to the WebAssembly runtimes and automation scripts under an open-source license, excluding the SPEC benchmarks which are protected by copyright.

4 EVALUATION

In the following section, we discuss the performance of each runtime and bounds checking mechanism configuration introduced in Section 3.1, when executing the benchmarks listed in Section 3.3. We collect a variety of execution statistics, using the native Clang and GCC benchmark runs as baselines.

4.1 Execution Time Statistics

We collected detailed execution time statistics for each benchmark in each configuration, with a minimum of ten runs of SPEC benchmarks, and a minimum of hundreds of Polybench/C runs on each CPU thread, excluding the warm-up and cool-down runs.

A comparison of the results for each single-threaded configuration, by taking the geometric mean of the ratios [4] of execution times to the Native Clang execution time for each benchmark, is shown in figures 2a, 2b, 2c. SPEC and Polybench/C (PBC) results are separated.

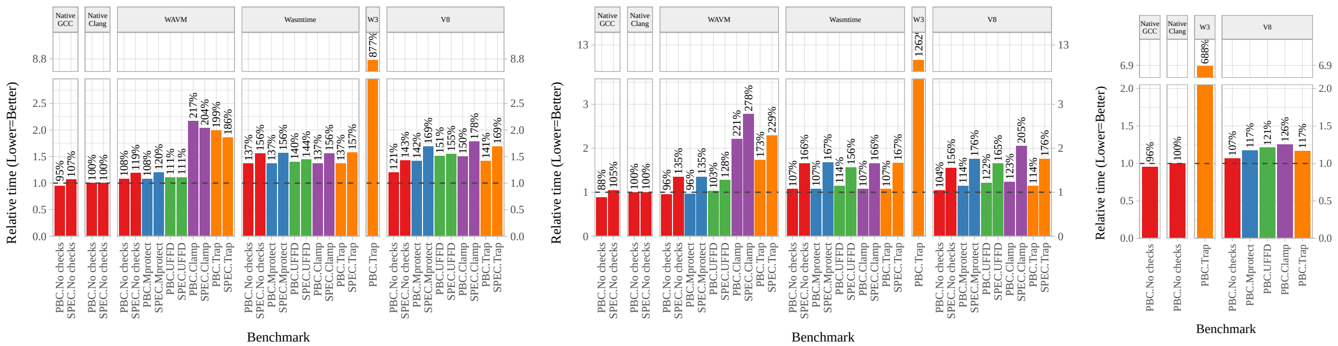
From these results we can see that the fastest WebAssembly runtime among the evaluated ones is WAVM, followed by Wasmtime, and then V8 very closely. No bounds checks is, as we would expect, the fastest, but the `mprotect()` and UFFD strategies have very little overhead, on the order of 1-2 percentage points, except for the 10 points difference for the V8 runtime.

Software checks are significantly slower in a number of configurations, most notably in WAVM, with clamping addresses unconditionally behaving worse than generating conditional traps.

Based on these results, we can say that WebAssembly runtimes with an advanced backend focusing on performance (such as WAVM backed by LLVM), can be used to sandbox code running in server environments with only a minor overhead. WAVM was able to generate better code with its LLVM frontend for some Polybench/C benchmarks than native LLVM, performing closer to a native GCC compiler which happens to generate faster code for this particular suite.

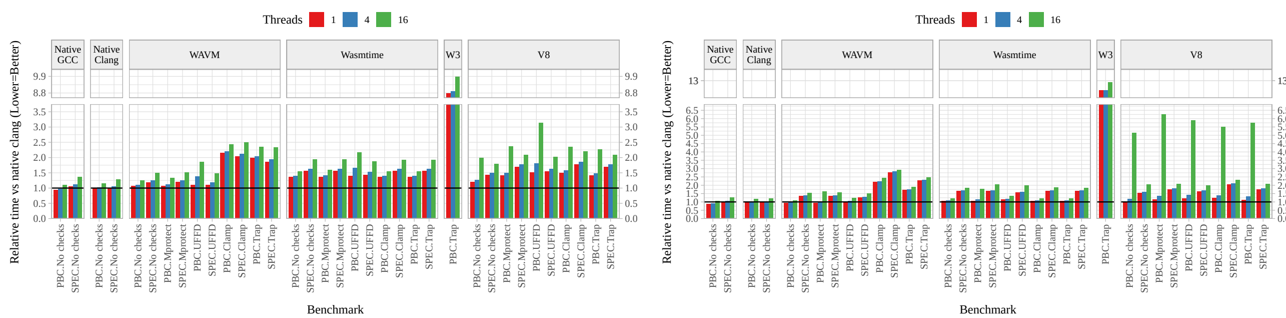
We investigate the causes of these differences in the following sections by looking at various system and CPU performance counters. This data is presented for the x86_64 and Armv8 architectures, because running the monitoring tools on the RISC-V board was causing significant changes to the benchmark results due to the slow, single-threaded CPU performance.

4.1.1 Scaling with thread counts. One interesting aspect of the various runtimes and bounds checking methods is to see how running multiple isolates in parallel on separate threads affects the overall performance. We investigated this by running multiple instances of each benchmark on worker threads pinned to chosen CPU cores to reduce the impact



(a) x86-64 **(b) Armv8** **(c) RISC-V**

Figure 2: Geometric mean of per-benchmark execution time medians divided by the native Clang time medians



(a) x86-64 **(b) Armv8**

Figure 3: Performance scaling with increased number of threads

of scheduling decisions about CPU migrations. The performance scaling at 1, 4 and 16 threads (all active CPU cores) is shown in Figures 3a and 3b.

We can see that in most cases running multiple parallel benchmark instances in separate threads does not affect the performance in a major way, the small slowdowns are easily explained by the usual causes, confirmed by monitoring the benchmarked systems during benchmarking: different frequency scaling characteristics when more CPU cores are busy in modern CPUs, increased memory bus contention and mutual exclusion when executing certain syscalls such as write operations.

One major difference between the runtimes visible is that V8 struggles when 16 worker threads are created, this is because V8 uses worker threads for some of its internal operations such as JIT compilation, and periodic garbage collection locks other worker threads from performing work. When all of the physical cores are already occupied by the benchmarks, additional work requires context switches, which are visible in Figure 5b – scaling the number of threads for V8 increases the measured switches by an order of magnitude.

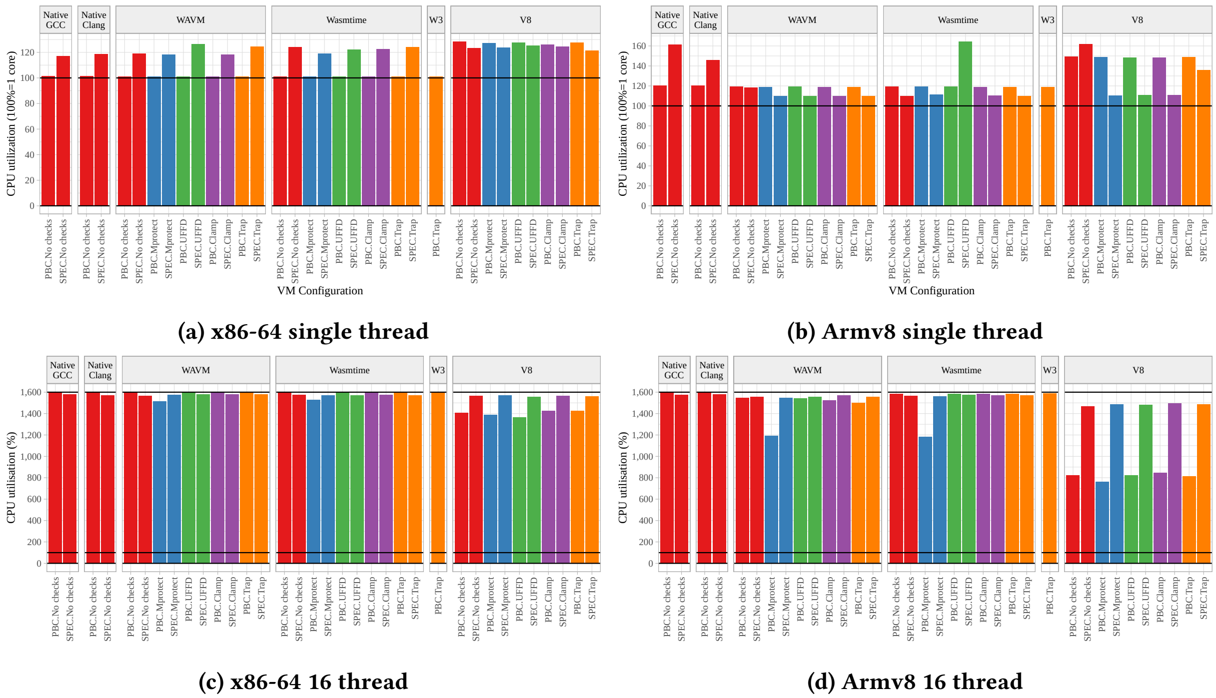
Another major difference, also visible in the context switch graphs, is the poor scaling of `mprotect()`-based memory protection to multiple threads. Especially visible in the PolybenchC benchmarks, which cause frequent allocation and deallocation of memory as they execute in a short span of time, this stresses the virtual memory management subsystem in the Linux kernel for the host process, causing excessive locking and pausing of thread execution.

4.2 CPU Statistics

4.2.1 CPU Utilisation. We define CPU utilisation as the total number of milliseconds, averaged across all CPU(s), that the Linux kernel reports in `/proc/stat`[4] spending in either user or kernel mode, offset by the total number of milliseconds spent idle, i.e.

$$\frac{us + sys + hi + si}{us + sys + hi + si + id} \tag{1}$$

[4]**us** represents user mode time including "nice" time, **sys** represents kernel mode time, **hi** represents time servicing interrupts, **si** represents time servicing softirqs and **id** represents idle time.



(a) x86-64 single thread

(b) Armv8 single thread

(c) x86-64 16 thread

(d) Armv8 16 thread

Figure 4: Average CPU load during benchmark execution

We rescale this quantity so that 100% utilization is a full utilization of one cpu core, so 1600% utilization is all 16 cores occupied.

In Figures 4a and 4b we can see that in the single-threaded configuration all of the runtimes are able to saturate a full CPU core, with the Arm machine having larger off-main-thread activity than x86. The V8 runtime uses extra worker threads for some tasks, mostly IO and background JIT compilation, in its implementation, therefore the utilization for it is larger than for the other runtimes.

In the case of the 16-threaded workload, presented in figures 4c and 4d, we can see that all runtimes except V8 are able to achieve full CPU saturation. The lower saturation in V8 is due to the periodically running JavaScript garbage collector which pauses the execution of other threads.

One big difference visible here between the bounds checking strategies is that `mprotect()`-based protection does not saturate the CPU like other mechanisms, as earlier discusses in Section 4.1.1. This is due to a mutex in the Linux kernel protecting the process' virtual memory areas tree [13], when WebAssembly resizes its memory to allocate or run the next iteration that mutex is acquired for significant periods of time that we confirmed by capturing stack traces of threads in a waiting state via bpftools. Software bounds checking requires less virtual memory manipulation, hence the effect there is not visible. The UFFD mechanism in the kernel does not acquire an exclusive lock over that structure, so the userspace code is able to use lockfree structures to manage its memory – in our implementation we use an atomic integer variable controlling the size of each memory arena, and a hazard pointer [18]-style implementation for adding and removing memory arenas, avoiding the need for locks most of the time.

Another option is to limit the number of executor thread per process, and instead build a multiprocess runtime. The locking effect was significantly more visible in short-running benchmarks, therefore we make a recommendation that for short-lived WebAssembly tasks, such as for certain classes of serverless applications, using userspace-managed pagefault handlers can be preferential to `mprotect()`-based handlers, unless the Linux kernel memory management switches to more fine-grained locking or lockfree data structures.

4.2.2 Context Switches. We also measure the total number of context switches per second, averaged across all CPU(s), for each configuration. The data can be seen in Figures 5a and 5b.

There is no significant impact of the bounds checking mechanism on the context switch rate, except for the previously discussed `mprotect()` scaling issue. When scaling V8 to multiple threads, care has to be taken to not saturate the CPU, as spawning 16 worker threads on a 16-core CPU negatively impacts performance because of the additional work the runtime does in its own worker threads.

4.3 Memory usage

We present the memory usage of the different runtimes in all of the bounds checking configurations in Figures 6a and 6b, as measured by the difference between total and "available" memory in `/proc/meminfo`.

There is no significant variance in memory usage between the different runtimes or bounds checking methods visible. One observable difference is the increased memory usage of the PolybenchC benchmark suite on the x86_64 architecture compared to the Armv8 architecture. This is due to the Linux kernel using huge pages to serve the WebAsssembly reservations, removing them from the pool of readily available memory, but that memory is reclaimable by splitting them into smaller pages. The transparent huge pages mechanism on the x86_64 ISA uses pages of up to 1 GiB size, while on Armv8 the limit is 2 MiB, leading to more fine-grained memory usage reporting.

4.4 Replicating previous results

In 2022, Titzer [29] measured Wasm3 to be roughly 10× slower than V8-TurboFan on the PolybenchC benchmark, which agrees with our results of 6x-11x difference depending on the CPU architecture on the same suite.

Rossberg et al. [25] in 2017 measured PolybenchC execution time on V8, showing that "WebAssembly is competitive with native code, with seven benchmarks within 10% of native and nearly all of them within 2× of native", with the measured performance for each benchmark closely matching our measurement in figure 1.

Jangda et al. [12] in 2019 reported a 1.55× geomean slowdown of SPEC on V8 compared to Native, we measured 1.69× slowdown on x86_64 and a 1.76x slowdown on Armv8. They were able to run a bigger subset of SPEC thanks to developing a custom POSIX layer that WebAssembly interacted with via JavaScript. Most of the runtimes we evaluated in this paper do not support JavaScript, therefore we could not use Browsix to run the same subset of benchmarks.

Comparing against these previous works, we can see that while Web-focused WebAssembly runtimes and interpreters have made very slow progress on the performance front since 2017, more performance-oriented runtimes have emerged recently approaching near-native performance levels for a wider set of programs.

5 RELATED WORK

Previous work did compare different WASM runtimes already, but none focused on the overhead added by bounds checking. Also, no previous work shows how the cost of bounds checking varies among different CPU ISAs. In the following points, we briefly summarize previous work:

With the introduction of WASM in their 2017 paper Rossberg et al. [25], the authors compared the WASM implementation in V8 and in SpiderMonkey on x86 only, without breaking down bounds checking overheads. Jangda et al. [12] introduces additional benchmarks to Rossberg et al. [25] so that also SPEC can be used to benchmark WebAssembly in addition to PolyBench, using a JavaScript POSIX emulation shim. They are the first to highlight that WASM is slower than what has been reported before, and one of the issue are the safety checks. However, their work do not explore why that is the case, is based on x86 only, and it does not introduce new bounds checking mechanisms enabled by latest advantages in OSes – which is the core contribution of this paper.

Yan et al. [36] presents a performance evaluation of WASM on a large collection of benchmarks, being the only work considering WASM execution on x86 (desktop) and ARM (mobile). At the same time, their work doesn't explain what is the cost of bounds checking, nor introduces anything new in WASM runtimes.

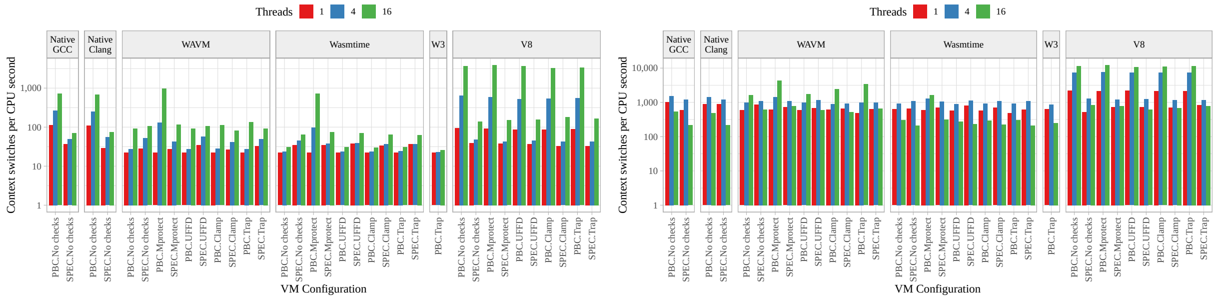
Hilbig et al. [11] also introduces a large collection of WASM benchmarks, WasmBench – the largest, which focuses on x86 only, and does not include considerations on bounds checking – while highlighting that memory errors can be propagated into WASM, further justifying our work.

Finally, Titzer [29] compares several engine runtimes (WAMR, WASM3, V8-liftoff and V8-turbofan, Spidermoneky, and JSC) on an Intel Core-i7, using PolyBenchC-4.2.1, showing execution time, translation time and space statistics. While we reported similar metrics, Titzer did not breakdown the cost of bounds checking, nor introduced any new bounds checking method on any engine runtime on different ISAs.

6 DISCUSSION

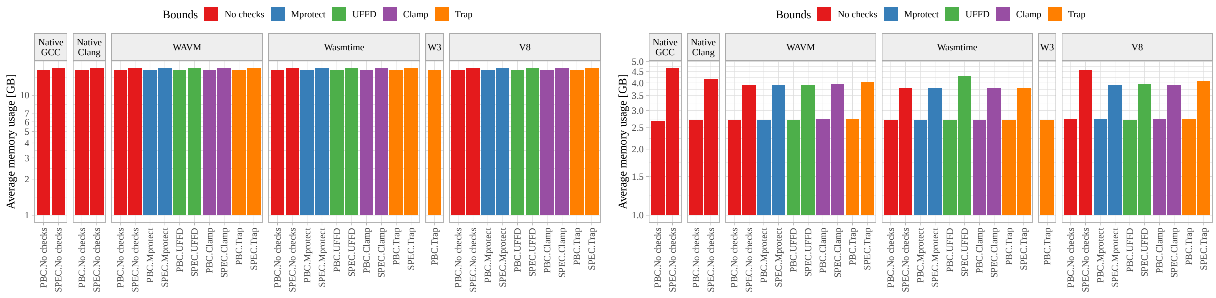
Our evaluation of four different WebAssembly runtimes against native GCC and Clang-compiled code on two benchmark suites shows that there is a variety of available runtimes, each striking a different balance between complexity, size, and runtime performance. WebAssembly has grown from its initial Web-focused applications to become a generic sandbox platform for server [30] and client [5] applications.

WebAssembly brings its own unique security mechanisms to the table, one of the major ones being bounds-checked memory accesses. While other languages often check array index bounds, WebAssembly limits all memory instructions to access a single, resizable block of memory, checking whether those accesses are within the current bounds on each load and store. we implemented alternative approaches to bounds checking into the WebAssembly runtimes that use



(a) x86_64 **(b) ARMv8**

Figure 5: Total number of context switches per CPU second induced by the tested runtimes



(a) x86_64 **(b) ARMv8**

Figure 6: Average memory usage by the tested runtimes

a compiler, and based on this quantified the exact impacts of pure software and virtual memory-accelerated bounds checking against disabled bounds checks. The exact overheads vary across architectures and benchmarks, but overall pure software checks cause significantly higher overhead compared to allocating large regions of virtual memory and using page fault handlers to catch illegal accesses.

7 CONCLUSION

We show that runtimes such as WAVM and Wasmtime are able to achieve performance on average within 20% of the native performance on x86_64 platforms, 35% for Armv8. On RISC-V V8 can achieve a 17% overhead over native code for simple numeric kernels. For simple numerical kernels, we have shown that there is no significant difference in the WebAssembly performance compared to native code across the three tested architectures: x86_64, Armv8 and RISC-V RV64GC.

In case of multithreaded scaling of the tested runtimes, which might for example be used to quickly scale up serverless instances for a single function without the overhead of spawning new processes, the default approach taken by WAVM, Wasmtime and V8 of using the `mprotect()` syscall to resize memory can cause excessive locking in the Linux kernel. This can be mitigated by using simpler, lockfree data structures for managing page permissions, which we were able to implement using Linux's recent userfaultfd mechanism for handling page faults in userspace.

We share our results and the entire set of tools and scripts under an open source license, except for the SPEC CPU benchmarks for which we only distribute the small patches required to compile them for WebAssembly due to its licensing terms. We hope that other researchers can use these tools in the future to replicate our results, and monitor the progress that WebAssembly runtimes make as this technology evolves.

REFERENCES

[1] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (1973), 370–372. https://doi.org/10.1145/362248.362270
[2] Bytecode Alliance. 2022. Cranelift. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/README.md. [Online; accessed 01-Mar-2022].
[3] Bytecode Alliance. 2022. Wasmtime. https://github.com/bytecodealliance/wasmtime. [Online; accessed 01-Mar-2022].
[4] Philip J. Fleming and John J. Wallace. 1986. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Commun. ACM* 29, 3 (1986), 218–221. https://doi.org/10.1145/5666.5673

[5] Nathan Froyd. 2020. Securing Firefox with WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/. [Online; accessed 01-Mar-2022].

[6] Google Corporation. 2020. Native Client developer documentation. https://developer.chrome.com/docs/native-client/.

[7] Google Corporation. 2022. The Chromium project. https://www.chromium.org/Home/. [Online; accessed 12-Jul-2022].

[8] Google Corporation. 2022. TurboFan V8. https://v8.dev/docs/turbofan. [Online; accessed 12-Jul-2022].

[9] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach, 5th Edition.* Morgan Kaufmann, -.

[10] David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js Specification. http://asmjs.org/spec/latest/.

[11] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2696–2708. https://doi.org/10.1145/3442381.3450138

[12] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. *login Usenix Mag.* 44, 3 (2019), 12–16. https://www.usenix.org/publications/login/fall2019/jangda

[13] Linux Foundation. 2022. Linux implementation of mprotect. https://github.com/torvalds/linux/blob/v5.19-rc4/mm/mprotect.c#L644. [Online; accessed 28-Jun-2022].

[14] Linux Foundation. 2022. Linux Userfaultfd documentation. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html. [Online; accessed 28-Jun-2022].

[15] LLVM Foundation. 2022. LLVM. https://llvm.org/. [Online; accessed 01-Mar-2022].

[16] LLVM Foundation. 2022. LLVM MCJIT. https://llvm.org/docs/MCJITDesignAndImplementation.html. [Online; accessed 01-Mar-2022].

[17] Steven Massey and Volodymyr Shymanskyy. 2022. WASM3. https://github.com/wasm3/wasm3. [Online; accessed 12-Jul-2022].

[18] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distributed Syst.* 15, 6 (2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[19] Microsoft Corporation. 2022. Common Language Runtime (CLR) overview. https://docs.microsoft.com/en-us/dotnet/standard/clr. [Online; accessed 12-Jul-2022].

[20] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings (Lecture Notes in Computer Science)*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.), Vol. 11543. Springer, Gothenburg, Sweden, 23–42. https://doi.org/10.1007/978-3-030-22038-9_2

[21] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 1–30.

[22] OpenJS Foundation. 2022. Node.js WASI support. https://nodejs.org/api/wasi.html. [Online; accessed 12-Jul-2022].

[23] Oracle Corporation. 2022. The Java® Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se18/html/index.html. [Online; accessed 12-Jul-2022].

[24] Louis-Noel Pouchet and Tomofumi Yuki. 2015. Polybench/C. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/. [Online; accessed 01-Mar-2022].

[25] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. 2018. Bringing the web up to speed with WebAssembly. *Commun. ACM* 61, 12 (2018), 107–115. https://doi.org/10.1145/3282510

[26] Andrew Scheidecker. 2022. WAVM. https://github.com/WAVM/WAVM. [Online; accessed 01-Mar-2022].

[27] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/Docs/overview.html. [Online; accessed 01-Mar-2022].

[28] Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. 2022. Mitigating Speculative Execution Attacks via Context-Sensitive Fencing. *IEEE Des. Test* 39, 4 (2022), 49–57. https://doi.org/10.1109/MDAT.2022.3152633

[29] Ben L. Titzer. 2022. A fast in-place interpreter for WebAssembly. https://doi.org/10.48550/ARXIV.2205.01183

[30] Kenton Varda. 2018. WebAssembly on Cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

[31] W3C. 2022. Use Cases - WebAssembly. https://webassembly.org/docs/use-cases/. [Online; accessed 12-Jul-2022].

[32] W3C. 2022. WASI. https://wasi.dev/. [Online; accessed 12-Jul-2022].

[33] W3C. 2022. WASI Libc. https://github.com/WebAssembly/wasi-libc. [Online; accessed 01-Mar-2022].

[34] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 457–468. https://doi.org/10.1109/ISCA.2014.6853201

[35] Peter Xu. 2020. Userfaultfd-wp Latency Measurements. https://xzpeter.org/userfaultfd-wp-latency-measurements/.

[36] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the Performance of Webassembly Applications. In *Proceedings of the 21st ACM Internet Measurement Conference* (Virtual Event) *(IMC '21)*. Association for Computing Machinery, New York, NY, USA, 533–549. https://doi.org/10.1145/3487552.3487827