Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code

Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha,
University of Massachusetts Amherst

https://www.usenix.org/conference/atc19/presentation/jangda

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code

Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha
University of Massachusetts Amherst

Abstract

All major web browsers now support WebAssembly, a low-level bytecode intended to serve as a compilation target for code written in languages like C and C++. A key goal of WebAssembly is performance parity with native code; previous work reports near parity, with many applications compiled to WebAssembly running on average 10% slower than native code. However, this evaluation was limited to a suite of scientific kernels, each consisting of roughly 100 lines of code. Running more substantial applications was not possible because compiling code to WebAssembly is only part of the puzzle: standard Unix APIs are not available in the web browser environment. To address this challenge, we build BROWSIX-WASM, a significant extension to BROWSIX [29] that, for the first time, makes it possible to run unmodified WebAssembly-compiled Unix applications directly inside the browser. We then use BROWSIX-WASM to conduct the first large-scale evaluation of the performance of WebAssembly vs. native. Across the SPEC CPU suite of benchmarks, we find a substantial performance gap: applications compiled to WebAssembly run slower by an average of 45% (Firefox) to 55% (Chrome), with peak slowdowns of 2.08× (Firefox) and 2.5× (Chrome). We identify the causes of this performance degradation, some of which are due to missing optimizations and code generation issues, while others are inherent to the WebAssembly platform.

1 Introduction

Web browsers have become the most popular platform for running user-facing applications, and until recently, JavaScript was the only programming language supported by all major web browsers. Beyond its many quirks and pitfalls from the perspective of programming language design, JavaScript is also notoriously difficult to compile efficiently [12, 17, 30, 31]. Applications written in or compiled to JavaScript typically run much slower than their native counterparts. To address this situation, a group of browser vendors jointly developed *WebAssembly*.

WebAssembly is a low-level, statically typed language that does not require garbage collection, and supports interoperability with JavaScript. The goal of WebAssembly is to serve as a universal compiler target that can run in a browser [15, 16, 18].[1] Towards this end, WebAssembly is designed to be fast to compile and run, to be portable across browsers and architectures, and to provide formal guarantees of type and memory safety. Prior attempts at running code at native speed in the browser [4, 13, 14, 38], which we discuss in related work, do not satisfy all of these criteria.

WebAssembly is now supported by all major browsers [8, 34] and has been swiftly adopted by several programming languages. There are now backends for C, C++, C#, Go, and Rust [1, 2, 24, 39] that target WebAssembly. A curated list currently includes more than a dozen others [10]. Today, code written in these languages can be safely executed in browser sandboxes across any modern device once compiled to WebAssembly.

A major goal of WebAssembly is to be faster than JavaScript. For example, the paper that introduced WebAssembly [18] showed that when a C program is compiled to WebAssembly instead of JavaScript (`asm.js`), it runs 34% faster in Google Chrome. That paper also showed that the performance of WebAssembly is competitive with native code: of the 24 benchmarks evaluated, the running time of seven benchmarks using WebAssembly is within 10% of native code, and almost all of them are less than 2× slower than native code. Figure 1 shows that WebAssembly implementations have continuously improved with respect to these benchmarks. In 2017, only seven benchmarks performed within 1.1× of native, but by 2019, this number increased to 13.

These results appear promising, but they beg the question: *are these 24 benchmarks representative of WebAssembly's intended use cases*?

[1]The WebAssembly standard is undergoing active development, with ongoing efforts to extend WebAssembly with features ranging from SIMD primitives and threading to tail calls and garbage collection. This paper focuses on the initial and stable version of WebAssembly [18], which is supported by all major browsers.

The Challenge of Benchmarking WebAssembly The aforementioned suite of 24 benchmarks is the PolybenchC benchmark suite [5], which is designed to measure the effect of polyhedral loop optimizations in compilers. All the benchmarks in the suite are small scientific computing kernels rather than full applications (e.g., matrix multiplication and LU Decomposition); each is roughly 100 LOC. While WebAssembly is designed to accelerate scientific kernels on the Web, it is also explicitly designed for a much richer set of full applications.

The WebAssembly documentation highlights several intended use cases [7], including scientific kernels, image editing, video editing, image recognition, scientific visualization, simulations, programming language interpreters, virtual machines, and POSIX applications. Therefore, WebAssembly's strong performance on the scientific kernels in PolybenchC do not imply that it will perform well given a different kind of application.

We argue that a more comprehensive evaluation of WebAssembly should rely on an established benchmark suite of large programs, such as the SPEC CPU benchmark suites. In fact, the SPEC CPU 2006 and 2017 suite of benchmarks include several applications that fall under the intended use cases of WebAssembly: eight benchmarks are scientific applications (e.g., `433.milc`, `444.namd`, `447.dealII`, `450.soplex`, and `470.lbm`), two benchmarks involve image and video processing (`464.h264ref` and `453.povray`), and all of the benchmarks are POSIX applications.

Unfortunately, it is not possible to simply compile a sophisticated native program to WebAssembly. Native programs, including the programs in the SPEC CPU suites, require operating system services, such as a filesystem, synchronous I/O, and processes, which WebAssembly and the browser do not provide. The SPEC benchmarking harness itself requires a file system, a shell, the ability to spawn processes, and other Unix facilities. To overcome these limitations when porting native applications to the web, many programmers painstakingly modify their programs to avoid or mimic missing operating system services. Modifying well-known benchmarks, such as SPEC CPU, would not only be time consuming but would also pose a serious threat to validity.

The standard approach to running these applications today is to use Emscripten, a toolchain for compiling C and C++ to WebAssembly [39]. Unfortunately, Emscripten only supports the most trivial system calls and does not scale up to large-scale applications. For example, to enable applications to use synchronous I/O, the default Emscripten `MEMFS` filesystem loads the entire filesystem image into memory before the program begins executing. For SPEC, these files are too large to fit into memory.

A promising alternative is to use BROWSIX, a framework that enables running unmodified, full-featured Unix applications in the browser [28, 29]. BROWSIX implements a Unix-compatible kernel in JavaScript, with full support for processes, files, pipes, blocking I/O, and other Unix features. Moreover, it includes a C/C++ compiler (based on Emscripten) that allows programs to run in the browser unmodified. The BROWSIX case studies include complex applications, such as $\LaTeX$, which runs entirely in the browser without any source code modifications.

Unfortunately, BROWSIX is a JavaScript-only solution, since it was built before the release of WebAssembly. Moreover, BROWSIX suffers from high performance overhead, which would be a significant confounder while benchmarking. Using BROWSIX, it would be difficult to tease apart the poorly performing benchmarks from performance degradation introduced by BROWSIX.

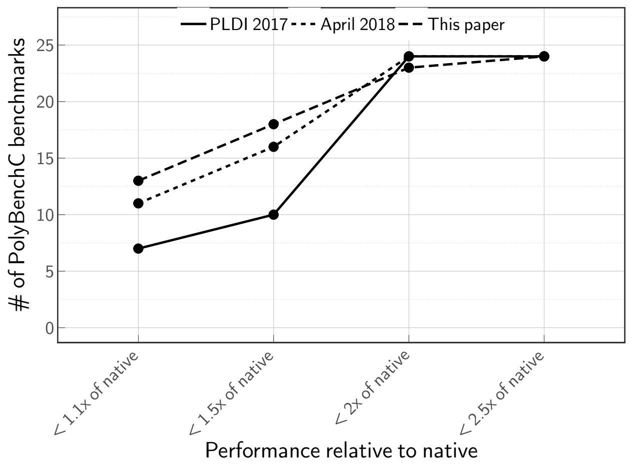


Figure 1: Number of PolyBenchC benchmarks performing within $x\times$ of native. In 2017 [18], seven benchmarks performed within $1.1\times$ of native. In April 2018, we found that 11 performed within $1.1\times$ of native. In May 2019, 13 performed with $1.1\times$ of native.

Contributions

- **BROWSIX-WASM:** We develop BROWSIX-WASM, a significant extension to and enhancement of BROWSIX that allows us to compile Unix programs to WebAssembly and run them in the browser with no modifications. In addition to integrating functional extensions, BROWSIX-WASM incorporates performance optimizations that drastically improve its performance, ensuring that CPU-intensive applications operate with virtually no overhead imposed by BROWSIX-WASM (§2).

- **BROWSIX-SPEC:** We develop BROWSIX-SPEC, a harness that extends BROWSIX-WASM to allow automated collection of detailed timing and hardware on-chip performance counter information in order to perform detailed measurements of application performance (§3).

- **Performance Analysis of WebAssembly:** Using BROWSIX-WASM and BROWSIX-SPEC, we conduct the first comprehensive performance analysis of WebAssembly using the SPEC CPU benchmark suite (both 2006 and 2017). This evaluation confirms that WebAssembly does run faster than JavaScript (on average 1.3× faster across SPEC CPU). However, contrary to prior work, we find a substantial gap between WebAssembly and native performance: code compiled to WebAssembly runs on average 1.55× slower in Chrome and 1.45× slower in Firefox than native code (§4).

- **Root Cause Analysis and Advice for Implementers:** We conduct a forensic analysis with the aid of performance counter results to identify the root causes of this performance gap. We find the following results:

  1. The instructions produced by WebAssembly have more loads and stores than native code (2.02× more loads and 2.30× more stores in Chrome; 1.92× more loads and 2.16× more stores in Firefox). We attribute this to reduced availability of registers, a sub-optimal register allocator, and a failure to effectively exploit a wider range of x86 addressing modes.
  2. The instructions produced by WebAssembly have more branches, because WebAssembly requires several dynamic safety checks.
  3. Since WebAssembly generates more instructions, it leads to more L1 instruction cache misses.

  We provide guidance to help WebAssembly implementers focus their optimization efforts in order to close the performance gap between WebAssembly and native code (§5, 6).

BROWSIX-WASM and BROWSIX-SPEC are available at `https://browsix.org`.

2 From BROWSIX to BROWSIX-WASM

BROWSIX [29] mimics a Unix kernel within the browser and includes a compiler (based on Emscripten [33, 39]) that compiles native programs to JavaScript. Together, they allow native programs (in C, C++, and Go) to run in the browser and freely use operating system services, such as pipes, processes, and a filesystem. However, BROWSIX has two major limitations that we must overcome. First, BROWSIX compiles native code to JavaScript and not WebAssembly. Second, the BROWSIX kernel has significant performance issues. In particular, several common system calls have very high overhead in BROWSIX, which makes it hard to compare the performance of a program running in BROWSIX to that of a program running natively. We address these limitations by building a new in-browser kernel called BROWSIX-WASM, which supports WebAssembly programs and eliminates the performance bottlenecks of BROWSIX.

Emscripten Runtime Modifications BROWSIX modifies the Emscripten compiler to allow processes (which run in WebWorkers) to communicate with the BROWSIX kernel (which runs on the main thread of a page). Since BROWSIX compiles native programs to JavaScript, this is relatively straightforward: each process' memory is a buffer that is shared with the kernel (a `SharedArrayBuffer`), thus system calls can directly read and write process memory. However, this approach has two significant drawbacks. First, it precludes growing the heap on-demand; the shared memory must be sized large enough to meet the high-water-mark heap size of the application for the entire life of the process. Second, JavaScript contexts (like the main context and each web worker context) have a fixed limit on their heap sizes, which is currently approximately 2.2 GB in Google Chrome [6]. This cap imposes a serious limitation on running multiple processes: if each process reserves a 500 MB heap, BROWSIX would only be able to run at most four concurrent processes. A deeper problem is that WebAssembly memory cannot be shared across WebWorkers and does not support the `Atomic` API, which BROWSIX processes use to wait for system calls.

BROWSIX-WASM uses a different approach to process-kernel communication that is also faster than the BROWSIX approach. BROWSIX-WASM modifies the Emscripten runtime system to create an auxiliary buffer (of 64MB) for each process that is shared with the kernel, but is distinct from process memory. Since this auxiliary buffer is a `SharedArrayBuffer` the BROWSIX-WASM process and kernel can use `Atomic` API for communication. When a system call references strings or buffers in the process's heap (e.g., `writev` or `stat`), its runtime system copies data from the process memory to the shared buffer and sends a message to the kernel with locations of the copied data in auxiliary memory. Similarly, when a system call writes data to the auxiliary buffer (e.g., `read`), its runtime system copies the data from the shared buffer to the process memory at the memory specified. Moreover, if a system call specifies a buffer in process memory for the kernel to write to (e.g., `read`), the runtime allocates a corresponding buffer in auxiliary memory and passes it to the kernel. In case the system call is either reading or writing data of size more than 64MB, BROWSIX-WASM divides this call into several calls such that each call only reads or writes at maximum 64MB of data. The cost of these memory copy operations is dwarfed by the overall cost of the system call invocation, which involves sending a message between process and kernel JavaScript contexts. We show in §4.2.1 that BROWSIX-WASM has negligible overhead.

Performance Optimization While building BROWSIX-WASM and doing our preliminary performance evaluation,

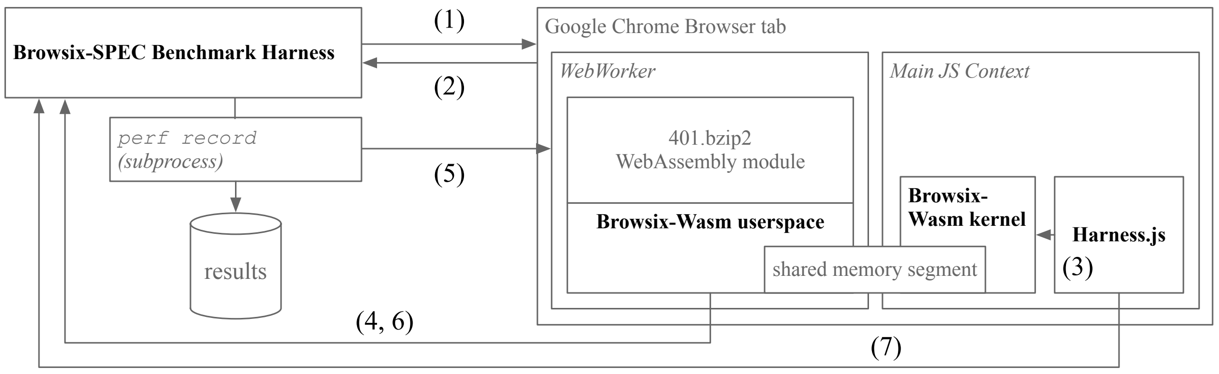


Figure 2: The framework for running SPEC benchmarks in browsers. **Bold** components are new or heavily modified (§3).

we discovered several performance issues in parts of the BROWSIX kernel. Left unresolved, these performance issues would be a threat to the validity of a performance comparison between WebAssembly and native code. The most serious case was in the shared filesystem component included with BROWSIX/BROWSIX-WASM, BROWSERFS. Originally, on each append operation on a file, BROWSERFS would allocate a new, larger buffer, copying the previous and new contents into the new buffer. Small appends could impose substantial performance degradation. Now, whenever a buffer backing a file requires additional space, BROWSERFS grows the buffer by at least 4 KB. This change alone decreased the time the `464.h264ref` benchmark spent in BROWSIX from 25 seconds to under 1.5 seconds. We made a series of improvements that reduce overhead throughout BROWSIX-WASM. Similar, if less dramatic, improvements were made to reduce the number of allocations and the amount of copying in the kernel implementation of pipes.

3 BROWSIX-SPEC

To reliably execute WebAssembly benchmarks while capturing performance counter data, we developed BROWSIX-SPEC. BROWSIX-SPEC works with BROWSIX-WASM to manage spawning browser instances, serving benchmark assets (e.g., the compiled WebAssembly programs and test inputs), spawning `perf` processes to record performance counter data, and validating benchmark outputs.

We use BROWSIX-SPEC to run three benchmark suites to evaluate WebAssembly's performance: SPEC CPU2006, SPEC CPU2017, and PolyBenchC. These benchmarks are compiled to native code using Clang 4.0, and WebAssembly using BROWSIX-WASM. We made no modifications to Chrome or Firefox, and the browsers are run with their standard sandboxing and isolation features enabled. BROWSIX-WASM is built on top of standard web platform features and requires no direct access to host resources – instead, benchmarks make standard HTTP requests to BROWSIX-SPEC.

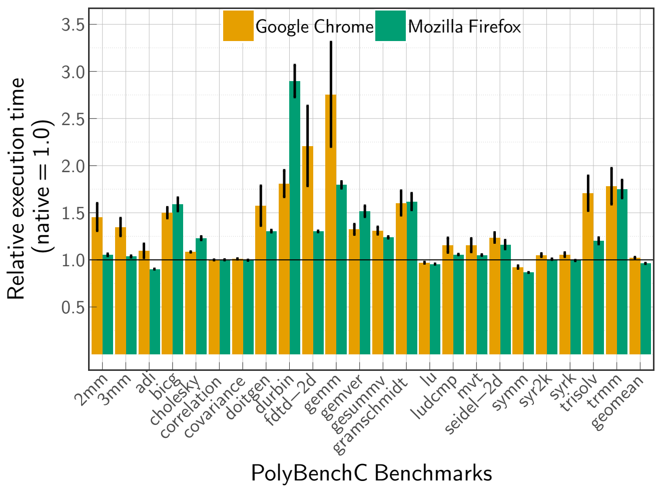
3.1 BROWSIX-SPEC Benchmark Execution

Figure 2 illustrates the key pieces of BROWSIX-SPEC in play when running a benchmark, such as `401.bzip2` in Chrome. First (1), the BROWSIX-SPEC benchmark harness launches a new browser instance using a WebBrowser automation tool, Selenium.[2] (2) The browser loads the page's HTML, harness JS, and BROWSIX-WASM kernel JS over HTTP from the benchmark harness. (3) The harness JS initializes the BROWSIX-WASM kernel and starts a new BROWSIX-WASM process executing the `runspec` shell script (not shown in Figure 2). `runspec` in turn spawns the standard `specinvoke` (not shown), compiled from the C sources provided in SPEC 2006. `specinvoke` reads the `speccmds.cmd` file from the BROWSIX-WASM filesystem and starts `401.bzip2` with the appropriate arguments. (4) After the WebAssembly module has been instantiated but before the benchmark's `main` function is invoked, the BROWSIX-WASM userspace runtime does an XHR request to BROWSIX-SPEC to begin recording performance counter stats. (5) The benchmark harness finds the Chrome thread corresponding to the Web Worker `401.bzip2` process and attaches `perf` to the process. (6) At the end of the benchmark, the BROWSIX-WASM userspace runtime does a final XHR to the benchmark harness to end the `perf record` process. When the `runspec` program exits (after potentially invoking the test binary several times), the harness JS POSTs (7) a `tar` archive of the SPEC results directory to BROWSIX-SPEC. After BROWSIX-SPEC receives the full results archive, it unpacks the results to a temporary directory and validates the output using the `cmp` tool provided with SPEC 2006. Finally, BROWSIX-SPEC kills the browser process and records the benchmark results.

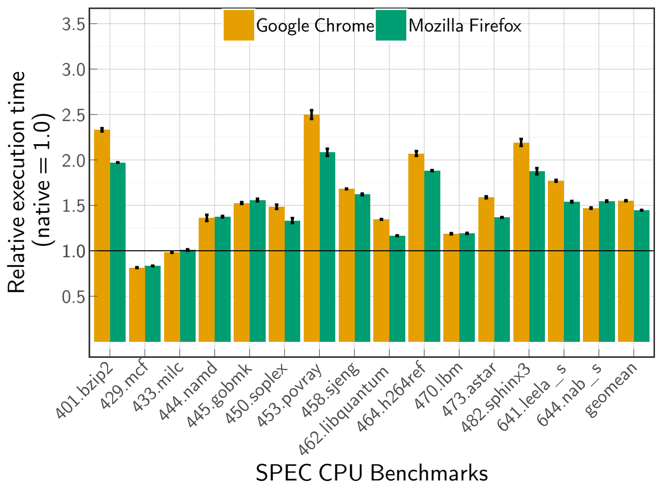
4 Evaluation

We use BROWSIX-WASM and BROWSIX-SPEC to evaluate the performance of WebAssembly using three benchmark

[2]`https://www.seleniumhq.org/`



(a)



(b)

Figure 3: The performance of the PolyBenchC and the SPEC CPU benchmarks compiled to WebAssembly (executed in Chrome and Firefox) relative to native, using BROWSIX-WASM and BROWSIX-SPEC. The SPEC CPU benchmarks exhibit higher overhead overall than the PolyBenchC suite, indicating a significant performance gap exists between WebAssembly and native.

suites: SPEC CPU2006, SPEC CPU2017, and PolyBenchC. We include PolybenchC benchmarks for comparison with the original WebAssembly paper [18], but argue that these benchmarks do not represent typical workloads. The SPEC benchmarks are representative and require BROWSIX-WASM to run successfully. We run all benchmarks on a 6-Core Intel Xeon E5-1650 v3 CPU with hyperthreading and 64 GB of RAM running Ubuntu 16.04 with Linux kernel v4.4.0. We run all benchmarks using two state-of-the-art browsers: Google Chrome 74.0 and Mozilla Firefox 66.0. We compile benchmarks to native code using Clang 4.0[3] and to WebAssembly using BROWSIX-WASM (which is based on Emscripten with Clang 4.0).[4] Each benchmark was executed five times. We report the average of all running times and the standard error. The execution time measured is the difference between wall clock time when the program starts, i.e. after WebAssembly JIT compilation concludes, and when the program ends.

4.1 PolyBenchC Benchmarks

Haas et al. [18] used PolybenchC to benchmark WebAssembly implementations because the PolybenchC benchmarks do not make system calls. As we have already argued, the PolybenchC benchmarks are small scientific kernels that are typically used to benchmark polyhedral optimization techniques, and do not represent larger applications. Nevertheless, it is still valuable for us to run PolybenchC with BROWSIX-WASM, because it demonstrates that our infrastructure for system calls does not have any overhead. Figure 3a shows the execution time of the PolyBenchC benchmarks in BROWSIX-WASM and when run natively. We are able to reproduce the majority of the results from the original WebAssembly paper [18]. We find that BROWSIX-WASM imposes a very low overhead: an average of 0.2% and a maximum of 1.2%.

4.2 SPEC Benchmarks

We now evaluate BROWSIX-WASM using the C/C++ benchmarks from SPEC CPU2006 and SPEC CPU2017 (the new C/C++ benchmarks and the speed benchmarks), which use system calls extensively. We exclude four data points that either do not compile to WebAssembly[5] or allocate more memory than WebAssembly allows.[6] Table 1 shows the absolute execution times of the SPEC benchmarks when running with BROWSIX-WASM in both Chrome and Firefox, and when running natively.

WebAssembly performs worse than native for all benchmarks except for `429.mcf` and `433.milc`. In Chrome, WebAssembly's maximum overhead is 2.5× over native and 7 out of 15 benchmarks have a running time within 1.5× of native. In Firefox, WebAssembly is within 2.08× of native and performs within 1.5× of native for 7 out of 15 benchmarks. On average, WebAssembly is 1.55× slower than native in Chrome, and 1.45× slower than native in Firefox. Table 2 shows the time required to compile the SPEC benchmarks

[3]The flags to Clang are `-O2 -fno-strict-aliasing`.

[4]BROWSIX-WASM runs Emscripten with the flags `-O2 -s TOTAL_MEMORY=1073741824 -s ALLOW_MEMORY_GROWTH=1 -fno-strict-aliasing`.

[5]`400.perlbench`, `403.gcc`, `471.omnetpp`, and `456.hmmer` from SPEC CPU2006 do not compile with Emscripten.

[6]From SPEC CPU2017, the `ref` dataset of `638.imagick_s` and `657.xz_s` require more than 4 GB RAM. However, these benchmarks do work with their `test` dataset.

Benchmark	Native	Google Chrome	Mozilla Firefox
401.bzip2	370 ± 0.6	864 ± 6.4	730 ± 1.3
429.mcf	221 ± 0.1	180 ± 0.9	184 ± 0.6
433.milc	375 ± 2.6	369 ± 0.5	378 ± 0.6
444.namd	271 ± 0.8	369 ± 9.1	373 ± 1.8
445.gobmk	352 ± 2.1	537 ± 0.8	549 ± 3.3
450.soplex	179 ± 3.7	265 ± 1.2	238 ± 0.5
453.povray	110 ± 1.9	275 ± 1.3	229 ± 1.5
458.sjeng	358 ± 1.4	602 ± 2.5	580 ± 2.0
462.libquantum	330 ± 0.8	444 ± 0.2	385 ± 0.8
464.h264ref	389 ± 0.7	807 ± 11.0	733 ± 2.4
470.lbm	209 ± 1.1	248 ± 0.3	249 ± 0.5
473.astar	299 ± 0.5	474 ± 3.5	408 ± 1.0
482.sphinx3	381 ± 7.1	834 ± 1.8	713 ± 3.6
641.leela_s	466 ± 2.7	825 ± 4.6	717 ± 1.2
644.nab_s	2476 ± 11	3639 ± 5.6	3829 ± 6.7
Slowdown: geomean	–	**1.55×**	**1.45×**
Slowdown: median	–	**1.53×**	**1.54×**

Table 1: Detailed breakdown of SPEC CPU benchmarks execution times (of 5 runs) for native (Clang) and WebAssembly (Chrome and Firefox); all times are in seconds. The median slowdown of WebAssembly is 1.53× for Chrome and 1.54× for Firefox.

Benchmark	Clang 4.0	Google Chrome
401.bzip2	1.9 ± 0.018	0.53 ± 0.005
429.mcf	0.3 ± 0.003	0.15 ± 0.005
433.milc	2.2 ± 0.02	0.3 ± 0.003
444.namd	4.6 ± 0.02	0.78 ± 0.004
445.gobmk	12.1 ± 0.2	1.4 ± 0.014
450.soplex	6.9 ± 0.01	1.2 ± 0.009
453.povray	15.3 ± 0.03	1.2 ± 0.012
458.sjeng	1.9 ± 0.01	0.35 ± 0.001
462.libquantum	6.9 ± 0.03	0.15 ± 0.002
464.h264ref	10.3 ± 0.06	1.0 ± 0.03
470.lbm	0.3 ± 0.001	0.14 ± 0.004
473.astar	0.73 ± 0.005	0.24 ± 0.004
482.sphinx3	3.0 ± 0.04	0.48 ± 0.007
641.leela_s	4.3 ± 0.05	0.74 ±0.003
644.nab_s	4.1 ± 0.03	0.41 ±0.001

Table 2: Compilation times of SPEC CPU benchmarks (average of 5 runs) for Clang 4.0 and WebAssembly (Chrome); all times are in seconds.

using Clang and Chrome. (To the best of our knowledge, Firefox cannot report WebAssembly compile times.) In all cases, the compilation time is negligible compared to the execution time. However, the Clang compiler is orders of magnitude slower than the WebAssembly compiler. Finally, note that Clang compiles benchmarks from C++ source code, whereas Chrome compiles WebAssembly, which is a simpler format than C++.

4.2.1 Browsix-Wasm Overhead

It is important to rule out the possibility that the slowdown that we report is due to poor performance in our implementation of Browsix-Wasm. In particular, Browsix-Wasm implements system calls without modifying the browser, and system calls involve copying data (§2), which may be costly. To quantify the overhead of Browsix-Wasm, we instrumented its system calls to measure all time spent in Browsix-Wasm. Figure 4 shows the percentage of time spent in Browsix-Wasm in Firefox using the SPEC benchmarks. For 14 of the 15 benchmarks, the overhead is less than 0.5%. The maximum overhead is 1.2%. On average, the overhead of Browsix-Wasm is only 0.2%. Therefore, we conclude that Browsix-Wasm has negligible overhead and does not substantially affect the performance counter results of programs executed in WebAssembly.

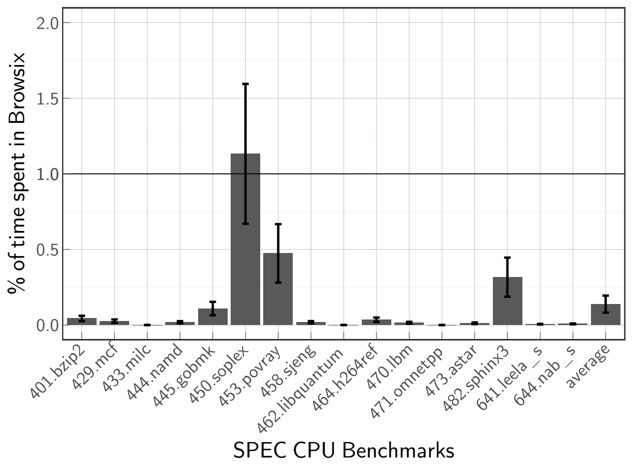


Figure 4: Time spent (in %) in Browsix-Wasm calls in Firefox for SPEC benchmarks compiled to WebAssembly. Browsix-Wasm imposes a mean overhead of only 0.2%.

4.2.2 Comparison of WebAssembly and `asm.js`

A key claim in the original work on WebAssembly was that it is significantly faster than `asm.js`. We now test that claim using the SPEC benchmarks. For this comparison, we modified Browsix-Wasm to also support processes compiled to `asm.js`. The alternative would have been to benchmark the `asm.js` processes using the original Browsix. However, as we discussed earlier, Browsix has performance problems that would have been a threat to the validity of our results. Figure 5 shows the speedup of the SPEC benchmarks using WebAssembly, relative to their running time using `asm.js` using both Chrome and Firefox. WebAssembly outperforms `asm.js` in both browsers: the mean speedup is 1.54× in

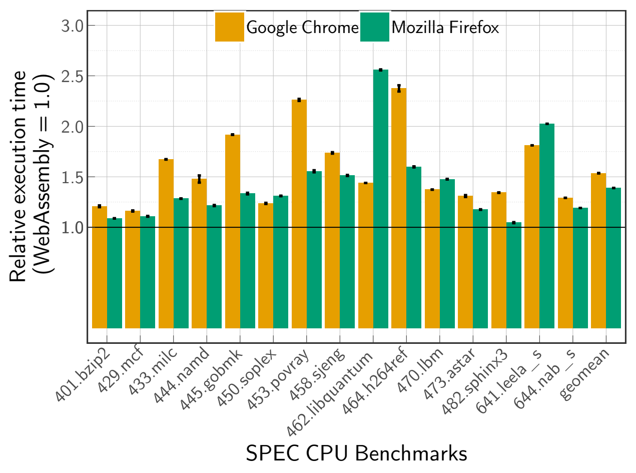


Figure 5: Relative time of `asm.js` to WebAssembly for Chrome and Firefox. WebAssembly is 1.54× faster than `asm.js` in Chrome and 1.39× faster than `asm.js` in Firefox.

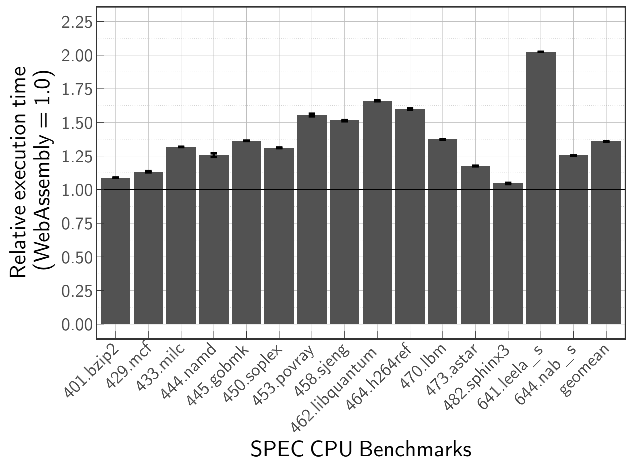


Figure 6: Relative *best* time of `asm.js` to the best time of WebAssembly. WebAssembly is 1.3× faster than `asm.js`.

Chrome and 1.39× in Firefox.

Since the performance difference between Chrome and Firefox is substantial, in Figure 6 we show the speedup of each benchmark by selecting the best-performing browser for WebAssembly and the best-performing browser of `asm.js` (i.e., they may be different browsers). These results show that WebAssembly consistently performs better than `asm.js`, with a mean speedup of 1.3×. Haas et al. [18] also found that WebAssembly gives a mean speedup of 1.3× over `asm.js` using PolyBenchC.

5 Case Study: Matrix Multiplication

In this section, we illustrate the performance differences between WebAssembly and native code using a C function that performs matrix multiplication, as shown in Figure 7a. Three matrices are provided as arguments to the function, and the results of `A` ($N_I \times N_K$) and `B` ($N_K \times N_J$) are stored in `C` ($N_I \times N_J$), where N_I, N_K, N_J are constants defined in the program.

In WebAssembly, this function is 2×–3.4× slower than native in both Chrome and Firefox with a variety of matrix sizes (Figure 8). We compiled the function with `-O2` and disabled automatic vectorization, since WebAssembly does not support vectorized instructions.

Figure 7b shows native code generated for the `matmul` function by `clang-4.0`. Arguments are passed to the function in the `rdi`, `rsi`, and `rdx` registers, as specified in the System V AMD64 ABI calling convention [9]. Lines 2 - 26 are the body of the first loop with iterator `i` stored in `r8d`. Lines 5 - 21 contain the body of the second loop with iterator `k` stored in `r9d`. Lines 10 - 16 comprise the body of the third loop with iterator `j` stored in `rcx`. Clang is able to eliminate a `cmp` instruction in the inner loop by initializing `rcx` with $-N_J$, incrementing `rcx` on each iteration at line 15, and using `jne` to test the zero flag of the status register, which is set to 1 when `rcx` becomes 0.

Figure 7c shows x86-64 code JITed by Chrome for the WebAssembly compiled version of `matmul`. This code has been modified slightly – `nops` in the generated code have been removed for presentation. Function arguments are passed in the `rax`, `rcx`, and `rdx` registers, following Chrome's calling convention. At lines 1– 3, the contents of registers `rax`, `rdx`, and `rcx` are stored on the stack, due to registers spills at lines 7 - 9. Lines 7–45 are the body of the first loop with iterator `i` stored in `edi`. Lines 18–42 contain the body of second loop with iterator `k` stored in `r11`. Lines 27–39 are the body of the third loop with iterator `j` stored in `eax`. To avoid memory loads due to register spilling at lines 7– 9 in the first iteration of the first loop, an extra jump is generated at line 5. Similarly, extra jumps are generated for the second and third loops at line 16 and line 25 respectively.

5.1 Differences

The native code JITed by Chrome has more instructions, suffers from increased register pressure, and has extra branches compared to Clang-generated native code.

5.1.1 Increased Code Size

The number of instructions in the code generated by Chrome (Figure 7c) is 53, including `nops`, while clang generated code (Figure 7b) consists of only 28 instructions. The poor instruction selection algorithm of Chrome is one of the reasons for increased code size.

Additionally, Chrome does not take advantage of all available memory addressing modes for x86 instructions. In Figure 7b Clang uses the `add` instruction at line 14 with register addressing mode, loading from and writing to a memory address in the same operation. Chrome on the other hand loads

```
 1 void matmul (int C[NI][NJ],
 2              int A[NI][NK],
 3              int B[NK][NJ]) {
 4   for (int i = 0; i < NI; i++) {
 5     for (int k = 0; k < NK; k++) {
 6       for (int j = 0; k < NJ; j++) {
 7         C[i][j] += A[i][k] * B[k][j];
 8       }
 9     }
10   }
11 }
```

(a) `matmul` source code in C.

```
 1 xor   r8d, r8d           #i <- 0
 2 L1:                      #start first loop
 3   mov   r10, rdx
 4   xor   r9d, r9d         #k <- 0
 5   L2:                    #start second loop
 6     imul  rax, 4*NK, r8
 7     add   rax, rsi
 8     lea   r11, [rax + r9*4]
 9     mov   rcx, -NJ     #j <- -NJ
10     L3:                  #start third loop
11       mov   eax, [r11]
12       mov   ebx, [r10 + rcx*4 + 4400]
13       imul  ebx, eax
14       add   [rdi + rcx*4 + 4*NJ], ebx
15       add   rcx, 1       #j <- j + 1
16     jne L3               #end third loop
17
18     add   r9,  1         #k <- k + 1
19     add   r10, 4*NK
20     cmp   r9,  NK
21   jne L2                 #end second loop
22
23   add  r8,  1            #i <- i + 1
24   add  rdi, 4*NJ
25   cmp  r8,  NI
26 jne L1                   #end first loop
27 pop  rbx
28 ret
```

(b) Native x86-64 code for `matmul` generated by Clang.

```
 1 mov [rbp-0x28],rax
 2 mov [rbp-0x20],rdx
 3 mov [rbp-0x18],rcx
 4 xor edi,edi                 #i <- 0
 5 jmp L1'
 6 L1:                         #start first loop
 7   mov ecx,[rbp-0x18]
 8   mov edx,[rbp-0x20]
 9   mov eax,[rbp-0x28]
10   L1':
11   imul r8d,edi,0x1130
12   add r8d,eax
13   imul r9d,edi,0x12c0
14   add r9d,edx
15   xor r11d,r11d             #k <- 0
16   jmp L2'
17   L2:                       #start second loop
18     mov ecx,[rbp-0x18]
19     L2':
20     imul r12d,r11d,0x1130
21     lea r14d,[r9+r11*4]
22     add r12d,ecx
23     xor esi,esi             #j <- 0
24     mov r15d,esi
25     jmp L3'
26     L3:                     #start third loop
27       mov r15d,eax
28       L3':
29       lea eax,[r15+0x1]     #j <- j + 1
30       lea edx,[r8+r15*4]
31       lea r15d,[r12+r15*4]
32       mov esi,[rbx+r14*1]
33       mov r15d,[rbx+r15*1]
34       imul r15d,esi
35       mov ecx,[rbx+rdx*1]
36       add ecx,r15d
37       mov [rbx+rdx*1],ecx
38       cmp eax,NJ            #j < NJ
39     jnz L3                  #end third loop
40   add r11,0x1               #k++
41   cmp r11d,NK               #k < NK
42   jnz L2                    #end second loop
43 add edi,0x1                 #i++
44 cmp edi,NI                  #i < NI
45 jnz L1                      #end first loop
46 retl
```

(c) x86-64 code JITed by Chrome from WebAssembly `matmul`.

Figure 7: Native code for `matmul` is shorter, has less register pressure, and fewer branches than the code JITed by Chrome. §6 shows that these inefficiencies are pervasive, reducing performance across the SPEC CPU benchmark suites.

the address in `ecx`, adds the operand to `ecx`, finally storing `ecx` at the address, requiring 3 instructions rather than one on lines 35−37.

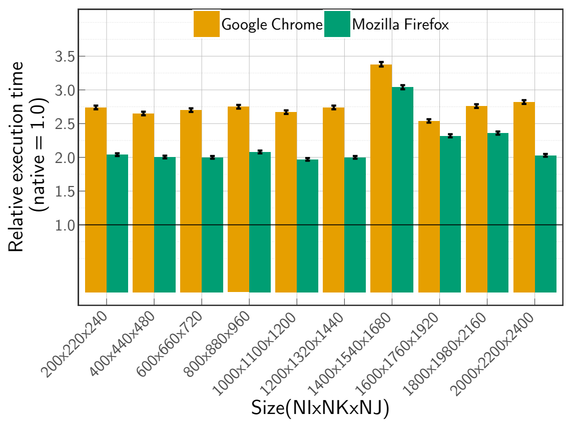


Figure 8: Performance of WebAssembly in Chrome and Firefox for different matrix sizes relative to native code. WebAssembly is always between 2× to 3.4× slower than native.

`perf` Event	Wasm Summary
`all-loads-retired (r81d0)` (Figure 9a) `all-stores-retired (r82d0)` (Figure 9b)	Increased register pressure
`branches-retired (r00c4)` (Figure 9c) `conditional-branches (r01c4)` (Figure 9d)	More branch statements
`instructions-retired (r1c0)` (Figure 9e) `cpu-cycles` (Figure 9f) `L1-icache-load-misses` (Figure 10)	Increased code size

Table 3: Performance counters highlight specific issues with WebAssembly code generation. When a raw PMU event descriptor is used, it is indicated by `rn`.

5.1.2 Increased Register Pressure

Code generated by Clang in Figure 7b does not generate any spills and uses only 10 registers. On the other hand, the code generated by Chrome (Figure 7c) uses 13 general purpose registers – all available registers (`r13` and `r10` are reserved by V8). As described in Section 5.1.1, eschewing the use of the register addressing mode of the `add` instruction requires the use of a temporary register. All of this register inefficiency compounds, introducing three register spills to the stack at lines 1–3. Values stored on the stack are loaded again into registers at lines 7–9 and line 18.

5.1.3 Extra Branches

Clang (Figure 7b) generates code with a single branch per loop by inverting the loop counter (line 15). In contrast, Chrome (Figure 7c) generates more straightforward code, which requires a conditional jump at the start of the loop. In addition, Chrome generates extra jumps to avoid memory loads due to register spills in the first iteration of a loop. For example, the jump at line 5 avoids the spills at lines 7– 9.

6 Performance Analysis

We use BROWSIX-SPEC to record measurements from all supported performance counters on our system for the SPEC CPU benchmarks compiled to WebAssembly and executed in Chrome and Firefox, and the SPEC CPU benchmarks compiled to native code (Section 3).

Table 3 lists the performance counters we use here, along with a summary of the impact of BROWSIX-WASM performance on these counters compared to native. We use these results to explain the performance overhead of WebAssembly over native code. Our analysis shows that the inefficiences described in Section 5 are pervasive and translate to reduced performance across the SPEC CPU benchmark suite.

6.1 Increased Register Pressure

This section focuses on two performance counters that show the effect of increased register pressure. Figure 9a presents the number of *load* instructions retired by WebAssembly-compiled SPEC benchmarks in Chrome and Firefox, relative to the number of load instructions retired in native code. Similarly, Figure 9b shows the number of *store* instructions retired. Note that a "retired" instruction is an instruction which leaves the instruction pipeline and its results are correct and visible in the architectural state (that is, not speculative).

Code generated by Chrome has 2.02× more load instructions retired and 2.30× more store instructions retired than native code. Code generated by Firefox has 1.92× more load instructions retired and 2.16× more store instructions retired than native code. These results show that the WebAssembly-compiled SPEC CPU benchmarks suffer from increased register pressure and thus increased memory references. Below, we outline the reasons for this increased register pressure.

6.1.1 Reserved Registers

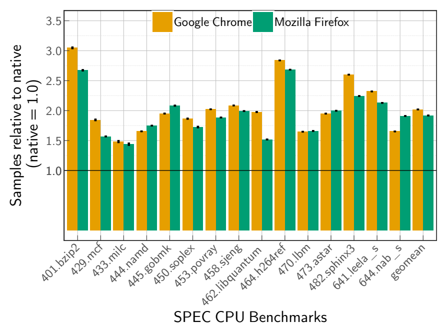
In Chrome, `matmul` generates three register spills but does not use two x86-64 registers: `r13` and `r10` (Figure 7c, lines 7– 9). This occurs because Chrome reserves these two registers.[7] For the JavaScript garbage collector, Chrome reserves `r13` to point to an array of GC roots at all times. In addition, Chrome uses `r10` and `xmm13` as dedicated scratch registers. Similarly, Firefox reserves `r15` as a pointer to the start of the heap, and `r11` and `xmm15` are JavaScript scratch registers.[8] None of these registers are available to WebAssembly code.

6.1.2 Poor Register Allocation

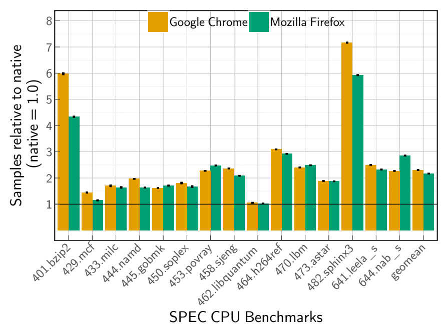
Beyond a reduced set of registers available to allocate, both Chrome and Firefox do a poor job of allocating the registers

[7] `https://github.com/v8/v8/blob/7.4.1/src/x64/register-x64.h`

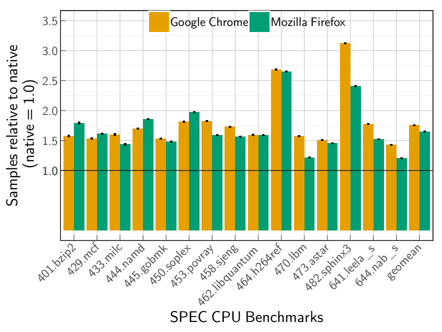
[8] `https://hg.mozilla.org/mozilla-central/file/tip/js/src/jit/x64/Assembler-x64.h`



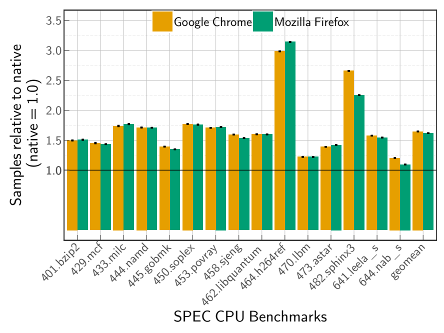
(a) `all-loads-retired`



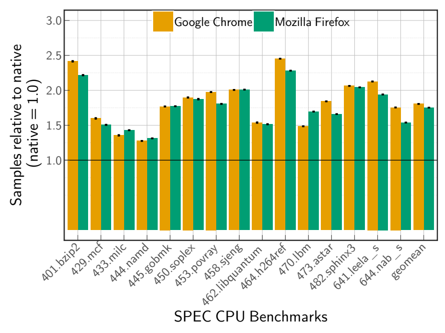
(b) `all-stores-retired`



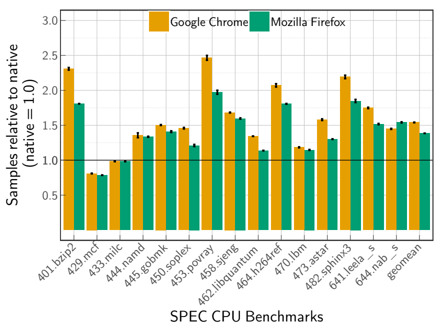
(c) `branch-instructions-retired`



(d) `conditional-branches`



(e) `instructions-retired`



(f) `cpu-cycles`

Figure 9: Processor performance counter samples for WebAssembly relative to native code.

they have. For example, the code generated by Chrome for `matmul` uses 12 registers while the native code generated by Clang only uses 10 registers (Section 5.1.2). This increased register usage—in both Firefox and Chrome—is because of their use of fast but not particularly effective register allocators. Chrome and Firefox both use a linear scan register allocator [36], while Clang uses a greedy graph-coloring register allocator [3], which consistently generates better code.

6.1.3 x86 Addressing Modes

The x86-64 instruction set offers several addressing modes for each operand, including a *register* mode, where the instruction reads data from register or writes data to a register, and memory address modes like *register indirect* or *direct offset* addressing, where the operand resides in a memory address and the instruction can read from or write to that address. A code generator could avoid unnecessary register pressure by using the latter modes. However, Chrome does not take advantage of these modes. For example, the code generated by Chrome for `matmul` does not use the register indirect addressing mode for the `add` instruction (Section 5.1.2), creating unnecessary register pressure.

6.2 Extra Branch Instructions

This section focuses on two performance counters that measure the number of branch instructions executed. Figure 9c shows the number of branch instructions retired by WebAssembly, relative to the number of branch instructions retired in native code. Similarly, Figure 9d shows the number of *conditional* branch instructions retired. In Chrome, there are 1.75× and 1.65× more unconditional and conditional branch instructions retired respectively, whereas in Firefox, there are 1.65× and 1.62× more retired. These results show that all the SPEC CPU benchmarks incur extra branches, and we explain why below.

6.2.1 Extra Jump Statements for Loops

As with `matmul` (Section 5.1.3), Chrome generates unnecessary jump statements for loops, leading to significantly more branch instructions than Firefox.

6.2.2 Stack Overflow Checks Per Function Call

A WebAssembly program tracks the current stack size with a global variable that it increases on every function call. The programmer can define the maximum stack size for the program. To ensure that a program does not overflow the stack, both Chrome and Firefox add stack checks at the start of each function to detect if the current stack size is less than the maximum stack size. These checks includes extra comparison and conditional jump instructions, which must be executed on every function call.

6.2.3 Function Table Indexing Checks

WebAssembly dynamically checks all indirect calls to ensure that the target is a valid function and that the function's type at runtime is the same as the type specified at the call site. In a WebAssembly module, the function table stores the list of functions and their types, and the code generated by WebAssembly uses the function table to implement these checks. These checks are required when calling function pointers and virtual functions in C/C++. The checks lead to extra comparison and conditional jump instructions, which are executed before every indirect function call.

6.3 Increased Code Size

The code generated by Chrome and Firefox is considerably larger than the code generated by Clang. We use three performance counters to measure this effect. (i) Figure 9e shows the number of *instructions retired* by benchmarks compiled to WebAssembly and executed in Chrome and Firefox relative to the number of instructions retired in native code. Similarly, Figure 9f shows the relative number of *CPU cycles* spent by benchmarks compiled to WebAssembly, and Figure 10 shows the relative number of *L1 instruction cache load misses*.

Figure 9e shows that Chrome executes an average of 1.80× more instructions than native code and Firefox executes an average of 1.75× more instructions than native code. Due to poor instruction selection, a poor register allocator generating more register spills (Section 6.1), and extra branch statements (Section 6.2), the size of generated code for WebAssembly is greater than native code, leading to more instructions being executed. This increase in the number of instructions executed leads to increased L1 instruction cache misses in Figure 10. On average, Chrome suffers 2.83× more I-cache misses than native code, and Firefox suffers from 2.04× more L1 instruction cache misses than native code. More cache misses means that more CPU cycles are spent waiting for the instruction to be fetched.

We note one anomaly: although `429.mcf` has 1.6× more instructions retired in Chrome than native code and 1.5× more instructions retired in Firefox than native code, it runs *faster* than native code. Figure 3b shows that its slowdown relative to native is 0.81× in Chrome and 0.83× in Firefox. The reason for this anomaly is attributable directly to its lower number of L1 instruction cache misses. `429.mcf` contains a main loop and most of the instructions in the loop fit in the L1 instruction cache. Similarly, `433.milc` performance is better due to fewer L1 instruction cache misses. In `450.soplex` there are 4.6× more L1 instruction cache misses in Chrome and Firefox than native because of several virtual functions being executed, leading to more indirect function calls.

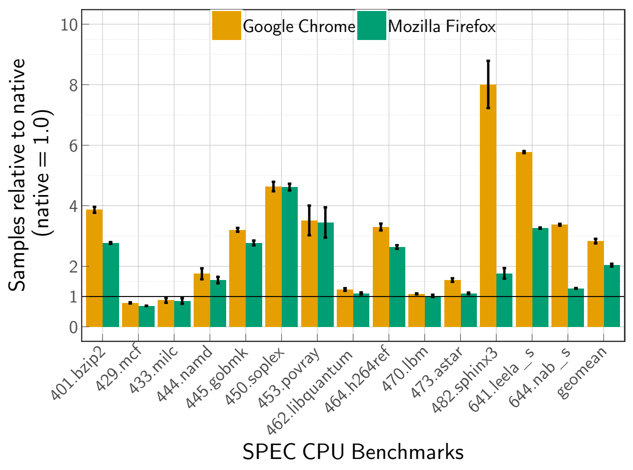


Figure 10: `L1-icache-load-misses` samples counted for SPEC CPU compiled to WebAssembly executed in Chrome and Firefox, relative to native. `458.sjeng` not shown in the graph exhibits 26.5× more L1 instruction cache misses in Chrome and 18.6× more in Firefox. The increased code size generated for WebAssembly leads to more instruction cache misses.

Performance Counter	**Chrome**	**Firefox**
`all-loads-retired`	2.02×	1.92×
`all-stores-retired`	2.30×	2.16×
`branch-instructions-retired`	1.75×	1.65×
`conditional-branches`	1.65×	1.62×
`instructions-retired`	1.80×	1.75×
`cpu-cycles`	1.54×	1.38×
`L1-icache-load-misses`	2.83×	2.04×

Table 4: The geomean of performance counter increases for the SPEC benchmarks using WebAssembly.

6.4 Discussion

It is worth asking if the performance issues identified here are fundamental. We believe that two of the identified issues are not: that is, they could be ameliorated by improved implementations. WebAssembly implementations today use register allocators (§6.1.2) and code generators (§6.2.1) that perform worse than Clang's counterparts. However, an offline compiler like Clang can spend considerably more time to generate better code, whereas WebAssembly compilers must be fast enough to run online. Therefore, solutions adopted by other JITs, such as further optimizing hot code, are likely applicable here [19, 32].

The four other issues that we have identified appear to arise from the design constraints of WebAssembly: the stack overflow checks (§6.2.2), indirect call checks (§6.2.3), and reserved registers (§6.1.1) have a runtime cost and lead to increased code size (§6.3). Unfortunately, these checks are necessary for WebAssembly's safety guarantees. A redesigned WebAssembly, with richer types for memory and function pointers [23], might be able to perform some of these checks at compile time, but that could complicate the implementation of compilers that produce WebAssembly. Finally, a WebAssembly implementation in a browser must interoperate with a high-performance JavaScript implementation, which may impose its own constraints. For example, current JavaScript implementations reserve a few registers for their own use, which increases register pressure on WebAssembly.

7 Related Work

Precursors to WebAssembly There have been several attempts to execute native code in browsers, but none of them met all the design criteria of WebAssembly.

ActiveX [13] allows web pages to embed signed x86 libraries, however these binaries have unrestricted access to the Windows API. In contrast, WebAssembly modules are sandboxed. ActiveX is now a deprecated technology.

Native Client [11, 37] (NaCl) adds a module to a web application that contains platform specific machine code. NaCl introduced sandboxing techniques to execute platform specific machine code at near native speed. Since NaCl relies on static validation of machine code, it requires code generators to follow certain patterns, hence, supporting only a subset of the x86, ARM, and MIPS instructions sets in the browser. To address the inherent portability issue of NaCl, Portable NaCl (PNaCl) [14] uses LLVM Bitcode as a binary format. However, PNaCl does not provide significant improvement in compactness over NaCl and still exposes compiler and/or platform-specific details such as the call stack layout. Both have been deprecated in favor of WebAssembly.

`asm.js` is a subset of JavaScript designed to be compiled efficiently to native code. `asm.js` uses type coercions to avoid the dynamic type system of JavaScript. Since `asm.js` is a subset of JavaScript, adding all native features to `asm.js` such as 64-bit integers will first require extending JavaScript. Compared to `asm.js`, WebAssembly provides several improvements: (i) WebAssembly binaries are compact due to its lightweight representation compared to JavaScript source, (ii) WebAssembly is more straightforward to validate, (iii) WebAssembly provides formal guarantees of type safety and isolation, and (iv) WebAssembly has been shown to provide better performance than `asm.js`.

WebAssembly is a stack machine, which is similar to the Java Virtual Machine [21] and the Common Language Runtime [25]. However, WebAssembly is very different from these platforms. For example WebAssembly does not support objects and does not support unstructured control flow.

The WebAssembly specification defines its operational semantics and type system. This proof was mechanized using the Isabelle theorem prover, and that mechanization effort found and addressed a number of issues in the specification [35]. RockSalt [22] is a similar verification effort for NaCl. It implements the NaCl verification toolchain in Coq, along with a proof of correctness with respect to a model of the subset of x86 instructions that NaCl supports.

Analysis of SPEC Benchmarks using performance counters Several papers use performance counters to analyze the SPEC benchmarks. Panda et al. [26] analyze the SPEC CPU2017 benchmarks, applying statistical techniques to identify similarities among benchmarks. Phansalkar et al. perform a similar study on SPEC CPU2006 [27]. Limaye and Adegija identify workload differences between SPEC CPU2006 and SPEC CPU2017 [20].

8 Conclusions

This paper performs the first comprehensive performance analysis of WebAssembly. We develop BROWSIX-WASM, a significant extension of BROWSIX, and BROWSIX-SPEC, a harness that enables detailed performance analysis, to let us run the SPEC CPU2006 and CPU2017 benchmarks as WebAssembly in Chrome and Firefox. We find that the mean slowdown of WebAssembly vs. native across SPEC benchmarks is 1.55× for Chrome and 1.45× for Firefox, with peak slowdowns of 2.5× in Chrome and 2.08× in Firefox. We identify the causes of these performance gaps, providing actionable guidance for future optimization efforts.

Acknowledgements We thank the reviewers and our shepherd, Eric Eide, for their constructive feedback. This work was partially supported by NSF grants 1439008 and 1413985.

References

[1] Blazor. https://blazor.net/. [Online; accessed 5-January-2019].

[2] Compiling from Rust to WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm. [Online; accessed 5-January-2019].

[3] LLVM Reference Manual. https://llvm.org/docs/CodeGenerator.html.

[4] NaCl and PNaCl. https://developer.chrome.com/native-client/nacl-and-pnacl. [Online; accessed 5-January-2019].

[5] PolyBenchC: the polyhedral benchmark suite. http://web.cs.ucla.edu/~pouchet/software/polybench/. [Online; accessed 14-March-2017].

[6] Raise Chrome JS heap limit? - Stack Overflow. https://stackoverflow.com/questions/43643406/raise-chrome-js-heap-limit. [Online; accessed 5-January-2019].

[7] Use cases. https://webassembly.org/docs/use-cases/.

[8] WebAssembly. https://webassembly.org/. [Online; accessed 5-January-2019].

[9] System V Application Binary Interface AMD64 Architecture Processor Supplement. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2013.

[10] Steve Akinyemi. A curated list of languages that compile directly to or have their VMs in WebAssembly. https://github.com/appcypher/awesome-wasm-langs. [Online; accessed 5-January-2019].

[11] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 355–366. ACM, 2011.

[12] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A Trace-based JIT Compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 708–725. ACM, 2010.

[13] David A Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[14] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. PNaCl: Portable Native Client Executables. https://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf, 2010.

[15] Brendan Eich. From ASM.JS to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/, 2015. [Online; accessed 5-January-2019].

[16] Eric Elliott. What is WebAssembly? https://tinyurl.com/o5h6daj, 2015. [Online; accessed 5-January-2019].

[17] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478. ACM, 2009.

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200. ACM, 2017.

[19] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, 2008.

[20] Ankur Limaye and Tosiron Adegbija. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, 2018.

[21] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[22] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404. ACM, 2012.

[23] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[24] Richard Musiol. A compiler from Go to JavaScript for running Go code in a browser. `https://github.com/gopherjs/gopherjs`, 2016. [Online; accessed 5-January-2019].

[25] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. Nigel Horspool, editor, *Compiler Construction*, pages 213–228. Springer, 2002.

[26] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, 2018.

[27] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 412–423. ACM, 2007.

[28] Bobby Powers, John Vilk, and Emery D. Berger. Browsix: Unix in your browser tab. `https://browsix.org`.

[29] Bobby Powers, John Vilk, and Emery D. Berger. Browsix: Bridging the Gap Between Unix and the Browser. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 253–266. ACM, 2017.

[30] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12. ACM, 2010.

[31] Marija Selakovic and Michael Pradel. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 61–72. ACM, 2016.

[32] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-in-time Compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 180–195. ACM, 2001.

[33] Luke Wagner. asm.js in Firefox Nightly | Luke Wagner's Blog. `https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/`. [Online; accessed 21-May-2019].

[34] Luke Wagner. A WebAssembly Milestone: Experimental Support in Multiple Browsers. `https://hacks.mozilla.org/2016/03/a-webassembly-milestone/`, 2016. [Online; accessed 5-January-2019].

[35] Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65. ACM, 2018.

[36] Christian Wimmer and Michael Franz. Linear Scan Register Allocation on SSA Form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 170–179. ACM, 2010.

[37] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 2009.

[38] Alon Zakai. asm.js. `http://asmjs.org/`. [Online; accessed 5-January-2019].

[39] Alon Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312. ACM, 2011.