Comparing the Energy Efficiency of WebAssembly and JavaScript in Web Applications on Android Mobile Devices

Max van Hasselt, Kevin Huijzendveld, Nienke Noort, Sasja de Ruijter
Tanjina Islam, Ivano Malavolta
Vrije Universiteit Amsterdam, The Netherlands
{ m.p.p.van.hasselt | k.j.huijzendveld | n.noort | s.n.de.ruijter }@student.vu.nl, { t.islam | i.malavolta }@vu.nl

ABSTRACT

Context. WebAssembly was created as an alternative to JavaScript for developing heavy loading web applications. Since JavaScript is known to have long execution times. A lot of research is already performed to compare the run-time performance of WebAssembly against that of JavaScript. However, little research is available that compares the energy consumption of WebAssembly versus JavaScript.

Goal. With this study we aim to identify the correlation between the energy consumption and the use of WebAssembly versus JavaScript. This will aid developers in deciding which method matches the needs of their project best in terms of energy efficiency.

Method. The subjects of the experiment are WebAssembly and JavaScript. During the experiment two research questions are defined. For the first research question the programming language is the independent variable. For the second research question the web browser is the independent variable. For both research questions is the energy consumption of the Android device in Joules the dependent variable.

Results. We can confirm that the energy consumption of WebAssembly is less than that of JavaScript. The browser also plays a role since the energy consumption of Firefox is significantly smaller than that of Chrome for both WebAssembly and JavaScript.

Conclusions. This study provides evidence that using WebAssembly for the development of web applications can reduce the energy consumption and thus improve the battery life of a user's Android device. Developers can use this information when choosing a programming language to develop a web application. Moreover, using Firefox over Chrome does also reduce the energy consumption of web applications developed both with WebAssembly and JavaScript.

ACM Reference Format:
Max van Hasselt, Kevin Huijzendveld, Nienke Noort, Sasja de Ruijter and Tanjina Islam, Ivano Malavolta. 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), June 13–15, 2022, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3530019.3530034

EASE 2022, June 13–15, 2022, Gothenburg, Sweden

ACM ISBN 978-1-4503-9613-4/22/06…$15.00
https://doi.org/10.1145/3530019.3530034

1 INTRODUCTION

JavaScript is one of the most used programming language supported by the web. However, interpreted JavaScript is known to run applications much slower than compiled Android machine code, from hereon referred to as native code [1] [2]. Problems arise when adopting high performance computing, like machine learning and 3D visualizations, on the web [3]. JavaScript is not equipped to perform such high performance computing functions, which leads to long execution times. In an effort to tackle these problems, a number of major web browser providers like Apple, Microsoft and Mozilla, decided to collaborate and join forces. This led to the creation of a new type of low-level code, namely WebAssembly[1]. WebAssembly is a standard that defines low-level binary code. The standard is designed to be as small as possible and to compile efficiently. It is language independent, meaning that it supports several programming language by using a compiler like Emscripten [4].

Various experiments have been conducted to analyse the run-time performance of WebAssembly by either comparing WebAssembly to the application's native code, or in comparison with JavaScript [2, 5–8]. Most of these papers use the execution time of benchmark functions to measure the run-time performance. However, little is known about the impact on energy consumption of WebAssembly. Oliveira et al. conducted an experiment on the run-time performance of WebAssembly where the battery consumption was measured in addition to the execution time [6]. Based on the obtained results, the authors concluded that using WebAssembly so as to improve the run-time performance of JavaScript reduces the battery consumption by 39%.

According to the book of Sasu Tarkoma et al., a crucial aspect of consumer satisfaction of smartphones is the battery life [9]. Software is often equipped with suboptimal functions which leads to inefficient use of the hardware. Since there is little knowledge about the impact of the use of WebAssembly on energy consumption, this research aims to examine the energy consumption using WebAssembly in comparison with JavaScript.

The goal of this paper is to identify the correlation between the energy consumption and the use of WebAssembly versus JavaScript. This will aid developers in deciding which of these two languages matches the needs of their project best in terms of energy efficiency.

2 RELATED WORK

Studies on the run-time performance of WebAssembly vs native/Java- Script code. The motivation behind the study of Herrera et al. [5] was the increase in power of hardware devices. They carried out the experiments on the Ostrich benchmark set,

[1] https://webassembly.github.io/spec/core/

which is a collection of numerical programs representing the numerical dwarf categories. The goal was to evaluate the current state of portable and web-based numerical computing. In their experiment, they compared the run-time performance of WebAssembly to the run-time performance of JavaScript. The results of their study reveal that the run-time performance of using WebAssembly is better than the run-time performance of using only the numeric benchmarks. In this paper, the Ostrich benchmark set is used as well, however, the focus lies solely on Android devices, whereas the focus of the experiment by Herrera et al. is on various types of devices. Thus, mobile phones are being compared to desktops, and browsers are being compared as well. Another difference is that the paper of Herrera et al. focuses on the run-time performance differences between WebAssembly and JavaScript, whereas this study focuses on the energy efficiency of WebAssembly and JavaScript.

The goal of Jangda et al. [2] is to analyze the run-time performance of WebAssembly compared to the native code, for which a SPEC CPU benchmark suite is used. This benchmark suite includes several applications that fall under the intended use cases of WebAssembly. Thus, where the study of Herrera et al. [5] and this study focuses on JavaScript, the study of Jangda et al. focuses on native code in general. Besides this, the results of the study of Jangda et al. are solely focused on the run-time performance of WebAssembly and native code in web browsers, specifically Chrome and Firefox. From this study follows that applications compiled to WebAssembly run slower than applications compiled to native code in both Firefox and Chrome.

Studies on the runtime efficiency of WebAssembly. Similar to the experiment of Herrera et al. [5], two other related works use the Ostrich benchmark set. Firstly, the experiment of Oliveira et al. [6], assesses the use of WebAssembly as a strategy to improve the run-time performance of JavaScript applications in an IoT environment. From this experiment follows that JavaScript run-time performance could be improved in terms of execution time, memory usage, and battery consumption is reduced when using WebAssembly. Their main objective is to assess if there is a performance gain and how much it impacts battery-powered devices. Specifically, they focus on the IoT environment. Whereas, in this paper, we analyze the impact on the energy consumption of web applications on Android mobile devices. Additionally, compared to their study, we analyze how the choice of web browsers impacts the energy efficiency of JavaScript and WebAssembly.

Secondly, the experiment of Alamari and Chow [10] follows from the new run-time performance capabilities in the browser. These being the possibility to build universal applications that run on every machine that has a web browser installed on it. They propose a design to build web applications that take advantage of these new run-time performance capabilities. Both of these experiments differ from this paper, along the lines of its goal. The goal of these experiments is to improve the run-time performance of web applications, whereas the goal of the experiment in this paper is to analyze the energy efficiency of WebAssembly and JavaScript.

An experiment done by Reiser and Bläser [7] does not make use of the Ostrich benchmark set, but this experiment is focused on runtimes. Its goal is faster and more predictable runtimes for performance-critical web code. In order to achieve this goal, Reiser and Bläser create a cross-compiler that translates JavaScript/TypeScript to WebAssembly. From this follows a compute-intense web code that is faster and has a reduced runtime.

Another experiment that focuses on runtimes is executed by Lehmann and Pradel [11]. They present Wasabi, a framework which is created to dynamically analyze WebAssembly. Wasabi is based on binary instrumentation. This binary instrumentation inserts calls to analysis functions that are written in JavaScript into a WebAssembly binary. This framework demands a runtime overhead that is reasonable for dynamic analysis, and it makes implementation of various dynamic analyses straightforward.

The run-time performance of JavaScript. So far, most related work that is discussed focuses on the combination of WebAssembly and JavaScript. However, there are also studies that focus only on the run-time performance of JavaScript.

Sandhu et al. [8] compare JavaScript to native languages by focusing on run-time performance and choice of optimal sparse matrix storage format for sequential parse matrix-vector multiplication (SpMV). SpMv is a kernel that is considered critical regarding the run-time performance of data-intensive applications. From this experiment follows that the best performing browser demonstrated a slowdown of only 1.2x to 3.9x, that double-precision SpMV is more efficient than single-precision, and that the optimal storage format choices are very different for C as compared to JavaScript, and even quite different between the two browsers. The main difference between this experiment and the experiment of this paper, is the use of SpMv to compare the run-time performance of JavaScript with the run-time performance of native languages.

An experiment done by Miettinen and Nurminen [12] analyzes the energy consumption of JavaScript based web applications on mobile phones. From this experiment follows that choosing a proper library can save up energy when implementing the same functionality. Thus, there are relevant differences when implementing a JavaScript based application when focusing on energy efficiency. Where the experiment of Miettinen and Nurminen focuses solely on the energy efficiency of JavaScript, the experiment of this paper analyzes the differences in the energy efficiency of both JavaScript and WebAssembly.

Energy consumption of Google Chrome vs Mozilla Firefox. Macedo et al. [13] present a study on the energy consumption of two popular browsers: Google Chrome and Mozilla Firefox. Their goal was to measure the energy consumption of the browsers to understand which browser is the most appropriate to be used if energy consumption is of concern to the user. In particular, they measure and analyze the energy consumed by the DRAM and CPU of Chrome and Firefox on different web applications. In general, they conclude that Chrome is a more energy-efficient browser than Firefox, but Firefox is more consistent in terms of energy efficiency.

3 STUDY DESIGN

The experiment is carried out according to known empirical software engineering guidelines [14–16] . Below we present the design of the experiment. For further details, a complete replication package is available[2]. The replication package also allows independent researchers to verify and replicate the study.

[2]https://github.com/S2-group/EASE-2022-energy-web-assembly-rep-pkg

3.1 Goal and Research Questions

The goal of this study is to *analyze the impact of using WebAssembly over JavaScript on the energy efficiency of Android browser processes.* Our goal is refined into the following two research questions:

[RQ1]: *How does the use of WebAssembly over JavaScript impact the energy efficiency of web applications on Android mobile devices?*

By answering this research question, software developers will be given an empirically substantiated perspective on the impact of compiling to WebAssembly, as opposed to JavaScript, on the energy efficiency of their web applications.

[RQ2]: *To what extent does the choice of browser influence the energy efficiency of JavaScript and WebAssembly applications?* Different browsers use different JavaScript engines [17][18] to compile both JavaScript and WebAssembly. These different engines could produce different results. As such, this study will factor in the type of browser used during the experiment. The answer to this question may help end-users who want to reduce their energy consumption in choosing a mobile web browser.

3.2 Subjects Selection

In our experiment, we consider benchmarking algorithms as the subject. The benchmark chosen for this experiment is the Ostrich benchmark [19]. This benchmark is chosen mainly because it is specifically designed to study the run-time performance of programming languages for numerical code. Each of the benchmark functions contain an equivalent implementation of both JavaScript and C. The Ostrich benchmark is also used in previous studies [5, 6, 10]. Using the compiler Emscripten, the Ostrich benchmark can be compiled from C to the equivalent implementation of WebAssembly. This will allow us to compare JavaScript and WebAssembly in terms of their energy consumption.

The Ostrich benchmark consists of 12 numerical benchmark functions. Since four of the functions in this benchmark were crashing when run multiple times in succession, this study uses only 8 of these algorithms. Table 1 contains the 8 used benchmark functions together with a description as stated by the developers of the Ostrich benchmark[19]. It also shows the number of times each function will be executed during a single run. For each function, the algorithm has a run time of approximately one minute. Thus, the amount of calls depends on the speed of the function. The benchmark functions used in this experiment are written both in JavaScript and C. The C implementation is compiled to WebAssembly.

3.3 Experimental Variables

This study has two research questions that both introduce one independent variable. For RQ1, the **independent variable** is the programming language. This independent variable has two treatments, namely the raw JavaScript code and the C code compiled to WebAssembly. Compiled means WebAssembly binary generated after the compilation step. For RQ2, the **independent variable** is the web browser in which the trials will run. This independent variable has the two treatments Chrome and Firefox.

The **dependent variable** for both research questions is the energy consumption (E) of the Android device in Joules. Energy consumption is computed as follows:

$$E = \frac{P}{10^6} W \times \frac{T}{1000} s, \tag{1}$$

where P is the average power consumed by the Android device in microWatts, and T is the time in milliseconds that each run takes.

3.4 Experimental Hypotheses

In order to answer the two research questions of our study, we have formulated the following hypotheses. Given that $\mu_{JavaScript}$ is the average energy consumption of web applications when using JavaScript and $\mu_{WebAssembly}$ is the average energy consumption of web applications when using WebAssembly, then the null and alternate hypotheses for the programming language (pl) in RQ1 are defined as:

$$H_0^{pl} : \mu_{JavaScript} = \mu_{WebAssembly}$$

$$H_a^{pl} : \mu_{JavaScript} \neq \mu_{WebAssembly}$$

The null hypothesis states that the average energy consumption of web applications when using JavaScript or WebAssembly is the same. Intuitively, the alternative hypothesis states that there is a significant difference in average energy consumption when using JavaScript versus WebAssembly.

Similarly, given that β_{chrome} is the average energy consumption of web applications launched in Google Chrome and $\beta_{firefox}$ is the average energy consumption of web applications launched in Mozilla Firefox, then the null and alternate hypotheses for the browser (b) in RQ2 are defined as:

$$H_0^{b} : \beta_{chrome} = \beta_{firefox}$$

$$H_a^{b} : \beta_{chrome} \neq \beta_{firefox}$$

Intuitively, the null hypothesis states that there is no significant difference between the average energy consumption of web applications when using either Google Chrome or Mozilla Firefox browser. Subsequently, the alternative hypothesis states that there is a significant difference in average energy consumption when using a different web browser.

The following two hypotheses are defined for the interaction of browser and programming language, where μ_i is the effect of treatment i (JavaScript/WebAssembly) of the programming language factor from RQ1 and β_j is the effect of treatment j (Google Chrome/Mozilla Firefox) of Web browser factor from RQ2. Then the null and alternate hypotheses are defined as:

$$H_0^{pl,b} : (\mu\beta)_{ij} = 0 \ \forall i, j$$

$$H_a^{pl,b} : \exists (i, j) \mid (\mu\beta)_{ij} \neq 0$$

The null hypothesis states that for each pair of treatments (*i.e.,* programming language and web browser), there is no significant difference between the average energy consumption of web applications. However, the alternative hypothesis states that the average energy consumption of at least one pair of treatments is significantly less than others.

Table 1: Ostrich benchmark functions

Number	Benchmark	Category	Description	Runs
1	fft	Spectral methods	Application of the Fast Fourier Transform function	10
2	hmm	Graphical models	A forward-backward algorithm which looks for the unknown parameters of a hidden Markov model	3
3	lavamd	N-body methods	Calculation of particle potential and relocation within a big 3D space	7
4	lud	Dense linear algebra	Application of a lower-uper decomposition on a 1024 x 1024 matrix	6
5	nqueens	Branch and bound	An algorithm that computes the number of different possibilities where there are n queens put on a n x n chess board and the queens are not attacking eachother	3
6	nw	Dynamic programming	Computation of the optimal alignment of two protein sequences	10
7	page-rank	Map reduce	An algorithm that ranks websites according to their importance, also used by Google Search	4
8	spmv	Sparse linear algebra	Multiplication of a sparse matrix with a vector	15

3.5 Experiment Design

Based on the subject selection, the variables, and the hypotheses constructed for this experiment, the design type will be two factors and two treatments (2F-2T). These two factors are the programming language from RQ1 and the web browser from RQ2. Each factor has two treatments. Firstly, the two treatments of the programming language factor are JavaScript and WebAssembly. Secondly, the two treatments of the web browser factor are Chrome and Firefox. Since there are two factors in this design type, it has to be taken into account that these two factors might interact with each other. Thus, not only the effect of the treatments is being modeled, the effect of the interaction between these is being modeled as well. In the experimental hypotheses section, this effect is included by the last hypothesis.

The factorial design considers all possible combinations of treatments, in which we use the same number of benchmark functions for all cases.

For answering RQ1, the 8 benchmark functions will be executed once in JavaScript, and once in C++, which will be compiled to WebAssembly. We used the Ostrich benchmark in our experiment, where the benchmark functions contain an equivalent implementation of both JavaScript and C. That is why we considered C/C++ over other programming languages. It is important to note that, to answer RQ1, the benchmark functions will be executed on Google Chrome for Android, since it is currently the most popular browser on the Android operating system [20].

For answering RQ2, all subjects are executed for each possible combination of browser and programming language. About the browsers, Google Chrome and Firefox browsers are chosen since they follow different approaches to compile towards WebAssembly. Specifically, Chrome's V8 compiles to machine code [17], and Firefox's SpiderMonkey compiles to intermediate bytecode [21]. Finally, each trial of our experiment is repeated 30 times, so for each pair of programming language and web browser. During each run, the total energy consumption (Joules) of the Android device is measured. These results are then used to calculate the energy efficiency of each algorithm in each web browser. These repetitions per trial are done to take into account the intrinsic variability of the energy measurement process [16].

3.6 Data Analysis

To answer RQ1 and RQ2, the collected measures are analyzed in four phases: data exploration, check for normality and transformations, hypothesis testing, and effect size estimation. In the data exploration phase, we get a first indication of the obtained energy consumption values via a combination of descriptive statistics, histograms, and box plots. Next, we analyze the distribution of the data. Specifically, we check if the energy consumption measures are normally distributed via (i) visually inspecting their Q-Q plots and (ii) applying the Shapiro-Wilk normality test. Since this experiment has more than one factor, it is greatly desired that these values are normally distributed. This is because an ANOVA test is possible when there is a normal distribution, but this is not the case when the data is not normally distributed. Since there is no non-parametric counterpart for a two-factor design, an ANOVA test cannot be used when there is no normal distribution of data [14]. Here, ANOVA is an example of a parametric test. In non-parametric tests, only very general assumptions are made and the data does not have to be normally distributed. If data is normally distributed, we will apply a single statistical test to test the statistical hypothesis of both research questions. As mentioned before, this test will be two-way ANOVA. The threshold for α will be set to 0.05. If the data is not normally distributed, the measurement data will be transformed. As was mentioned by Stevens in his book "Applied Multivariate Statistics for the Social Sciences", the impact on the validity of non-normal data is small enough for an ANOVA analysis so that the statistical test still produces realistic results [22]. The significance level will again be $\alpha = 0.05$. If this transformation does not lead to normality, logistic regression is used. Finally, to statistically assess the magnitude of the differences between the energy consumption of the treatments of both factors, the Cliff's Delta measure will be used [23]. This is a non-parametric effect size measure that will be used complementary to the hypothesis testing. Where the ANOVA test will determine whether there is a statistical significance, the Cliff's Delta measure will determine whether there is a practical significance in the differences between the treatment pairs [24]. The Cliff's Delta statistics will be interpreted as proposed by Grissom et al. [25].

4 EXPERIMENT EXECUTION

In this section, the technical setup used for executing the experiment will be described. This includes the different devices used, the infrastructure that connects these devices, and the software tools used to run the experiment.

Figure 1 shows a schematic overview of the infrastructure of our experiment. The infrastructure of the experiment consists of two devices: (i) an Android device on which the subjects of the experiments will be executing, and (ii) a Raspberry Pi, which will be used to orchestrate the experiment and is responsible for collecting experimental data from the Android device. The Raspberry Pi is convenient for this experiment because the USB ports can be programmatically disabled, ensuring that the phone is not charging during the experiment. The trials are run on a Huawei Nexus 6P phone. The energy consumption of the Android device is measured using the Trepn plugin[3], a software-based power profiler for Android devices. Trepn is widely used in empirical research on energy-efficient software [26–28], and it has been tested to be sufficiently accurate like the hardware-based power monitors (e.g., the Monsoon Power Monitor[4]), with an error margin of 99% [29]. Due to version limitations of the Trepn framework, the phone will run Android 6. The system specs for the Nexus 6P can be seen in Table 2.

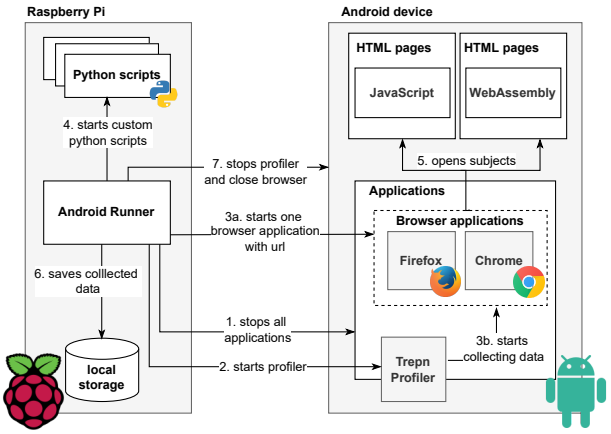

Figure 1: Overview of the experiment execution

The browsers that are used to execute the subjects are Firefox Nightly version 95.0a1 and Google Chrome 94.0. Before every run in Firefox, due to the duration of the runs, the setting "dom.max_script_run_time" is set to 0, to allow scripts to run indefinitely. Firefox sets this setting to 20 seconds by default. Such a configuration is not necessary for Chrome, which allows long-running scripts by default.

We take special care in keeping the execution environment as clean as possible. Specifically, the application data of the browser will be cleared, so that no caching will influence the power measurements and the run time of the run. Furthermore, to avoid charging the Android device during execution, before every run, the Raspberry Pi USB port to which the Android device is connected is disabled. After each run, it is re-enabled to allow for charging between runs. Android-runner provides functionality to achieve this.

For the orchestration of the experiment, a Raspberry Pi 4B will be used, which is a convenient, low-power micro-computer with decent specifications and which runs Linux. This device will control the phone and initiate the different trials. Furthermore, the Raspberry Pi will record the Trepn logs of the power consumption of the phone during the trials. The technical specification of the Raspberry Pi can be found in Table 3.

To control the experiment, the Raspberry Pi will use Android Runner [30], a Python framework used to execute experiments on Android devices. The experiment will be defined using a JSON file describing the steps of the experiment. Then, the entire execution of the experiment is managed by Android Runner with a combination of Python scripts and Android Debug Bridge (ADB) commands. The Raspberry Pi will instruct the Nexus 6P to execute the browsers and open the HTML files containing the subjects. To eliminate the influence that a fluctuating internet connection may have, the HTML files will be served by the Raspberry Pi over WiFi to the Nexus 6P.

Table 2: Huawei Nexus 6P specs

OS	Android 6 (Marshmallow)
Chipset	Qualcomm MSM8994 Snapdragon 810
CPU	Octa-core (4x1.55 GHz Cortex-A53 & 4x2.0 GHz Cortex-A57)
GPU	Adreno 430
Storage	32 GB eMMC 5.0
RAM	3GB
Screen	5.7 inch OLED QHD, (2560 x 1440)
WLAN	Wi-Fi 802.11 a/b/g/n/ac, dual-band, Wi-Fi Direct
USB	Type-C 2.0
Battery	LiPo 3450 mAh, Fast charging 15W

Table 3: Raspberry Pi 4B specs

OS	Raspian GNU/Linux 10 (Buster)
CPU	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8)
RAM	8GB
USB	USB 3

The timeline of the experiment will be as follows. Firstly, all running applications on the Nexus 6P will be stopped, represented by 1 in Figure 1. This way the influences of other processes on the energy consumption of the device will be minimized. Secondly, the profiler will be started, 2 in Figure 1. Third, the browser will be opened and then instructed to, one by one, open the HTML documents which directs the browser to load the subject with specific treatment (JavaScript or WebAssembly). Each treatment will be executed 30 times and each run will last for about 1 minute. After each execution, the execution time and power consumption during execution will be measured and stored. The order of the execution of the trials will be randomized to counteract the possible variability of the energy measurements. After each trial execution, the browser cache will be cleared. Then step three will be repeated for the other browser. Finally, the data of all runs will be aggregated, and the power measurements over time will be converted to Joules.

[3]https://github.com/S2-group/android-runner/tree/master/AndroidRunner/Plugins/trepn
[4]https://www.msoon.com/

5 RESULTS

5.1 Impact on energy efficiency (RQ1)

5.1.1 Data Exploration. An overview of the amount of energy that is consumed (in Joules) when running the experiment is given in Table 4, and Figure 2. From Table 4 follows a clear average energy consumption throughout the experiment of 56,847 Joules for both programming languages. When looking at JavaScript and WebAssembly separately, a large difference was found between the average energy consumption of both languages. Namely, the average energy consumption of JavaScript is 81,785 Joules. This average is larger than 31.91 Joules, which is the average energy consumption of WebAssembly.

Table 4: Descriptive statistics for RQ1

	Energy Consumption (Joules)		
	Both	JavaScript	WebAssembly
Minimum	3.251	3.251	4.039
1st quartile	24.996	35.912	22.226
Median	42.073	72.302	27.913
Mean	56.847	81.785	31.910
3rd quartile	72.280	130.752	46.031
Maximum	164.219	164.219	68.531

What is noticed as well when looking at Table 4, is the maximum energy consumption of JavaScript and WebAssembly. This maximum is almost 100 Joules bigger for JavaScript compared to WebAssembly.

To get a better understanding of the data, two boxplots are created. Figure 2 illustrates a boxplot of the energy consumption per language. From the diagram, we get an overview of the energy consumption for each benchmark function per language type. From the boxplot, we can infer that there is a large difference in energy consumption values for JavaScript and WebAssembly.

5.1.2 Check for normality. In order to check the normality of the data, a histogram for the distribution of the consumption of

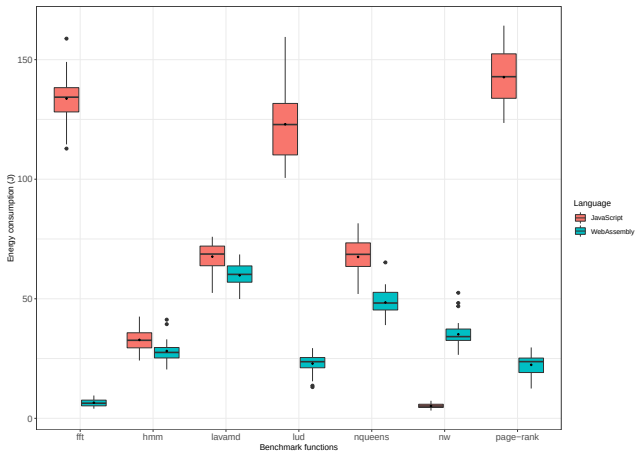

Figure 2: Energy consumption of JavaScript (red) compared to WebAssembly (green)

JavaScript and WebAssembly is given in Figure 3. Based on these histograms, we can infer that the data for both treatments are not normally distributed.

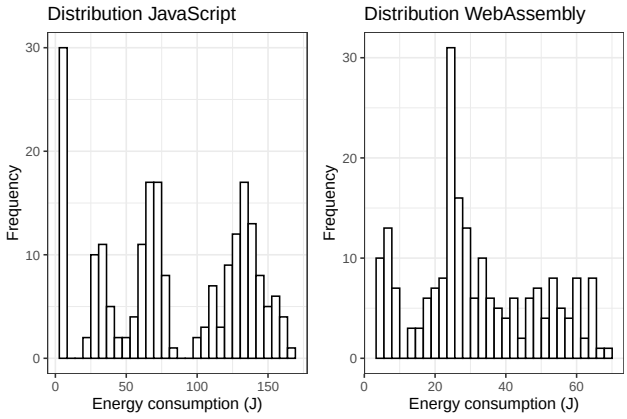

Figure 3: Frequency of energy consumption of RQ1

To confirm this presumption, the Shapiro-Wilks test is executed on the data. From this follows W = 0.91868 and p-value = 2.39e-09 for JavaScript, and W = 0.95126, p-value = 1.476e-06 for WebAssembly. Thus, since the p-value of both JavaScript and WebAssembly is below the significance threshold of 0.05, the data is not normally distributed. In order to be able to execute an ANOVA test, the data is transformed to normalized data. This is done by using orderNorm for both JavaScript and WebAssembly, which results in the Q-Q-plots shown in Figure 4.

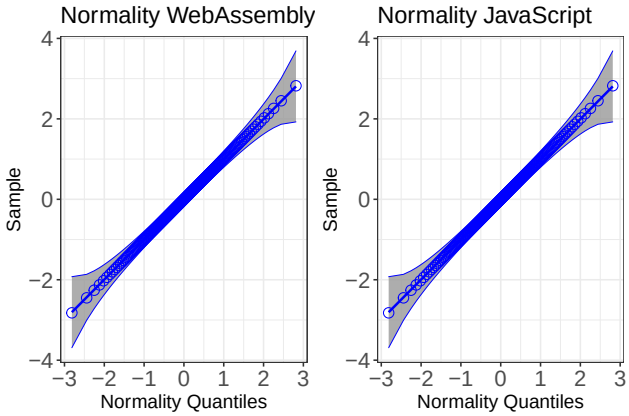

Figure 4: Q-Q-plot of JavaScript and WebAssembly

By looking at these Q-Q-plots, the data seems normally distributed. To confirm this, the Shapiro-Wilks test is executed again. This leads to a value of W = 0.99978 and p-value = 1 for both JavaScript and WebAssembly. Thus, this data is normally distributed.

5.1.3 Hypothesis Testing. Since the transformed data is normally distributed and we will only focus on the treatment language, a one-way ANOVA test is executed. Due to the influence of the difference in the output of the benchmarks on the data, a within-subjects factor is added as well. By adding the benchmarks as a within-subjects

factor, it is possible to compare these benchmarks under different conditions. Thus, a one-way repeated measure is executed. Here the different conditions are the languages used. As can be seen in Table 5, the p-value is below 0.05 which means that we can reject the null hypothesis H_0^{RQ1}. Therefore it can be said that the use of JavaScript or WebAssembly has a different impact on the energy consumption of web applications on Android mobile devices.

Table 5: One-way ANOVA for RQ1

	Df	Sum Sq	Mean Sq	F value	p-value
Language	1	261197	261197	188.06	< 2.2e-16
Residuals	418	580563	1389		

5.1.4 Effect Size Estimation. By calculating the effect size estimation using Cliff's Delta and a density plot, the strength of the differences between the treatment pairs will be estimated. Looking at the density plot in Figure 5, it is expected that effect size is big considering the height and spread of both curves. The value of Cliff's delta estimation is large (0.5907937). This means that there is a strong difference between the use of JavaScript versus WebAssembly in Chrome.

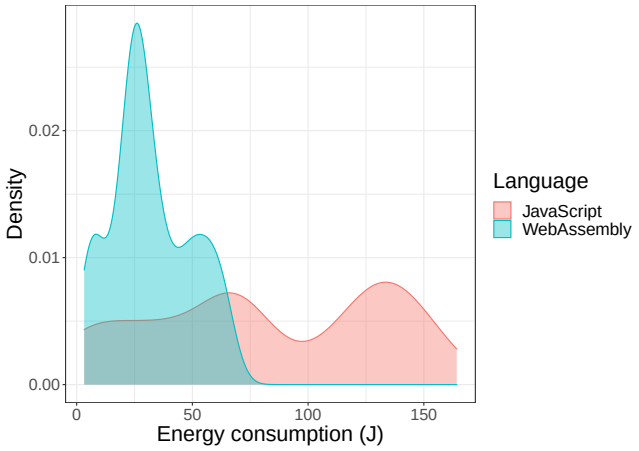

Figure 5: Energy consumption density curve of JavaScript compared to WebAssembly

5.2 Influence of the choice of browser (RQ2)

5.2.1 Data Exploration. In Table 6, and Figure 6 we give an overview of the energy consumption of each run of the experiment for the browsers Chrome and Firefox. Similar to Table 4, Table 6 shows that the average energy consumption of JavaScript is a lot larger than the average energy consumption of WebAssembly. It also shows that the difference in the average energy consumption between Chrome and Firefox is not very large. However, Table 6 shows that Chrome has a larger average energy consumption than Firefox.

Another data point that stands out, is the maximum value of Firefox combined with JavaScript. This maximum value is larger than the maximum value of Chrome combined with JavaScript, even though the mean of Firefox and JavaScript is a lot smaller than the mean of Chrome and JavaScript. Likewise, the minimum energy consumption of Firefox and JavaScript is larger than the minimum energy consumption of Chrome and JavaScript. This probably means that the data points that are measured for Firefox and JavaScript have more outliers than the data points measured for Chrome and JavaScript.

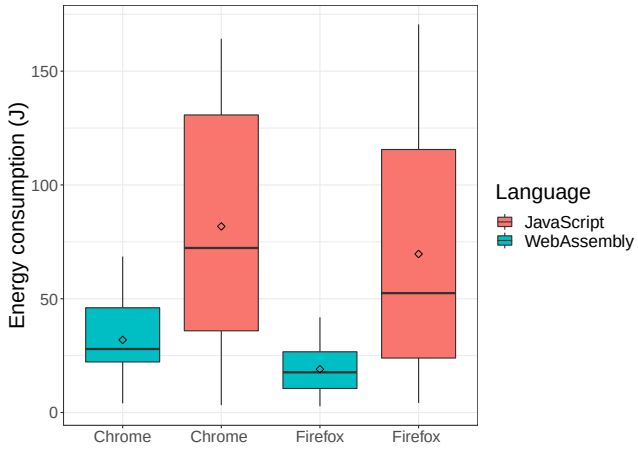

Figure 6: Boxplot of the energy consumption of JavaScript, WebAssembly, Chrome and Firefox

The boxplot in Figure 6 shows the energy consumption of JavaScript and WebAssembly combined with Chrome and Firefox. From the diagram, we get an overview of the energy consumption per language to the type of browser used. Again, the big difference in the energy consumption between JavaScript and WebAssembly immediately stands out. However, when looking at the different types of browsers, the difference in energy consumption is smaller. But, as discussed earlier in this paragraph, this boxplot clearly shows that this difference does exist between the energy consumption of Chrome and Firefox. Namely, Chrome consumes more energy than Firefox. Since the mean of each treatment is larger than the median, it follows that the data is positively skewed. Thus, the data is most likely not normally distributed.

5.2.2 Check for normality. For the normality check of the data, a histogram for the treatments is created to give a first impression. These histograms are shown in Figure 7, and it shows the distribution of the consumption of JavaScript and WebAssembly combined with Chrome and Firefox. When solely looking at these histograms, it looks like none of these data sets are normally distributed.

However, it cannot be assumed that data is not normally distributed by just looking at a histogram of the data. Thus, the Shapiro-Wilks test is executed as well. The value of W and the p-value of this test is given in Table 7 for each of the treatments. From this follows that the data is indeed not normally distributed, since the p-value is remarkably smaller than 1 for each language-browser combination.

Since the data has to be normally distributed in order to execute a two-way ANOVA test, the data is transformed. This is done by applying orderNorm to all of the data. The transformed data is shown in the Q-Q-plots of Figure 8.

Table 6: Descriptive statistics for RQ2

	Energy Consumption (Joules)			
	Chrome and JavaScript	Chrome and WebAssembly	Firefox and JavaScript	Firefox and WebAssembly
Minimum	3.251	4.039	4.226	2.755
1st quartile	35.912	22.226	23.940	10.570
Median	72.302	27.913	52.466	17.669
Mean	81.785	31.910	69.669	19.017
3rd quartile	130.752	46.031	115.576	26.689
Maximum	164.219	68.531	170.553	41.812

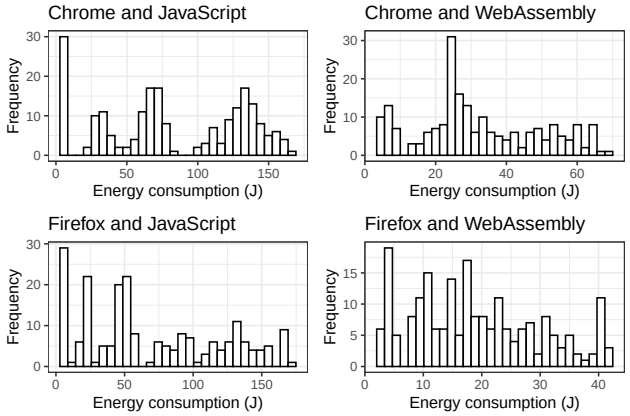

Figure 7: Distribution of energy consumption of RQ2

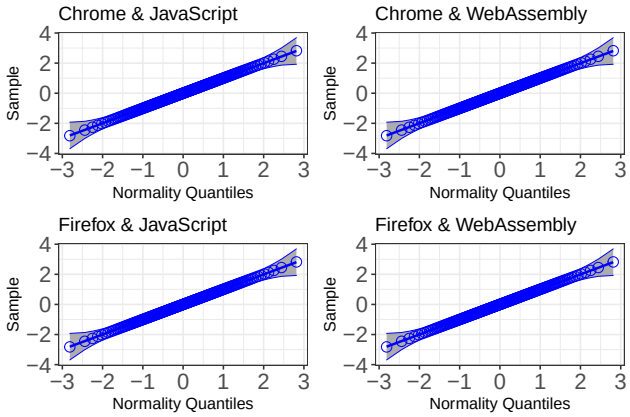

Figure 8: Q-Q-plot of JavaScript and WebAssembly combined with Chrome or Firefox

Table 7: Shapiro-Wilks test for RQ2

	W	p-value
Chrome and JavaScript	0.91868	2.39e-09
Chrome and WebAssembly	0.95126	1.476e-06
Firefox and JavaScript	0.91861	2.36e-09
Firefox and WebAssembly	0.95285	2.138e-06

These Q-Q-plots look like they show normalized data, but a Shapiro-Wilks test is executed again to confirm this presumption. From this follows the value of W = 0.99978 and a p-value of 1 for all of the treatments. Thus, the transformed data is normally distributed.

5.2.3 Hypothesis Testing. Assuming the data is normally distributed after transformation, a two-way ANOVA test is executed to calculate the contribution of both the different languages and browsers. In Table 8 the results are presented. It can be seen that both the language and the browser affect on the energy consumption of the web applications since they have a p-value of respectively < 2.2e-16 and 9.355e-07. However, the two treatments are not significantly interacting with each other considering the p-value of 0.8782.

Table 8: Two-way ANOVA for RQ2

	Df	**Sum Sq**	Mean Sq	F value	p-value
Language	1	530554	530554	394.6031	<2.2e-16
Browser	1	32834	32834	24.4207	9.355e-07
Language: Browser	1	32	32	0.0235	0.8782
Residuals	836	1124025	1345		

5.2.4 Effect Size Estimation. In Table 9 the effect size measures are shown of all the possible pairs of the two treatments. For three out of the six treatment pairs, the effect size is large. This means that according to Cliff's Delta estimation, the difference between the treatments of these pairs is noticeable strong. In one of the treatment pairs, Firefox JavaScript versus Chrome JavaScript, the effect size is negligibly weak. Looking at the density plot presented in Figure 9, it is also shown that the differences between 'Firefox and JavaScript' versus 'Chrome and JavaScript' are smaller compared to the other treatment pairs. Whereas the other treatment pairs are different in height, spread, or both.

Table 9: Effect Size Estimation for RQ2

Treatment pair	**Cliff's Delta estimate**
Chrome JS - Chrome WASM	0.5907937 (large)
Firefox JS - Firefox WASM	0.6470295 (large)
Firefox JS - Chrome JS	0.1351927 (negligible)
Firefox WASM - Chrome WASM	0.4539229 (medium)
Firefox WASM - Chrome JS	0.7060317 (large)
Firefox JS - Chrome WASM	-0.4078912 (medium)

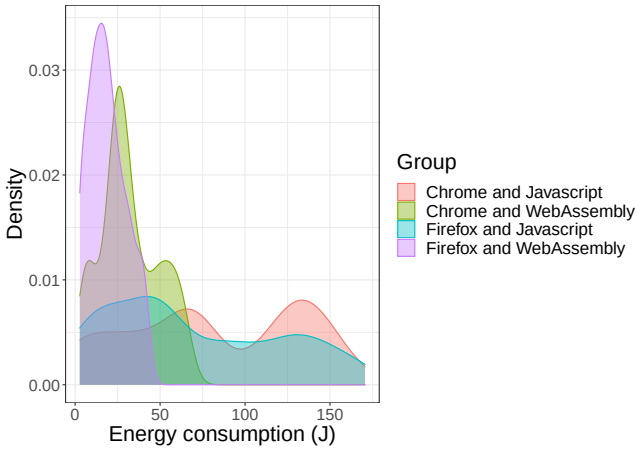

Figure 9: Energy consumption density curves

6 DISCUSSION

The results of the application of the one-way ANOVA test for RQ1 allow us to reject the H_0^{RQ1} null hypothesis. This means that the use of WebAssembly or WebAssembly for web applications has a significantly impact on the energy consumption with respect to JavaScript on Android mobile devices. Besides a one-way ANOVA test, the Cliff's Delta effect size measure was applied to the data as well. This measure shows a large effect size between the use of JavaScript versus WebAssembly. Thus, generally, it can be concluded that the difference in energy consumption between WebAssembly and JavaScript is statistically confirmed and it is *large*.

The results of this paper give developers an indication of the energy consumption of JavaScript and WebAssembly. This can be used by developers when they have to decide between these two types of languages for creating a web application.

For what concerns RQ2, the two-way ANOVA test reveals that both the language and the browser significantly affect the energy consumption of web applications. From this test follows as well that the language type and the browser type are not significantly interacting with each other. After an ANOVA test, the Cliff's Delta effect size measure was applied to the data. This effect size is large for the pairs Chrome JavaScript versus Chrome WebAssembly, Firefox JavaScript versus Firefox WebAssembly, and Firefox WebAssembly versus Chrome JavaScript. There is a medium effect size for the pairs Firefox WebAssembly versus Chrome WebAssembly, and Firefox JavaScript versus Chrome WebAssembly. The effect size is negligible for Firefox JavaScript versus Chrome JavaScript. Based on these statistical tests, it can generally be stated that both the language and the browser have an impact on the energy consumption of web applications.

When looking at the results from the perspective of **web developers**, the focus is mostly on the fact that WebAssembly consumes consistently less energy compared to JavaScript. For a web developer, this means that if they have JavaScript functions implemented which are constantly running in the background (e.g., a poller for fetching remote data, a thread for real-time image manipulation, etc.), it might be convenient to implement these functions in C and then to compile them to WebAssembly. This will make their web applications more energy efficient with respect to using a plain JavaScript implementation.

Based on our empirical results, Firefox tends to consume less energy. This result can be useful for users in some specific conditions. For example, in the case that their device only has 1% of energy left, and they want to keep working on their device for the largest amount of time possible. The mean energy consumption of WebAssembly on the Chrome browser is 31.91 Joules, and on Firefox is 19.02 Joules, which is 12.89 joules less than Chrome. This leads to a 40.39% energy saving on average when switching to Firefox from Chrome. To put this into perspective, if 5 million users switch to Firefox from Chrome, then a total of 2 million Joules of energy will be saved. This amount of energy is equivalent to the energy consumed by a 100W bulb for 6 hours.

For **browser vendors**, it is important to notice that Mozilla Firefox is an open-source project. Thus, browser vendors can look at its engine for executing JavaScript and WebAssembly. From this, they can take inspiration in order to make their execution engines more energy efficient.

7 THREATS TO VALIDITY

This section will provide an analysis of possible threats to validity of the study. The threats are categorized into four different types, as defined by Cook and Campbell: internal validity, external validity, construct validity, and conclusion validity [31].

Internal Validity. System processes may impact the energy consumption of the Android device. To mitigate this, the order of execution of the subjects was randomized, and before every run, the application data of the to be used browser is cleared. As such, the risk of system processes impacting the results of a subject is minimized. It could also be possible that the consumption of the Android device is influenced by the condition of the network over which the subjects are served, for example when the download time is increased due to fluctuations in the WiFi connection. To mitigate this, a dedicated router is used which only connects the Raspberry Pi and the Android device. Furthermore, the Android device is placed at a constant distance from the router and the Raspberry Pi is connected with an ethernet cable. A replication package is available for independent verification of the setup and the generated data.

External Validity. The most relevant threat to the external validity of this study is the fact that the subjects are not real-life applications, but benchmarking algorithms designed to be compute-intensive. This means that the results of this study can be misleading when compared to the situation in the industry, where mobile web applications typically do not do a minute of intense computation on the client-side. However, the benchmark that was used in this study, was used in multiple other studies [5] [6] [10]. A future study might compare close to real-life web applications implemented in JavaScript and WebAssembly. We used a Huawei Nexus 6P to run the trials on. Therefore, the version of Android (Android 6, Marshmallow) used may not represent the current state of Android (Android 12, Snow Cone). We choose to run the phone on Android 6 due to the compatibility issues with Trepn. Nevertheless, newer devices running newer Android releases may lead to different energy

measurements; further replications of the performed experiments can help in mitigating this potential threat to validity.

Construct Validity. To avoid inadequate preoperational explication of constructs, we defined our constructs a priori, before the experiment execution. All the details related to the design of the experiment (e.g., the goal, research questions, variables, data analysis procedures) was defined before executing the experiment. We used the GQM approach to define our goal, which then guided the definition of the research questions of this study. The hypotheses, dependent and independent variables, and treatments were all defined during the planning phase of the experiment.

Conclusion Validity. To mitigate this type of threat, we used a fixed number of treatments for both the research questions. Depending on the research question those treatments changed. We made use of 8 subjects so that means in total 32 web applications. We then executed 30 runs per web application and as a result, we have a relatively large total sample size of 960. The procedures and tests in our statistical analysis were defined a priori. During the statistical analysis, it was checked whether the assumptions of the applied statistical tests were met. This check validated that the right statistical tests was used on the data. The replication package contains all the raw data and the analysis scripts, this can be used for independent inspection.

8 CONCLUSIONS

In this paper, the impact on the energy consumption of web applications is measured between web browsers and programming languages. In our experiment, we used eight different benchmark functions to measure the energy consumption across the two treatments. The results of this experiment show that there is a significant difference in energy consumption between the use of WebAssembly and JavaScript. In addition, the choice of browser also contributed to the differences in energy consumption. Therefore to improve the energy efficiency of web applications, we recommend software developers to compile web applications to WebAssembly when possible. In the same sense, we suggest user to use the browser Firefox over Chrome, when energy is an issue. Combining these two factors will lead to the optimization of energy consumption when using web applications.

As future work, this experiment can be extended by using real-life web applications instead of benchmark functions. Such extension can make the results more representative wrt real-life situations. Additionally, we are interested in running the same experiment on a different device, for example, a smartphone with an iOS operating system. Lastly, the experiment can be repeated on different browsers. Since the use of web browsers Firefox and Chrome have an impact on energy consumption, it will be interesting to compare other web browsers as well. In the future, it will be interesting to perform a trade-off analysis between energy consumption and the run-time performance of WebAssembly.

REFERENCES

[1] A. Rossberg, "Webassembly: high speed at low cost for everyone," in *ML16: Proceedings of the 2016 ACM SIGPLAN Workshop on ML*, 2016.

[2] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of webassembly vs native code," *Proceedings of the 2019 USENIX Annual Technical Conference*, pp. 107–120, 2019.

[3] D. Frankl, "Machine learning in javascript," 2014.

[4] "Building to webassembly." [Online]. Available: https://emscripten.org/docs/compiling/WebAssembly.html#webassembly

[5] 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.

[6] F. Oliveira and J. Mattos, "Analysis of WebAssembly as a Strategy to Improve JavaScript Performance on IoT Environments," *Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais*, pp. 133–138, 11 2020.

[7] M. Reiser and L. Bläser, "Accelerate javascript applications by cross-compiling to webassembly," 10 2017, pp. 10–17.

[8] P. Sandhu, D. Herrera, and L. Hendren, "Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in javascript and webassembly," 09 2018, pp. 1–13.

[9] S. Tarkoma, M. Siekkinen, E. Lagerspetz, and Y. Xiao, *Smartphone energy consumption: modeling and optimization*. Cambridge University Press, 2014.

[10] J. Alamari and C. E. Chow, "Computation at the edge with webassembly," in *ITNG 2021 18th International Conference on Information Technology-New Generations*, S. Latifi, Ed. Cham: Springer International Publishing, 2021, pp. 229–238.

[11] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[12] A. Miettinen and J. Nurminen, "Analysis of the energy consumption of javascript based mobile web applications," vol. 45, 05 2010, pp. 124–135.

[13] J. de Macedo, J. Aloísio, N. Gonçalves, R. Pereira, and J. Saraiva, "Energy wars - chrome vs. firefox: Which browser is more energy efficient?" in *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2020, pp. 159–165.

[14] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[15] F. Shull, J. Singer, and D. I. Sjøberg, *Guide to advanced empirical software engineering*. Springer, 2007.

[16] L. Ardito, R. Coppola, M. Morisio, and M. Torchiano, "Methodological guidelines for measuring energy consumption of software applications," *Scientific Programming*, vol. 2019, p. 1–16, 2019.

[17] G. inc. V8. [Online; Consulted 20 September 2021]. [Online]. Available: https://v8.dev/

[18] M. inc. Spidermonkey. [Online; Consulted 20 September 2021]. [Online]. Available: https://spidermonkey.dev/

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

[20] S. G. Stats. Browser market share worldwide. [Online; Consulted 9 January 2022]. [Online]. Available: https://gs.statcounter.com/browser-market-share#monthly-202110-202110-bar

[21] M. inc. Firefox source documentation. [Online; Consulted 20 September 2021]. [Online]. Available: https://firefox-source-docs.mozilla.org/js/index.html

[22] K. A. Pituch, J. Stevens, and J. Stevens, *Applied multivariate statistics for the social sciences*. Routledge, 2016.

[23] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, p. 494–509, 1993.

[24] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11-12, p. 1073–1086, 2007.

[25] J. J. K. Robert J. Grissom, *Effect Sizes for Research: A Broad Practical Approach*, 1st ed. Routledge Academic, 2005. [Online]. Available: libgen.li/file.php?md5=c8d59f4f79837368fea6d52a6a3da9d2

[26] I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirovic, "Assessing the impact of service workers on the energy efficiency of progressive web apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 35–45.

[27] Y. Hu, J. Yan, D. Yan, Q. Lu, and J. Yan, "Lightweight energy consumption analysis and prediction for android applications," *Science of Computer Programming*, vol. 162, pp. 132–147, 2018, special Issue on TASE 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642317300953

[28] M. Couto, J. Saraiva, and J. P. Fernandes, "Energy refactorings for android in the large and in the wild," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 217–228.

[29] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, "Modeling, profiling, and debugging the energy consumption of mobile devices," vol. 48, no. 3, dec 2015. [Online]. Available: https://doi.org/10.1145/2840723

[30] I. Malavolta, E. M. Grua, C.-Y. Lam, R. de Vries, F. Tan, E. Zielinski, M. Peters, and L. Kaandorp, "A Framework for the Automatic Execution of Measurement-based Experiments on Android Devices," in *35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '20)*. ACM, 2020.

[31] C. Cook, *Quasi-experimentation - Design and Analysis Issues for Field Settings.*, 1979.