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
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]. 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.ml, 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.

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 ($\S$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 ($\S$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 the 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,
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 append 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 suites. 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.

²https://www.seleniumhq.org/
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 2 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.

\(^5\)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.
## 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.

### 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

### 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.

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

| Slowdown: geomean   | 1.55×   |
| Slowdown: median    | 1.53×   | 1.54×   |

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

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

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%.
WebAssembly gives a mean speedup of $1.3 \times$ over \texttt{asm.js} that WebAssembly consistently performs better than \texttt{asm.js} with a mean speedup of $1.3 \times$ (i.e., they may be different browsers). These results show that WebAssembly 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 $2 \times 3.4 \times$ slower than native in both Chrome and Firefox with a variety of matrix sizes (Figure 8). We compiled the function with --no and disabled automatic vectorization, since WebAssembly does not support vectorized instructions.

Figure 5: Relative time of \texttt{asm.js} to WebAssembly for Chrome and Firefox. WebAssembly is $1.54 \times$ faster than \texttt{asm.js} in Chrome and $1.39 \times$ faster than \texttt{asm.js} in Firefox.

Figure 6: Relative \textit{best} time of \texttt{asm.js} to the best time of WebAssembly. WebAssembly is $1.3 \times$ faster than \texttt{asm.js} in Chrome and $1.39 \times$ 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 \texttt{asm.js} (i.e., they may be different browsers). These results show that WebAssembly consistently performs better than \texttt{asm.js}, with a mean speedup of $1.3 \times$. Haas et al. [18] also found that WebAssembly gives a mean speedup of $1.3 \times$ over \texttt{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
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 $2 \times$ to $3.4 \times$ slower than native.

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

<table>
<thead>
<tr>
<th>perf Event</th>
<th>Wasm Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-loads-retired (r$1d0$) (Figure 9a)</td>
<td>Increased register</td>
</tr>
<tr>
<td>all-stores-retired (r$2d0$) (Figure 9b)</td>
<td>pressure</td>
</tr>
<tr>
<td>branches-retired (r$0uc4$) (Figure 9c)</td>
<td>More branch statements</td>
</tr>
<tr>
<td>conditional-branches (r$0ic4$) (Figure 9d)</td>
<td></td>
</tr>
<tr>
<td>instructions-retired (r$1c0$) (Figure 9e)</td>
<td>Increased code size</td>
</tr>
<tr>
<td>cpu-cycles (Figure 9f)</td>
<td></td>
</tr>
<tr>
<td>L1-cache-load-misses (Figure 10)</td>
<td></td>
</tr>
</tbody>
</table>

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 $2.02 \times$ more load instructions retired and $2.30 \times$ more store instructions retired than native code. Code generated by Firefox has $1.92 \times$ more load instructions retired and $2.16 \times$ 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: r$13$ and r$10$ (Figure 7c, lines 7–9). This occurs because Chrome reserves these two registers.\footnote{https://github.com/v8/v8/blob/7.4.1/src/x64/register-x64.h} For the JavaScript garbage collector, Chrome reserves r$13$ to point to an array of GC roots at all times. In addition, Chrome uses r$10$ and xmm$13$ as dedicated scratch registers. Similarly, Firefox reserves r$15$ as a pointer to the start of the heap, and r$11$ and xmm$15$ are JavaScript scratch registers.\footnote{https://hg.mozilla.org/mozilla-central/file/tip/js/src/jit/x64/Assembler-x64.h} 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...
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 include 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 L1 cache misses than native code, and Firefox suffers from 2.04× more L1 instruction cache misses than native code. More cache misses mean 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

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.

<table>
<thead>
<tr>
<th>Performance Counter</th>
<th>Chrome</th>
<th>Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-loads-retired</td>
<td>2.02×</td>
<td>1.92×</td>
</tr>
<tr>
<td>all-stores-retired</td>
<td>2.30×</td>
<td>2.16×</td>
</tr>
<tr>
<td>branch-instructions-retired</td>
<td>1.75×</td>
<td>1.65×</td>
</tr>
<tr>
<td>conditional-branches</td>
<td>1.65×</td>
<td>1.62×</td>
</tr>
<tr>
<td>instructions-retired</td>
<td>1.80×</td>
<td>1.75×</td>
</tr>
<tr>
<td>cpu-cycles</td>
<td>1.54×</td>
<td>1.38×</td>
</tr>
<tr>
<td>L1-icache-load-misses</td>
<td>2.83×</td>
<td>2.04×</td>
</tr>
</tbody>
</table>

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

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.

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


