Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code

Abhinav Jangda
University of Massachusetts Amherst

Bobby Powers
University of Massachusetts Amherst

Arjun Guha
University of Massachusetts Amherst

Emery Berger
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 [22] 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 50% (Firefox) to 89% (Chrome), with peak slowdowns of 2.6× (Firefox) and 3.14× 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 client-side application developers. Until recently, the only programming language available to web developers was JavaScript. Beyond its various quirks and pitfalls from a programming languages perspective, JavaScript is also notoriously difficult to compile efficiently. 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* (a.k.a. *Wasm*).

WebAssembly is specifically intended to serve as a universal compilation target for web browsers [13, 14, 15][1]. Unlike past approaches to achieve native or near-native speed for code running inside the browser (e.g., ActiveX [11], Google Native Client and Portable Native Client [4, 12], and `asm.js` [26]), WebAssembly is intended to not only be fast but also to be safe (providing formal safety guarantees) and highly portable. It is now supported by all major browsers [7, 23].

Adoption of WebAssembly has been swift; compilers and runtimes targeting Wasm now support a wide range of programming languages, including C, C++, C#, Go, and Rust [1, 2, 18, 27]. A curated list currently includes more than a dozen others [9]. Today, code written in these languages can be safely executed in browser sandboxes across any modern device once compiled to WebAssembly.

Because WebAssembly is low level and strictly typed, it offers the promise of higher performance than JavaScript. The paper introducing WebAssembly [15] presents results showing that code compiled to WebAssembly executing on Google Chrome's V8 engine runs 34% faster than code compiled to JavaScript (`asm.js`). It also achieves competitive performance versus native code: of the 24 benchmarks evaluated, seven run within 10% of native execution and almost all of them run less than 2× slower.

Challenges of Benchmarking WebAssembly: While these results are promising, the evaluation conducted in this paper is severely limited. It relies exclusively on the PolyBenchC benchmark suite [5]. This suite, meant to measure the effect of polyhedral loop optimizations, consists of a number of small scientific computing kernels like matrix multiplication. Each benchmark is roughly 100 lines of code. These workloads are not necessarily representative of applications that target the browser.

[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 as described in [15] and currently supported by all major browsers.

A more comprehensive evaluation would use an established, large-scale benchmark suite consisting of large programs such as SPEC CPU. However, it is not generally possible to run larger applications inside the browser. Compiling them to WebAssembly is not enough: browsers lack support for the kind of system services that applications expect, such as a file system, synchronous I/O, and processes.

For example, the SPEC benchmark suite reads in files from the file system and executes applications at the command line, spawning processes that expect a Unix environment. WebAssembly does not provide support for any of these features. Instead, WebAssembly relies on the JavaScript environment it is embedded in to provide callable functions implementing such interactions, and the browser provides no such support.

The standard approach to running these applications today is to use Emscripten, a toolchain for compiling C and C++ to WebAssembly [27]. 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.

One promising alternative is BROWSIX, a framework that enables running unmodified, full-featured Unix applications in the browser [21, 22]. BROWSIX comprises both compiler support and a Unix kernel written in JavaScript. BROWSIX integrates modified versions of both the Emscripten compiler (for C/C++) and GopherJS (for Go) compilers that allows applications compiled to JavaScript to run with the BROWSIX kernel. BROWSIX can run applications as complex and full-featured as $\LaTeX$without any code modifications.

Unfortunately, BROWSIX is a JavaScript-only solution: it predates the introduction of WebAssembly, and only supports the execution of code compiled to JavaScript. It also suffers from high performance overhead which not only slows program execution but also acts as a confounder that would make it difficult to tease apart the performance of compiled code from the framework itself.

Contributions: This paper makes the following contributions.

- **BROWSIX-WASM:** We develop BROWSIX-WASM, a significant extension to and enhancement of BROWSIX that enables running directly in the browser Unix programs compiled to WebAssembly. 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). We find that while WebAssembly runs faster than JavaScript (on average 1.3× faster across SPEC CPU), there remains a substantial gap between WebAssembly and native performance: code compiled to WebAssembly runs on average 1.5× slower in Firefox and 1.9× slower in Chrome than native code (§4).

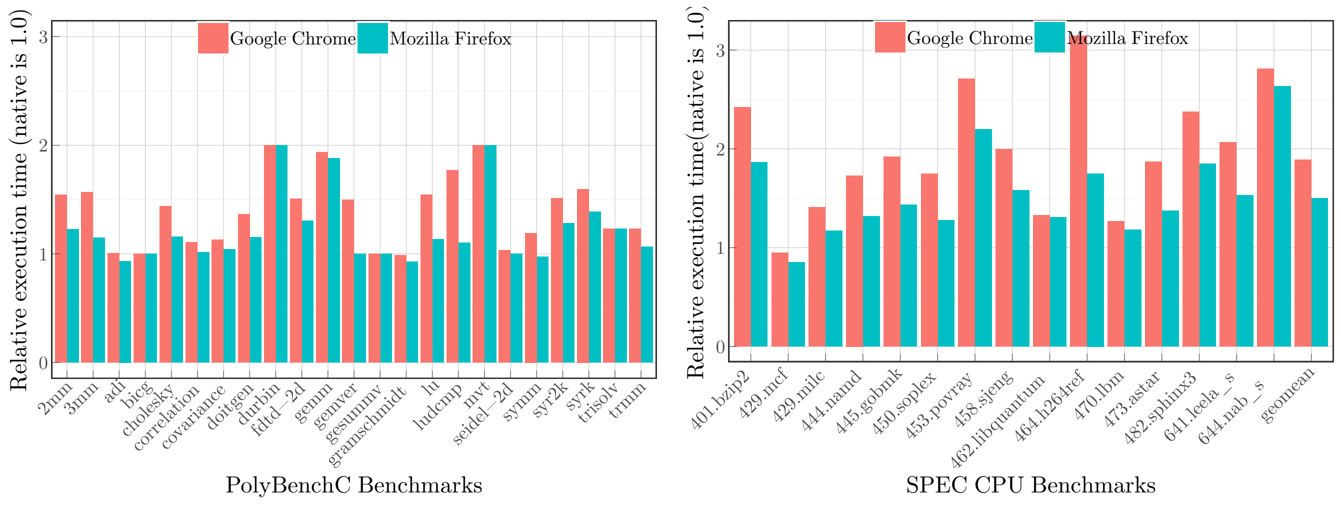
- **Root Cause Analysis and Advice for Implementers:** We conduct a forensic analysis with the aid of performance counter results to identify the root causes of this performance gap. We find the following results: (1) code compiled to WebAssembly yields more loads and stores than native code (2.1× more loads and 2× more stores in Chrome; 1.6× more loads and 1.7× 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) increased code sizes lead to more instructions being executed and more L1 instruction cache misses; and (3) generated code has more branches due to safety checks for overflow and indirect function calls. We provide guidance to WebAssembly implementers on where to focus their optimization efforts to close the performance gap between WebAssembly and native code (§5, 6).

2 From BROWSIX to BROWSIX-WASM

To enable running code compiled to WebAssembly, we build a new framework, BROWSIX-WASM, that extends both the Emscripten toolchain and the BROWSIX kernel. This section describes the extensions and enhancements that BROWSIX-WASM comprises.

BROWSIX comprises modified compiler support and a JavaScript-only Unix kernel that lets it run unmodified applications written in a range of languages (including C, C++, and Go). The compiler component is based on Emscripten [17, 27], a backend for LLVM and clang that can emit JavaScript code. BROWSIX itself predates WebAssembly and only supported JavaScript. BROWSIX-WASM required significant modifications both to the compiler and kernel, which we outline below:

Emscripten Runtime Modifications: The Emscripten integration with BROWSIX-WASM required a complete rewrite to support programs compiled to WebAssembly. For `asm.js` modules, BROWSIX depended on a `SharedArrayBuffer`



(a) (b)

Figure 1: The performance of the PolyBenchC and the SPEC CPU benchmarks compiled to WebAssembly (executed in Firefox and Chrome) 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.

rather than a `ArrayBuffer` for a process's linear memory to enable both the process and the kernel to read from and write to the heap.

In addition to the fact that WebAssembly instances do not yet support backing linear memory with a `SharedArrayBuffer`, 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.

To overcome these limitations and support WebAssembly at the same time, we modified the Emscripten runtime to create a `SharedArrayBuffer` for communication between the kernel and process separate from the process's linear memory. For system calls that reference strings or buffers in the process's heap, such as `writev` and `stat`, data is copied from the linear memory into the `SharedArrayBuffer` (or out of it, on syscall return, for syscalls like `readv`). The cost of this memory copy is dwarfed by the current method of invoking the system call: sending a message between JavaScript contexts that results in the kernel's syscall handler callback being scheduled on the event loop of the main JavaScript context. We show in §4.2.1 that overhead of BROWSIX-WASM on WebAssembly programs is minimal.

Performance Optimization: During the development of BROWSIX-WASM and during our initial evaluations with the SPEC benchmark suite, we discovered significant and previously unknown performance issues in core parts of the BROWSIX kernel. 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 that 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 a test harness called BROWSIX-SPEC. BROWSIX-SPEC works with BROWSIX-WASM to manage spawning browser instances, serving the benchmark assets (such as 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 Selenium. (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 2016. `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 begins monitoring performance counters for it. (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 three benchmark suites in our evaluation: SPEC CPU2006, SPEC CPU2017 and PolyBenchC (§3). We include all benchmarks in the PolyBenchC benchmark suite and all C/C++ benchmarks in SPEC CPU2006 benchmark suite except `400.perlbench` and `403.gcc`, which we were unable to compile with Emscripten. We include only the speed benchmarks of SPEC CPU2017 and the new C/C++ benchmarks added to SPEC CPU2017. Although there are four new C/C++ benchmarks added to SPEC CPU2017, we could not execute the `ref` dataset of `638.imagick_s` and `657.xz_s` in WebAssembly because these benchmarks allocate more memory than the 4GB memory WebAssembly can reference. Both of these benchmarks do work under BROWSIX-WASM with `test` dataset, showing that BROWSIX-WASM can support the SPEC CPU benchmarks. All benchmarks were executed on a system with 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. Benchmarks were compiled to native code using clang 4.0 with flags `-O2 -fno-strict-aliasing`. BROWSIX-WASM (with Emscripten based on clang/llvm 4.0) was used to compile all benchmarks to WebAssembly with flags `-O2 -s TOTAL_MEMORY=1073741824 -s ALLOW_MEMORY_GROWTH=1 -fno-strict-aliasing`. WebAssembly code was executed in two state-of-the-art browsers: Google Chrome 67.0 and Mozilla Firefox 64.0.

4.1 PolyBenchC Benchmarks

Due to no support for system calls, WebAssembly could only support benchmarks using no system calls like PolybenchC used to evaluate WebAssembly with native in [15]. PolybenchC benchmarks are small scientific kernels and are standard benchmarks for polyhedral techniques but they are not representative of larger applications. We reproduce previous results [15] in our benchmark setup and shows that BROWSIX-WASM imposes minimal overhead.

We executed PolyBenchC benchmarks with BROWSIX-WASM and without BROWSIX-WASM, to determine if BROWSIX-WASM introduces any overhead over these benchmarks. In addition to compiling PolyBenchC benchmarks using BROWSIX-WASM, these benchmarks were also compiled using vanilla Emscripten. Figure 1a shows the execution time of each PolyBench benchmark compiled to WebAssembly relative to native. We are able to reproduce majority of the results from [15]. We found that BROWSIX-WASM imposes a mean overhead of 0.6% and maximum overhead of 1%. Since, PolyBenchC Benchmarks does not perform any system calls other than allocating memory, we expected to find minimal overhead.

4.2 SPEC Benchmarks

Figure 1b shows the execution time of SPEC CPU Benchmarks compiled to WebAssembly and executed under BROWSIX-WASM in Firefox and Chrome relative to the execution time of native code. Table 1 shows the absolute execution times of SPEC Benchmarks compiled to WebAssembly and executed under BROWSIX-WASM in Firefox and Chrome and native code execution times.

WebAssembly performs worse than native for all benchmarks except for `429.mcf`. In Firefox, WebAssembly is within 2.6× of native and for 9 out of 15 performing with 1.5× of native. In Chrome, the WebAssembly gives max-

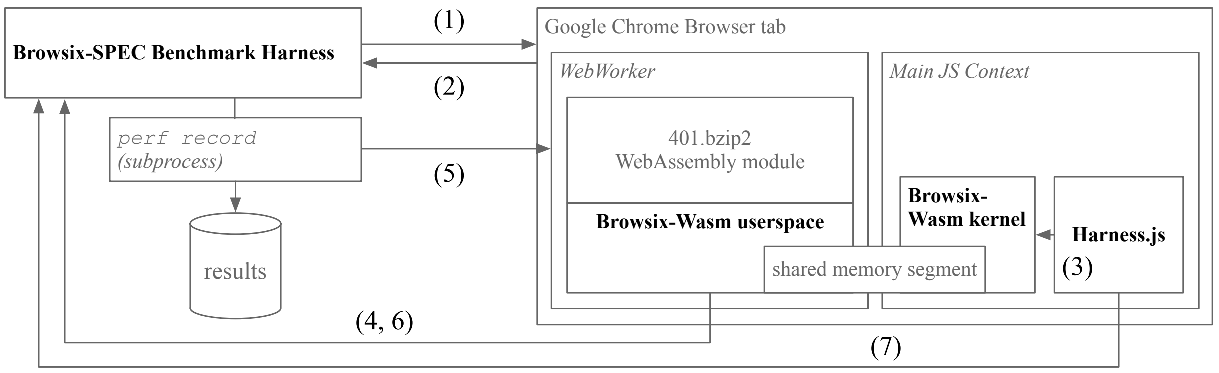


Figure 2: **Running and observing SPEC in the browser.** The parts in **bold** were created (BROWSIX-SPEC benchmark harness, benchmark JS) or heavily expanded (BROWSIX-WASM) in this paper. See Section 3 for a detailed description.

Benchmark	**Native**	**Mozilla Firefox**	**Google Chrome**
`401.bzip2`	365 ± 3.6	681 ± 5.1	883 ± 7.5
`429.mcf`	220 ± 2.1	188 ± 1.9	209 ± 2.1
`429.milc`	357 ± 2.3	418 ± 4.8	503 ± 7.1
`444.namd`	267 ± 3.1	352 ± 2.4	462 ± 6.2
`445.gobmk`	343 ± 2.8	492 ± 4.5	658 ± 4.3
`450.soplex`	180 ± 1.3	230 ± 2.7	315 ± 4.4
`453.povray`	106 ± 0.9	233 ± 2.1	287 ± 2.9
`458.sjeng`	350 ± 3.6	554 ± 6.2	699 ± 7.9
`462.libquantum`	335 ± 2.7	438 ± 6.8	445 ± 4.4
`464.h264ref`	389 ± 4.5	681 ± 7.7	1223± 9.8
`470.lbm`	215 ± 1.4	254 ± 2.5	273 ± 2.7
`473.astar`	279 ± 1.4	383 ± 3.8	521 ± 5.2
`482.sphinx3`	362 ± 3.6	670 ± 6.7	860 ± 8.1
`641.leela_s`	466 ± 5.6	714 ± 6.4	963 ± 9.8
`644.nab_s`	1464± 8.4	3853± 18	4118± 24
Slowdown: geomean	-	**1.5×**	**1.9×**
Slowdown: median	-	**1.43×**	**1.92×**

Table 1: Detailed breakdown of SPEC CPU Benchmarks execution times (of 5 runs) for native (clang) and WebAssembly (Firefox and Chrome); all times are in seconds. Firefox is always faster than Chrome. The median slowdown of WebAssembly is 1.43× for Firefox, and 1.92× for Chrome.

imum overhead of 3.1× over native and only 4 out of 15 benchmarks running within 1.5× over native. On average WebAssembly in Firefox runs at 1.9× over native code and in Chrome runs at 1.75× over native code. For all benchmarks except `462.libquantum`, WebAssembly in Chrome performs worse than in Firefox. On average WebAssembly in Chrome runs at 1.27× slower than Firefox.

4.2.1 BROWSIX-WASM Overhead

The amount of time spent in BROWSIX-WASM for a program contains (i) the time taken to execute all system calls in the program, and (ii) extra copy overhead (§2). To determine the BROWSIX-WASM overhead we performed experiment to retrieve the amount of time spent in BROWSIX-WASM. Figure 3 shows the percentage of time spent in BROWSIX-WASM in Firefox when executing SPEC Benchmarks. For 10 out of 15 benchmarks, the overhead is less than 0.5% and for all benchmarks the overhead is less than 2%. On average the BROWSIX-WASM overhead is only 1.15%. Low overhead of BROWSIX-WASM shows that execution in BROWSIX-WASM does not affect the times and performance counter results of programs executed in WebAssembly.

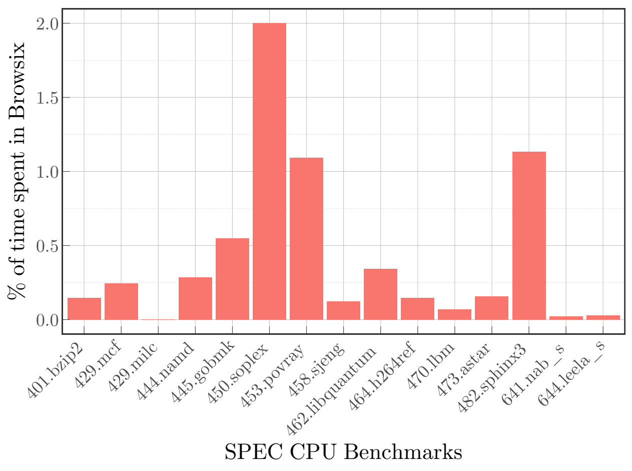


Figure 3: Time spent (in %) in BROWSIX-WASM calls in Firefox for SPEC Benchmarks compiled to WebAssembly. BROWSIX-WASM imposes mean overhead of only 1.15%.

4.2.2 Comparison of WebAssembly and `asm.js`

Figure 4 shows the execution times of SPEC Benchmarks compiled to `asm.js` relative to execution times of benchmarks compiled to WebAssembly and executed in Firefox and Chrome under BROWSIX-WASM. WebAssembly performs better than `asm.js` in Firefox for all benchmarks with

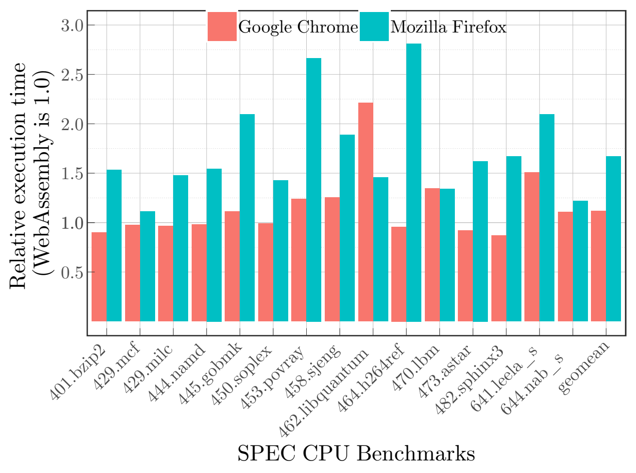


Figure 4: Relative SPEC CPU Benchmarks times of `asm.js` to WebAssembly for Firefox and Chrome. WebAssembly is 1.68× faster than `asm.js` in Firefox and 1.10× faster than `asm.js` in Chrome.

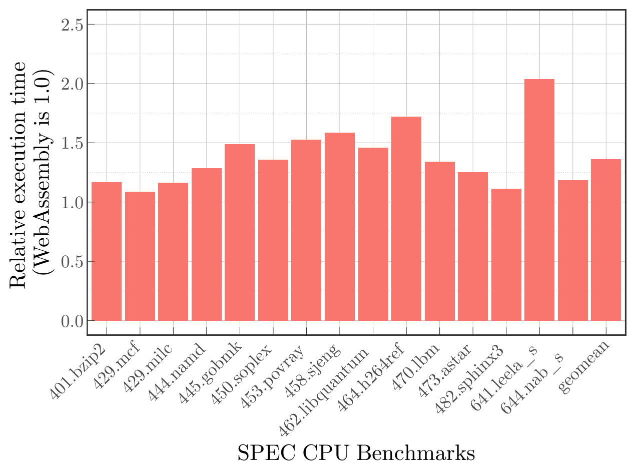


Figure 5: Relative SPEC CPU Benchmarks best time of `asm.js` to best time of WebAssembly. WebAssembly is 1.3× faster than `asm.js`.

mean speedup of 1.68×. In Chrome, WebAssembly performs competitive to `asm.js` with mean speedup of 1.10 ×.

Since the difference in performance between Firefox and Chrome is substantial, we also compare the best performance of `asm.js` out of Firefox and Chrome for each benchmark with best performance of WebAssembly in Firefox and Chrome. Such an evaluation provides better comparison between `asm.js` and WebAssembly because of the differences in the code generator and optimizations in both browsers. Figure 5 shows best `asm.js` times out of Chrome and Firefox with respect to best WebAssembly times out of Chrome and Firefox. Results shows that WebAssembly continously performs better than `asm.js` with mean speedup of 1.3×.

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 6a. 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.

Figure 7 shows the performance of `matmul` compiled to WebAssembly and executed in both Chrome and Firefox compared to native x86-64 code. For both WebAssembly and native, `matmul` was compiled at the `-O2` optimization level. For native code, we additionally disable automatic vectorization with `-fno-tree-vectorize`, as WebAssembly does not yet support vector instructions. The generated WebAssembly code runs between 2× and 2.6× slower.

Figure 6b 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 [8]. Lines 2 – 26 are the body of first loop with iterator `i` stored in `r8d`. Lines 5 – 21 contain the body of second loop with iterator `k` stored in `r9d`. Lines 10 – 16 comprise the body of 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 status register, which is set to 1 when `rcx` becomes 0.

Figure 6c shows x86-64 code JITed by Chrome for the WebAssembly compiled version of `matmul`. This code has been modified slightly – `nop`s in the generated code have been removed for presentation. Function arguments are passed in `rax`, `rcx`, and `rdx` registers, following Chrome's calling convention. At lines 1 – 2, the contents of registers `rax` and `rdx` are stored to the stack, due to registers spills are introduced at lines 38 – 39. Lines 6 – 41 are the body of first loop with iterator `i` stored in `rsi`. Lines 12 – 34 contain the body of second loop with iterator `k` stored in `r11`. Lines 18 – 30 are the body of third loop with iterator `j` stored in `rbx`. Chrome generates two branches for each loop – one is a conditional jump to exit the loop, while the other is an unconditional jump to the start of the loop. For third loop at line 27 `eax` is compared with N_J; the conditional jump at line 28 is taken when the preceding `cmp` evaluates to true. Otherwise, the unconditional jump at line 30 is taken.

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.

```c
void matmul (int C[NI][NJ],
             int A[NI][NK],
             int B[NK][NJ]) {
  for (int i = 0; i < NI; i++) {
    for (int k = 0; k < NK; k++) {
      for (int j = 0; k < NJ; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

(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, 4800, r8
 7      add   rax, rsi
 8      lea   r11, [rax + r9*4]
 9      mov   rcx, -1100     #j <- -1100
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 + 4400], 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, 4400
20      cmp   r9,  1200
21    jne L2                 #end second loop
22
23    add   r8,  1           #i <- i + 1
24    add   rdi, 4400
25    cmp   r8,  1000
26  jne L1                   #end first loop
27  pop   rbx
28  ret
```

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

```
 1  mov  [rsp - 16], rax
 2  mov  [rsp - 24], rdx
 3  xor  rsi, rsi                             #i <- 0
 4  mov  rdi, 0
 5  L1:                                       #start first loop
 6      imul  r8d, 4800, esi
 7      add   r8,  rdx
 8      imul  r9d, 4400, esi
 9      add   r9,  rax
10      xor   r11, r11                        #k <- 0
11      L2:                                   #start second loop
12          imul  r12d, 4400, r11d
13          lea   r14,  [r8 + r11*4]
14          add   r12,  rcx
15          xor   rbx,  rbx                   #j <- 0
16          mov   r15d, ebx
17          L3:                               #start third loop
18              lea   rax, [r15 + 1]          #j <- j + 1
19              lea   rbx, [r12 + r15*4]
20              mov   edx, [rdi + r14*1]
21              lea   r15, [r9  + r15*4]
22              mov   ebx, [rdi + rbx*1]
23              imul  edx, ebx
24              mov   ebx, [rdi + r15*1]
25              add   edx, ebx
26              mov   [rdi + r15*1], ebx
27              cmp   eax, 1100
28              jz    L3END
29              mov   r15d, eax
30              jmp   L3
31          L3END:                            #end third loop
32          add  r11d, 1                      #k <- k + 1
33          cmp  r11d, 1200
34          jnz  L2                           #end second loop
35      add  esi, 1                           #i <- i + 1
36      cmp  esi, 1000
37      jz   L1END
38      mov  rdx, [rsp - 24]
39      mov  rax, [rsp - 16]
40      jmp   L1
41  L1END:                                    #end first loop
42  ret
```

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

Figure 6: 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.

5.1.1 Increased Code Size

The number of instructions in the code generated by Chrome (Figure 6c) are 48, including nops, while clang generated code (Figure 6b) consists of only 28 instructions. The poor instruction selection algorithm of Chrome is one of the reasons for increased code size. As described earlier, Chrome generates 3 instructions for incrementing the loop counter and checking for loop termination condition at lines 18, 27, and 29. Clang, in comparison, functionally identical code in 2 instructions at lines 9, and 15.

Additionally, Chrome does not take advantage of all available memory addressing modes for x86 instructions. In Figure 6b 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 the loads the address in ebx, adds the operand to ebx, finally storing ebx at the address, requiring 3 instructions rather than one on lines 24–26.

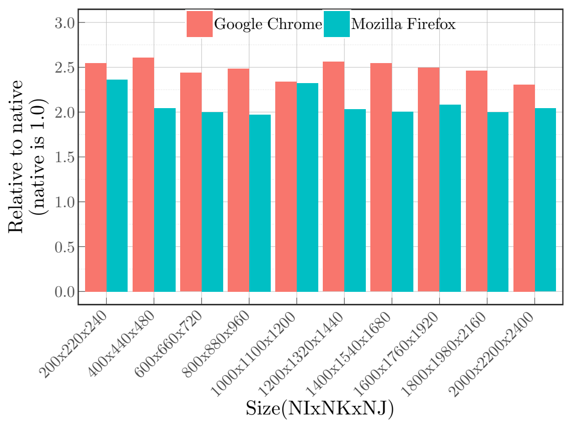


Figure 7: Performance of WebAssembly in Chrome and Firefox for different matrix sizes relative to native code. WebAssembly always performs between 2× to 2.6× vs native.

perf Event	**Wasm Summary**
`all-loads-retired (r81d0)` (Figure 8a) `all-stores-retired (r82d0)` (Figure 8b)	Increased register pressure
`branches-retired (r00c4)` (Figure 9a) `conditional-branches (r01c4)` (Figure 9b)	More branch statements
`instructions-retired (r1c0)` (Figure 10a) `cpu-cycles` (Figure 10b) `L1-icache-load-misses` (Figure 11)	Increased code size

Table 2: Measurements from these performance counters highlight specific issues with WebAssembly code generation. When a raw PMU event descriptor is used, it is indicated by (`rXXXX`).

5.1.2 Increased Register Pressure

Code generated by clang in Figure 6b does not generate any spills and uses only 10 registers. On the other hand, code generated by Chrome (Figure 6c) 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 `add` instruction requires the use of a temporary register. All of this register inefficiency compounds, introducing two register spills to the stack at lines 38 – 39.

5.1.3 Extra Branches

Clang is able to generates code with a single branch per loop, due to clever inversion of the loop counter, like on line 15. Chrome generates more straightforward code, with a conditional branch for the case of loop exit followed by an unconditional jump to the start of the loop.

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 Firefox and Chrome, and the SPEC CPU benchmarks compiled to native code (Section 3).

Table 2 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 8a 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 8b 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).

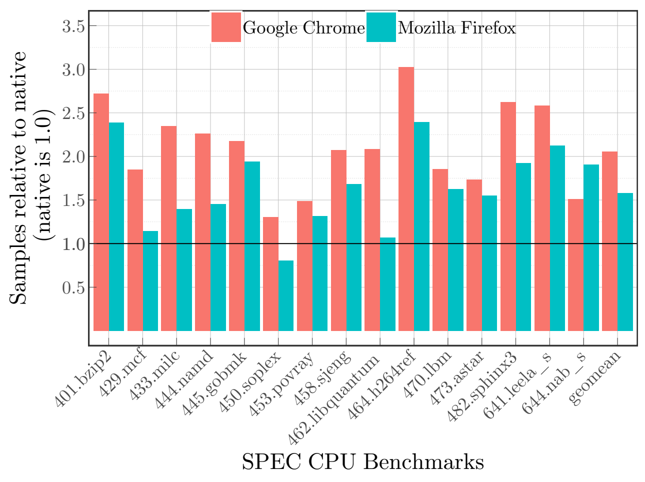
Code generated by Firefox has 1.6× load instrucions retired and 1.7 × store instructions retired than native code. Code generated by Chrome has 2.1× load instructions retired and 2× store instructions retired than the native code. These results show that the WebAssembly-compiled SPEC CPU benchmarks suffer from increased register pressure and thus increased memory references. Below, we outline the reasons for this increased register pressure.

6.1.1 Reserved Registers

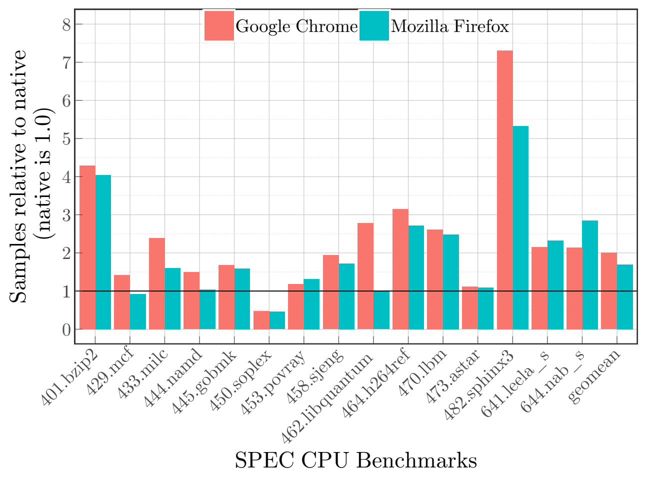
Chrome reserves some registers for its own uses; this increases overall register pressure. The code generated by Chrome for `matmul` generates two register spills but does not use two x86-64 registers: `r13` and `r9`. As Figure 6c shows, there are two register spills at lines 38 – 39. Had these two extra registers been available, these spills could have been avoided. We see this effect across the SPEC benchmark suite.

Chrome stores all the root objects of the JavaScript program in an array and uses `r13` to point to the start of that array. This optimization enables fast accessing of root objects during garbage Collection. Since `r13` always has to point at the start of that array, `r13` cannot be used in the generated code. Chrome also dedicates `r9` for integers and `xmm13` for floats as scratch registers, which are neither callee-saved nor caller-saved. However, Chrome does not use `r9` in WebAssembly generated code.

Firefox also reserves registers, using register `r15` as a heap register that points to the start of the heap; it reserves `r11` as an integer scratch register and `xmm15` as a float scratch register.



(a) `all-loads-retired`



(b) `all-stores-retired`

Figure 8: `all-loads-retired` and `all-stores-retired` samples counted for SPEC CPU benchmarks compiled to WebAssembly executed in Firefox and Chrome, relative to native. The increased number of memory references in WebAssembly is due to the use of reserved registers, a less-effective register allocator, and poor instruction selection.

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 13 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 [24], while clang/llvm 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 Increased Branches

This section focuses on two performance counters that measure the effect of increased branch instructions. Figure 9a shows the number of branch instructions retired by WebAssembly compiled SPEC benchmarks in Chrome and Firefox, relative to the number of branch instructions retired in native code. Similarly, Figure 9b shows the number of *conditional* branch instructions retired. Code generated by Firefox has 1.15× more branch instructions retired and 1.21× more conditional branch instructions retired than native code, while code generated by Chrome has 4.13× branch instructions retired and 5.06× more conditional branch instructions retired.

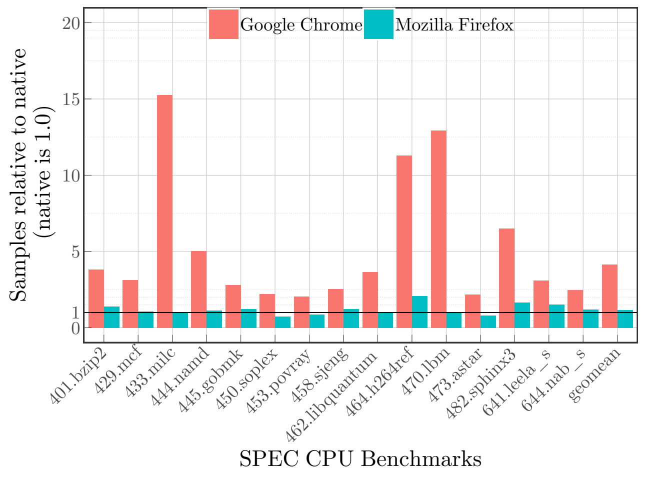
These results shows, as with `matmul` (Section 5.1.3), the WebAssembly compiled SPEC CPU benchmarks suffer from extra branches. Below, we outline the causes of the increased number of branches.

6.2.1 Extra Jump Statements for Loops

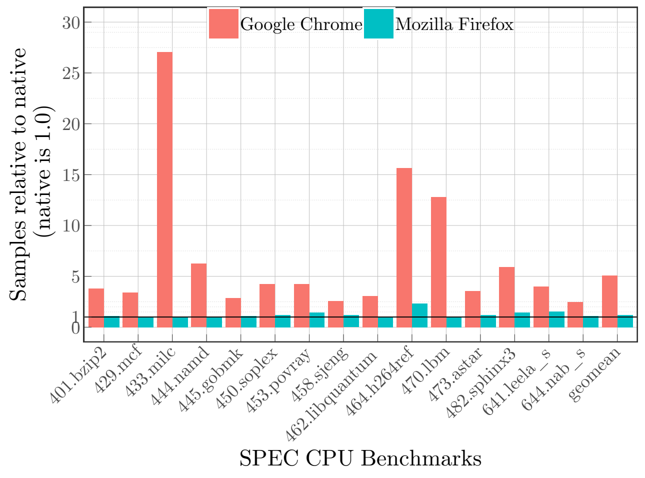
As with `matmul` (Section 5.1.3), Chrome generates unnecessary jump statements for loops, leading to significantly more branch instructions than Firefox.

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.



(a) `branch-instructions-retired`



(b) `conditional-branches`

Figure 9: `branch-instructions-retired` and `conditional-branches` samples counted for SPEC CPU benchmarks compiled to WebAssembly executed in Firefox and Chrome, relative to native. The higher number of branches in code generated for WebAssembly code is due to extra jump statements in loops, stack overflow checks, and indirect function call checks.

6.2.3 Function Table Indexing Checks

WebAssembly supports indirect calls using a function table defined for each module, which has to be indexed to retrieve the correct function code. Function pointers and virtual functions in C/C++ are supported by WebAssembly using this mechanism. A function pointer in WebAssembly is represented as an index to be indexed in the function table. An indirect function call in WebAssembly contains a check to ensure that the index is within the bounds of the function table. Such checks include extra comparison and conditional jump instructions, which are executed before every indirect function call.

6.3 Increased Code Size

As with `matmul` (Section 5.1.1), the WebAssembly compiled SPEC CPU benchmarks also suffers from larger code size. The code generated by Chrome and Firefox for WebAssembly is invariably larger than that generated by clang.

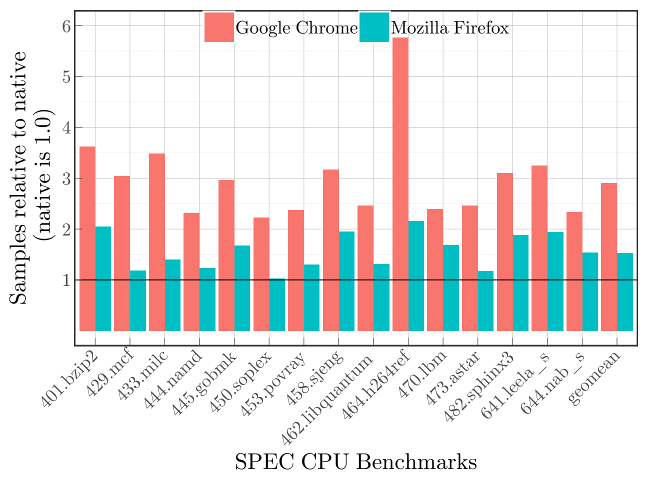
We use three performance counters to measure this effect. (i) Figure 10a shows the number of *instructions retired* by benchmarks compiled to WebAssembly and executed in Chrome and Firefox relative to the number of instructions retired in native code. Similarly, Figure 10b shows the relative number of *CPU cycles* spent by benchmarks compiled to WebAssembly, and Figure 11 shows the relative number of *L1 instruction cache load misses*.

Figure 10a shows that Chrome executes 2.9× more instructions and Firefox executes 1.53× more instructions on average 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 11. On average, Chrome suffers from 3.88× more L1 instruction cache misses than native code, and Firefox suffers from 1.8× 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.

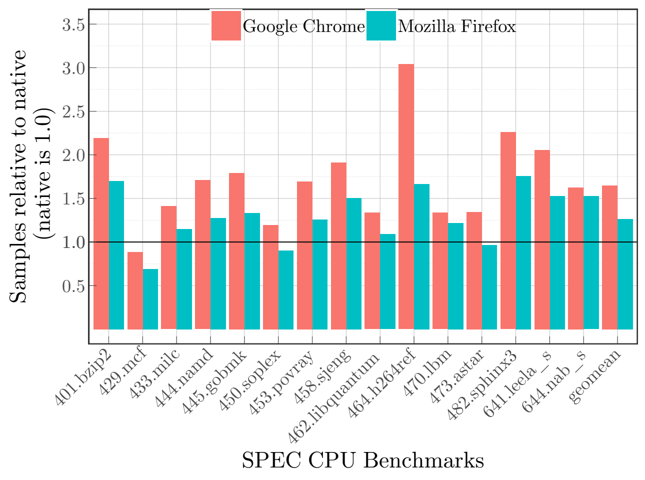
We note one anomaly: although `429.mcf` has 1.1× more instructions retired in Firefox than native code and 3× more instructions retired in Chrome than native code, it runs *faster* than native code. It takes 0.85× as much time in Firefox and 0.95× as much in Chrome. The reason for this anomaly is attributable directly to its lower number of L1 instruction cache misses.

6.4 Comparison of Chrome and Firefox

Section 4 shows that SPEC CPU in Chrome runs 1.27× slower than in Firefox. Performance counter measurement results in Table 3 shows that Chrome's measurements are always higher than Firefox. As compared to Firefox, Chrome's generated code also suffers from unnecessary jump statements for loops and less effective register allocation, leading to more memory references and branch statements. Hence, the performance of code generated by Chrome is worse than that generated by Firefox.



(a) `instructions-retired`



(b) `cpu-cycles`

Figure 10: `instructions-retired` and `cpu-cycles` samples counted for SPEC CPU compiled to WebAssembly executed in Firefox and Chrome, relative to native. The increased code size generated for WebAssembly leads to more instructions being executed.

7 Related Work

Previous approaches to running native code in the browser sacrificed safety, speed, or portability. Microsoft's ActiveX [11] framework enabled interop between native code and web pages, but only ran on Windows and had perennial security issues. Native Client [10, 25] and Portable Native Client [12] restrictions assembly or LLVM bitcode to make it amenable to static analysis [12], but only ran on Chrome.

Several papers use performance counters to analyze the SPEC benchmarks. Panda et al. [19] analyzes the SPEC CPU2017 benchmarks, applying statistical techniques to identify similarities among benchmarks. Phansalkar et al. perform a similar study on SPEC CPU2006 [20]. Limaye and Adegija characterize SPEC CPU2017 identify workload differences between CPU2006 and CPU2017 [16]. Here we use performance counters to evaluate and analyze the performance of WebAssembly vs. native code.

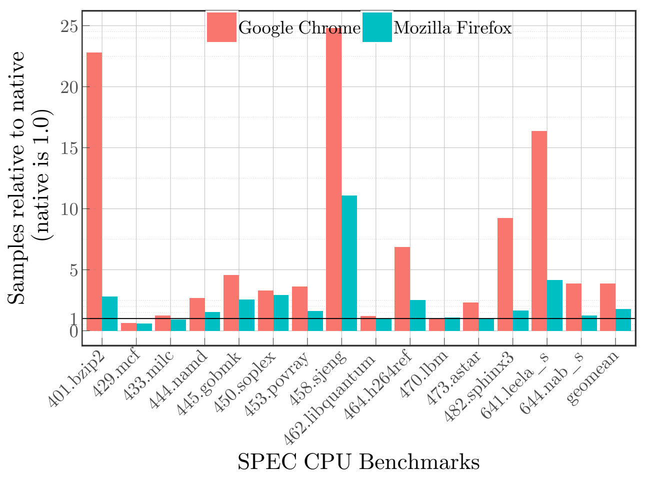


Figure 11: `L1-icache-load-misses` samples counted for SPEC CPU compiled to WebAssembly executed in Firefox and Chrome, relative to native. The increased code size generated for WebAssembly leads to more instruction cache misses.

Performance Counter	**Chrome**	**Firefox**
`all-loads-retired`	2.1×	1.6×
`all-stores-retired`	2×	1.7×
`branch-instructions-retired`	4.13×	1.15×
`conditional-branches`	5.06×	1.21×
`instructions-retired`	2.9×	1.53×
`cpu-cycles`	1.65×	1.26×
`L1-icache-load-misses`	3.88×	1.8×

Table 3: The geometric mean of increases in performance counters for the SPEC benchmarks executed in Chrome and Firefox.

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 1.43× for Firefox, and 1.92× for Chrome, with peak slowdowns of 2.6× in Firefox, and 3.14× in Chrome. We identify the causes of these performance gaps, providing actionable guidance for future optimization efforts.

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

[8] System v application binary interface amd64 architecture processor supplement. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2013.

[9] 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].

[10] 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, New York, NY, USA, 2011. ACM.

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

[12] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. PNaCl: Portable native client executables. *Google White Paper*, 2010.

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

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

[15] 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, New York, NY, USA, 2017. ACM.

[16] 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.

[17] Mozilla. Mozilla is unlocking the power of the web as a platform for gaming. https://blog.mozilla.org/blog/2013/03/27/mozilla-is-unlocking-the-power-of-the-web-as-a-platfo
[Online; accessed 7-January-2019].

[18] 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].

[19] Reena Panda, Shuang Song, Joseph Dean, and Lizy Kurian 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.

[20] 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, New York, NY, USA, 2007. ACM.

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

[22] 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, New York, NY, USA, 2017. ACM.

[23] 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].

[24] 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, New York, NY, USA, 2010. ACM.

[25] 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, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009.

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

[27] 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, New York, NY, USA, 2011. ACM.