Performance Evaluation of WebAssembly

Md Shafiuzzaman *Punnal Ismail Khan*

Contents

1 Introduction

WebAssembly (abbreviated WASM) is developed and introduced in the developers' community as an assembly-like language for the web. It is originally designed to act as a compilation target for other languages [1]. Applications written in or compiled to JavaScript (JS) typically run much slower than their native counterparts. Therefore, one of the main objectives of WebAssembly is to run faster than JS [2]. Additionally, it is expected that the low-level control over the memory layout and close mapping to hardware instructions provide a near-native performance [4]. However, in practice, there are some mixed experiences reported by the developers' community about the performance of WebAssembly. This motivates us to investigate the parameters that influence the performance of WebAssembly. In this project, we evaluate the performance of WebAssembly by running several experiments and reasoning behind their experimental results.

2 WebAssembly

WebAssembly is a low-level, statically typed language that can be run in all major browsers. It does not require Garbage Collection (GC) and supports interoperability with JS. It is claimed to have a memory-safe, sand boxed environment, and when embedded in the web will apply the same-origin and permissions security policies of the browser [4]. Furthermore, it supports modular secure compilation that helps to run it as a multi-language platform. The main key points behind its popularity are its aim to execute at native speed in web and act as an universal compilation target.

Programs written in other languages like C, C++ or Rust can be compiled to WebAssembly to obtain bytecode (Figure 1). Then the bytecode can be run directly on the web browsers (or a virtual machine). It uses a linear memory model (untyped arrays of bytes) to load and store its instructions. It preserves modular secure compilation by allotting different modules for different languages so that each language can be compiled independently and then linked together to perform a single task. WebAssembly is also designed to be programmer-friendly. Though a program written in a certain language can be directly compiled to WebAssambly byte code (to run on the browser), it can also be compiled to a pretty printed human-readable text format called WebAssambly text (Figure 1) to ease the debugging and testing activities.

C++	WebAssembly Text	WebAssembly Bytes

`int add(){`
`  int b = 9;`
`  int a = 1 + b;`
 	
`  return a;`
`}`
	`(module`
	`  (memory 1)`
	`  (func $add (result i32)`
	`  (local $var0 i32)`
	`  i32.const 0`
	`  i32.load offset=4`
	`  i32.const 16`
	`  i32.sub`
	`  tee_local $var0`
	⋮
		`00000000: 0061 736d 0100`
		`0000 0185 8080 8000 0160`
		`00000010: 0001 7f03 8380`
		`8080 0002 0000 0484 8080`
		`00000020: 8000 0170 0000`
		`0583 8080 8000 0100 0106`
		`00000030: 8180 8080 0000`
		`079b 8080 8000 0306 6d65`
		`00000040: 6d6f 7279 0200`
		⋮

Figure 1: A C++ code compiled to WebAssembly

3 Native code compilation in web

Compiling native codes in web is not new. Node.js has supported C++ addons in the browsers since the beginning. These addons are dynamically linked shared objects written in C++. The require() function in JS can be used to load these objects inside JS and execute the methods exposed by these objects. In short, these addons are used to link C/C++ libraries to JS code. That JS code can then use these fast libraries to perform tasks on web browsers.

Emscripten can also compile native languages like C++, rust, and go to asm.js. asm.js is a subset of JS that offers performance improvements as compared to native JS. For example, it uses type coercions to avoid dynamic checks of JS.

So why does we need a new one? Node.js C++ addons and asm.js are great alternatives to improve performance but they have certain drawbacks. For example, Node.js C++ addons require updates for every new version of Nodejs. asm.js on the other hand is faster than native JS but it is still JS and much slower than native codes (C, C++ or Python). Moreover, since JS does not support 64-bit integers, asm.js also does not support them.

4 Performance of WebAssembly

Web Assembly is safe and portable like C++ addons and asm.js. Having low-level control over the memory layout and close mapping to hardware instructions, WebAssembly is expected to be faster than JS. The paper [3] that introduced WebAssembly reported that a C program compiled to WebAssembly instead of JavaScript (asm.js) runs 34% faster in Google Chrome. They also claimed WebAssembly runs on average 10% slower than native code.

However, we see there are mixed experiences reported in literature. Several stack overflow contributors [5, 6] identified WebAssembly functions are much slower than same JS functions. Samsung engineers observed WebAssembly much slower than JS in Samsung Internet browser (v7.2.10.12) when performing multiplications on matrices of certain sizes [7]. On the other hand, eBay calimed their barcode scanners implemented in WebAssembly performs 50 times better than JS [8]. That disparate data across the software development community motivates us to investigate the performance of WebAssmebly and identify the root cause behind this.

To understand the overall performance, we divide our investigation into three questions - (i) Is WebAssembly faster than JS? (ii) How does WebAssembly work comparing to native implementations? (iii) How does choice of languages compiled to WebAssembly affect its performance?

5 Is WebAssembly faster than JS?

To run this evaluation we use iswasmfast [9] tool. This tool was originally implemented to compare WebAssembly with different Node.js C++ addons. We modify it to compare WebAssembly programs and JS native implementations. For levenstein distance and simple linear regression

JS outperforms WebAssembly (details are added in Appendix I).

Reasoning by intuition: To identify the root cause intuitively, we go through all the source codes of the algorithms. We identify a difference for those two algorithms that both of them accept arrays as their input. Therefore, conversion from and to arrays might affect performance of WebAssembly. More interestingly, the performance issue identified by the stack overflow contributors [5, 6] suffer for same reason. WebAssembly works with compiled code, it does not optimize anything while running the programs on web whereas JS engine can apply a lot of dynamic optimisations. And one important thing is that both cases (our investigation and stack overflow case) discussed here are very far from practical solutions. WebAssembly has been built in mind to deal with the different issues working with browser, server-client and cloud. On those real life scenarios those execution time trade-offs might come with different benchmark.

To understand the performance issue in real application scenario, we go for another experiment prepared by Yan et. al [10]. They use PolyBenchC [11] which combines 41 widely-used C benchmarks. For this experiment, 41 WebAssembly binaries and 41 JavaScript programs compiled from 41 C benchmarks of PolyBenchC are run into Chrome browser. To identify the effect of input size five different sizes (XS: smallest, XL: largest) data (details are added to Appendix II) are provided to each benchmark. The experiment collects execution time and average memory usages statistics shown in Figure 2.

Observations: From Figure 2, we can see that WASM outperforms 97.5 % cases (40 out of 41) for the smallest data size while it is 56 % for the largest dataset. This ensures input size has a noticeable effect on the performance of WASM. The most important observation is the pattern of average memory usages. Memory used by JS remains constant regardless of input size. However, WASM usage much memory when it deals with larger input size. It can be referred that the optimization provided by JS helps to execute much lesser instructions than WASM. Though WASM runs instructions faster, it has to execute more instructions while the input size grows larger.

(a)

Input Size	isWasmFaster	isJsFaster
XS	40	1
S	39	2
M	23	18
L	25	16
XL	23	18

(b)

Input Size	WASM	JS
XS	2,001.54	879.41
S	2,077.27	878.73
M	2,985.78	880.54
L	26,991.05	883.10
XL	100,943.88	889.20

Figure 2: WASM vs JS in PolyBenchC running in Chrome (a) Execution time (b) Average Memory Usages

6 How does WebAssembly work comparing to native implementations?

A major goal of WebAssembly is to provide near-native performance. To investigate this, we take help of Browsix tool from Abhinav et. al [2]. Browsix mimics a Unix kernel inside the browser and includes a compiler based on Emscripten that compiles native codes to JS. Abhinav et. al [2] also use PolyBenchC for benchmark. As the WebAssembly codes are also generated by Emscripten, it simulates a more apples-to-apples comparison for WASM vs JS. The experiments have been run into Chrome and Firefox and reported in Figure 3. We can see most of the cases WebAssembly runs more than 10% slower than the native codes. For some cases it is more than 150% slower. The other thing can be noted that Firefox outperforms Chrome for most of the cases. It signifies the environmental effect on the performance of WebAssembly. However, we are more interested in identifying the root cause of the bottleneck of near-native performance.

Observations: To identify the root cause, Abhinav et. al [2] uses matrix multiplication problem as their candidate program. Comparing the assembly instructions generated by both implementations (native and WASM) in browsix, we see a significant difference in their instruction sets (detailed added in Appendix III). Native code uses 10 registers while WebAssembly uses 13 registers. WebAssembly reserves a register to point to an array of GC roots at all times (as WebAssembly does not have own GC). WebAssembly uses other two registers as dedicated scratch registers to hold intermediate values generated by WebAssembly. WebAssembly also uses some additional branch conditions such as extra jumps to avoid memory loads, stack overflow checks per function call and function table indexing checks. These also affect the performance of WebAssembly.

Takeaways: Though WebAssembly suffers for some optimizations, some security guarantees such as stack overflow checks, indirect call checks, and reserved registers checks are necessary to be a worthy programming language. Developers may skip those performance difference as WASM is the best alternatives to them considering its both of its performance and security guarantees.

7 How does choice of languages compiled to WebAssembly affect its performance?

One of the most distinctive features of WebAssembly is it can be used as universal compilation target for different programming languages. This leads to a key performance question: Does choice of languages have affect on the performance of WebAssembly? To get this answer, [12] prepare an interesting implementation setup by collecting off-the-shelf implementations of matrix multiplication (100X100 arrays with values 1-10,000), insertion sort (int values between 0-999 in an array of size 1000) and addition (a function that took two values and added them together) both in C and RUST. They identify a significant difference in the number of instructions for C compiled WebAssembly to Rust complied WebAssembly. Most of the cases, C compiled

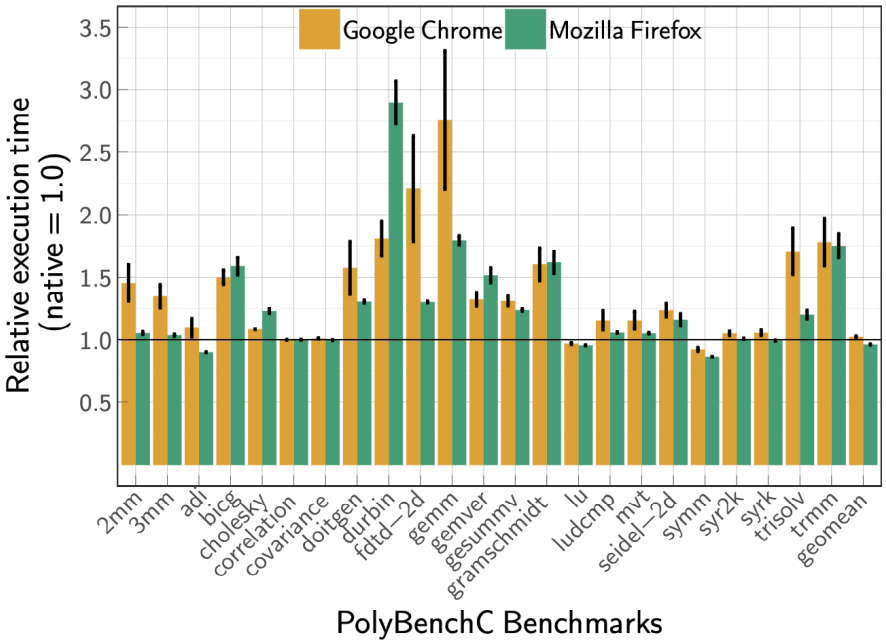


Figure 3: The performance of the PolyBenchC compiled to WebAssembly (executed in Chrome and Firefox) relative to native

WebAssembly uses nearly 3 times more instructions. Unsurprisingly, C compiled WebAssembly also underperforms in execution time benchmark.

We get a clear message WebAssembly performs differently when it compiles from different languages. Rust works better in demanding calculations. For shorter and simpler tasks it is not as clear which language to choose. However, those observations are not free from nepotism of implementation excellence of the developers. However, the most interesting findings from this experiment is performance of WebAssembly depends on the execution environment (browser engine) rather than file transfer from the serve as We see in all cases fetch time remains constant.

8 Conclusion

This project provides a comprehensive performance study for WebAssembly. All the experiments depend on several environmental and human factors. Therefore, impartiality cannot be completely achieved here. However, we feel the experiments investigated here identifies several important parameters that influence the performance of WebAssembly. Some of them like runtime optimizations can be ameliorated by improved implementations. The others such as stack overflow checks, indirect call checks, and reserved registers checks are necessary for security features. Considering all the aforementioned factors, we may conclude even after providing all those unique important features like universal compilation target, module secure compilation, and developer-friendly implementation, WebAssembly do provide manageable performance.

References

[1] Group, WebAssembly Community, and Andreas Rossberg. WebAssembly Specification, n.d., 185.

[2] Jangda, Abhinav, Bobby Powers, Emery D Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code, n.d., 15.

[3] 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 Lan- guage Design and Implementation, PLDI 2017, pages 185–200. ACM, 2017.

[4] Lehmann, Daniel, and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly." In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1045–58. Providence RI USA: ACM, 2019.

[5] Stack Overflow Contributor Blindman67. 2018. Why is webAssembly function almost 300 time slower than same JS function. https://stackoverflow.com/questions/48173979/why-is-webassembly-function- almost- 300- time- slower- than- same- js- function. Accessed in 06 December 2021.

[6] Stack Overflow Contributor ColinE. 2017. Why is my WebAssembly function slower than the JavaScript equivalent? https://stackoverflow.com/questions/46331830/ why-is-my-webassembly-function-slower-than-the-javascript-equivalent. Accessed in 06 December 2021.

[7] Winston Chen. 2018. Performance Testing Web Assembly vs JavaScript. https://medium.com/samsung-internet-dev/performance-testing-web- assembly- vs-javascript- e07506fd5875. Accessed in 06 December 2021.

[8] SenthilPadmanabhanandPranavJha.2020.WebAssemblyateBay:A Real-World Use Case. https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real- world- use- case. Accessed in 06 December 2021.

[9] JavaScript C++: Modern Ways to Use C++ in JavaScript Projects. https://github.com/zandaqo/iswasmfast. Accessed in 06 December 2021.

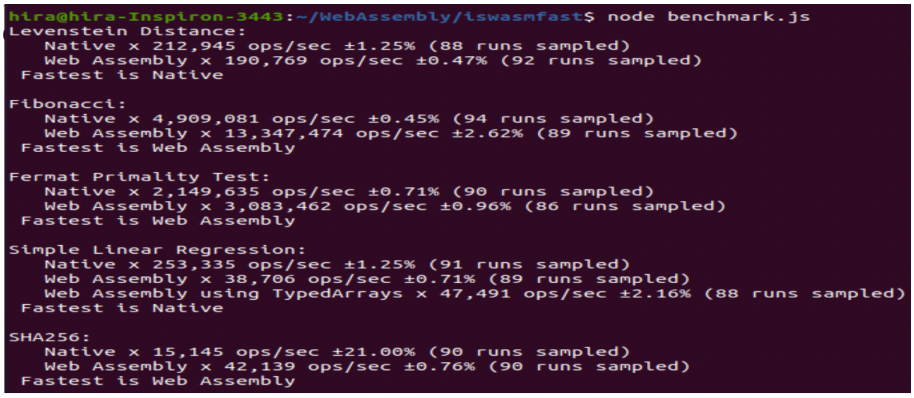
[10] Yan, Yutian, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. "Understanding the Performance of Webassembly Applications." In Proceedings of the 21st ACM Internet Measurement Conference, 533–49. Virtual Event: ACM, 2021.

[11] 41 widely-used C benchmarks (PolyBenchC). https://github.com/MatthiasJReisinger /PolyBenchC-4.2.1. Accessed in 06 December 2021.

[12] Medin, Magnus, and Ola Ringdahl. "Bachelor's Degree in Computer Science," n.d.,16.

A Appendix I

Comparison of execution time of some off-the-shelf C++ implementations of various algorithms (Levenstein Distance, Fibonacci, Fermat Primality Test, Simple Linear Regression, SHA256) compiled to WebAssembly and native implementations in JS:



Courtesy: https://github.com/zandaqo/iswasmfast

Figure 4: WASM vs JS

B Appendix II

Different input sizes used for the experiment WASM vs JS in PolyBenchC running in Chrome to collect execution time and average memory usages statistics:

Input Size	XS	S	M	L	XL
Input size in benchmarks	MINI_DATASET	SMALL_DATASET	MEDIUM_DATASET	LARGE_DATASET	EXTRALARGE_DATASET
CORRELATION	M=28 N =32	M=80 N=100	M=240 N=260	M=1200 N=1400	M=2600 N=3000
COVARIANCE	M=28 N =32	M=80 N=100	M=240 N=260	M=1200 N=1400	M=2600 N=3000
GEMM	NI=20 NJ=25 NK=30	NI=60 NJ=70 NK=80	NI=200 NJ=220 NK=240	NI=1000 NJ=1100 NK=1200	NI=2000 NJ=2300 NK=2600
GEMVER	N=40	N=120	N=400	N=2000	N=4000

Yan, Yutian, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. "Understanding the Performance of Webassembly Applications." In Proceedings of the 21st ACM Internet Measurement Conference, 533–49. Virtual Event: ACM, 2021. https://doi.org/10.1145/3487552.3487827

Figure 5: Different input sizes for each benchmark of PolybenchC

C Appendix III

Assembly instructions generated by both implementations native and WASM in browsix:

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

Figure 6: Instructions generated from native

```
mov [rbp-0x28],rax
mov [rbp-0x20],rdx
mov [rbp-0x18],rcx
xor edi,edi                    #i <- 0
jmp L1'
L1:                            #start first loop
  mov ecx,[rbp-0x18]
  mov edx,[rbp-0x20]
  mov eax,[rbp-0x28]
  L1':
  imul r8d,edi,0x1130
  add r8d,eax
  imul r9d,edi,0x12c0
  add r9d,edx
  xor r11d,r11d                #k <- 0
  jmp L2'
  L2:                          #start second loop
    mov ecx,[rbp-0x18]
    L2':
    imul r12d,r11d,0x1130
    lea r14d,[r9+r11*4]
    add r12d,ecx
    xor esi,esi                #j <- 0
    mov r15d,esi
    jmp L3'
    L3:                        #start third loop
      mov r15d,eax
      L3':
      lea eax,[r15+0x1]        #j <- j + 1
      lea edx,[r8+r15*4]
      lea r15d,[r12+r15*4]
      mov esi,[rbx+r14*1]
      mov r15d,[rbx+r15*1]
      imul r15d,esi
      mov ecx,[rbx+rdx*1]
      add ecx,r15d
      mov [rbx+rdx*1],ecx
      cmp eax,NJ               #j < NJ
    jnz L3                     #end third loop
  add r11,0x1                  #k++
  cmp r11d,NK                  #k < NK
  jnz L2                       #end second loop
add edi,0x1                    #i++
cmp edi,NI                     #i < NI
jnz L1                         #end first loop
retl
```

Figure 7: Instructions generated from native WebAssembly