How fast is WebAssembly?

Arne Vogel
arne.vogel@tu-bs.de
University of Technology Braunschweig
Institute of Operating Systems and Computer Networks
Seminar: WebAssembly, the standard for the future?
Brunswick, Germany

Abstract

WebAssembly was introduced to be an efficient and fast alternative to JavaScript. The goal is to achieve execution times within the range of native code performance. But can WebAssembly achieve this goal? The original paper which introduced WebAssembly already tries to answer this question. But doubts have been expressed about the validity of the benchmarks used for this purpose.

This paper explores how fast WebAssembly is compared to native code. For this purpose new benchmarks of WebAssembly and native code are evaluated. Furthermore, the reasons for WebAssembly's slow execution speed are examined, and whether these reasons can be alleviated in the future is discussed.

1 Introduction

WebAssembly strives to provide execution speeds close to native execution speed [12]. But can WebAssembly meet these self-imposed reqirements? In this paper we examine how fast WebAssembly is.

In the paper introducing WebAssembly [15], a benchmark for WebAssembly has already been performed. For this benchmark the authors used PolybenchC [15]. PolybenchC is a benchmark suite with various mathematical procedures. However, according to Jangda et al. these benchmarks are not representative for the actual execution of WebAssembly. The use cases for which WebAssembly was introduced go far beyond mathematical procedures [11]. These include among others image / video editing, live video augmentation, CAD applications and developer tooling (editors, compilers, debuggers, ... etc). Since these use cases are only a small part of the PolybenchC benchmark, the expressiveness of the benchmark about the execution speeds of WebAssembly programs remains low. For this reason, Jangda et al. use the SPEC CPU benchmark suite which better matches the use cases of WebAssembly. To run these benchmarks the authors additionally developed Browsix-WASM a tool to simulate Linux system calls in the browser.

The structure of the paper is as follows. In the Section 2 background information necessary for the understanding of this paper is provided. In Sections 3 and 4 the newly created tools by Jandga et al. are presented. Furthermore, the performance of WebAssembly is examined in Section 5, the current state of performance is analysed and the reasons for this state are examined. Finally, related works are presented in section 6 and a conclusion is made in Section 7.

2 Background

In this section background information for the understanding of the paper is provided.

Browsix provides Unix interfaces like processes, sockets, pipes, system calls and a shared file system to JavaScript, asm.js or WebAssembly running in the browser [2, 19]. This is achieved by mapping low level Unix primitive to existing browser APIs. In this way different Unix functionalities can be utilized by JavaScript in the browser. This includes processes (e.g. fork, spawn, exec, and wait4) and signals between processes (e.g. kill) [19].

The **SPEC CPU benchmark suite** is a benchmark designed to compare workloads on different computer systems [10]. The goal of the SPEC CPU Benchmarks is to test the CPU in such a way as it were stressed from real applications. For this reason, the usage of this benchmark is widespread, as for instance the developers of Native Client used this benchmark to test Native Client [22].

PolyBenchC is the C version of the PolyBench benchmarks. Polybench is a collection of benchmarks containing mathematical operations like matrix multiplication or LU decomposition [8].

asm.js is a subset of the JavaScript language. The intention is that asm.js has a better performance than regular JavaScript. This is important because asm.js is used as a compilation target for languages like C. This allows the development of web applications with languages other than JavaScript. Based on asm.js the binary format WebAssembly was developed.

2.1 WebAssembly

Here we provide some detailed information about WebAssembly required for the understanding of this paper.

WebAssembly is a byte code format for execution in web browsers [15]. The first minimal viable product of WebAssembly was released in 2017. The goal of WebAssembly is to deliver execution times faster than JavaScript and as close to native code as possible [12]. WebAssembly is not intended to replace JavaScript but to support JavaScript in CPU intensive applications. As an open standard developed by the W3C WebAssembly Working Group [13], WebAssembly is available in all popular browsers.

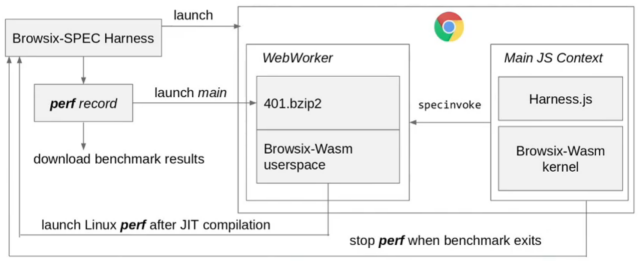

Figure 1. Browsix-SPEC workflow overview [16].

The **memory** of WebAssembly is stored in an array of bytes. This memory must be protected from access outside the array. For this reason, all memory accesses are dynamically checked at runtime. Access outside the array result in a trap. This memory is disjoint from code space. For this reason, no additional memory checks need to be made on memory accesses or storage.

To enable dynamic linking of modules function pointers can be emulated with the instruction call_indirect. This is done via a runtime index in a **global table of functions** defined by the module. The function type is checked dynamically at runtime to ensure it matches the expected function type supplied by call_indirect. This check is needed to guarantee the integrity of the execution environment.

3 SPEC Benchmark

In this section we describe the additions developed by Jangda et al. for the SPEC benchmark.

To investigate the speed of WebAssembly, Jangda et al. utilize the SPEC CPU benchmarks. They chose this benchmark suite because it covers the diverse use cases of WebAssembly.

Jangda et al. have developed Browsix-SPEC Harness for the automatic execution of the SPEC Benchmark in the browser. The behaviour of Browsix-SPEC can be seen in Figure 1. The Browsix-SPEC Harness starts a browser with the *Browsix-WASM kernel*, Browsix-WASM userspace and *Harness.js*. Harness.js is used to load the WebAssembly module to be tested. The Browsix-WASM kernel runs beside the WebAssembly module and is responsible for processing the system calls of this application. After the JIT compilation of the WebAssembly module the Linux tool *perf* is launched for the benchmark. Perf then starts the main and records the performance during execution. The tool perf is a utility that can use *performance counters* to analyse the performance of programs. Performance counters are special CPU registers which are dedicated to store a number of hardware-related activities. The Browsix-SPEC benchmark collected the following performance counters: `all-loads-retired`, `all-stores-retired`, `branches-retired`, `conditional-branches`, `cpu-cycles`, `instructions-retired`, `L1-icache-load-misses`.

4 Browsix-WASM

The SPEC benchmarks require system calls. Jangda et al. could not use Browsix for this because Browsix only supports JavaScript programs. For this reason, Jangda et al. have developed Browsix-WASM as an extension of Browsix. In this section the structure and function of Browsix-WASM are presented.

Browsix was developed in JavaScript [19]. As Jangda et al. wanted to investigate the performance of WebAssembly, they could not use Browsix for this reason. For this reason, they developed Browsix-WASM, an extension of Browsix. Browsix-WASM offers the same Unix interfaces as Browsix, but is implemented in WebAssembly.

Browsix uses SharedArrayBuffer for communication between the process and the kernel. WebAssembly does not support SharedArrayBuffers. For this reason Browsix-WASM an extension to Browsix was developed by Jangda et at. [16]. The extension provides system calls for WebAssembly applications with a low overhead. Programs compiled with Browsix-WASM generate a JavaScript module in which the generated WebAssembly binary is embedded. Additionally, the Browsix-WASM runtime is included in the JavaScript module. The Browsix-WASM runtime provides the libmusl C library which makes the standard C/POSIX library avaliable to WebAssembly programs [5, 16]. Since WebAssembly does not support the SharedMemoryBuffer required by Browsix, it must be supported by Browsix-WASM. The simple solution to copy the memory of the WebAssembly application into a SharedMemoryBuffer results in a high copy overhead and 2x memory usage. For this reason, the authors use a 64mb auxiliary buffer for each process. For a system call, only the required memory is copied into this auxiliary buffer. If more than 64mb is required, the system call is divided into several calls. After the system call has been processed, the memory is copied back into the WebAssembly memory. This process can be seen in Figure 2. Jandga et al. showed that this solution provides a low copy overhead and low memory requirements. In contrast to the naive approach with a high copy overhead and 2x memory usage this approach uses only the resources that are actually needed for the operations. To make sure that the performance of WebAssembly is actually measured and that Browsix-WASM does not cause slower execution performance, Jangda et al. additionally have measured the overhead of Browsix-WASM. For this they instrumented the system calls and looked at how much time is spent during the execution of benchmarks in Browsix-WASM. It was found that for 14 out of 15 benchmarks the Browsix-WASM overhead was below 0.5%. The maximum overhead was 1.2% and the average overhead was only 0.2%.

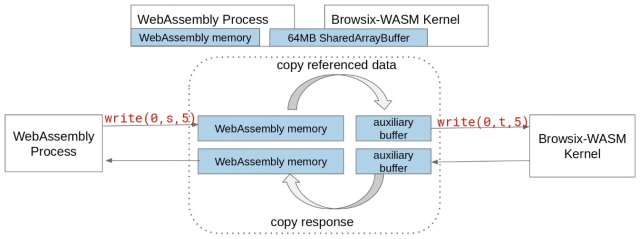

Figure 2. Browsix-WASM converting WebAssembly memory to a SharedMemoryBuffer as required by Browsix [16].

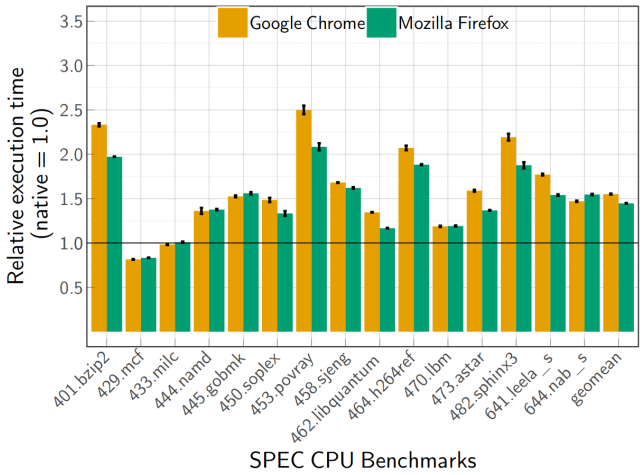

Figure 3. Results of the Browsix-SPEC Benchmarks [16]. Execution times are down relative to the execution times archieved by native code.

5 Performance Analysis

In this section we look at the performance of WebAssembly. We look at the speed in relation to native code and in relation to JavaScript code.

5.1 Benchmarks

Various benchmarks have been performed to determine the speed of WebAssembly. In the paper introducing WebAssembly, Haas et al. already measured the execution performance of WebAssembly. For this purpose they used PolyBenchC. With these benchmarks they found that nearly all tests stayed under 2x the performance of native code and 7 tests could even achieve 1.1x the performance of native code.

Jangda et al. were able to reproduce the results from Haas et al. in their work. However, Jangda et al. also pointed out the weaknesses of PolyBenchC for evaluating WebAssembly. According to their argumentation, PolyBenchC is not representative for the WebAssembly intended use cases. They argue that PolyBenchC as a mathematical benchmark suite reflects only a small amount of the use cases of WebAssembly. For this reason, Jangda et al. have adapted and performed the SPEC CPU benchmark for WebAssembly. The results of the benchmark can be seen in Figure 3. They discovered that WebAssembly code is on average 1.45x slower in Firefox and 1.55x slower in Chrome compared to native code. The highest slowdown was 2.08x in Firefox and 2.5x in Chrome.

In the following we will take a closer look at the SPEC CPU benchmark and examine how the results can be explained.

5.2 Measured performance

With the help of the performance counters as described in Section 3 Jandga et al. were able to analyse precisely the behaviour of the compiled WebAssembly code. The outcomes can be observed in Figure 5.

`all-stores-retired`, **`all-loads-retired`**: These counters indicate how high the register pressure is. Retired instructions are instructions that have been completely executed [4]. In the speculative execution of instructions, more instructions are executed than the control flow requires. As soon as it is certain that an instruction is indeed needed, it becomes "retired". Load and store instructions are involved when reading and writing to registers. For this reason, the number of retired load and store instructions is an indicator of how high the register pressure is. A higher number of load and store instructions is an indicator of a greater register pressure.

In Chrome there were 2.02x and in Firefox 1.92x more load instructions were retired by WebAssembly code in comparison to native code. Furthermore, in Chrome there were 2.30x and in Firefox 2.16x more store instructions being retired by WebAssembly code in contrast to native code.

`branches-retired`, **`conditional-branches`**: Just as more retired load and store instructions indicate a higher register pressure, the number of retired branch instructions can be used as an indicator for more branches in the generated code.

In Chrome there were 1.75x and in Firefox 1.65x more branch instructions were retired by WebAssembly code in comparison to native code. Furthermore, in Chrome there were 1.65x and in Firefox 1.62x more conditional branches in WebAssembly code in contrast to native code.

`instructions-retired`, **`cpu-cycles`**, **`L1-icache-load-misses`**: With L1 instruction load cache misses, more CPU cycles must be used to wait for the requested data. For this reason, L1 instruction cache misses together with the total number of instructions that are retired and the number of CPU cycles required for execution are an indicator of an increased code size.

In Chrome there were 1.80x and in Firefox 1.75x more branch instructions were retired by WebAssembly code in comparison to native code. Furthermore, in Chrome there were 1.53x and in Firefox 1.38x more cpu cycles in WebAssembly code in contrast to native code. In addition, in

Chrome there were 2.83x and in Firefox 2.04x more L1 instruction cache load misses in WebAssembly code in contrast to native code.

5.3 Slow Performance

As it was shown in the benchmarks, the execution speed of WebAssembly is not yet at the same level as the execution speed of native code. Although WebAssembly is faster than JavaScript, it lags behind native code. In this subchapter the different reasons for this are presented which Jangda et al. identified for it.

5.3.1 Increased Register Pressure

Jangda et al. identify increased register pressure as one of the causes for slower execution times compared to native code. Register pressure as defined by Braun et al. denotes "the number of simultaneously live variables at an instruction" [14]. At a high register pressure, variables must be transferred from registers to the memory. This results in slower variable access times for these variables and thus a slower overall execution time. The reasons for the increased register pressures in WebAssembly are multifaceted:

Register allocation algorithm Chrome and Firefox both use a linear scan register allocator while Clang uses a greedy graph-coloring register allocator [6, 21]. The results of the algorithms can differ in the number of registers used for the code. In their work, Jangda et al. showed that for an example matrix multiplication program Clang compiled a program with only 10 registers while Chrome needed 12 registers for the same program. Firefox and Chrome use these algorithms for their fast performance [16]. While fast compilations are important for browsers in order to provide a smooth user experience, native compilers can spend more time on optimizations.

Reduced register number: both Chrome and Firefox have registers that are not available for WebAssembly. These registers are used for browser internal variables. For example, Chrome uses a register to point to an array of GC roots [9], while Firefox uses a register to point to the start of heap memory [1]. Thus, the WebAssembly compiler has fewer registers available than a compiler for native code. For this reason, WebAssembly code executed in browsers has fewer registers available for allocations than native code. This additionally increases the demand on the register allocation algorithms used. Since they have fewer registers to work with, the probability of register spilling is higher.

5.3.2 Extra Branch Instructions

WebAssembly has to do checks before certain instructions for security reasons. These checks are the verification that the stack does not overflow, checks that indirect function calls are valid, and tests that prevent register spilling in loops. All these checks result in a higher number of branches and conditional branches.

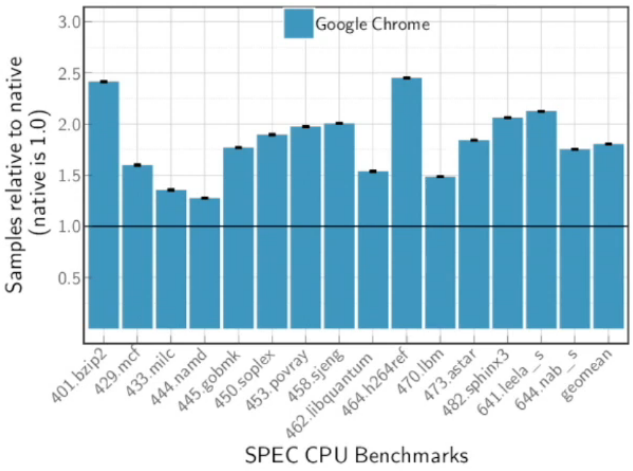

Figure 4. Number of instructions executed by WebAssembly compared to the native code baseline [16].

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

Figure 5. Measurements of the performance counters comparing WebAssembly to a native code baseline [16].

The current stack size is stored in a global variable in both Firefox and Chrome and increased with each function call. For this reason, the stack size must be compared with the maximum stack size allowed by the developer for each function call. This results in both a larger code size and more branches.

For indirect calls, WebAssembly must dynamically check at runtime whether the specified function is valid and whether the types match. These checks are made using the function table. All these checks lead to a larger code size and more branches.

5.3.3 Increased Code Size

For the reasons set out in 5.3.2 WebAssembly code generated in browsers has a larger code size compared to native code. The larger code size can also be recognized in the number of instructions that are executed. This effect can be seen in Figure 4. On average, code generated for WebAssembly will require 1.80x more instructions for the same program as code generated by a native compiler.

5.4 Illustration of the problems on an example

Jangda et al. have illustrated the problems of the WebAssembly compilers using a small example program snippet. The snippet was a matrix multiplication of two matrices. They compiled this once with Clang and once with Chromes WebAssembly compiler. Afterwards the two different outputs were examined.

Firstly, this example demonstrates how not all x86 addressing modes are used by the WebAssembly compiler. This can be seen in Figure 6 and Figure 7. It can be observed how Clang performs the addition in one operation while WebAssembly has to perform two operations to achieve this.

```
add [rdi + rcx*4 + 4*NJ], ebx
```

Figure 6. Addition in Clang

```
add ecx,r15d
mov [rbx+rdx*1],ecx
```

Figure 7. Addition in WebAssembly

Furthermore, Jangda et al. observed how the register pressure has already led to problems in the relatively small snippet. The code created by the WebAssembly compiler contains 3 register spills, although it uses 3 more registers in total. This illustrates the weakness of the linear scan register allocator used by the Chrome WebAssembly compiler. Finally, the code example has made clear how the WebAssembly adds extra jump instructions. Due to additional memory loads at the beginning of loops the WebAssembly compiler needs extra jumps. These jumps skip the memory loads in the first iteration.

As a result of all these problems the WebAssembly code size is much greater compared to the size of native code. While the native code needs 28 lines for the matrix multiplication, the WebAssembly compiler needs 46.

5.5 Possible Improvements

Jangda et at. also considered whether the current reasons for the slower execution time can be fixed or whether they are the result of the design of WebAssembly. They come to the conclusion that the use of the current registers allocators and code generators offers room for improvement compared to the optimizations of native compilers. However, because WebAssembly in browsers requires less compilation time for a good user experience, these optimizations are difficult to implement at initial compilation without affecting the user experience. Therefore, Jangda et al. suggest adopting optimizations from other JIT compilers like hot code operation.

Four other problems, namely stackoverflow checks, indirect call checks, reserved registers and the resulting increased code size, are problems resulting from the design constrains of WebAssembly. The cause of these problems lies in the design of WebAssembly and the resource dependency on the browser. For this reason, these cannot be solved with improved algorithms or other alternative strategies.

5.6 Continuous improvements

In spite of the unsolvable problems, the performance of WebAssembly has improved since its creation. The first release of WebAssembly was only a minimum viable product [7]. Therefore, not all possibilities for improving execution time are likely to be available yet. Since WebAssembly only exists since 2017 and it was released as minimum viable product there are still many areas in which WebAssembly has areas for optimization [7]. For example, fixed-width SIMD and threads are planned as future features [3]. For this reason, further improvements in the execution speed can be expected.

The fact that the speed of WebAssembly is continuously increasing can be observed in Figure 8. It shows the relative execution speed of WebAssembly compared to native code. Over time, the number of WebAssembly PolyBenchC benchmarks running within 1.1x of the speed of native code gradually increases. While in 2017 there were only seven benchmarks within 1.1x speed, in 2019 the number has almost doubled to 13.

These figures indicate that there is continuous work on improvements for WebAssembly and that it is likely that further improvements can be anticipated in the future. As discussed in 5.5, there are still some adjustments where further improvements can be found. So it is unlikely that the optimum has already been reached at this point. Research around Native Client (NaCl) and PortableNative Client (PNaCl) has found lower limits between 5% and 10% performance reduction for code with dynamic security safeguards as found in WebAssembly [18, 20, 22]. Since these works were not measuring the performance in the browser and since browsers are reducing the performance further, e.g. with reserved registers, the lower limit of the performance of WebAssembly in browsers will probably be higher than found in those works.

5.7 WebAssembly vs asm.js

Additionally, Jangda et al. investigated whether WebAssembly is faster than asm.js. For the investigation, the authors have made additional changes to Browsix-WASM to support code compiled to asm.js.

WebAssembly is faster than asm.js in all tested benchmarks. The results of the comparison can be seen in Figure 9. In all benchmarks, the performance of WebAssembly was better than the performance of the same programs compiled to asm.js. They revealed that WebAssembly is 1.54x faster in Chrome and 1.39x faster in Firefox compared to asm.js. Haas et al. [15] have found similar results using the PolyBenchC benchmark with a mean speed up of 1.3x from WebAssembly

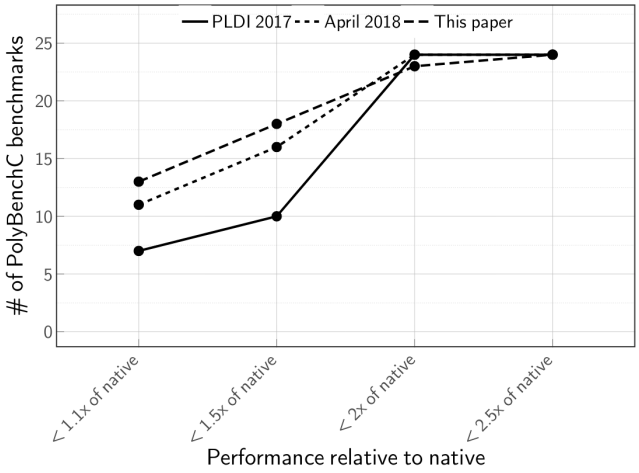

Figure 8. Number of PolyBenchC benchmarks performing within x of native [16]. The number of benchmarks within 1.1x and 1.5x of native code execution times increases with time.

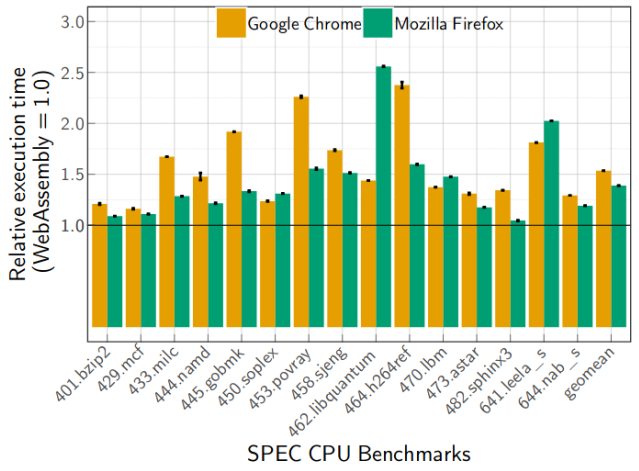

Figure 9. Comparison of the execution time of asm.js compared to the WebAssembly baseline in both Firefox and Chrome [16]. In all benchmarks, the performance of WebAssembly was better than the performance of the same programs compiled to asm.js.

compared to asm.js. This shows that WebAssembly is consistently faster than JavaScript and thus meets one of the self-imposed design criteria.

6 Related Work

One of the predecessors of WebAssembly was Native Client (NaCl) from Google [22]. Just like WebAssembly, certain security precautions had to be taken for NaCl in order to ensure secure execution on end devices. When running the SPEC2000 CPU benchmarks as Linux binary compiled with the NaCl compiler Yee et al. have experienced an on average a slower performance of 5% compared to binaries compiled with the gcc compiler.

Sehr et al. have investigated the software fault isolation (SFI) of NaCl [20]. They found that SFI decrease execution speed between 5%-7% on both ARM and x86-64 architectures. Similar to Jangda et al. they found a bigger code size and cache pressure in code with SFI.

In a technical report from 2015, Lopez et al. examined the performance of PNaCl compared to other web technologies [18]. In the paper, the authors examined among other things the Ostrich benchmark suite [17] for both native code and PNaCl code. The Ostrick benchmark suite contains several numerical algorithms and must therefore be interpreted with caution in the same respect as the PolyBenchC benchmarks. In this benchmark PNaCl code was able to compete with native code with a geometric mean slowdown of around 9%.

7 Conclusion

In this paper we have examined the execution speed of WebAssembly. For this we examined new benchmarks results which better cover the self-imposed use cases of WebAssembly. Though WebAssembly is faster than JavaScript, it is still lagging well behind the performance of native code. Although WebAssembly is on average 1.3x faster than JavaScript, with an average 1.55x slower execution time in Chrome and 1.45x for Firefox, the performance of WebAssembly is still below the speed that can be achieved with native code. We have investigated the reasons for the discrepancy between the execution time of WebAssembly and native code and identified several factors that can be attributed to this discrepancy. We identified the factors of increased code size, increased register pressure and increased number of branch instructions in comparison between WebAssembly and native code. Finally, we investigated if these reasons can be fixed in the future or if they are inherent to the architecture of WebAssembly.

References

[1] [n.d.]. Assembler-x64.h. https://hg.mozilla.org/mozilla-central/file/tip/js/src/jit/x64/Assembler-x64.h. Accessed: 12-01-2020.
[2] [n.d.]. Browsix: Unix in your browser tab. https://browsix.org/. Accessed: 2019-11-15.
[3] [n.d.]. Features to add after the MVP. https://webassembly.org/docs/future-features/. Accessed: 2019-12-18.
[4] [n.d.]. Instructions Retired Event. https://software.intel.com/en-us/vtune-help-instructions-retired-event. Accessed: 2019-12-15.
[5] [n.d.]. Introduction to musl. https://www.musl-libc.org/intro.html. Accessed: 2019-11-20.
[6] [n.d.]. LLVM Reference Manual. https://llvm.org/docs/CodeGenerator.html. Accessed: 2019-11-30.
[7] [n.d.]. Minimum Viable Product. https://webassembly.org/docs/mvp/. Accessed: 2019-11-30.
[8] [n.d.]. PolyBench/C the Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/. Accessed: 2019-11-15.

[9] [n.d.]. register-x64.h. https://github.com/v8/v8/blob/7.4.1/src/x64/register-x64.h. Accessed: 12-01-2020.

[10] [n.d.]. SPEC Benchmarks. https://www.spec.org/benchmarks.html. Accessed: 2019-11-14.

[11] [n.d.]. Use Cases. https://webassembly.org/docs/use-cases/. Accessed: 2019-11-26.

[12] [n.d.]. WebAssembly. https://webassembly.org/. Accessed: 2019-11-14.

[13] [n.d.]. WebAssembly Working Group. https://www.w3.org/wasm/. Accessed: 2019-12-15.

[14] Matthias Braun and Sebastian Hack. 2009. Register spilling and live-range splitting for SSA-form programs. In *International Conference on Compiler Construction*. Springer, 174–189.

[15] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 185–200.

[16] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: Analyzing the Performance of WebAssembly vs. native code. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 107–120.

[17] Khan, Faiz and Foley-Bourgon, Vincent and Kathrotia, Sujay and Lavoie, Erick. [n.d.]. *Ostrich Benchmark Suite*. https://github.com/Sable/Ostrich

[18] Lei Lopez. 2015. Halophile: Comparing PNacl to Other Web Technologies. (2015).

[19] Bobby Powers, John Vilk, and Emery D Berger. 2017. Browsix: Bridging the gap between Unix and the browser. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 253–266.

[20] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. (2010).

[21] Christian Wimmer and Michael Franz. 2010. Linear scan register allocation on SSA form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 170–179.

[22] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 79–93.