Comparative Analysis Of JavaScript And WebAssembly In The Browser Environment

Tushar
Department of Information technology
National Institute of Technology Surathkal
Karnataka, India
tushar.202it030@nitk.edu.in

Biju R Mohan
Department of Information technology
National Institute of Technology Surathkal
Karnataka, India
biju@nitk.edu.in

2022 IEEE 10th Region 10 Humanitarian Technology Conference (R10-HTC) | 978-1-6654-0156-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/R10-HTC54060.2022.9929829

***Abstract*—As World Wide Web is evolving, larger and high-performance applications are being entirely run on the browsers. Web applications have their own advantages like they are more accessible and platform independent. JavaScript was the only programming-language which was historically supported to be ran on the web browsers, but it is quite limited to high-performance applications as it is dynamically-typed and interpreted language. So, as the high-performance applications started to come to web there have been always a need for another language which could run in the browser environment but also take advantage of system resources. WebAssembly was one such effort by the vendors of different browsers coming together. WebAssembly is claimed to be portable and size and time efficient binary format which could be compiled to run on the web browsers at near native speed. This paper will try to verify the claim by running various experiments on both WebAssembly and JavaScript and measuring resource used and time taken by those programs to execute and will later do a comparative analysis between the both.**

***Index Terms*—JavaScript, WebAssembly, wasm, compilation targets, v8 engine**

I. Introduction

JavaScript is the integral part of the web, all the browsers comes pre-loaded with JavaScript engine. JavaScript is a high-level just-in-time compiled language i.e. it is compiled while running and not before unlike languages like C/C++. It is dynamically and weakly typed language, means type of variables can change at runtime and certain types are implicitly cast depending on the operation used. JavaScript is used as scripting language of the web, there have been various frameworks and toolchain made over it to make working with JavaScript efficient and easy. But in the end everything converts to JavaScript, which is the only language browsers support natively.

Decades later JavaScript remains the only primary programming language used for web technologies. It has worked quite well until some modern use cases arose like gaming on the web, image/video rendering, VR and AR technologies, simulators etc. With use cases like these JavaScript is not able to provide native-like performance. So, there have been a need for an another programming language which are only focused for these high performance tasks and could run alongside JavaScript in the browser.

WebAssembly which was first released in 2017 was the solution to the rise in demanding web applications. WebAssembly is a portable low-level bytecode made to safe, efficient compilation and almost no-overhead execution. WebAssembly happened when many major browser vendors came together to collaboratively to create something to enable high performance applications on the web. However WebAssembly do not make any web-specific assumptions, it is language-independent, hardware-independent and platform-independent and can be employed beyond the web, compilers like Wastime, wasm3 could be used to run WebAssembly in non-browser environments[8]. It became World Wide Web Consortium(W3C) recommendation in 2020[6] i.e. it became the fourth language to be natively supported on the browsers, so as a result Web Browsers can run any combinations of HTML, CSS, JavaScript and WebAssembly now. As of 2021 95% of browsers natively supports WebAssembly. WebAssembly also provides compilation targets for many other languages as it is pre-compiled language. Initially support for C and C++ is developed as compilation target to WebAssembly but now almost forty languages have been reportedly supporting WebAssembly as compile target as of now.[1]

WebAssembly had the following goals to be achieved:

- To be portable, size and time-efficient to be provided as compilation target. It should be able to take advantage of common hardware capabilities of different platforms to execute the binary codes at near-native speed.[12]
- WebAssembly should be able to achieve roughly same functionality as *asm.js*, using *asm.js* users were able to serve languages like C/C++ as compilation target to the web. Additionally features like threads and SIMD are also to be implemented.[13]
- WebAssembly should be able to run on the existing Web platforms, should be backward compatible. It should be designed to run with JavaScript in the web browsers, users should be able to make calls to/from existing JavaScript while maintaining the same security standards.[15]
- A human readable form of the binary formatted WebAssembly code should be developed in order to make debugging easy.[15]
- As discussed earlier WebAssembly makes no assump-

tions about the Web and should support non-browser environments too.
- Discussing about the other languages that is to be supported as compilation target to the WebAssembly, the developers initially focused on C/C++ building LLVM[9] support for WebAssembly. LLVM is a set of compiler and tooling technologies. The tooling support expects to include editors, language virtual machines, debuggers, security enhancements, profilers, JavaScript with WebAssembly optimization tools etc.[14]

WebAssembly's main focus is the web browsers, we could write entire codebase in WebAssembly or mainframe in WebAssembly and UI in JavaScript/HTML, the last method is mostly used in the industry as of now. Domains which could benefit using WebAssembly inside the web browsers are image/video editing, VR/AR applications, better execution of the languages that are cross-compiled to the web, games and games editors, CAD applications, visualization applications, developer tools, remote desktops, VPN, encryption tasks etc. However WebAssembly could be useful outside browser environment too in variety of use cases like server side applications, native applications of mobile devices, multi-node computations etc.

II. Literature Survey

WebAssembly was not the first attempt at developing bytecode for the web. Bytecode is very low-level and very compactly represented, hence these are very easier for the engines to read, just-in-time compile and execute these. WebAssembly being bytecode can take advantage of the arithmetic capabilities of the processors, it operates on very simple and floating point numbers rather than complex JavaScript object system which result in large overheads for the programs.

Microsoft and Oracle tried to achieve bytecode support in the web browsers with .NET and Java, but these plugins needed to be installed to support these languages in web browser, this is like installing another engine alongside JavaScript manually[2].

Google also tried to come up with techniques which extends the browser ability beyond JavaScript. Native Client(NaCl)[16] could able to run x86(ARM) validated programs in sandboxed enviroment, Portable NaCl(PNaCl)[7] made it portable. This model allowed the NaCl code to reside with sensitive data which is serious concern with security and it was unable to access JavaScript or Web APIs synchronously. Also other browser vendors did not install it and it just got restricted to Chrome.

A framework known as Emscripten[17] was able to convert C/C++ code to be ran on web to a special JavaScript subset file called *asm.js*[3], it provided good performance over regular JavaScript by eliminating dynamic type guards, boxed values, and garbage collection. So basically *asm.js* is a low level JavaScript subset which is compiled from C/C++ programs and retains their performance.

As our computers became more capable and JavaScript also became faster, many JavaScript based programs which could do powerful things on the browsers also emerged. Like WebGL[10] exposed 3D graphics which are hardware accelerated to the JavaScript developers. Many APIs[11] were also developed which could give users access to game controllers, microphone, camera and other hardware in the browser which extended the scope of what JavaScript could do in the web browsers. Many languages other than JavaScript also emerged which makes the development easier and efficient like TypeScript[4]. These language compilers produce JavaScript code at the end.

But despite of all such advancements the need of bytecode for the web was not ruled out. For this developers of four major browser vendors came together, namely, Mozilla, Microsoft, Google and Apple[2]. It resulted in solution for low-level code for the web which satisfies all the essential design goals[8] which are:

- **Safe:** WebAssembly inherently provided safety by introducing another VM alongside JavaScript one. It have a managed language runtime i.e. it will prevent programs from compromising user data or system state.
- **Fast:** WebAssembly code being low-level is optimised ahead-of-time to be fast and close to native. However some performance overhead is introduced in order to implement sandboxing techniques.
- **Portable:** The web runs over different devices, operating systems, browsers, processor architectures hence code targeting web must be platform- and hardware-independent so that applications could run across all these types with the same behaviour. Previous low-level web codes discussed have these portability problems.
- **Compact:** To reduce bandwidth usage, reduce load times and improving overall responsiveness the code that is transferred over the network should be compact. JavaScript code even when minified is far less compact than binary code.

This paper tries to make sense of WebAssembly if it really gives users the performance benefits as promised. This paper runs benchmark tests over different kind of algorithms namely iterative and recursive and try to compare the time and other resources needed to execute those programs and conclude if WebAssembly really gave us any performance benefits. In section III we'll discuss about how a WebAssembly code is executed, how a C/C++ program can be targeted for web browser using WebAssembly and the structure of WebAssembly file. In section IV we go in details about the benchmark tests we have ran and how WebAssembly and JavaScript compared in those tests. In section V we conclude our benchmark results and discuss about the future scope of WebAssembly and this project.

III. Execution of WebAssembly Programs

WebAssembly bytecode in the browser is run in separate Virtual Stack Machine and it interfaces with JavaScript code with the help of modules. The separate VM in the browser is designed to be faster to execute programs than JavaScript and to have compact code representation. Instruction Set

```
    function fetchAndInstantiate(url,
importObject) {
  return fetch(url).then(response =>
    response.arrayBuffer()
  ).then(bytes =>
    WebAssembly.instantiate(bytes,
importObject)
  ).then(results =>
    results.instance
  );
}
```

Fig. 1. Loading wasm module in JavaScript

Architecture(ISA) of WebAssembly defines the operations to be executed by the VM. The list of instruction contains memory load/store instructions, numeric, parametric, control of flow instruction types and Wasm-specific variable instructions. WebAssembly have 128-bit SIMD opcodes. The SIMD opcodes are portable and translate to native instruction sets like x64 and ARM.

WebAssembly also provides compile targets for other languages to be run on the web. Rust, C/C++ are those languages which have really good support in WebAssembly. WebAssembly binary format have the extension `.wasm`, this format is not designed to be read by humans, for that purpose WebAssembly have another textual representation format `.wat`. This paper for benchmark tests uses C as compilation targets for WebAssembly in web browsers.

Modules written in C/C++ can be compiled to WebAssembly using Emscripten tool, which can compile C/C++ to any language which uses LLVM which is true in case of WebAssembly. Emscripten will generate binary WebAssembly module code, A JavaScript file which contains glue code to facilitate communication between C functions through wasm and JavaScript and A html file to load, compile and instantiate the wasm code. The C functions needed to be called through JavaScript have to be flagged as `EMSCRIPTEN_KEEPALIVE`, this is done by putting this keyword before a C function, this will create this function callable from JavaScript

Ecmascripten given us the glue code so that we can directly call our C/C++ functions right into our JavaScript code, if done manually it is little complicated as of now, the could be seen in Figure 1.

Structure of WebAssembly file: As of now the WebAssembly functions can use integers and floating point numbers as parameters or return values. For more complex datatypes such as strings we have to use WebAssembly module's memory, it is an array of bytes where array indexes serve as memory addresses. Since indexes are integers they can be passed in WebAssembly functions and first character of the string(or character code equivalent) can be used as pointer in JavaScript.

WebAssembly code can be written directly in text format

```
    long fibonacci(int n){
        if (n <= 1) return n;
        return fibonacci(n - 1) +
fibonacci(n - 2);
    }
```

Fig. 2. Example code in C

WebAssembly

```
(module
  (type $t0 (func (param i32) (result
i32)))
  (func $fibonacci (export "fibonacci")
(type $t0) (param $p0 i32) (result i32)
    get_local $p0
    i32.const 2
    i32.lt_s
    if $I0
      get_local $p0
      return
    end
    get_local $p0
    i32.const -1
    i32.add
    call $fibonacci
    get_local $p0
    i32.const -2
    i32.add
    call $fibonacci
    i32.add)
  (memory $memory (export "memory") 1))
```

Fig. 3. Example code in WebAssembly text format

i.e. `.wat` format but most likely it will be written in some higher level language C/C++ optimally and then compiled to bytecode using some toolchain, Emscripten in case of C/C++. Let us try to compile C code shown in Figure 2. When this code is compiled to `.wat` format we get the output as shown in Figure 3

But the code that machine executes is in binary format, the binary format of the same code which is `.wasm` format looks like as shown in Figure 4. To make it more clear lets consider three lines of WebAssembly code `get_local $p0` gets the value of 1st param and push it on the top of stack, `i32.const -2` pushes -2 on the top of stack `i32.add` adds the top two values on the stack. in binary this part of code is represented as `20 00 41 7E 6A`. As clear from the wat format this part of code is `n-2` in the C code.

Running of wasm file: After the creation of `.wasm` file here are the steps that would be taken by the browser to execute it. These are the steps taken by JavaScript to get executed, we will discuss these in context with WebAssembly.

```
    00 61 73 6D 01 00 00 00 01 06 01 60
01 7F 01 7F 03 02 01 00 05 03 01 00 01 07
16 02 06 6D 65 6D 6F 72 79 02 00 09 66 69
62 6F 6E 61 63 63 69 00 00 0A 1E 01 1C 00
20 00 41 02 48 04 40 20 00 0F 0B 20 00 41
7F 6A 10 00 20 00 41 7E 6A 10 00 6A 0B
```

Fig. 4. Example code in WebAssembly Binary Format

- **Fetching:** This is the part where the wasm file is fetched from the network. As WebAssembly files are more compact than JavaScript files, fetching them would take relatively lesser time. On slower networks this would be very beneficial.
- **Parsing:** In this step JavaScript source gets parsed into Abstract Syntax Tree(AST) and gets converted to the bytecode which is specific to the JS engine. As WebAssembly is already in bytecode form it just needs to be decoded and validated. Parsing is often done lazily in browsers i.e. functions are parsed when they are needed.
- **Compiling:** Compilation of WebAssembly code starts much closer to the machine code, also it do not need much optimizations as these are done ahead of time in LLVM. For these reasons Compiling and optimizing of WebAssembly code takes reasonably less faster.
- **Reoptimization:** A program may have to go through reoptimizing cycles due to deoptimizations, which may happen due to variety of reasons, if variables changes in successive iterations or new functions are inserted in the prototype chains. So, JIT may sometimes decide to go thorough compilation again. But in WebAssembly types are explicit so it does not make any assumptions about type based on data it gathers during runtime. Hence, it does not go through reoptimization cycles.
- **Execution:** Performant codes in JavaScript needs to be in accord with the optimizations that JIT make. Many coding practices we perform to make our code readable makes code less performant. These JIT optimizations aren't necessary in WebAssembly so it is generally faster than JavaScript. WebAssembly is also a compiler target i.e. it is designed for compilers to generate and not for human programmers to write hence it can be more ideal code for machines.
- **Garbage Collection:** As of now WebAssembly does not have any garbage collector and memory is managed manually. It makes performance more consistent as It is in developer's hands when garbage collection need to kick in.

Looking at the steps it becomes clear that WebAssembly outperforms JavaScript for ample of reasons when doing a task. However WebAssembly functions as of now are called through JavaScript code which will have some runtime overhead that may result in slower performance of simpler tasks in WebAssembly relative to JavaScript. Also as WebAssembly is designed to perform larger computation related tasks hence interaction with browser APIs is currently not supported in WebAssembly.

IV. Benchmark tests and Results

In this section we will run benchmark tests and compare the time taken by JavaScript and C compiled to WebAssembly to run the same program. We will be testing both the technologies over recursive and iterative programs. Later we will also be compiling other modern languages namely Rust and Golang to WebAssembly and comparing the performance.

For the first benchmark test we have chosen program to calculate Fibonacci of a number. Fibonacci of a number *n* is the *nth* number of the sequence which have a number as sum of previous two numbers, the sequence starts with 1. We will be using simple recursive Fibonacci function, which will be a good test of how well function calling is, test of crunching big numbers and test of how a language deals with recursive calls. Another program for benchmark is to test if a number is prime or not, for this purpose we will be running a simple loop over the number without any optimisations to test if it is prime or not. This will be a good test of how well a language deals with iterative programs.

We have carried our tests on Ubuntu machine, with 16GB Memory and octa-core 3GHz Intel i7 processor. Results of the tests can vary depending on the machine type the tests are ran on. The tests are ran for 5 times and average time have been taken to avoid any inconsistency in the results.

In the first test we calculated Fibonacci sequence of numbers from 1 to 51 using both WebAssembly and JavaScript on Chrome as well as Mozilla Firefox. This is plotted against time taken for each sequence to get calculated as shown in Figure 5. Further we also compiled targeted WebAssembly using Rust and Go. For Rust we have used wasm-pack to compile Rust code to WebAssembly, for Go we have used TinyGo compiler for the task. The results of the same are shown in Figure 6, we have scaled x-axis of the graph to start from 40 in order to better see the difference when the program becomes hard to compute.

As it is clear from the figure that the difference between the time to calculate Fibonacci sequence of starting number is negligible for JavaScript and WebAssembly but the gap widens as the numbers get large. This clear separation when the numbers got large is clearly visible when we plot log scaled graph of the same as shown in Figure 7. The conclusion of the graphs shown is also provided in Table I for better readability. WebAssembly performed better on Mozilla Firefox as compared to Chrome but JavaScript on Mozilla performed far worse than Chrome. While running of the program one CPU out of 8 was maxed out on both the languages and browsers. WebAssembly also occupied relatively lesser memory as compared to JavaScript, while running the program, JavaScript was taking around 40Mbs while WebAssembly around 33Mbs. Talking about the language support, it is quite evident from the results that C/C++ have better target compilation support for WebAssembly followed by Rust.

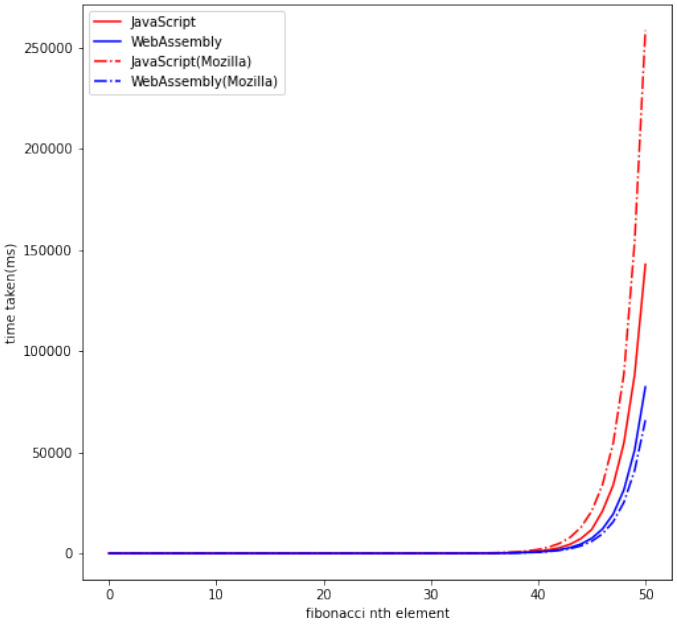

Fig. 5. Time taken to calculate Fibonacci of a number

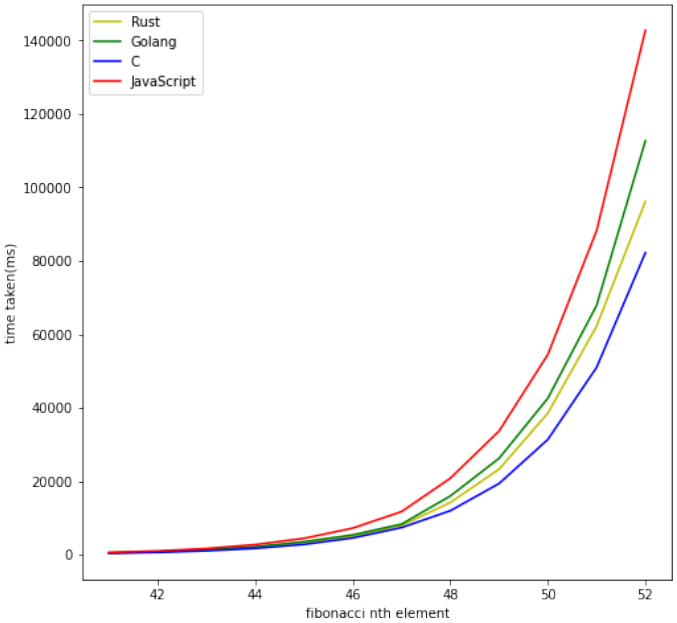

Fig. 6. Fibonacci number calculation time using multiple languages

For another test we have a list of prime numbers in the range 1009 to 2124749677 this is to fit the numbers in the range of C long, we would simply iterate over those prime numbers and record the time taken for iteration over a number, this will test the ability of our programs to make large iterations. This test is done at 6x slowdown CPU throttling in order to get larger times. Our recorded results are shown in Figure 8, as it is clearly conclusive from the plot that when the prime numbers are small there is no clear winner but

TABLE I
RUNNING TIME OF THE TESTS TO CALCULATE FIBONACCI SEQUENCE

	nth element of Fibonacci series			
Language	**10**	**30**	**45**	**51**
JavaScript(Chrome)	0.00	5.60	7267.00	142749.80
JavaScript(Mozilla)	0.00	10.00	12854.00	258423.00
WebAssembly(Chrome)	0.00	4.00	4588.90	82212.60
WebAssembly(Mozilla)	0.00	4.00	3694.00	65788.00
WebAssembly(Rust, Chrome)	0.00	4.60	4993.20	96164.3
WebAssembly(Go, Chrome)	0.00	4.90	5423.20	112651.4

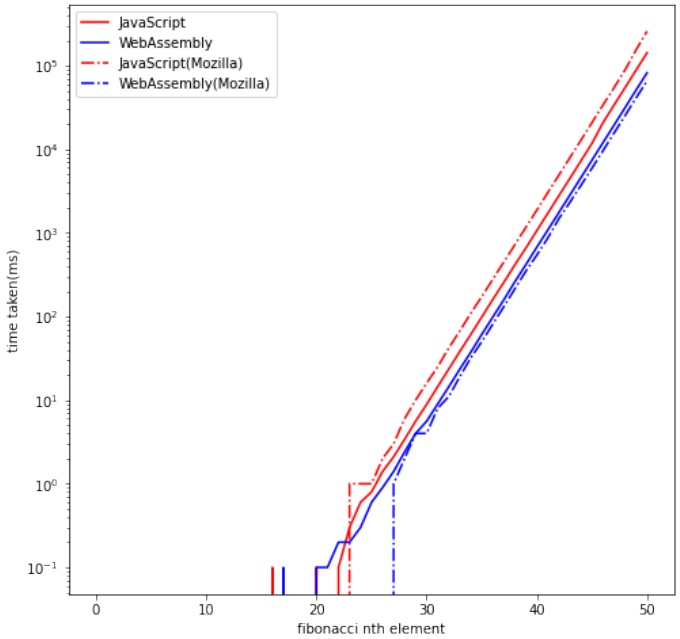

Fig. 7. Time taken to calculate Fibonacci of a number(Log scaled)

as the prime numbers get large WebAssembly clearly takes lesser time as compared to JavaScript to iterate over a large number. Time taken to check a number is prime or not is highly inconsistent but WebAssembly clearly takes the lesser time. As the time taken to check a number is prime is too short, measuring CPU usage over such short duration will not be meaningful. Time taken on Mozilla to run the same tasks also gave similar results where WebAssembly outperformed JavaScript for large prime numbers.

The CPU usage for both WebAssembly and JavaScript programs are almost same. In memory usage WebAssembly edges JavaScript by a close margin. File sizes of such small programs for WebAssembly and JavaScript may not be much of a difference but WebAssembly being a bytecode is smaller and this would prove to be very crucial when we write larger programs.

V. CONCLUSION

It is clear from the discussion above that there have always been need for the bytecode for web which could work with JavaScript in the browser to facilitate it and could handle tasks like iterating larger arrays, working with large numbers

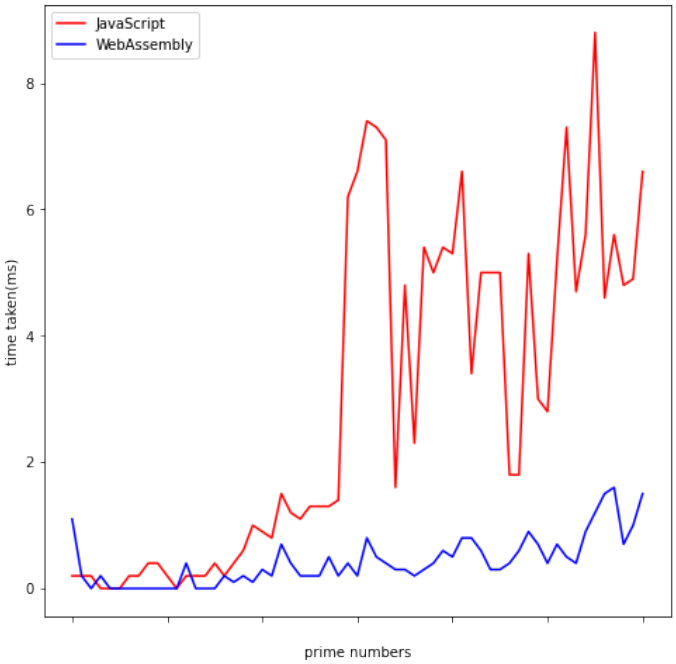

Fig. 8. Time taken to calculate if a number is prime or not

and is computationally focused. WebAssembly was claimed to be solution to the bytecode for web. By running benchmark tests it became conclusive that WebAssembly is better than JavaScript in handling large calculations, iterating on large arrays and tasks which involve large computations. In our case JavaScript outperformed WebAssembly when the computations made are easy but as the calculations got complicated WebAssembly outperformed JavaScript by a large margin, same was true when the iterations got longer. Also WebAssembly appeared to be faster in Mozilla Firefox than on Chrome, this also confirms the claim made by Lin Clark[5]. From the results we could also conclude that WebAssembly have better support for C/C++ for now as compared to other languages.

Introducing new standards also introduces new costs and challenges like maintenance, code size etc. but these must be offset by the benefits it brings in. Big applications like Tensorflow.js, Unity, AutoCAD WebApp, Google Earth etc. have started using WebAssembly in order to get performance gains that we get using WebAssembly. WebAssembly is relatively new and it is adding features day by day. In future features like Exception Handling, larger linear memory sizes which extends current limit of 2^{32} bits, ability for single module to use multiple memories, module linking within WebAssembly etc. There could also be more features which could make it easy for developers to write WebAssembly code.

REFERENCES

[1] @appcypher. *Awesome wasm Languages*. https://github.com/appcypher/awesome-wasm-langs. 2021.

[2] Ars Staff. *The Web is getting its bytecode: WebAssembly*. https://arstechnica.com/information-technology/2015/06/the-web-is-getting-its-bytecode-webassembly/. June 2015.

[3] *asm.js*. http://asmjs.org. Accessed: 2022-3-17.

[4] Boris Cherny. *Programming TypeScript: making your JavaScript applications scale*. O'Reilly Media, 2019.

[5] Lin Clark. "Making webassembly even faster: Firefox's new streaming and tiering compiler". In: *Mozilla Hacks–the Web developer blog, January* (2018).

[6] World Wide Web Consortium. *WebAssembly Core Specification-W3C Recommendation*. https://www.w3.org/TR/wasm-core-1/. 2019.

[7] Alan Donovan et al. *PNaCl: Portable native client executables*. 2011.

[8] WebAssembly Community Group. *Introduction- WebAssembly 1.1*. https://webassembly.github.io/spec/core/intro/introduction.html. 2022.

[9] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[10] Tony Parisi. *WebGL: up and running*. " O'Reilly Media, Inc.", 2012.

[11] *Web APIs*. en. https://developer.mozilla.org/en-US/docs/Web/API. Accessed: 2022-3-17.

[12] WebAssembly. *WebAssebmly-High Level Goals*. https://webassembly.org/docs/high-level-goals/. 2021.

[13] WebAssembly. *WebAssebmly-Roadmap*. https://webassembly.org/roadmap/. 2022.

[14] WebAssembly. *WebAssebmly-Toolings*. https://webassembly.org/docs/tooling/. 2021.

[15] WebAssembly. *WebAssebmly-Web Embeddings*. https://webassembly.org/docs/web/. 2021.

[16] Bennet Yee et al. "Native client: A sandbox for portable, untrusted x86 native code". In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE. 2009, pp. 79–93.

[17] 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*. 2011, pp. 301–312.