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

Raven Szewczyk[1], Kimberley Stonehouse[1], Antonio Barbalace[1], and Tom Spink[2]

[1]*University of Edinburgh, UK*
[2]*University of St Andrews, UK*

Abstract

WebAssembly is gaining more and more popularity, finding applications beyond the Web browser for which it was initially designed. However, its performance, which developers intended to be comparable with native, has not been extensively studied to identify overheads and pinpoint their causes. This paper identifies that WebAssembly's bounds-checked memory access safety mechanism may introduce up to a 650% overhead, and requires further tuning.

Based on that, we extend four popular WebAssembly runtimes with modern bounds checking mechanisms and compare the performance of each with native compiled code. The runtimes are evaluated on three different instruction set architectures: x86-64, Armv8, and RISC-V RV64GC.

We show that, for simple numerical kernels from PolyBench/C, there are no significant differences in the bounds checking performance overheads across different instruction set architectures. With the default bounds checking mechanism, performance-oriented runtimes are able to achieve execution times within 20% of native on x86-64 platforms, within 35% on Armv8 platforms, and within 17% on RISC-V.

We also show that, when scaling the tested runtimes to multiple threads, the default bounds checking approach taken by WAVM, Wasmtime, and V8 of using the mprotect syscall to resize memory can cause excessive locking in the Linux kernel. Such scaling might be used to quickly start up serverless instances for a single function without the overhead of spawning new processes. We present an alternative userfaultfd-based solution to mitigate this issue.

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

1. Introduction

Language virtual machines are incredibly popular, enabling programs to be written once and executed on a variety of CPUs of different Instruction Set Architectures (ISAs) without the need for recompilation, and often providing enhanced security guarantees relative to native execution. WebAssembly [9] is a language that is steadily gaining traction. 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 [32]. Despite the name, WebAssembly applications are not confined to the web. The WebAssembly System Interface (WASI) [34] provides a uniform way for WebAssembly code to communicate with the underlying system (e.g., the browser or the operating system), thus extending the benefits of WebAssembly far beyond the web.

WebAssembly models a simple virtual stack machine. However, it is distinct from other languages that use virtual stack machines, in that it is an assembly-like language with unmanaged memory access [26]. Rather than having memory management features like garbage collection or a managed heap, operations happen 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 targets can only be valid WebAssembly functions. This suggests that the bounds checking mechanism for validating linear memory accesses (and, less frequently, function table accesses) is likely a performance overhead that is largely unique to 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 the bounds checking overhead that is specific to WebAssembly. We examine several 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: 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

As detailed by Rossberg et al. [26], the first goal of WebAssembly (Wasm) is memory safety, that is, preventing programs from compromising user data or system state, and the second goal is speed. With WebAssembly now having a myriad of use cases [33] and becoming widely adopted [21],

978-1-6654-8798-6/22/$31.00 ©2022 IEEE
DOI 10.1109/IISWC55918.2022.00030

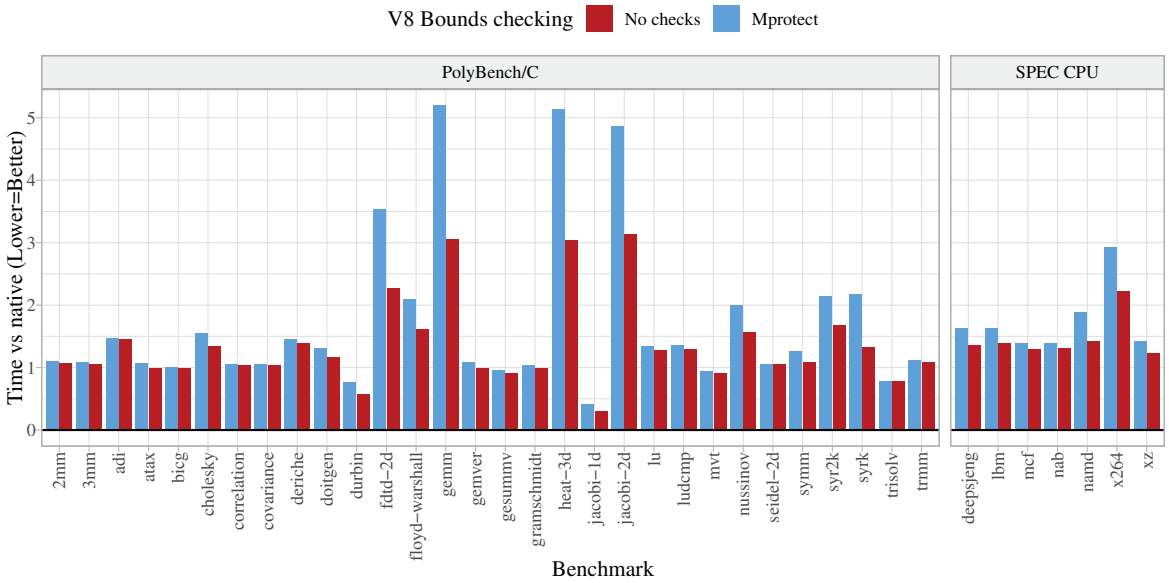


Figure 1: Cost of default bounds checking strategies in a WebAssembly runtime

achieving these goals is increasingly important. However, memory safety and fast execution are notoriously conflicting objectives, because safety mechanisms usually introduce additional code that negatively impacts performance. Therefore, it is crucial to identify the available memory safety mechanisms and their comparative performance.

Herein, we focus exclusively on software memory safety mechanisms, which are portable across different ISAs, at least when using the same operating system or different operating systems exposing the same system calls. At the same time, there are several different Wasm runtimes available to choose from, with a diverse set of designs and implementations and each with a unique approach to code generation and bounds checking. Like many implementation details, the choice of bounds checking strategy can introduce significant overhead and ultimately impact application execution time. Whilst other issues such as register allocation and dealing with inlining can also impact performance, these are encountered by many language virtual machines, whereas the memory bounds checking is specific to Wasm. In this paper, we focus on POSIX-compliant operating systems due to the wide adoption of Linux in data centers. To the best of our knowledge, despite their diversity, most Wasm runtimes implement bounds checking with `mprotect()` on POSIX operating systems. Because new alternative mechanisms have recently been made available at the operating system level [15], we believe an evaluation of the different mechanisms is necessary.

Is bounds checking the real culprit of the performance disparity with native execution? The work of Jangda et al. [13] is the first highlighting that Wasm safety checks, including stack overflow checks, indirect call checks, and reserved registers, affect performance. To assess their claim, we ran two sets of benchmarks in four Wasm runtimes on three different ISAs (detailed further in subsection 3.3), both with and without bounds checking. Figure 1 shows the resulting execution times for V8-TurboFan on x86-64, normalized on native execution (with no bounds checking). The results show that whilst around half of the PolyBench/C suite is unaffected, bounds checking may introduce between 20% (Cholesky) and 220% (gemm) overhead in application execution time for the other half. For the SPEC benchmarks, the overhead is between 10% and 80%. We obtained similar results with different runtimes and ISAs, recording overheads of up to 650% on Arm/Wasmtime, and a peak 50% overhead on RISC-V/V8. Thus, for many applications – whilst not the only source of overhead – 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 and evaluating how well they achieve WebAssembly's primary goals.

1.2. Key Contributions

Our key contributions are:

- An extensive comparison of the performance of prominent WebAssembly runtimes on three different ISAs.
- Isolation of the impact of various bounds checking mechanisms on that performance.
- Implementations of alternative bounds checking strategies for several WebAssembly runtimes, including a novel approach based on user-mode page fault handlers.
- A reproducible benchmark suite with automatic execution and data collection that can be reused on new platforms.

- 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:

- There are *no significant differences* in the relative costs of bounds checking methods across architectures for simple numerical kernels from PolyBench/C: the cost of each method seems to be roughly the same on x86-64, Armv8 and RISC-V.
- Using `mprotect()` on Linux to dynamically adjust the size of WebAssembly memory causes *poor multi-threaded scaling*, decreasing maximum CPU utilization by up to 25% for short-running benchmarks. This is fully mitigated by our userfaultfd-based bounds checking approach.
- 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 an 8-20% average overhead overall on x86-64.

2. Background

2.1. WebAssembly

WebAssembly is a simple and portable bytecode format that can also be thought of as a programming language [26]. It was designed to be an easy target for compilation of high-level 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 to run safe, fast, and portable code on the Web, replacing previous attempts such as asm.js [11] 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) [34] were co-developed alongside WebAssembly and explicitly create a POSIX-like environment rather than a Web-based one.

In this way, WebAssembly runtimes can be compared with other programming language virtual machines, like the Java Virtual Machine [24], 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 [20] for languages such as C#, F#, and Visual Basic.

However, despite the similarities, WebAssembly is significantly less complex in its design than the aforementioned languages. 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 organizational 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 a security mechanism for indirect branches to avoid exposing the host's instruction pointer directly; and exports – supporting named references to various other module elements 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 allow 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 [18], follow a fetch-execute loop, reading the bytecode and then executing native code specific to the fetched instruction. Various implementation techniques have emerged for interpreters, and currently, the most prevalent one for fast execution is threaded interpretation [1]. Such interpreters 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 program execution, often inserting instrumentation and using the collected data to choose functions to recompile for better performance. The V8 [8] 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. Despite using JIT frameworks to load the compiled code at runtime into the host, WAVM [27] and Wasmtime [3] are in fact AOT compilers, since 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 to an address within the bounds of the active linear memory. This is similar to inserting checks that indices lie inside the bounds of an array for each array indexing operation, but here, it happens for every memory access. 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. On average, load and store instructions form 40% of x86-64 programs [10], and inserting a branch instruction before every single one results in a significant cost, even if some proportion of the branches can be eliminated by an optimization pass.

Therefore, high-performance runtimes instead use operating system mechanisms to manage virtual memory themselves and catch out-of-bounds accesses more efficiently. This is done by over-allocating a large virtual memory region and only populating the valid memory range with read-write-allowed pages. The rest of the region generates a CPU exception if illegally accessed, which the runtime can subsequently catch and handle. Because the current WebAssembly standard limits the memory access instruction operands to a 32-bit integer address base and a 32-bit integer offset, the total addressable space is 8 GiB, which can be reserved in a single allocation on 64-bit machines with virtual memory. The two 32-bit integers mean that mathematically, the generated machine code cannot access the area outside of this allocation. The downside of this approach is that managing such large allocations in the operating system can be costly, especially on less powerful hardware. In Linux, changing the size of such an allocation requires adjusting shared process VMAs, which requires taking an exclusive modification lock [14]. If modification happens frequently, this locking can have a negative scaling impact for multithreaded applications.

An alternative mechanism for managing virtual memory is Userfaultfd [15], which lets applications reserve a region of virtual memory and handle page faults on that region in userspace, with no kernel-side locking and VMAs remaining untouched. The page fault handler can choose to populate the faulted page (or a larger range of pages) with zero-filled pages or with content copied from another range of pages. It can also choose not to populate the pages at all and instead raise an exception, if it determines that the access is illegal. The handler can operate either as a thread polling the userfault file descriptor and being notified of events, or as a signal handler for SIGBUS signals that the kernel sends to the page faulting thread. Because the SIGBUS handler gets executed within the same thread that caused the page fault, it avoids back-and-forth context switches and can therefore achieve lower latency [37]. Thus, this is the method we decided to use in our Userfaultfd-accelerated bounds checking implementation.

Various hardware-accelerated bounds checking methods have also been proposed and implemented for array accesses, some of which could be reused for WebAssembly bounds checking. For example, some Intel processors had an MPX extension providing bounds-checked pointer access instructions. However, such extensions have a high overhead (50% on average [22]), leading Intel to discontinue the MPX extension and remove it from the x86 processor manuals in 2019. An upcoming, promising approach is CHERI [36], which provides capability-checked memory accesses to multiple CPU architectures with a single mechanism. However, CHERI is still in a relatively early phase of development and hardware availability is very limited, so we did not evaluate it in this work.

3. Benchmark Design

In this section, we present how we analyze and compare the performance of different bounds checking techniques in selected WebAssembly runtimes on multiple 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:** A conditional branch to error handling code. If an access is out of bounds, a trap to the host is generated by jumping to an invalid instruction, 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` caught by the runtime. The runtime invokes `mprotect()` to modify the process' virtual memory area (VMA) to grant the necessary permissions when memory is resized. As mentioned previously, this requires acquiring a lock on the VMA of the process, since all threads within a process share one VMA protecting the entire virtual address space.
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 be handled in user space. Any attempt to write a missing page generates a `SIGBUS`, which prompts either an `ioctl` call to copy or zero the page, or a new signal to 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 four are WebAssembly runtimes, where the benchmarks are first compiled to WebAssembly using Clang 13 from the WASI SDK (target wasm32-wasi) before being translated to native code and executed by the runtime. The WebAssembly runtimes are able to provide isolation, so for each benchmark instance, the WebAssembly benchmark runner spawns one instance of the runtime in an isolated thread, with all threads contained within the same process. On the other hand,

the native benchmark runner spawns one process for each benchmark instance. The four WebAssembly runtimes are:

1) **WAVM** [27]: A standalone virtual machine that uses the LLVM [16] compiler infrastructure (specifically the MCJIT framework [17]) to AOT compile WebAssembly to machine code. 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 to add alternative bounds checking methods.
3) **Wasm3** [18]: 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 [23]: A standalone JavaScript and WebAssembly runtime, used as a part of the Chromium [7] web browser and focused on striking a balance between speed of compilation and speed of the executed code for Web applications. We modified 400 lines of code to add alternative bounds checking methods.

The WAVM, Wasmtime, and V8 runtimes all use *mprotect* to implement bounds checking by default. We augmented those three runtimes with implementations for *none*, *clamp*, *trap*, and *uffd* strategies. Since Wasm3 does not generate compiled code, the way that the memory instruction interpreter code is written means that it effectively uses an equivalent of the *trap* mechanism. Since the Wasm3 runtime is already significantly slower at executing WebAssembly, we did not modify 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) [34], 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 [25] in the MEDIUM configuration for evaluation, in order to allow us to compare with earlier results [13] [26]. We also decided to use the SPEC CPU 2017 Rate benchmark suite [28] in order to provide a more comprehensive evaluation. PolyBench/C consists mostly of simple numerical kernels, while SPEC CPU is made up of more complex, real-world applications such as deepsjeng (chess engine), xz (compressor), and x264 (video codec). Together, the two suites cover a wide range of applications, providing a comprehensive evaluation of WebAssembly overheads in different scenarios.

Since some of the SPEC benchmarks rely on libc and C++ functionality (e.g., signal handling, non-local exits, exceptions) and the WASI libc [35] implementation is still under development, we were only able to compile a subset[1] of the SPEC benchmarks to WebAssembly for evaluation. However, we were able to use all of the PolyBench/C benchmarks. We are optimistic that as the WASI and WebAssembly standards evolve and a Fortran to WebAssembly compiler is developed, the rest of the SPEC CPU suite will run under WASI.

Due to the very long execution times of the SPEC benchmarks and the very high number of tested configurations, we chose to use the Train configuration rather than the Refrate configuration. Based on trial runs, we estimate that running all of the benchmark configurations in the Refrate mode would take about a month of CPU time on each machine, and possibly more on RISC-V, if our platform had enough memory available to run SPEC there.

1. Subset of SPEC CPU 2017 Rate suite used: 505.mcf_r, 508.namd_r, 519.lbm_r, 525.x264_r, 531.deepsjeng_r, 544.nab_r, and 557.xz_r

3.4. Hardware

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

1) **x86-64**: Intel Xeon Gold 6230R, with 16 hardware threads enabled, no simultaneous multithreading, 768 GiB of system memory.
2) **AArch64**: Cavium ThunderX2 CN9980 v2.2, configured to have 16 hardware threads, no simultaneous multithreading, 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 [30]. The CPU governors were set to performance mode where possible, to prefer higher operating frequency over power saving.

On each system, we ran 1, 4 and 16 copies of the benchmarks 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 PolyBench/C suite, and only in a single-threaded mode. This is because the 1 GiB physical memory available made it impossible to run the SPEC suite, and because the CPU only has one physical core with no simultaneous multi-threading capabilities. Additionally, the WAVM and Wasmtime runtimes do not have RISC-V backends to test – when WAVM was forced to generate RISC-V code via LLVM, this led to crashes in the MCJIT framework, and Wasmtime's Cranelift backend does not have a RISC-V target implemented – leaving the RISC-V platform with the native, Wasm3 and V8 runtimes only.

3.5. Benchmarking Harness

In order to make consistent measurements between all of the execution environments, we implemented a custom benchmarking harness in about 2000 lines of C++ code. The harness interacts directly with the WebAssembly runtimes via their C and C++ APIs.

The harness first ensures that the Wasm code is fully loaded into the runtime and compiled where appropriate,

before executing a clone of the benchmark module in a timed loop in each worker thread that is pinned to a CPU core. Only the module execution is timed; the setup and teardown between loop iterations is not included in the reported time.

To ensure that all physical CPU threads are equally busy before the timed execution runs begin, there is a warm-up phase. Once each thread finishes its timed workload, it continues to run the WebAssembly code for a few more iterations, until all of the threads finish their measured runs. This ensures that the final measurements are not affected by other CPU cores becoming less busy.

For native code, the same overall procedure is followed, except rather than simply calling the JITted code, a new process is spawned with a `vfork()` and `fexecve()` syscall combination (on a pre-opened executable file descriptor to reduce process creation overhead). This is done because loading multiple copies of native Linux executables into the same process is not achievable via any standard system interfaces. The downside of this is that it includes the process spawning and teardown overhead in the native code measurements. However, we measured this overhead to be in the order of a hundred microseconds once the benchmarks warm up, so it does not affect the results significantly.

Our benchmarking harness, patches to the WebAssembly runtimes, and automation scripts are all available under an open-source license, excluding the SPEC benchmarks, which are protected by copyright. See the Appendix for instructions on how to obtain and execute our code [29].

4. Evaluation

In the following section, we discuss the performance of each bounds checking mechanism and runtime configuration introduced in subsection 3.1 and subsection 3.2 when executing the benchmarks listed in subsection 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 SPEC benchmark runs 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, obtained 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, and 2c. SPEC and PolyBench/C (PBC) results are separated.

From these results, we can see that the fastest WebAssembly runtime among those evaluated is WAVM, followed by Wasmtime and then very closely by V8. As we would expect, no bounds checking is the fastest. However, the `mprotect()` and UFFD strategies also have very little overhead, in the order of 1-2 percentage points, except in the case of the V8 runtime, which has a 10 point difference.

The conditional 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.

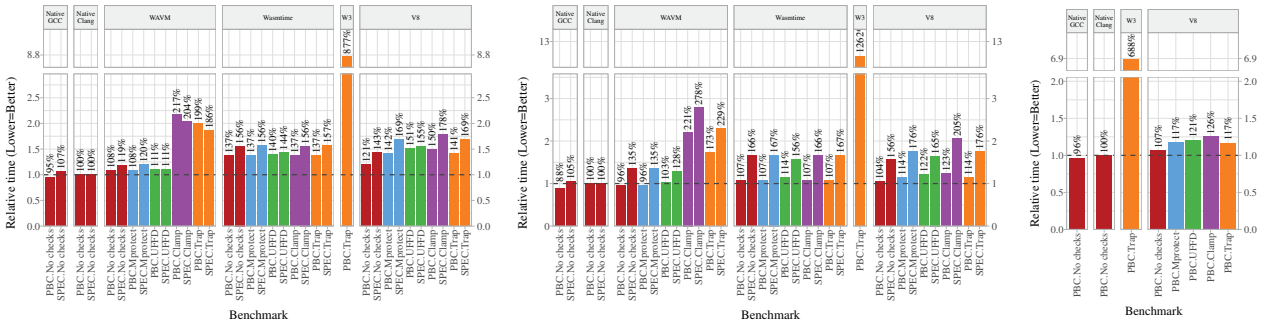
We investigate the causes of these differences in the following sections by looking at various system and CPU performance counters. This data is presented for the x86-64 and Armv8 architectures, because running the monitoring tools on the RISC-V board was causing significant changes to the results due to the slow, single-threaded CPU performance.

4.1.1. Scaling With Thread Counts. One interesting characteristic of the various runtimes and bounds checking methods is how their overall performance is affected by running multiple isolates in parallel on separate threads. We investigated this by running multiple instances of each benchmark on worker threads, pinned to chosen CPU cores to reduce the impact 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.

In most cases, we can see that 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 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.

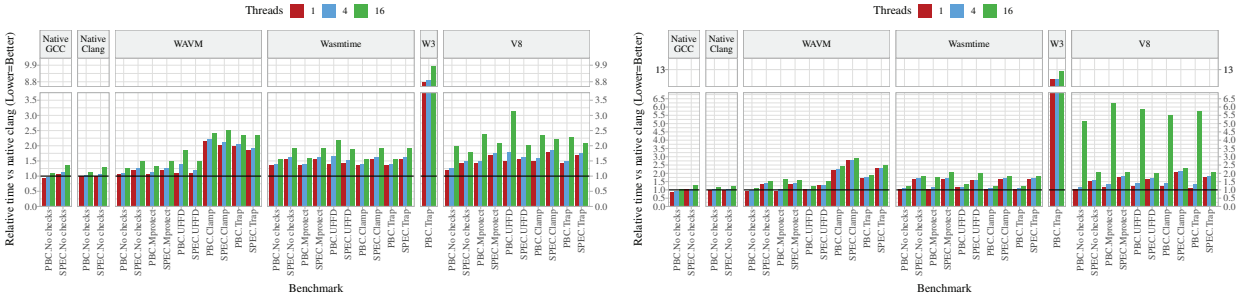
One major visible result is that V8 struggles when 16 worker threads are created. This is because V8 uses worker threads for some of its internal operations, such as JIT compilation. Internal worker threads are also used for periodic garbage collection, which locks other worker threads from performing work. When all of the physical cores are already occupied by the benchmarks, this internal 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. This is especially visible in the PolyBench/C benchmarks, which execute in a short span of time, causing frequent allocation and deallocation of memory. This stresses the virtual memory management subsystem in the Linux kernel for the host process, causing excessive locking and pausing of thread execution.



(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

4.2. System Performance 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` spending in either user or kernel mode[2], offset by the total number of milliseconds spent idle, i.e.

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

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

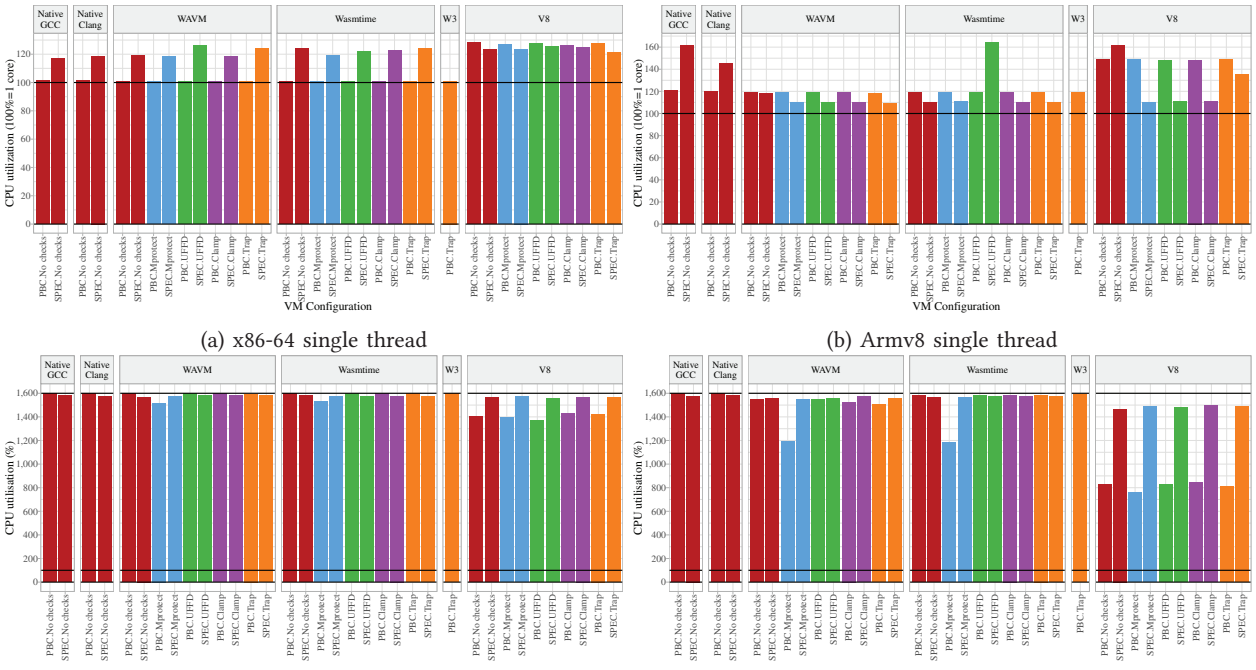
In figures 4a and 4b, we can see that in the single-threaded configuration, all of the runtimes are able to saturate a full CPU core, with the Arm machine having larger off-main-thread activity than x86. As mentioned in subsubsection 4.1.1, the V8 runtime implementation uses extra worker threads for internal operations, therefore the utilization for V8 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. Again, the lower saturation in V8 is due to the periodically running JavaScript garbage collector, which pauses the execution of other threads.

Similar to the results in subsection 4.1.1, one big, visible difference here between the bounds checking strategies is that `mprotect()`-based protection does not saturate the CPU like other mechanisms. As discussed in subsection 3.1, this is due to a mutex in the Linux kernel protecting the process VMA. When WebAssembly resizes its memory to allocate or run the next iteration, that mutex is acquired for significant periods of time, which we were able to confirm by capturing stack traces of threads in a waiting state via bpftools.

The effect is not visible with the conditional clamp and trap bounds checking methods because they require less virtual memory manipulation. 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 [19]-style implementation for adding and removing memory arenas, avoiding the need for locks most of the time.

The locking effect was significantly more visible in shorter-running benchmarks. Therefore, we make a recommendation that for short-lived WebAssembly tasks, such as for certain classes of serverless applications, using

2. **user** 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

userspace-managed pagefault handlers can be preferential to `mprotect()`-based handlers, unless the Linux kernel memory management switches to finer-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. The bounds checking mechanism has no significant impact 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 due to the internal worker threads.

4.3. Memory Usage

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

There is no visibly significant variance in memory usage between the different runtimes or bounds checking methods. When considering architectures, one observable difference is the increased memory usage of the PolyBench/C benchmark suite on the x86-64 architecture when compared with 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. Reclaiming that memory then requires splitting the pages into smaller ones. The transparent huge pages mechanism on the x86-64 ISA uses pages of up to 1 GiB in size, whilst on Armv8, the limit is 2 MiB, leading to more fine-grained memory usage reporting.

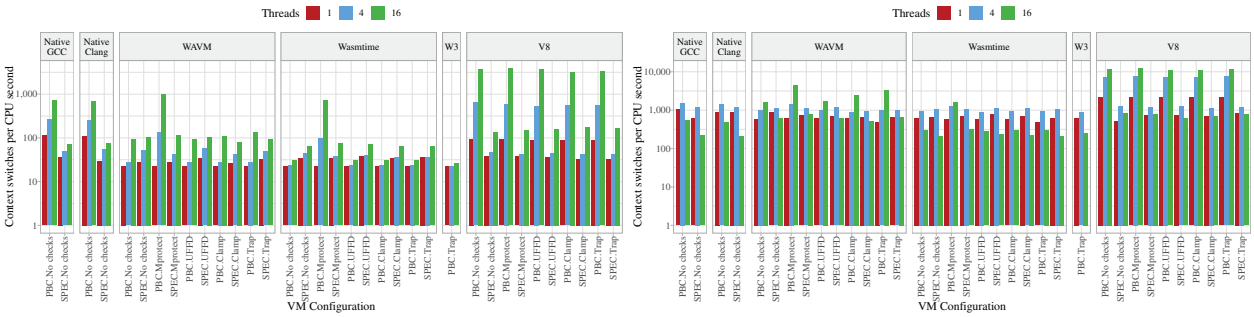
4.4. Replicating Previous Results

In 2022, Titzer [31] measured Wasm3 to be roughly 10× slower than V8-TurboFan on the PolyBench/C benchmarks, which agrees with our results of between 6 and 11× slower on the same suite, depending on the CPU architecture.

In 2017, Rossberg et al. [26] measured PolyBench/C 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". The measured performance for each benchmark closely matches our measurement in figure 1.

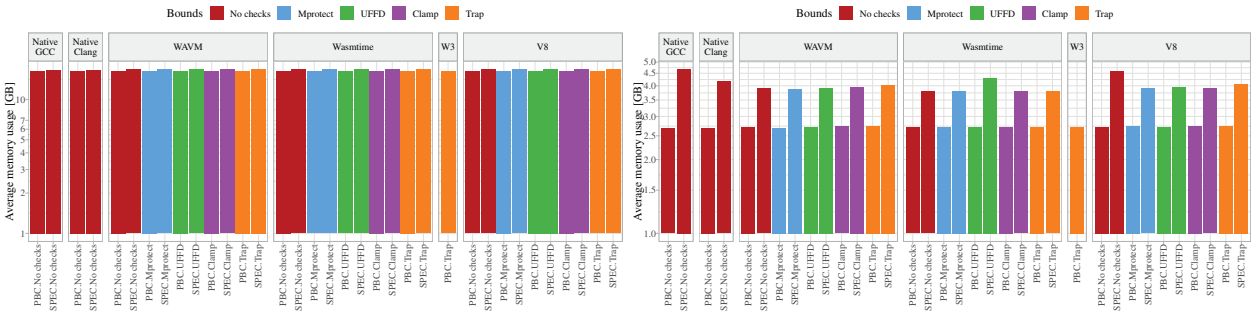
In 2019, Jangda et al. [13] reported a 1.55× geomean slowdown of SPEC on V8 compared to native. We measured a 1.69× slowdown on x86-64 and a 1.76× slowdown on Armv8. Jangda et al. were able to run a bigger subset of SPEC, thanks to developing a custom POSIX layer that WebAssembly interacted with via JavaScript. Most of the runtimes evaluated in this paper do not support JavaScript, therefore we could not use Browsix to run the same subset of benchmarks.

Using these previous works as a reference point, we can see that whilst Web-focused WebAssembly runtimes and interpreters have made very slow progress on the performance front since 2017, more performance-oriented



(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

runtimes have emerged recently and are approaching near-native performance levels for a wider set of programs.

5. Related Work

Whilst previous work already compared different WASM runtimes already, none focused on the overhead added by bounds checking. Additionally, previous work does not show how the cost of bounds checking varies across different CPU ISAs. We briefly summarize this previous work here.

With the introduction of Wasm in their 2017 paper, Rossberg et al. [26] compared the Wasm implementation in V8 and in SpiderMonkey on x86 only, without breaking down the bounds checking overhead. Jangda et al. [13] introduces additional benchmarks to Rossberg et al. [26] using a JavaScript POSIX emulation shim, so that SPEC can also be used to benchmark WebAssembly, in addition to PolyBench/C. They are the first to highlight that Wasm is slower than what has been reported before, and that one of the issues is the safety checks. However, their work does not explore why that is the case, is based on x86 only, and does not introduce new bounds checking mechanisms enabled by the latest OS advances – which is the core contribution of this paper.

Yan et al. [38] 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 does not explain the cost of bounds checking, nor does it introduce anything new in Wasm runtimes.

Hilbig et al. [12] also introduces a large collection of Wasm benchmarks, WasmBench – which focuses on x86 only, and does not include considerations on bounds checking – highlighting that memory errors can be propagated into Wasm and further justifying our work. Finally, Titzer [31] compares several engine runtimes (WAMR, Wasm3, V8-Liftoff and V8-TurboFan, SpiderMonkey, and JSC) on an Intel Core-i7, using PolyBench/C-4.2.1, showing execution time, translation time and space statistics. Whilst we reported similar metrics, Titzer did not breakdown the cost of bounds checking, nor did they introduce any new bounds checking methods on any engine runtime, nor did they compare 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 exists a variety of available runtimes, each striking a different balance between complexity, size, and runtime performance. WebAssembly has grown from initially having Web-focused applications into being a generic sandbox platform for server [32] and client [5] applications.

WebAssembly brings unique security mechanisms to the table, with one of the major ones being bounds-checked

memory accesses. Whilst other languages often check array index bounds, WebAssembly limits all memory instructions to accessing a single, resizable block of memory, checking whether those accesses are within the current bounds on every load and store. We augmented the WebAssembly runtimes that use a compiler with alternative bounds checking approaches, and based on this, we 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 a significantly higher overhead when compared to allocating large regions of virtual memory and using page fault handlers to catch illegal accesses.

7. Conclusion

We show that, on average, runtimes such as WAVM and Wasmtime are able to achieve performance within 20% of the native performance on x86-64 platforms, and within 35% on Armv8. On RISC-V, V8 can achieve a 17% overhead over native code for simple numeric kernels from the PolyBench/C benchmark suite. For such kernels, we have shown that there is no significant difference in the WebAssembly execution time overheads across the three tested architectures: x86-64, Armv8, and RISC-V RV64GC.

When considering 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 achieve 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, with the exception of the SPEC CPU benchmarks, for which we only distribute the small patches required to compile them for WebAssembly due to the SPEC 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.

Acknowledgment

This work was partially funded by the Edinburgh Huawei Research Lab.

Appendix

1. Abstract

The artifact contains the source code of the benchmarks and WebAssembly runtimes evaluated in the paper, Docker containers used for building and executing them, the original experimental data as shown in the plots in the paper, and R code to generate the plots used in the paper – either from the original data, or a new benchmark run by the artifact user.

Users can reproduce all the presented results (all figures) in the paper, provided they can supply a copy of the SPECcpu2017 benchmark suite and own three platforms compatible with the systems used in the paper (x86_64, Armv8 and RISC-V RV64GC). Any subset of the benchmarks can be run on any subset of the machines to reproduce the results for those specific suites and architectures.

2. Artifact check-list (meta-information)

- **Program:** PolybenchC 4.2 (included, public), SPECcpu2017 (not provided, optional, proprietary, patches provided)
- **Compilation:** Standard GCC 11, Clang 11, Clang 13, Rust 1.62 – all provided in the toolchain Docker images
- **Binary:** Benchmark binaries compiled for the x86_64 architecture provided, scripts to build for other architectures provided, harness binaries provided for all architectures in the form of Docker images
- **Data set:** Included in the source code of the benchmark suites
- **Run-time environment:** Linux 5.7, R 4.2.1 and Rust 1.62 compiler required, Ubuntu 22.04 installation recommended. Dependencies provided
- **Hardware:** x86_64, 64-bit Armv8 or RISC-V RV64GC CPU required
- **Run-time state:** Sensitive to background applications, run isolated
- **Execution:** Sole user of the machine, can take up to 3 days to run depending on SPECcpu2017 presence
- **Metrics:** Reports execution time (each sample stored individually for later aggregation), system performance counters (load, process counts, etc.), CPU performance counters (branches predicted, cache misses, etc. where available)
- **Output:** Benchmarks produce a logfile, which can be converted to CSV via the included script and plotted to PDF graphs via the provided R script
- **Experiments:** Automation shell scripts are provided, manual actions needed for the initial setup.
- **How much disk space required (approximately)?:** 25 GiB
- **How much time is needed to prepare workflow (approximately)?:** 15 minutes, 1 hour if also providing a copy of SPECcpu2017
- **How much time is needed to complete experiments (approximately)?:** 1 day, 3 days if also providing a copy of SPECcpu2017
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT, Full source code provided except for SPECcpu2017
- **Data licenses (if publicly available)?:** MIT
- **Workflow framework used?:** Custom C++ code with Docker containers and scripts for reproducibility and ease-of-use
- **Archived (provide DOI)?:** Yes, on Zenodo: https://doi.org/10.5281/zenodo.7068161 [29], also available on GitHub: https://github.com/wasmbounds/wasmbounds and https://github.com/wasmbounds/wasmbounds/releases/tag/v1.0.0

3. Description

3.1. How to access.

1) Download all the archives from Zenodo https://doi.org/10.5281/zenodo.7068161 [29] (or GitHub if that is more convenient, the files are identical: https://github.com/wasmbounds/wasmbounds/releases/tag/v1.0.0).
2) Extract `wasmbounds-oss-v1.0.0.tar.gz` to a folder named `wasmbounds`
3) Follow the instructions in the README.MD plain text file in the extracted folder.

3.2. Hardware dependencies.

- Generating plots with R requires about 10 GiB free system memory while parsing the input CSV files
- The Polybench/C benchmarks have been successfuly run on the RISC-V system with 1GiB of memory (singlethreaded), while SPECcpu2017 requires up to 2GiB per thread run
- Plotting software and the WASM compiler run on the x86_64 architecture
- The benchmarks and WebAssembly runtimes run on x86_64, Armv8 (AArch64) and RISC-V 64GC architectures (some WebAssembly runtimes don't successfully run on RISC-V)

3.3. Software dependencies. The required software for reproducing the results is as follows:

- A recent Linux distribution is assumed as the operating system, e.g. Ubuntu 22.04 LTS. Use `mitigations=off nr_cpus=16` to turn off Spectre mitigations and limit CPU core count to 16 to match the configuration used in the paper.
- For the plotting: R 4.2.1, with the tidyverse and other libraries, specifically: tidyverse, rio, scales, xtable, ggbreak, patchwork, rmarkdown, xfun.
- For system monitoring (optional, recommended for full results): Rust compiler v1.62.0
- For compiling and running benchmarks via containers: Docker - Dockerfiles are provided which will fetch, compile and run all necessary dependencies
- For running benchmarks on bare metal:
  - For Ubuntu 22.04-specific installation instructions see the commands in the Dockerfiles included in the repository
  - Python 3.10
  - Docker 20.10
  - Linux kernel, version at least 5.7
  - GCC 11.2 for native benchmark compilation
  - LLVM 11 + Clang SDK for the WAVM runtime
  - Clang 13 for native benchmark compilation
  - WASI SDK 15
  - CMake 3.22
  - Boost 1.79.0, Abseil C++ 20220623.1

4. Installation

Full step-by-step instructions are provided in the `README.MD` file.

A summarized version of commands to run in the system shell is:

```
# In first shell run the performance monitoring tool
cd statmon
cargo build --release
# Root permission might not be needed if unprivileged access to performance counters is enabled via the relevant sysctls in Linux
sudo ./target/release/statmon --port 8125 --host-prefix local --netdev lo

# In second shell
# Load's last line of output has the sha256 of the imported image, tags have to be recreated manually
docker load -i wasmbounds-runtime-base.ARCHITECTURE.tar
docker tag sha256:xxxxxxx wasmbounds-runtime-base
docker load -i wasmbounds-toolchain-base.ARCHITECTURE.tar
docker tag sha256:xxxxxxx wasmbounds-toolchain-base
docker load -i wasmbounds-runners.ARCHITECTURE.tar
docker tag sha256:xxxxxxx wasmbounds-runners
```

5. Experiment workflow

Full step-by-step instructions are provided in the `README.MD` file.

Summarized version of commands to run benchmarks:

```
# If not using the provided x86\_64 binaries:
./build_binaries.sh polybenchc
# Dry run (no benchmark execution)
./benchrunner.sh --monitor-host 127.0.0.1 --monitor-port 8125 --output-dir runs --dry-run --suites polybenchc
# Test run (each benchmark ran once)
./benchrunner.sh --monitor-host 127.0.0.1 --monitor-port 8125 --output-dir runs --one-run --min-seconds 0 --min-runs 1 --suites polybenchc
# Full run
./benchrunner.sh --monitor-host 127.0.0.1 --monitor-port 8125 --output-dir runs --one-run --min-seconds 20 --min-runs 10 --suites polybenchc

# Convert the run logfile into a csv file for the plotting script
ls runs # note the filename
./scripts/log2csv.py ./runs/benchrunner-HOST-regular-DATE-TIME.log ./runs/myrun.csv
```

6. Evaluation and expected results

Full step-by-step instructions for generating plots are provided in the `README.MD` file. Minor variations in the specific values as quoted in the paper are to be expected, but the relative differences (larger vs. smaller) between bounds checking methods should remain the same.

The short `plots/knitall.R` script contains an array of machines plots are made for, add `myrun` to it to generate plots from your experiment run(s). Run the following command to regenerate plots from the data files in `runs/`:

```
cd graphs
Rscript ./knitall.R
```

The plots will be generated as `graphs/plots/*.pdf` files, the naming convention for them is `wasmbounds_[MACHINE]_[VARIABLE]_[CONFIGURATION].pdf`. The `knitall.R` script "knits" the `wasmbounds.Rmd` file in the same folder three times, switching out the machine parameter

7. Experiment customization

Detailed description of the code structure, recompilation and modification instructions are provided in the `README.MD` file.

Each of the WebAssembly (Wasm) runtimes and benchmark suites lives in a separate folder named after the upstream project. As much as it was possible, we made minimum modifications to the runtimes and instead put most of the shared code in the `runner-src/ directory`.

Each Wasm runtime has a corresponding benchmark runner executable, built from the `runner-src/impl_NAME.cpp` C++ source code, the build is defined in the `CMakeLists.txt` file at the root of the bundle.

`runner-src/runner.cpp` has the main benchmarking loop, which calls the setup and run functions for each runner. The runner interface `RunnerImpl` is defined in `runner-src/runner.h` file, it provides 3 main functions: the constructor `RunnerImpl (const RunnerOptions &opts)` initializing the object, called once per thread; `void prepareRun (const RunnerOptions &opts)` which is executed before each execution step untimed; and `void runOnce (const RunnerOptions &opts)` which is executed at each step, timed by the runner loop.

To implement bounds checking methods like mprotect and uffd control over the memory allocation is needed, so we implemented a small library `runner-src/vm-library` which provides header files for C and C++ at `wasmbounds_rr.h[pp]`, defining memory allocation, resizing and deallocation stubs allowing to use the same implementation across all tested WebAssembly runtimes. The runtimes were patched to call into this library instead of their own platform abstraction layer for managing WebAssembly memory objects for the purpose of evaluating the impact of different bounds checking methods in our paper.

The difference between upstream projects and our patched versions can be seen either in the .patch files in our bundle, or as the difference between commit `56643dd0` and the latest commit in the GitHub repository.

References

[1] J. R. Bell, "Threaded code," *Commun. ACM*, vol. 16, no. 6, pp. 370–372, 1973. [Online]. Available: https://doi.org/10.1145/362248.362270

[2] Bytecode Alliance, "Cranelift," https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/README.md, 2022, [Online; accessed 01-Mar-2022].

[3] ——, "Wasmtime," https://github.com/bytecodealliance/wasmtime, 2022, [Online; accessed 01-Mar-2022].

[4] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," *Commun. ACM*, vol. 29, no. 3, pp. 218–221, 1986. [Online]. Available: https://doi.org/10.1145/5666.5673

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

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

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

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

[9] W. C. Group, "WebAssembly Core Specification," https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition.* -: Morgan Kaufmann, 2012.

[11] D. Herman, L. Wagner, and A. Zakai, "asm.js Specification," http://asmjs.org/spec/latest/, 2014.

[12] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2696–2708. [Online]. Available: https://doi.org/10.1145/3442381.3450138

[13] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of webassembly vs. native code," *login Usenix Mag.*, vol. 44, no. 3, pp. 12–16, 2019. [Online]. Available: https://www.usenix.org/publications/login/fall2019/jangda

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

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

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

[17] ——, "LLVM MCJIT," https://llvm.org/docs/MCJITDesignAndImplementation.html, 2022, [Online; accessed 01-Mar-2022].

[18] S. Massey and V. Shymanskyy, "WASM3," https://github.com/wasm3/wasm3, 2022, [Online; accessed 12-Jul-2022].

[19] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distributed Syst.*, vol. 15, no. 6, pp. 491–504, 2004. [Online]. Available: https://doi.org/10.1109/TPDS.2004.8

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

[21] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "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*, ser. Lecture Notes in Computer Science, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds., vol. 11543. Gothenburg, Sweden: Springer, 2019, pp. 23–42. [Online]. Available: https://doi.org/10.1007/978-3-030-22038-9_2

[22] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–30, 2018.

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

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

[25] L.-N. Pouchet and T. Yuki, "Polybench/C," https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2015, [Online; accessed 01-Mar-2022].

[26] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman, "Bringing the web up to speed with webassembly," *Commun. ACM*, vol. 61, no. 12, pp. 107–115, 2018. [Online]. Available: https://doi.org/10.1145/3282510

[27] A. Scheidecker, "WAVM," https://github.com/WAVM/WAVM, 2022, [Online; accessed 01-Mar-2022].

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

[29] R. Szewczyk, K. Stonehouse, A. Barbalace, and T. Spink, "Leaps and bounds: Analysing WebAssembly's performance with a focus on bounds checking," Nov. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7068161

[30] M. Taram, A. Venkat, and D. M. Tullsen, "Mitigating speculative execution attacks via context-sensitive fencing," *IEEE Des. Test*, vol. 39, no. 4, pp. 49–57, 2022. [Online]. Available: https://doi.org/10.1109/MDAT.2022.3152633

[31] B. L. Titzer, "A fast in-place interpreter for webassembly," 2022. [Online]. Available: https://arxiv.org/abs/2205.01183

[32] K. Varda, "WebAssembly on Cloudflare workers," https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, Dec 2018.

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

[34] ——, "WASI," https://wasi.dev/, 2022, [Online; accessed 12-Jul-2022].

[35] ——, "WASI Libc," https://github.com/WebAssembly/wasi-libc, 2022, [Online; accessed 01-Mar-2022].

[36] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. M. Norton, and M. Roe, "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, 2014, pp. 457–468. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853201

[37] P. Xu, "Userfaultfd-wp Latency Measurements," https://xzpeter.org/userfaultfd-wp-latency-measurements/, 2020.

[38] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the performance of webassembly applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 533–549. [Online]. Available: https://doi.org/10.1145/3487552.3487827