Comparing the Energy Consumption of WebAssembly and JavaScript in Mobile Browsers

Dennis Pockstaller[1][a], Stefan Huber[2][b] and Lukas Demetz[1][c]
[1] *University of Applied Sciences Kufstein, 6330 Kufstein, Austria*
[2] *University of Innsbruck, 6020 Innsbruck, Austria*

Keywords: WebAssembly, Energy-Efficiency, Mobile Web Engineering.

Abstract: With WebAssembly, a new web technology has been developed that allows compiled bytecode to be executed directly in the browser, which, unlike JavaScript code, does not have to be initially compiled by the browser and can therefore be executed faster. This allows the development of complex web applications. A challenge for these complex web applications is the increasing importance of mobile devices and their limited battery capacity. The goal of this study is to determine whether the energy consumption of web applications can be reduced by using WebAssembly instead of JavaScript. For this purpose, an automated experiment was performed on Android smartphones with different algorithms using WebAssembly and JavaScript using common browsers. The energy consumption was measured hardware-based with the Monsoon HVPM measuring device. The results show that WebAssembly consumes about 20% to 30% less energy than JavaScript. In addition, differences between the two tested browsers, Chrome and Firefox, in the energy consumption of JavaScript and WebAssembly were found. This potential reduction of energy consumption also allows to reduce the user's CO2 footprint. The flexible study design used, allows for further investigations with other types of devices and other compilers.

1 INTRODUCTION

Applications are being increasingly developed as browser-based web applications to achieve a consistent user experience between different platforms, such as smartphones, tablets and desktops. New web technologies are continuously emerging and being standardized, making the browser an attractive target for modern application development. WebAssembly (Wasm) is one of these technologies, which allows compiled bytecode to be executed within the browser, unlike JavaScript (JS) code, which must be processed by an interpreter and just-in-time compiler when called.

A promised advantage of Wasm is its ability to achieve near-native performance, when executing computationally intensive or performance-critical code. In addition, existing code, created in programming languages such as C/C++ or Rust, can be compiled into Wasm and thus executed in the browser (Mozilla Developer Network, 2022). This highly increases the portability of existing codebases.

The battery capacity of mobile devices is a significant factor that can limit the user experience of modern web applications. This is especially true for resource-intensive use cases such as machine learning, games, and virtual or augmented reality experiences. Given the key attributes of Wasm, we state the hypothesis, that Wasm could be used as a substitute to JS for demanding workloads and, as a result, make web applications more energy efficient. To test this hypothesis, this study has defined the following research questions:

- RQ1: How does the energy consumption of executing intensive workloads in Wasm bytecode compare to that of JS?
- RQ2: How does the energy consumption of executing intensive workloads in Wasm and JS vary across different browsers?

To address these research questions, we conducted a controlled experiment where we executed intensive workloads using both JS and Wasm within two different browsers and measured the resulting energy

[a] https://orcid.org/0000-0003-0138-4117
[b] https://orcid.org/0000-0001-7229-9740
[c] https://orcid.org/0000-0001-8317-5049

Pockstaller, D., Huber, S. and Demetz, L.
Comparing the Energy Consumption of WebAssembly and JavaScript in Mobile Browsers.
DOI: 10.5220/0012205600003584
In *Proceedings of the 19th International Conference on Web Information Systems and Technologies (WEBIST 2023)*, pages 121-127
ISBN: 978-989-758-672-9; ISSN: 2184-3252

consumption. The execution was carried out on two different mobile devices to provide a comprehensive understanding of how browser and device variations impact energy consumption. For measuring the energy consumption, we used the Monsoon High Voltage Power Monitor[1] (HVPM) measuring device.

The remainder of this paper is structured as follows. Section 2 gives an overview of relevant related research. The research method is introduced in Section 3, followed by the results in Section 4. A discussion on the results is given in Section 5. Finally, Section 6 concludes the paper.

2 RELATED WORK

A part of existing studies focuses on a comparison of different types of performance indicators between Wasm in the browser and a native implementation on workstations or desktops. An early study from 2017 (Haas et al., 2017) examines the speed of Wasm in browsers with a benchmark compared to native C code. Results show that Wasm execution times of 7 out of 24 algorithms are up to 10% above native code, and almost all in the range of a factor of 2. Sandhu et al. (Sandhu et al., 2018) address the performance of Wasm with a focus on the calculation of sparse matrix-vector multiplications and show close to native slowdown factors with Firefox and factors of 2.2 for single- and 1.8 for double-precision floating points with Chrome. The authors Jangda et al. (Jangda et al., 2019) use a benchmark software for examining Wasm and show on average a slower execution of 45% with Chrome and 55% with Firefox than native code.

Other studies use the Node.js runtime environment instead of browsers to investigate the performance of Wasm. Oliveira et al. (Oliveira and Mattos, 2020) compare Wasm with JS and native C code with a benchmark software and show that Wasm is on average 21% slower in runtime compared to native C and around 40% faster than JS. Macedo et al. (De Macedo et al., 2021) compare different algorithms and benchmarks to show that Wasm is faster than JS in most cases, but only with small factors up to 1.17.

Some studies compare Wasm against JS in the browser. Sandhu et al. (Sandhu et al., 2018) also compare JS to Wasm in browsers and show slowdown factors of 1.9 and 2.1 with Chrome and 2.6 and 3.0 with Firefox. Herrera et al. (Herrera et al., 2018) achieve an advantage of between 1.5 and 2.5 with Wasm over JS on different device types and browsers with a benchmarking tool. In a comprehensive study in 2021, Yan et al. (Yan et al., 2021) compare code with a benchmark software compiled with different Wasm compilers to a JS implementation on desktop and mobile devices. They find fundamental performance differences in the execution on mobile and desktop devices, as well as the different browsers.

However, only few studies investigate the energy consumption of Wasm. Oliveira et al. (Oliveira and Mattos, 2020) investigate the energy consumption of Wasm in their Node.js IoT setup with a self-built measuring device and found an advantage of about 40% over JS. Studies by De Macedo et al. (De Macedo et al., 2021; De Macedo et al., 2022) use Wasm with benchmarks in different browsers and the software-based approach to measure the energy consumption. The results in their newer study show an average improvement in energy consumption of 30% with Wasm compared to JS.

A recent study by van Hasselt et al. (van Hasselt et al., 2022) compares the energy efficiency of Wasm against JS by using the Ostrich benchmark software on one Android device. The software-based energy profiler trepn[2] of Qualcomm is used, which is now available as Snapdragon Profiler[3]. As a result, on the choice of implementation, a large statistical effect size is determined for the energy-efficiency of Wasm compared to JS for two browsers. For the choice of the browser, they show a negligible difference for JS between Chrome and Firefox. For Wasm, their results show a medium effect size.

3 RESEARCH METHOD

The study was planned with a focus on reproducibility and extensibility for different device types and programming languages. Thus, implementations of algorithms were chosen instead of benchmarking libraries, as done in other studies (Haas et al., 2017; van Hasselt et al., 2022; Herrera et al., 2018; Jangda et al., 2019; Oliveira and Mattos, 2020; Yan et al., 2021), which are only available for specific programming languages. Furthermore, our setup also differs by using a hardware-based measuring of energy consumption and by using two devices from the low and high-end category. The experimental procedure, the measurements and the analysis is publicly made available as a replication package for a detailed

[1]https://www.msoon.com/high-voltage-power-monitor

[2]https://web.archive.org/web/20180514003122/https://developer.qualcomm.com/software/trepn-power-profiler

[3]https://developer.qualcomm.com/software/snapdragon-profiler

investigation[4].

3.1 Workloads

We carefully selected 19 algorithms from the Rosetta Code platform[5] as workloads for the experimental evaluation. To ensure extensibility, we set several selection criteria:

- Each algorithm must have a C and JS implementation.
- Each algorithm must provide a full implementation that does not require external dependencies, such as external libraries.
- The runtime of the algorithm should be controllable by varying its input parameters. This allows for adapting the algorithm to different experimental scenarios.

Based on these requirements, the nine sorting algorithms bubblesort, countingsort, gnomesort, heapsort, insertionsort, mergesort, pancakesort, quicksort and shellsort were selected. In addition, the following ten intensive processing algorithms were selected: Ackermann, Fibonacci, happy numbers, humble numbers, k-means, matrix multiplication, n-queens, perfect numbers, sequence of non squares and towers of Hanoi.

For executing the algorithms on the mobile devices, a minimal website was created for each implementation of an algorithm. The algorithms were embedded into the corresponding website, directly as JS code or compiled as Wasm bytecode.

The Wasm bytecode was created by compiling the C code with the emscripten compiler toolchain[6]. Wasm was compiled with the highest optimization level provided by the compiler. To trigger the execution of an algorithm, we added a button to the website. This button is also clickable by the automatized execution pipeline provided as part of the experimental setup.

3.2 Test Devices & Browsers

To provide enough variability, two different Android devices were selected for the execution of this experiment. The Samsung Galaxy S21 device was the latest Samsung model at the time of the experiment and represents the high-end class. The more than 7 years older Nexus 5 device represents the low-end class.

As the experiment is executed on real smartphones, there are several confounding factors, such as background services, other active apps or notifications. Therefore, the devices were prepared accordingly before starting the experiment. The display brightness is dimmed to approximately 50% to avoid influences on the energy consumption by automatic changes due to the ambient light sensor. All services and apps, which are not required for the experiment, such as mobile data networks, Bluetooth, GPS tracking or notifications, are deactivated or stopped.

We selected Google Chrome and Mozilla Firefox as the two execution targets for our experiment. Chrome has the largest worldwide mobile market share (StatCounter, 2022) and can be considered the default web-browser on Android phones. In terms of popularity, Firefox is far behind Chrome, but the only non-Chromium alternative on Android, which provides fully-fledged Wasm support.

3.3 Energy Measurement Procedure

For the measurement of the energy consumption during the experiment, the Monsoon HVPM measurement device is used. Compared to software-based measurements found in previous studies (De Macedo et al., 2021; De Macedo et al., 2022; van Hasselt et al., 2022), the hardware-based approach provides a more accurate representation of the real energy consumption of the device under test. The software-based measurements rely on mathematical models to estimate energy consumption, whereas hardware-based approaches directly measure the actual energy usage. Furthermore, a replication of the experiment with other device types, such as iOS smartphones, would be feasible with this approach. A disadvantage of this method is that the battery needs to be disassembled from the mobile device, which makes the device useless for other purposes.

A controlling device was used in this experiment to execute the experimental procedure. For this purpose, during the experiment the Android smartphones were connected with the controlling device via the Android Debug Bridge (ADB) using Wi-Fi. The power monitor was connected to the controlling device via USB to start and stop the measurement of the energy consumption and to transfer the measured values after the execution of an experiment run.

The full execution procedure of this experiment is automated by Python scripts to handle the large number of test cases and runs. This also ensures that all test cases are performed under the same conditions and can be replicated.

Each test run starts with opening one test website

[4]https://github.com/dpockstaller/webist23-wasm-js-energy

[5]https://rosettacode.org

[6]https://emscripten.org

in a new and clean browser window after resetting the browser application data. After a fixed waiting period for the browser to load the experimental setting, the energy consumption measurement procedure is started by tapping the button on the website. The tapping was generated with the ADB connection and the pixel coordinates of the button in the web-browser. After an execution time of 5 seconds, the measurement procedure is stopped. It was ensured that the execution of the algorithms took at least 5 seconds. At least 30 randomized samples for each test case were obtained. The measured energy consumption is per sample given in Joules.

3.4 Data Analysis

The experimental procedure yielded a total of 4,826 samples. Since a visual inspection of the sample distributions and also a Shapiro Wilk test did not indicate normally distributed data, a non-parametric statistical procedure was selected for analysis. To address the research questions, hypothesis tests (Mann Whitney U) were conducted to identify significant differences.

Regarding RQ1, we aimed to test a statistically significant difference in energy consumption between the execution of JS and Wasm within the two selected web-browsers. Thus, we formulated the following two-tailed hypothesis:

$$H1_{0,b} : \mu_{JS,b} = \mu_{Wasm,b} \qquad \forall b \in \{Chrome, Firefox\}$$
$$H1_{a,b} : \mu_{JS,b} \neq \mu_{Wasm,b} \qquad \forall b \in \{Chrome, Firefox\}$$

The $H1_{0,b}$ hypothesis states that there is no significant difference between the energy consumption of JS and Wasm within the two tested web-browsers. This is in contrast to the alternative hypothesis $H1_{a,b}$, that there is a significant difference.

For RQ2, the focus of our experiment was to determine any significant differences in energy consumption across the two selected web-browsers. Therefore, the following two-tailed hypothesis was formulated:

$$H2_{0,t} : \mu_{Chrome,t} = \mu_{Firefox,t} \qquad \forall t \in \{JS, Wasm\}$$
$$H2_{a,t} : \mu_{Chrome,t} \neq \mu_{Firefox,t} \qquad \forall t \in \{JS, Wasm\}$$

With $H2_{0,t}$ hypothesis, it was stated that there is no significant difference in energy consumption between Chrome and Firefox, when executing either Wasm or JS workloads. In contrast, the alternative hypothesis $H2_{a,t}$ states that there is a significant difference.

To add practical relevance to the findings, the analysis incorporated the calculation of appropriate effect sizes. In accordance with the non-statistical hypothesis test, we selected Cliff's delta effect sizes.

4 RESULTS

Figure 1 provides a graphical overview of the energy distribution of the collected samples per device in the form of violin plots. The violin plot shows the distribution of the energy consumption along the y-axis. The left side shows the distributions for the lower-end device and the right side the distributions for the higher-end device. The used web-browser can be distinguished by the blue and orange colour. Within each violin plot, the left column resembles the JS workload, the right column the Wasm workload.

From a pure visual interpretation, we can conclude that the higher-end device uses less energy for the same workloads. Also, the distribution of Wasm is skewed more towards lower energy-consumption values compared to that of JS. For the difference between the browsers, a more in depth statistical analysis is provided below.

Table 1 provides an overview of the descriptive results. We listed the values for the two web-browsers, with reference to the device class, within the columns. Typical descriptive statistics, such as mean, median and standard deviation are listed row-wise. When considering the difference in means of Chrome, we can observe a consistent improvement of approximately 20% across the devices, when using Wasm instead of JS. Firefox has a similar reduction of approximately 21% in mean on the lower-end device, but up to approximately 31% on the higher-end device.

For a proper statistical analysis, we conducted relevant hypothesis tests and a following calculation of corresponding effect sizes. The full list of results is given in Table 2. Within the major rows, we separated the results per device class. Per our defined α level of 0.05 all $H0$ hypotheses, as defined in Section 3.4, were rejected.

Looking in more detail into the results regarding RQ1, with reference to Table 2, it is observable that there is a significant difference in energy-consumption between Wasm and JS within the two selected web-browsers. Using Wasm over JS has a medium effect size also across the device classes. These results allow for the conclusion that Wasm uses less energy than JS for executing the tested algorithmic workloads within mobile web-browsers.

The results with respect to RQ2 and in reference to Table 2 show a significant difference in all the considered data pairs. Firefox in this case uses more energy than Chrome when executing JS workloads or Wasm workloads. This result is also consistent across the tested device classes. When considering the effect size and by that give an indication for the practical relevance of the results, only a small or even negligible

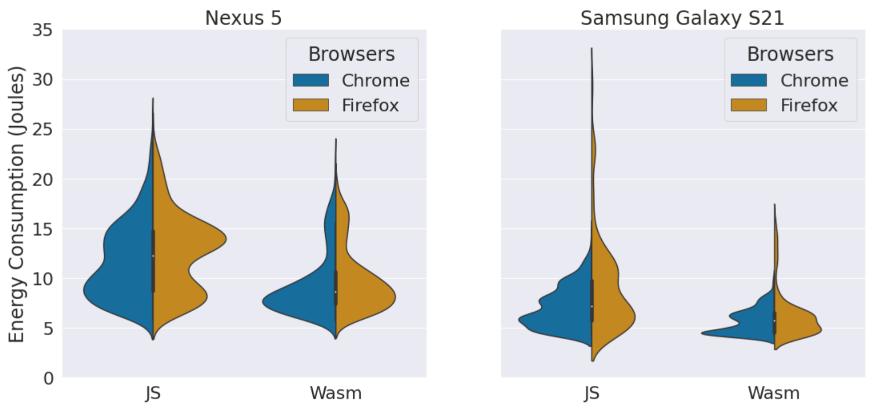

Figure 1: Energy consumption of Wasm and JS workloads executed on different mobile devices and browsers.

Table 1: Descriptive statistics.

Device	Low-End				High-End			
Browser	Chrome		Firefox		Chrome		Firefox	
	JS	Wasm	JS	Wasm	JS	Wasm	JS	Wasm
Mean	11.64	9.34	12.61	9.91	7.05	5.67	9.21	6.32
Std.Dev	3.75	3.16	3.91	3.19	2.00	1.25	4.82	2.43
Min	5.92	5.67	6.06	5.79	4.30	4.16	4.39	4.22
Q1	8.53	7.23	9.03	7.63	5.46	4.49	5.99	4.63
Median	10.93	8.24	13.13	9.04	6.62	5.47	7.67	5.96
Q3	14.44	10.02	15.01	10.99	8.29	6.46	11.03	6.62
Max	26.04	22.29	24.44	19.78	13.96	10.14	30.46	16.11

All values are given in Joules.

Table 2: Hypotheses tests and effect sizes on a high-end device.

Device	Dataset Pair	p-Value	Cliffs Delta	Interpretation
Low-End	Chrome JS vs. Chrome Wasm (RQ1)	3.872448e-33	0.395187	medium
	Firefox JS vs. Firefox Wasm (RQ1)	3.617754e-33	0.396013	medium
	Chrome JS vs. Firefox JS (RQ2)	9.598055e-06	-0.146084	negligible
	Chrome Wasm vs. Firefox Wasm (RQ2)	0.000023	-0.139392	negligible
High-End	Chrome JS vs. Chrome Wasm (RQ1)	1.591990e-36	0.423401	medium
	Firefox JS vs. Firefox Wasm (RQ1)	1.160505e-42	0.459227	medium
	Chrome JS vs. Firefox JS (RQ2)	9.184663e-15	-0.259970	small
	Chrome Wasm vs. Firefox Wasm (RQ2)	0.000402	-0.118702	negligible

interpretation can be attested.

5 DISCUSSION

Previous studies on Wasm focused mainly on runtime performance or hardware usage compared to a native implementation or JS and mostly on desktop devices. This paper focuses on the energy consumption of Wasm compared to JS in the browser on mobile devices.

To the best of our knowledge, only one recently published study investigated the energy consumption of Wasm and JS on Android smartphones. In contrast to our results, van Hasselt et al. (van Hasselt et al., 2022) found a large effect size between Wasm and JS for both browsers Chrome and Firefox, while this study identifies a medium interpreted effect size for both browsers. Also, a medium effect size for the difference in the energy consumption of Wasm between Chrome and Firefox was reported in their study. This cannot be confirmed by this paper, as a negligible effect size was achieved. Furthermore, the results of the browsers in their study are the opposite to this study, as in their results Firefox consumes less energy than Chrome. The results of their descriptive values show that Wasm consumes around 61% less energy with Chrome and 73% less energy with Firefox than JS in the mean value. These values are significantly larger than those found in this paper. A possible explanation for the diverging results could be that our study is based on a hardware-based measurement approach and that we selected different workloads. Thus, we consider the results of our study an important complement to the existing body of research in this field. For future work, it would be interesting to combine the experimental subjects and measurement methods in a single large experiment.

In other studies (Sandhu et al., 2018; Herrera et al., 2018), a larger difference between JS and Wasm implementations was found for Firefox than for Chrome. This could be confirmed in this study by RQ2 and is probably related to the lower base performance of JS in Firefox. Apart from the influence of the chosen browser, a study by Oliveira et al. (Oliveira and Mattos, 2020) using the Node.js environment, which comes closest to a comparison with Chrome due to the same V8 JS engine, showed an average improvement of about 40% in the runtime and the energy consumption using Wasm. In comparison, this study was able to determine a lower energy consumption of around 20% through the use of Wasm compared to JS using Chrome.

The findings of our study indicate that using Wasm in web application development brings a notable advantage in energy-efficiency across all tested browsers and devices. This outcome is particularly valuable in highly competitive markets, where such advantages could make a crucial difference in user satisfaction. These results would also indicate that it is beneficiary to transfer existing code bases in C to the web-browser via Web Assembly instead of a re-implementation in JS.

As a practical recommendation for mobile users, the results of this study suggest considering using Chrome over Firefox to conserve battery life. The results show significant results for our experiments, however, only with a small or negligible effect size. This small difference would result in an overall smaller CO2 footprint for users.

6 CONCLUSIONS

The goal of this study was to determine whether Wasm can be used to reduce the energy consumption of executing intensive workloads within mobile web applications. In an experiment with different Android devices and different browsers, a variety of algorithms to generate load were compared as implementations in Wasm and JS. In addition, the influence of the chosen browser on the energy consumption on mobile Android devices was investigated.

Overall, the results show that using Wasm instead of JS for intensive processing tasks can reduce energy consumption by a range of approximately 20% to 30% in the browser on Android smartphones. Furthermore, it was found that Chrome consumes less energy when executing both JS and Wasm compared to Firefox.

It is important to note that our study is limited to Android devices and Wasm bytecode generated by compiling C source code with emscripten. Further research could explore the differences in energy consumption across different device types, such as iOS smartphones, and other programming languages and their compilers, such as Rust, Go, or AssemblyScript. The study design is flexible enough and can be adapted to enable further research in these areas.

REFERENCES

De Macedo, J., Abreu, R., Pereira, R., and Saraiva, J. (2021). On the runtime and energy performance of webassembly: Is webassembly superior to javascript yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 255–262.

De Macedo, J., Abreu, R., Pereira, R., and Saraiva, J. (2022). Webassembly versus javascript: Energy and runtime performance. In *2022 International Conference on ICT for Sustainability (ICT4S)*, pages 24–34.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200.

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

Jangda, A., Powers, B., Guha, A., and Berger, E. (2019). Not so fast: Analyzing the performance of webassembly vs. native code. *login Usenix Mag.*, 44.

Mozilla Developer Network (2022). Webassembly concepts — mdn.

Oliveira, F. and Mattos, J. (2020). Analysis of WebAssembly as a strategy to improve JavaScript performance on IoT environments. In *Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC Estendido 2020)*. Sociedade Brasileira de Computação - SBC.

Sandhu, P., Herrera, D., and Hendren, L. (2018). Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in javascript and webassembly. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA. Association for Computing Machinery.

StatCounter (2022). Mobile browser market share worldwide.

van Hasselt, M., Huijzendveld, K., Noort, N., de Ruijter, S., Islam, T., and Malavolta, I. (2022). Comparing the energy efficiency of webassembly and javascript in web applications on android mobile devices. In *The International Conference on Evaluation and Assessment in Software Engineering 2022*, EASE 2022, page 140–149, New York, NY, USA. Association for Computing Machinery.

Yan, Y., Tu, T., Zhao, L., Zhou, Y., and Wang, W. (2021). Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 533–549, New York, NY, USA. Association for Computing Machinery.