Analysis of WebAssembly as a Strategy to Improve JavaScript Performance on IoT Environments

Fernando L. Oliveira
Lardev Research Group – Federal Institute Farroupilha – IFFar
Technological Development Center – CDTec
Federal University of Pelotas – UFPel
Pelotas, Brazil
fernando.oliveira@iffarroupilha.edu.br

Júlio C. B. Mattos
Technological Development Center – CDTec
Federal University of Pelotas – UFPel
Pelotas, Brazil
julius@inf.ufpel.edu.br

***Abstract*—JavaScript language (JS) has been widely used in recent years applied to browsers-context. Yet JS is being applied to other backgrounds such as server-side programming, mobile applications, games, robotics, and the Internet of Things (IoT). JavaScript is suitable for programming IoT devices due to event-driven oriented architecture. However, it is an interpreted language, so it has a lower performance than a compiled language. This paper assesses the use of WebAssembly as a strategy to improve the performance of JavaScript applications in the IoT environment. The experiments were performed on a Raspberry Pi using the Ostrich Benchmark Suite. We run the algorithms in JavaScript, WebAssembly, and C language while collecting data about device resource consumption. Our results showed that JavaScript performance could be improved by 39.81% in terms of execution time, a tiny gain in memory usage, and reduced battery consumption by 39.86% when using WebAssembly.**

***Index Terms*—JavaScript, WebAssembly, Raspberry PI, Internet of Things**

I. Introduction

The Internet of Things (IoT) is a paradigm in which ordinary objects can be connected to the Internet and has processing capacity to analyze the environment and make decisions without human intervention [1]. According to Transforma Insights [2], the forecast until the year 2030 will be a total of 24.1 billion active IoT devices, with a growth rate of 11% per year.

Programming IoT devices requires specific knowledge about embedded systems and compiled language like the C language [1]. However, high-level languages can facilitate programming and may are more suitable to the context of smart devices.

JavaScript is a high-level, lightweight, dynamic, and untyped programming language. It allows the use of functional and object-oriented paradigms, and its execution model is based on the interpretation of the source code [3].

Over time the language has gained prominence, the vast majority of sites use JS, and modern frameworks are designed based on it [3]. According to Stack Overflow [4], for the eighth year in a row, JavaScript has maintained it as the most commonly used programming language with an estimated community of 12 million developers worldwide [5].

JavaScript might be used to simplify the programming of IoT devices due to its event-oriented architecture. However, it is an interpreted language. Hence the programs cannot be executed directly on the CPU (Central Processing Unit) [6].

Typically, interpreted languages can be penalized in terms of performance when compared to compiled programs, and mainly in embedded systems can consume more resources like CPU, memory, and energy. For this kind of device, battery consumption is an essential issue. Therefore, to allow the JavaScript language in IoT programming, it is necessary to implement techniques or tools to improve JS performance.

The vast majority of approaches to improve JavaScript performance focus on the dynamic code translation process to reduce the translation overhead [7]. Oppositely, WebAssembly (WASM) presents a new proposal that can improve the performance of JS applications. WASM is a low-level language similar to assembly, which promises performance closest to compiled languages [8].

This paper presents the analysis of a strategy to improve the performance of the JS language in IoT environments. We adopt WebAssembly as an optimization technique and evaluate how much it consumes in terms of resources when compared to JavaScript.

Our experiment was performed from the execution of benchmark algorithms executed through WebAssembly under the NodeJS interpreter on a Raspberry Pi.

The experimental results point out that JavaScript performance could be improved by 39% in terms of execution time, a tiny gain in memory usage, and reduced battery consumption by 39%. In addition, we have identified which situations are most suitable for applying the WebAssembly approach.

The outline of this paper is as follows: Section II presents the essential concepts about WebAssembly and JS execution; Section III show the related work; Section IV presents the methodological process of the research; Section V shows the results, and finally, section VI presents the conclusions.

II. Background

The main goal of WebAssembly is to improve the JavaScript application performance while being a flexible and platform-independent model. To provide some essential background, we survey the most directly relevant ancestors about the execution model of JavaScript and WebAssembly.

A. JavaScript Execution Model

The JavaScript execution model is based on the interpreter, which executes the JS code. The JavaScript engine can be implemented as a standard interpreter or just-in-time compiler that compiles JS into bytecode [9]. To present the concepts, we use the Google V8 [10] interpreter as a model.

The V8 engine is used inside Google Chrome browser and is the basis for NodeJS [11] (the most popular server environment runtime for JS). The Figure 1 shows the overview of the JavaScript program execution.

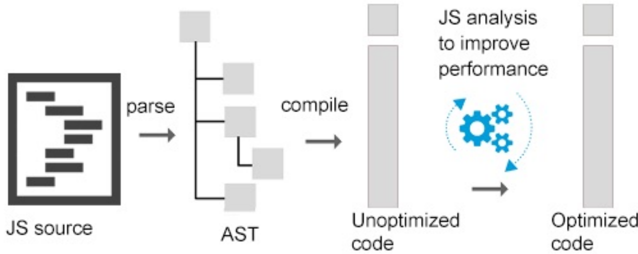

Fig. 1. Overview of the JavaScript pipeline

Figure 1 presents the typical execution flow of the JS program. First, the source code needs to be analyzed to convert the text format into tokens and generate an AST (Abstract Syntax Tree).

The V8 uses a mechanism called TurboFan for analysis if something is running slow if there are bottlenecks and access points to optimize them. While it is a powerful approach, the verification code, and decision about what needs to optimize, implies in CPU use. In other words, it means higher power consumption. Its an issue at the IoT devices powered by the batteries.

In this sense, to overcome these limitations, solutions are proposed to face performance issues and overhead of verification, one good strategy is the WebAssembly.

B. WebAssembly Approach

Designed from the four major browser vendors – Google, Microsoft, Mozilla, Apple – WebAssembly (WASM) is a portable low-level bytecode that the goal is to serve a universal compiler target [8]. Towards this end, WASM provides a set of architectural instructions that can be incorporated into several host environments, such as web browsers, cloud computing platforms, or IoT devices [12].

WebAssembly brings an innovative proposal to enable low-level code on the environment target [8]. The WASM code is produced through compilation from C/C++, Rust, JavaScript/TypeScript, using the official toolchain [12].

The WASM module is loaded, decoded, and compiled. The module exposed one API that the instance functions which can be directly reached through JavaScript application.

Figure 2 shows the JavaScript execution with WASM module. The module has already undergone optimization during the compilation phase. In addition, the analysis of code is also not necessary because a few of the steps in the process can be skipped.

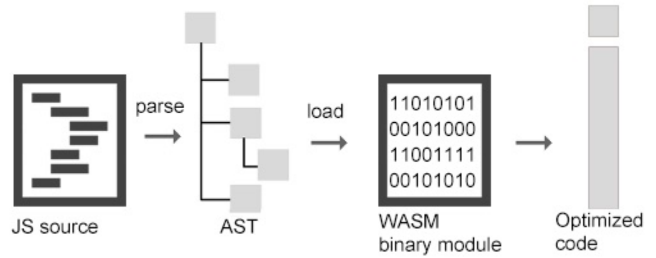

Fig. 2. JavaScript execution pipeline throught WASM

On the other hand, modern JavaScript engines have been investing in research to achieve highly optimize levels and improve their runtimes. The load WASM bundle and the execution time might not be very advantageous for simple, specific issues. In this sense, We seek to explore the performance trade-off of JavaScript through the WASM module in the IoT environment.

III. Related work

Due to its original purposes, the vast majority of works investigate the use of WebAssembly as a tool associated with the browser environment.

The introductory study was proposed by Haas et al. [8]. In their work, WebAssembly is presented in detail, and some tests were applied using PolyBench benchmark employed in browser-based environments. They conclusion show that WebAssembly is a competitive alternative running 10% slower than native code.

The author Conrad Watt [13], present Speedy.js, a cross-compiler that translates JavaScript/TypeScript to WebAssembly. They work aiming at the performance-critical web code, to provide translates in an optimization way. With this approach, it can manage to make compute-intense web code up to four times faster, while reducing runtime fluctuations to the half.

Herrera et al. [14], provided Apart from the Ostrich benchmark suite, five research questions were framed to investigate the improvement of JavaScript-based browser engines, the relative performance of JavaScript and WebAssembly with variations in portable versus vendor-specific browsers. In particular, the authors included a Raspberry Pi in their evaluation. In general, they registered a gain of speedup of 19% performance against native code.

On the opposite way, Jangda et al. [15] built an extension to BROWSIX to enable the execution of unmodified WebAssembly-compiled Unix applications directly inside the browser. The goal was to analyze the performance of WebAssembly compared to native code. For that, they choose the SPEC CPU suite as benchmarks and conclude that applications compiled to WebAssembly run slower by an average of 45% (Firefox) to 55% (Chrome).

There are divergent points in the literature regarding the performance of WebAssembly and, commonly, analyzes are

performed using browser-based environments. Thus it is essential to evaluate it from other perspectives and in different architectures.

In this sense, we contextualize WebAssembly as a means to enable the execution of JavaScript in embedded systems. Moreover, this paper differs from the others because of the focus on the IoT environment. In particular, how WebAssembly stacks against JavaScript in terms of performance and consumption of resources like the battery, CPU, and memory.

IV. Research Methodology

Regarding the evaluation of WebAssembly to improve JavaScript application performance, we performed a series of tests with a set of algorithms for different purposes. The goal is to assess if there is a performance gain and how much it impacts battery-powered devices.

The Raspberry Pi (RPi) is a single-board computer (SBC). Due to its versatility and low cost, it has become popular. We chose RPi because it is widely used, easy to manage, and enables fast prototyping for the IoT environment. Also, it is suitable for a wide variety of applications [16]. Table I shows the technical details about the evaluation setup.

TABLE I
Development environment

Component	Description
SBC	Raspberry PI3 model B 1.2GHz (64-bit) quad-core ARM Processor 1GB of RAM
Memory Card	SanDisk 16GB Class 10
Operating Systems	Raspberry Pi OS 32-bit Lite (5.4.51-v7+)
JS Engine	Node.js (12.18.2)
WASM toolchain	Emscripten (1.40.0)
C compiler	gcc (Raspbian 8.3.0-6+rpi1)
Profiling tool	Perf (4.9.82)
Current sensor	INA219 Zerø-Drift
Microcrontroler	Arduino UNO R3

Raspberry Pi is able to run a full operating system. The official image is a Debian-based Linux distribution knows as Raspberry OS. We keep the basic OS version to reduce the overhead of software that will not be used.

Regarding the JavaScript engine, we proceed with the NodeJS [11]. NodeJS represents state of the art in terms of server-side platform. It is widely used for JavaScript execution and has a wide range of tools/frameworks for application development [3].

Concerning the benchmarks, we select the Ostrich Benchmark Suite [17]. This benchmark provides some facilities for evaluating JavaScript against WebAssembly because it gives the same implementation of the algorithm in JS and C. Furthermore, Ostrich provides a make file to automate the build process of C compiler and Wasm module generation. Table II shows the benchmarks used.

The Ostrich Benchmark Suite is composed of 12 algorithms. However, in our analysis, we will work with only eight because four algorithms did not compile on the Raspberry Pi due to the incompatibility of versions and architecture.

TABLE II
Ostrich Benchmarks [17]

Benchmark	Description
nw	An algorithm to compute the optimal alignment of two protein sequences
crc	An error-detecting code which is designed to detect errors caused by network transmission or any other accidental error
nqueens	An algorithm to compute the number of ways to put down n queens on an n x n chess board where no queens are attacking each other
lud	A LU decomposition is performed on a 1024 x 1024, randomly-generated matrix
bfs	A breadth-first search algorithm that assigns to each node of a randomly generated graph its distance from a source node
levamd	An algorithm to calculate particle potential and relocation due to mutual forces between particles within a large 3D space
fft	The Fast Fourier Transform (FFT) function is applied to a random data set
srad	A diffusion method for ultrasonic and radar imaging applications based on partial differential equations

To generate the WASM modules, we use as input the C source code and the Emscripten compiler [18]. The module compilation required to rename the parameter TOTAL_MEMORY to INITIA_MEMORY to meet the restrictions of the new version of the WebAssembly compiler, and we included the setting ALLOW_MEMORY_GROWTH = 1 to allow memory allocation when necessary.

A. Power Measurement

To evaluate energy consumption, we built an external measurement based on the INA219 [19] current sensor. We make the power meter using the Arduino board to capture the current sensor's measured values. The data are stored on the SD card. The rate of samples is 150 registries per second, and the current consumption is recorded in mA.

B. System State Monitoring

We install the required software to measure the performance and proceed with the evaluation from the clean Linux image. Moreover, unnecessary services like ssh, dhcp, and sbus were disabled.

Thus, to collect the performance information, we selected the Perf profiling tool [20]. Perf is a profiling mechanism for Linux-based systems, and It can instrument CPU performance from events counters. It is possible to collect useful information from the counters to understand the execution of the applications, such as instructions executed, memory allocation, or cache-misses suffered

Perf offers different ways to collect profiling information. We record the execution section using the options [-d -e] to obtain, respectively, detailed information about the cache memory allocation and the number of instructions and cycles

executed by the CPU. Furthermore, it also provides the execution time in seconds for each algorithm.

C. Evaluation

The experiment consists of measuring each JS benchmark execution through NodeJS. Every single operation was repeated 30 times to obtain data consistency and more accurate statistical information [21].

We run the benchmarks and use tools to extract performance statistics. However, our analysis is performed to exclude a potential system overhead.

We have established a methodology for the execution of each test. Before performing the test, we collect statistical data about the idle state. In the sequence, we load the JavaScript interpreter. This action aims to remove the load time of the application engine. In this phase, we collect the performance without running the test. Finally, we execute and capture the statistics of each benchmark.

To control the test execution, we created a bash file that runs the experiments in an automated way. Each algorithm is repeated 30 times. Between one performance and another, we included ten seconds pause for the system to go back to idle mode and the drained current returns to usual variation.

V. Results and Discussion

This section presents the results of the comparative experiments between JavaScript and WebAssembly. The reports are segmented by language and grouped by benchmark.

Although our focus is not on compiled languages, we make comparisons with it. In this way, we decided to include test results for the C language. However, the main analysis continues around WebAssembly.

Table III shows the raw data regarding the performance of the benchmarks. The information represents the average from 30 runs per algorithm by approach performed on the Raspberry Pi. Overall, WebAssembly showed better results; however, memory consumption requires a more in-depth analysis. Table IV summarizes the comparison between the approaches.

From the perspective of absolute numbers, it is evident that there have been substantial improvements in execution time and energy consumption, while memory has a positive value, but far from the other metrics. Figure 3 presents the analysis of execution time.

Figure 3 highlights the executions of the *bfs* and *srad* algorithms. In these cases, the performance has increased by more 50%. To better understand the difference between the approaches, we analyzed the *srad* algorithm to know how it works and why WASM was superior.

The goal is to find out which of the implementation aspects take the longest to execute and, consequently, to identify more favorable scenarios for the application of WASM. Srad algorithm consists in diffusion method tailored to ultrasonic and radar imaging applications [17]. In a simplified way, the objective of the algorithm is to reduce speckle noise from images. For that, the implementation consists of some operations at the pixel level through arrays representing regions of the image.

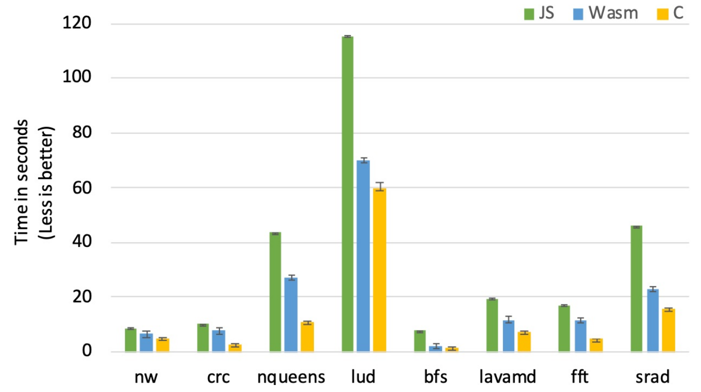

Fig. 3. Execution time analysis

In this sense, the operation of the algorithm considers functions with nested loops. Each loop interaction implies calculations to determine the next index and its values; this operation is repeated for a massive number of items. The problem is that in JavaScript, all numbers are floating-point numbers. Hence, it consumes more resources, and the JS interpreter requires an extra effort to speculate during compilation what data types will be used.

On the other hand, WebAssembly has not achieved the same performance in other algorithms that also use similar structures. WASM can be penalized when applied to simple tasks; loading codes or functions external to the application implies costs for proceeding between the JavaScript engine and the WebAssembly module. For instance, to execute a function that adds two numbers exclusively via JavaScript is faster than running the same function through WebAssembly, because to perform the sum requires to load the WASM module. Thus, the tradeoff between the load and execution might become inconvenient according to the scenario.

Therefore, loading a WASM module does not mean a performance-boost if applied for developing simple issues. Figure 4 shows the memory usage of each algorithm.

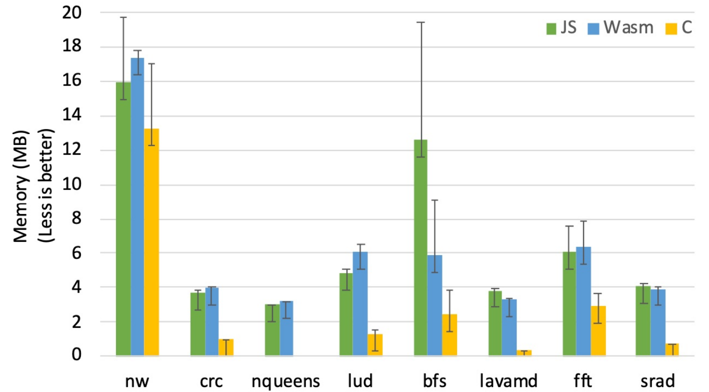

Fig. 4. Memory analysis

Regarding memory consumption, we highlight the difference in the *bfs* algorithm. To execute it, we need to reduce the number of entries sets since using the default values indicated

TABLE III
RAW EXECUTION DATA AND STANDARD DEVIATION

	Execution time (sec.)				Memory (MB)				Energy (J)			
	JavaScript		WASM		JavaScript		WASM		JavaScript		WASM	
Benchmark	SM	SD	SM	SD	SM	SD	SM	SD	SM	SD	SM	SD
nw	8.95	0.002	6.62	0.001	15.93	3.765	17.40	0.392	22.75	1.165	18.03	1.24
crc	10.35	0.003	7.86	0.002	3.67	0.147	3.94	0.122	26.30	0.800	19.99	0.87
nqueens	43.51	0.001	27.10	0.001	2.98	0.008	3.18	0.016	105.22	2.721	66.93	4.13
lud	115.30	2.281	70.06	0.032	4.87	0.231	6.04	0.508	270.64	7.984	166.62	1.53
bfs	7.79	0.041	2.24	0.000	12.64	6.844	5.85	3.245	23.77	1.830	6.00	1.11
lavamd	19.33	0.002	11.87	0.001	3.84	0.061	3.33	0.063	44.30	4.435	27.88	5.48
fft	17.25	0.012	11.79	0.004	6.10	1.425	6.41	1.443	43.22	1.249	27.43	2.06
srad	45.95	0.054	22.95	0.064	4.11	0.116	3.93	0.082	107.98	3.296	52.90	2.85

SM: Standard Mean
SD: Standard Deviation

TABLE IV
COMPARATIVE RESULTS BETWEEN JAVASCRIPT AND WEBASSEMBLY

	JavaScript Vs. WebAssembly		
Benchmark	Execution time (%)	Memory (%)	Energy (%)
nw	26.00	-9.27	20.73
crc	24.03	-7.35	23.97
nqueens	37.72	-6.48	36.38
lud	39.24	-24.09	38.43
bfs	71.20	53.72	74.74
lavamd	38.61	13.12	37.07
fft	31.66	-5.08	36.53
srad	50.05	4.46	51.01
Average	**39.81**	**2.38**	**39.86**

by the benchmark suite implied in out of memory error. This algorithm performs searches for graphs using nested loops, so each call allocates memory for parameters, local, and control variables varying according to the depth level of the graph.

WebAssembly had the worst memory consumption on *lud* algorithm. This algorithm operates on matrices inside loops, and matrix indices are calculated at each interaction based on variables that are also manipulated by the loop.

Theoretically, WebAssembly manages memory through a resizable ArrayBuffer that contains a linear array of bytes which is read and written by low-level memory instructions [8]. In this way, the dynamic indexes of the matrix imply readings that are not sequential. In general, when going through a matrix or array, the next value to be read is usually the subsequent record; however, in this scenario, the reading becomes unpredictable directly affecting the cache spatial locality.

The algorithm profiling data shows 46% of miss rate at last level cache (LLC). Furthermore, memory loads spilling registers, and extra jumps are necessary [22]. Thus, the processor needs more clock cycles to search for information.

JS garbage collection system uses non-deterministic algorithms that work when some memory allocation occurs in contrast to WASM that have a heap which allocate from the bottom of the heap, and grow the stack from the top of the heap [22].

Regarding the power consumption, the measurements are performed by second discarding the overhead for the system (idle mode). The results represent the geometric average from 30 runs recorded over the whole executions. Figure 5 shows the energy consumption.

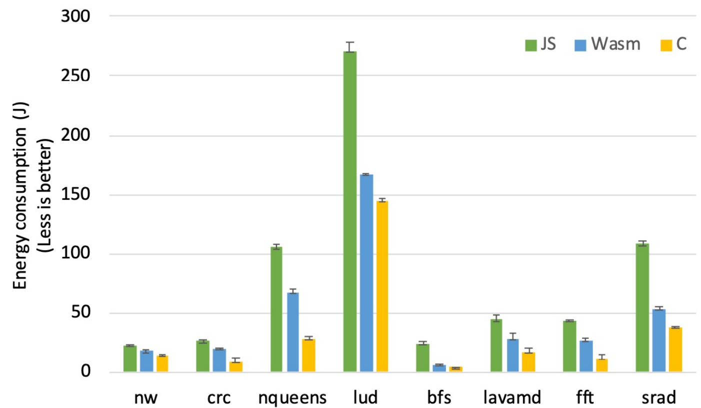

Fig. 5. Energy consumption analysis

Figure 5 exposes the average energy spend by execution. In the experimental environment, the algorithm execution did not compete directly with other applications for CPU usage. However, we have no control over the scheduling of processes by the operating system. Hence, the energy cost may be impacted by some sub-process that eventually has been scheduled during the test.

In general, JavaScript is perceived to consume more energy to develop the algorithms. It happens because the compiler performs optimizations (Turbofan [10]) in order to gain performance. Such analysis implies directly on CPU usage, so it has a higher energy cost.

Directly comparing WebAssembly against JavaScript, there is an overall gain of about 39%. Further, there is still a substantial performance in time-execution and energy saving, with an extra cost in memory consumption. We observe a more efficient performance with situations for computationally intensive tasks, as Haas et al. [8] points out.

Typically, the works found in the literature compare WebAssembly with the C language. Table V presents the execution statistics in the native language.

TABLE V
C EXECUTION STATISTICS

Benchmark	C language		
	Exec. Time (Sec.)	Memory (MB)	Energy (J)
nw	5.23	13.26	14.69
crc	2.99	0.99	8.97
nqueens	11.24	0.00	28.82
lud	59.77	1.26	145.18
bfs	1.63	2.45	4.26
lavamd	7.50	0.30	17.62
fft	4.81	2.89	11.88
srad	16.07	0.68	37.57

Comparing WebAssembly with C language, we reported that WASM runs 21% slower than native code. This rate is double that recorded by [8]. Nevertheless, a value very close to that achieved by [14] 19%. WASM proves to be a viable alternative for performance optimization, reducing the gap between compiled and interpreted languages.

On the other hand, in his work, Jangda et al. [15] highlights situations in which WebAssembly performs poorly. In our experiments, this limitation was perceived in situations in which arrays and matrices were involved because of excessive access to information that was not at the top of the memory hierarchy.

Herrera et al. [14] achieved more expressive rates between WebAssembly and JavaScript. However, in our tests, JS had a better performance. This difference probably occurs because JS interpreters have been investing a lot of effort into the optimization processes of their virtual machines, ensuring continuous improvements.

VI. CONCLUSION

This paper evaluated whether WebAssembly could improve the performance of the JavaScript language in the IoT environment. We were able to collect information about the consumption of resources such as CPU, memory, and battery from the execution of benchmarks.

The evaluations presented in this paper shown that WebAssembly is an efficient strategy to improve JavaScript performance, being superior in all evaluated items, emphasizing reducing battery consumption. On a battery-powered device with limited resources, energy consumption can be more significant than the speed of execution.

Finally, WebAssembly is a promising technology that can be applied to different contexts, mainly because it is compiled code suitable for the IoT environment optimizing the trade-off between performance and resource consumption.

For future studies, we intend to explore the use of WebAssembly through other JavaScript interpreters. We will also deepen the analysis of memory usage since this proved to be a limitation of WASM and represents a research opportunity to enable JavaScript as a language for the IoT environment.

ACKNOWLEDGMENT

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

REFERENCES

[1] L. Amorim, R. Barreto, and M. Alencar, "Tiny thing blocks: Integrating everyday objects into iot context," in *IX Brazilian Symposium on Computing Systems Engineering*. Porto Alegre, RS, Brasil: SBC, 2019, pp. 131–136.

[2] Transforma Insights, "The internet of things (iot) market 2019-2030," https://transformainsights.com/news/iot-market-24-billion-usd15-trillion-revenue-2030, 2020, accressed: jun. 2020.

[3] S. Delcev and D. Draskovic, "Modern javascript frameworks: A survey study," in *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, May 2018, pp. 106–109.

[4] Stack Overflow, "Stack overflow annual developer survey 2020," https://insights.stackoverflow.com/survey/2020, 2020, accressed: jun. 2020.

[5] Developer Economics, "State of the developer nation - 18 edition," https://www.developereconomics.com, 2020, accressed: jun. 2020.

[6] A. E. Kwame, E. M. Martey, and A. G. Chris, "Qualitative assessment of compiled, interpreted and hybrid programming languages," *Communications*, vol. 7, pp. 8–13, 2017.

[7] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: An empirical study," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 61–72.

[8] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 185–200. [Online]. Available: https://doi.org/10.1145/3062341.3062363

[9] D. Flanagan, *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[10] V8 JavaScript Engine, "Javascript and webassembly engine," https://v8.dev/, 2020, accressed: jul. 2020.

[11] Node JS, "Javascript runtime," https://nodejs.org, 2020, accressed: jul. 2020.

[12] WebAssembly Official Website, "Webassembly official website," https://webassembly.org/, 2020, accressed: jul. 2020.

[13] M. Reiser and L. Bläser, "Accelerate javascript applications by cross-compiling to webassembly," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 2017, pp. 10–17.

[14] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, "Webassembly and javascript challenge: Numerical program performance using modern browser technologies and devices," *University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2*, 2018.

[15] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: analyzing the performance of webassembly vs. native code," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 107–120.

[16] Q. He, B. Segee, and V. Weaver, "Raspberry pi 2 b+ gpu power, performance, and energy implications," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2016, pp. 163–167.

[17] Khan, Faiz and Foley-Bourgon, Vincent and Kathrotia, Sujay and Lavoie, Erick, "Ostrich benchmark suite." [Online]. Available: https://github.com/Sable/Ostrich

[18] Emscripten, "Emscripten toolchain," https://emscripten.org, 2020, accressed: jul. 2020.

[19] Texas Instruments, "Ina219 data sheet," https://www.ti.com/lit/gpn/ina219, 2020, accressed: jul. 2020.

[20] Perf, "Linux profiling with performance counters," https://perf.wiki.kernel.org, 2020, accressed: jul. 2020.

[21] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 57–76. [Online]. Available: https://doi.org/10.1145/1297027.1297033

[22] A. Guermouche and A.-C. Orgerie, "Experimental analysis of vectorized instructions impact on energy and power consumption under thermal design power constraints," 2019.