WebAssembly and JavaScript Performance Across Browsers A Comprehensive Study Using Automated Testing and Reporting

Zahidullah Sherzad, Prof. Dr. Kai Eckert

Computer science Department (Mannheim University of applied sciences)

z.sherzad@hs-mannheim.de

eckert@hs-mannheim.de

Abstract— this research investigates the performance attributes of WebAssembly (Wasm) and JavaScript across major web browsers

Keywords—— WebAssembly performance, JavaScript performance, Real-world Benchmarking, Execution Speed, Performance Optimization, Real-world applications.

I. INTRODUCTION

In the realm of web development, the choice between practical small web applications. The results consistently illuminate WebAssembly's remarkable 2.226 time's superiority in execution speed when compared to JavaScript. Microsoft Edge emerges as the top performer, showcasing consistently superior execution times across various computational tasks. Opera closely follows, securing the second position for both Wasm and JS. Google Chrome ranks third, offering competitive a comprehensive understanding of both technologies and their offshoots, thereby aiding developers in making informed decisions when selecting the most suitable option

Materials

1. Algorithms Implemented:

The research study involved the implementation and performance evaluation of several algorithms across different web browsers and technologies. Each algorithm was carefully selected to represent a diverse range of computational tasks and to provide insights into the performance of WebAssembly (Wasm) and JavaScript (JS) implementations.

NO	*Algorithm*	*Browsers*
1	*Image generation and sort*	*Chrome, Firefox, Edge*
2	*bubble array sort*	*Chrome, Firefox, Edge*
3	*Reverse array*	*Chrome, Firefox, Edge*
4	*Threshold image processing*	*Chrome, Firefox, Edge*
5	*Fibonacci*	*Chrome, Firefox, Edge*
6	*Quick sort*	*Chrome, Firefox, Edge*
7	*Convolution image processing*	*Chrome, Firefox, Edge*

Table 1: List of implemented algorithms

2. Technologies Used

The implementation and evaluation of algorithms were conducted using the following technologies:

WebAssembly (Wasm): A binary instruction format designed for high-performance execution of code in web browsers[1]. Wasm allows for near-native speed execution of algorithms written in languages such as C/C++ and Rust within web environments.

JavaScript (JS): The standard scripting language for web development, widely supported by web browsers[2]. JavaScript was used as a benchmark for comparison against WebAssembly in terms of performance and efficiency.

3. Automated Testing Setup

Automated testing was conducted utilizing the Selenium WebDriver framework within Python, offering a platform-agnostic interface for browser automation. This facilitated the programmatic interaction with web elements and the execution of test scripts. The setup automation encompassed various libraries and utilities, including Pytest for test writing and execution, along with platform [3] and psutil for system resource information retrieval [4]. Random and time modules were employed for generating random numbers and time-related functions, respectively, while os facilitated operating system-dependent functionality. Additionally, csv was utilized for reading and writing CSV files, and selenium was crucial for browser automation. The WebDriver class enabled automated browser control, while By provided mechanisms for element location on web pages. WebDriverWait, along with expected_conditions, facilitated waiting for specific conditions before code execution. Statistical calculations were carried out using the statistics module, and keyboard actions were simulated using Keys. Complex actions with WebDriver were accomplished using Action Chains [5], while matplotlib.pyplot aided in creating visualizations[6]. For document creation, reportlab was utilized, with reportlab.lib.pagesizes defining standard page sizes, and reportlab.platypus allowing for PDF document creation[7]. The Paragraph class was used to add paragraphs, Spacer for adding space, and Image for incorporating images into PDF documents. Predefined styles were provided by reportlab.lib.styles, and sys provided access to interpreter-related variables and functions.

Regimen, facilitating a thorough examination of performance across various environments. Testing was conducted across different platforms such as Windows, ensuring inclusivity and accounting for potential performance variations specific to each platform.

Method

1. Implementation and Compilation

The algorithms were initially implemented in C programming languages to leverage their computational efficiency.

emcc abc.c -o abc.js -s NO_EXIT_RUNTIME=1 -o2

emcc: This is the command-line interface (CLI) for the Emscripten compiler, which is used to compile C/C++ code into WebAssembly (Wasm) or JavaScript (JS) for execution in web browsers.

-s NO_EXIT_RUNTIME=1: This option is used to control the behavior of the Emscripten runtime environment. Setting NO_EXIT_RUNTIME=1 indicates that the program should not exit the runtime environment when it finishes execution. This can be useful if you plan to call functions from the compiled code multiple times without reloading the runtime environment.

-O2: This option specifies the optimization level for the compiler. In this case, -O2 indicates a moderate level of optimization to improve the performance of the compiled code.

3. Browser Selection

For comprehensive evaluation, we chose a diverse array of web browsers, focusing on the latest versions of three major players: Google Chrome, Mozilla Firefox, and Microsoft Edge. This selection ensures relevance to users relying on these widely-used platforms. Moreover, our testing extends beyond these browsers, as our framework easily accommodates additional browsers and versions. By utilizing Selenium web drivers, one can seamlessly incorporate new browsers or versions into the testing

4. Experiment Execution

Experiments were conducted in a controlled environment to ensure consistency and accuracy of performance measurements. For the experiment, we installed the VNC server with 4 GB of Intel64 Family 15 Model 107 Stepping 1 GenuineIntel RAM. Additionally, we utilized the latest versions of Chrome (123.0.6312.107), Firefox (124.0.2), and Microsoft Edge (123.0.2420.97) for our testing. We automated each algorithm in Python using the Selenium web driver, executing each algorithm approximately 100 times for both WebAssembly (Wasm) and JavaScript (Js) implementations. The performance times were stored in separate CSV files for each browser, facilitating efficient data management. Findings and facilitates easy interpretation of the results.

5. Data Collection and Analysis

After collecting the data, we processed the CSV files, calculating the mean and median execution times for Wasm and Js in different browsers. Subsequently, we generated graphical representations of the performance data, creating graphs for Wasm and Js in each browser. To ensure comprehensive analysis, we stored the complete reports for each algorithm separately. Finally, we merged all the individual PDF reports into a single PDF file, providing a comprehensive overview of the performance of each algorithm across different browsers and their respective versions. It's worth noting that the automation process took approximately 2 hours and 10 minutes to complete, from executing the algorithms to generating and merging the reports into a single PDF file. This thorough approach ensures reliable findings and facilitates easy interpretation of the results.

RESULTS

Below is a performance comparison table showcasing the execution times (in milliseconds) of WebAssembly (Wasm) and JavaScript (Js) implementations across various algorithms in three major web browsers: Chrome, Firefox, and Microsoft edge. Each row represents a specific algorithm, and the corresponding execution times are provided for both Wasm and Js implementations in each browser.

		WebAssembly			**JavaScript**		
S.NO	**Algorithms**	**Chrome/ms**	**Firefox/ms**	**Edge/ms**	**Chrome/ms**	**Firefox/ms**	**Edge/ms**
1	*Image Generation and Sort*	1691.952	1672.420	3371.76	3119.460	3814.690	3371.767
2	*Bubble Array Sort*	85.445	93.280	126.655	48.568	126.655	48.711
3	*Reverse Array*	0.019	0.005	0.013	0.019	0.030	0.011
4	*Threshold image processing*	4.251	3.665	4.253	12.587	11.325	12.402
5	*Fibonacci*	328.581	278.985	360.134	295.288	495.220	293.649
6	*Quick sort*	2.239	2.065	2.095	4.665	4.295	4.766
7	*Convolution Image processing*	139.384	176.615	139.933	1006.370	1848.795	1024.342

Table 2: List of implemented algorithms with performance in different browsers

Image generation and sort algorithm

The results of the tests showed diverse performance among the three browsers. Our result showed that Chrome browsers demonstrated 1691.952 ms in Wasm and 3119.460 ms in Js technology respectively in image Generation tasks. Image generation tasks in Firefox Browser with Wasm technology was 1672.420 and in Js it was 3814.690 MS. the image generation in Wasm technology with Edge Browser was 1690.483 ms while in Js technology the image generation took 3371.767 ms to be generated as show in Fig. 1.

Bubble sort algorithm

The Bubble Sort Report compares the performance of Bubble Sort algorithm implementations in three major Browsers: Chrome, Firefox, and Edge. For each browser, the report provides performance statistics in terms of mean and median execution times for both WebAssembly (Wasm) and JavaScript (Js) implementations of the Bubble Sort algorithm. In Chrome, the mean execution time for Wasm is 85.445 ms with a median of 82.550 ms, while for Js its 48.568 ms mean and 47.800 ms median. Firefox shows Wasm mean and median at 93.280 ms and 92.000 ms respectively, while Js has 126.655 ms mean and 125.000 ms median. In Edge, Wasm mean and median are 85.153 ms and 82.300 ms, and Js mean and median are 48.711 ms and 47.800 ms. these statistics offer insights into the relative performance of Bubble Sort across different browsers and implementations. Fig. 2.

Reverse Array algorithm

The Reverse Array Report examines the performance of reversing arrays in three prominent web browsers: Chrome, Firefox, and

Edge. It presents performance statistics in terms of mean execution times for both WebAssembly (Wasm) and JavaScript (Js) implementations of the reverse array operation. In Chrome, the mean execution time for Wasm is 0.019 milliseconds, while for Js it's also 0.019 milliseconds. Firefox, on the other hand, demonstrates a faster execution time for Wasm, with a mean of 0.005 milliseconds, compared to Js with a mean of 0.030 milliseconds. Similarly, Edge shows a faster mean execution time for Wasm at 0.013 milliseconds compared to Js at 0.011 milliseconds. These statistics provide insights into the proficiency of array reversal across dissimilar browsers and implementations. Fig. 3.

Threshold image processing algorithm

The Threshold Image Processing Report examines the performance of threshold image processing operations across three leading web browsers: Chrome, Firefox, and Edge. It provides performance metrics focusing solely on mean execution times for both WebAssembly (Wasm) and JavaScript (Js) implementations of the threshold image processing algorithm. In Chrome, the mean execution time for Wasm is 4.251 milliseconds, while Js averages at 12.587 milliseconds. Similarly, Firefox exhibits a mean execution time of 3.665 milliseconds for Wasm and 11.325 milliseconds for Js. Meanwhile, Edge showcases comparable results to Chrome, with mean execution times of 4.253 milliseconds for Wasm and 12.402 milliseconds for Js. These statistics illuminate the relative performance of threshold image processing across different browsers and implementations. Fig. 4.

Fibonacci algorithm

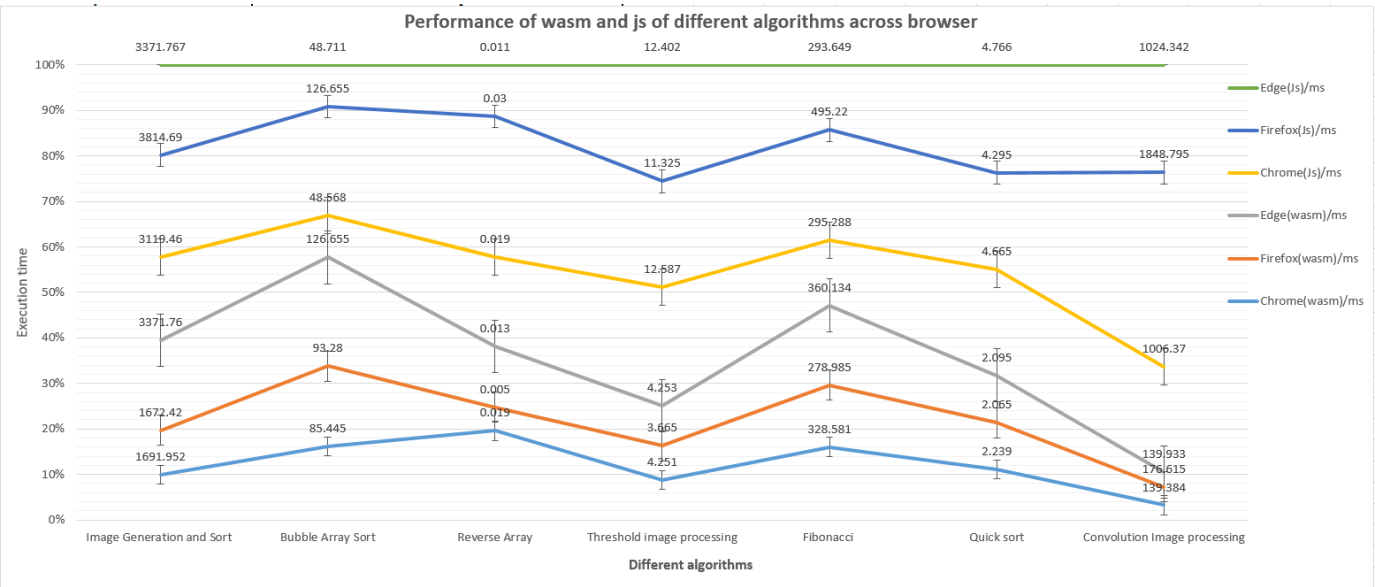
The Fibonacci Report evaluates the performance of Fibonacci sequence calculations across three prominent web browsers: Chrome, Firefox, and Edge. It presents performance statistics regarding mean execution times for both WebAssembly (Wasm) and JavaScript (Js) implementations of the Fibonacci algorithm. In Chrome, the mean execution time for Wasm is 328.581 milliseconds, while Js averages at 295.288 milliseconds. Firefox demonstrates a mean execution time of 278.985 milliseconds for Wasm and 495.220 milliseconds for Js. Meanwhile, Edge shows a mean execution time of 360.134 milliseconds for Wasm and 293.649 milliseconds for Js. These statistics provide insights into the relative performance of Fibonacci sequence calculations across different browsers and implementations. Fig. 5.

Quick sort algorithm

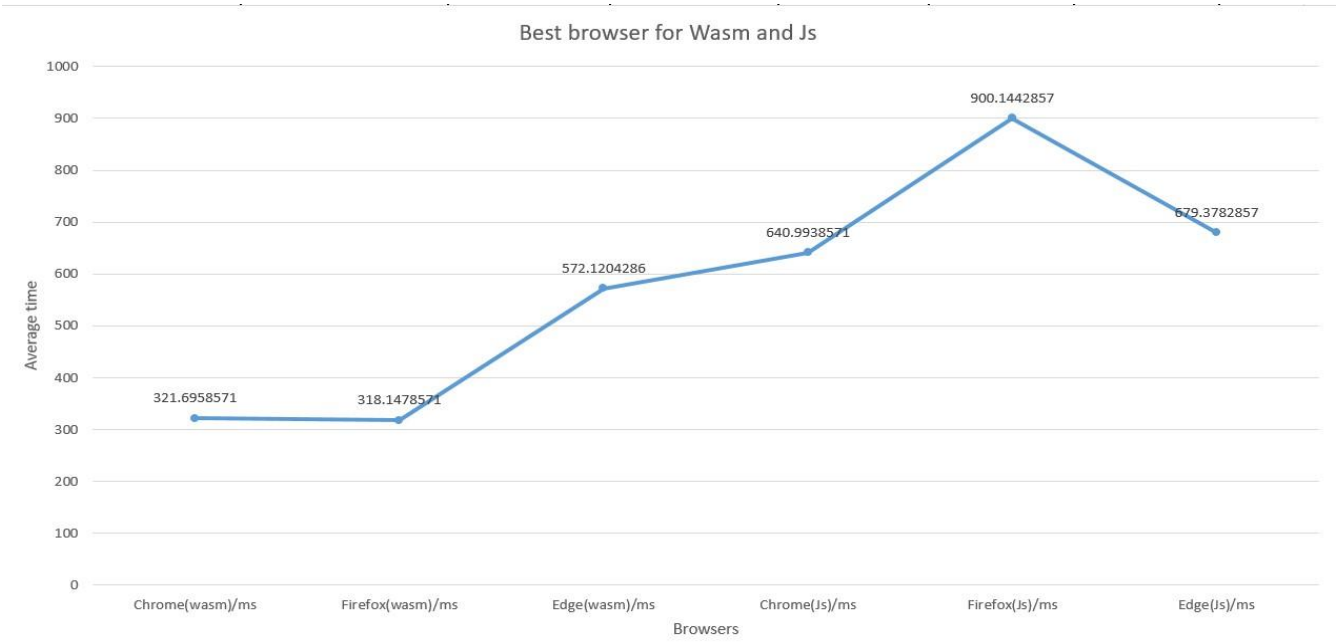
The Quick Sort Report evaluates the performance of the Quick Sort algorithm in three major web browsers: Chrome, Firefox, and Edge. It provides performance statistics focusing solely on mean execution times for both WebAssembly (Wasm) and JavaScript (Js) implementations. In Chrome, the mean execution time for Wasm is 2.239 milliseconds, while Js averages at 4.665 milliseconds. Firefox exhibits a mean execution time of 2.065 milliseconds for Wasm and 4.295 milliseconds for Js. Edge shows similar trends, with a mean execution time of 2.095 milliseconds for Wasm and 4.766 milliseconds for Js. These statistics shed light on the relative performance of Quick Sort across different browsers and implementations. Fig. 6.

Convolution image processing algorithm

The Convolution Image Processing Report analyzes the performance of convolution image processing operations across three major web browsers: Chrome, Firefox, and Edge. It presents performance statistics focusing solely on mean execution times for both WebAssembly (Wasm) and JavaScript (Js) implementations. In Chrome, the mean execution time for Wasm is 139.384 milliseconds, while Js averages at 1006.370 milliseconds. Firefox exhibits a mean execution time of 176.615 milliseconds for Wasm and 1848.795 milliseconds for Js. Edge shows similar trends, with a mean execution time of 139.933 milliseconds for Wasm and 1024.342 milliseconds for Js. These statistics offer insights into the relative performance of convolution image processing across different browsers and implementations. Fig. 7.



In our comprehensive analysis across various algorithms, we have observed distinct performance trends among different web browsers for both WebAssembly (Wasm) and JavaScript (Js). Notably, our findings reveal that Firefox emerges as the top-performing browser for Wasm, demonstrating superior execution times across our algorithmic implementations. Following closely, Chrome exhibits commendable performance, securing the second position for Wasm, while Edge attains the third position. Conversely, for JavaScript, Chrome emerges as the leading browser, showcasing optimal performance across our test cases. Edge follows suit as the second best-performing browser for Js, with Firefox trailing behind in the third position. These findings shed light on the nuanced performance variations among popular web browsers, providing valuable insights for developers and users alike. For a visual representation of our results, please refer to the following graphical representation.



These individual graphs represent the performance of each algorithm across different web browsers, meticulously generated using Python web automation techniques. Each graph provides a visual depiction of the execution times for the respective algorithm in popular browsers such as Chrome, Firefox, and Edge. By automating the data collection and visualization process, we ensure accuracy and consistency in our analysis, allowing for comprehensive insights into browser performance variations. These graphical representations serve as valuable tools for understanding the comparative efficiency of algorithms across different browser environments. The following graphs are explained in the result section of the paper.

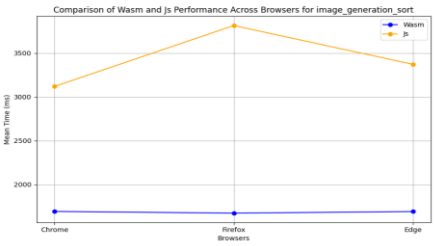


Figure 2: image generation and sort

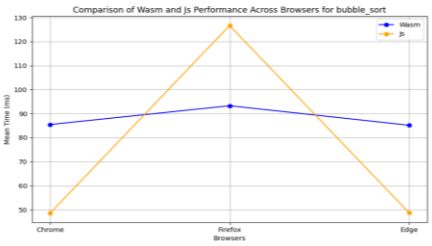


Figure 2: Bubble sort

Figure 3: Reverse array

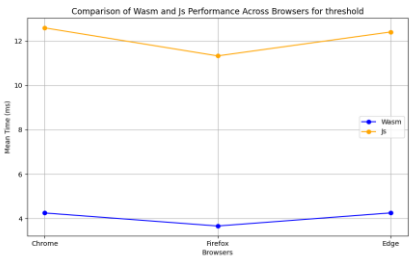


Figure 4: Threshold image processing

Figure 5: Fibonacci algorithm

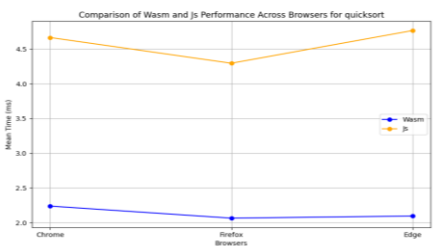


Figure 6: Quick sort

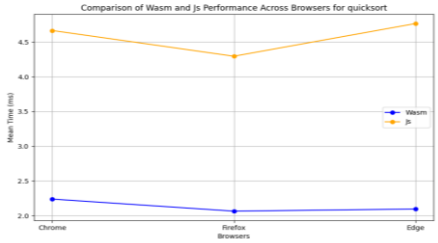


Figure 7: Convolution algorithm

As there is no any data available for data automation in theses technology. In this regard, the main objective of this

II. REFERENCES

[1] W. Wang, “Empowering Web Applications with WebAssembly: Are We There Yet?,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Melbourne, Australia: IEEE, Nov. 2021, pp. 1301–1305. doi: 10.1109/ASE51524.2021.9678831.

[2] D. Flanagan, *Java Script: o guia essencial*. Bookman, 2012.

[3] N. T. T. Soe, N. Wild, S. Tanachutiwat, and H. Lichter, “Design and Implementation of a Test Automation Framework for Configurable Devices,” in *2022 4th Asia Pacific Information Technology Conference*, Virtual Event Thailand: ACM, Jan. 2022, pp. 200–207. doi: 10.1145/3512353.3512383.

[4] J. Hunt, “Performance Monitoring and Profiling,” in *Advanced Guide to Python 3 Programming*, in Undergraduate Topics in Computer Science. , Cham: Springer International Publishing, 2023, pp. 505–517. doi: 10.1007/978-3-031-40336-1_44.

[5] O. Pursky, V. Babenko, O. Nazarenko, O. Mandych, T. Filimonova, and V. Gamaliy, “Framework Development for Testing Automation of Web Services Based on Python,” in *Green Sustainability: Towards Innovative Digital Transformation*, vol. 753, D. Magdi, A. A. El-Fetouh, M. Mamdouh, and A. Joshi, Eds., in Lecture Notes in Networks and Systems, vol. 753. , Singapore: Springer Nature Singapore, 2023, pp. 375–388. doi: 10.1007/978-981-99-4764-5_24.

[6] “Comparative Analysis of Data Visualization Libraries Matplotlib and Seaborn in Python,” *Int. J. Adv. Trends Comput. Sci. Eng.*, vol. 10, no. 1, pp. 277–281, Feb. 2021, doi: 10.30534/ijatcse/2021/391012021.

[7] “reportlab-userguide.pdf.”