On the Energy Consumption and Performance of WebAssembly Binaries across Programming Languages and Runtimes in IoT

Linus Wagner
Vrije Universiteit Amsterdam
l3.wagner@student.vu.nl

Maximilian Mayer
Vrije Universiteit Amsterdam
m.mayer@student.vu.nl

Andrea Marino
Vrije Universiteit Amsterdam
a.marino@student.vu.nl

Alireza Soltani Nezhad
Vrije Universiteit Amsterdam
a.soltaninezhad@student.vu.nl

Hugo Zwaan
Vrije Universiteit Amsterdam
h.zwaan@student.vu.nl

Ivano Malavolta
Vrije Universiteit Amsterdam
i.malavolta@vu.nl

ABSTRACT

Context. WebAssembly (WASM) is a low-level bytecode format that is gaining traction among Internet of Things (IoT) devices. Because of IoT devices' resources limitations, using WASM is becoming a popular technique for virtualization on IoT devices. However, it is unclear if the promises of WASM regarding its efficient use of energy and performance gains hold true.

Goal. This study aims to determine how different source programming languages and runtime environments affect the energy consumption and performance of WASM binaries.

Method. We perform a controlled experiment where we compile three benchmarking algorithms from four different programming languages (*i.e.,* C, Rust, Go, and JavaScript) to WASM and run them using two different WASM runtimes on a Raspberry Pi 3B.

Results. The source programming language significantly influences the performance and energy consumption of WASM binaries. We did not find evidence of the impact of the runtime environment. However, certain combinations of source programming language and runtime environment leads to a significant improvement of its energy consumption and performance.

Conclusions. IoT developers should choose the source programming language wisely to benefit from better performance and a reduction in energy consumption. Specifically, Javy-compiled JavaScript should be avoided, while C and Rust are better options. We found no conclusive results for the choice of the WASM runtime.

ACM Reference Format:
Linus Wagner, Maximilian Mayer, Andrea Marino, Alireza Soltani Nezhad, Hugo Zwaan, and Ivano Malavolta. 2023. On the Energy Consumption and Performance of WebAssembly Binaries across Programming Languages and Runtimes in IoT . In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE '23), June 14–16, 2023, Oulu, Finland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3593434.3593454

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EASE '23, June 14–16, 2023, Oulu, Finland
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0044-6/23/06…$15.00
https://doi.org/10.1145/3593434.3593454

1 INTRODUCTION

The *Internet of Things* (IoT) refers to the system of networks of physical devices which allow for detailed data collection, processing, and exchange [1]. Generally, IoT devices feature sensors, low powered processing units and delegate computationally intensive tasks to centralized servers, most likely in the cloud. While centralized cloud paradigms are likely to remain prevalent in the industry for most use-cases, some IoT applications simply cannot rely on persistent network connections due to performance or security issues [2]. These concerns, eventually resulted in the idea of *Edge Computing*, which bypasses the traditional data offload in favor of a distributed architectural style [3]. Leveraging IoT devices for processing data close to the source or even locally can only be supported via heavily optimized and lightweight software. However, ease of deployment and platform compatibility also play a crucial role for such distributed designs to be a viable solution in the real world. As a result, virtualization and containerization technologies like Docker are now commonly employed in the IoT ecosystem [2]. In this context, the need for performant and energy efficient containers is even more pressing than that for their cloud counterparts.

Due to the intrinsic limitations of IoT devices, technologies that reduce boot-up overhead, memory footprint, and overall container size, such as *WebAssembly* (WASM), show greater potential than traditional implementations meant for more powerful machines [4]. WASM refers to both an assembly-like low level language and to its corresponding binary format. It was first introduced as a cross-platform compilation target for languages that are traditionally not executed on the web, e.g., C/C++, and developed as an open standard by W3C [5]. More specifically, most modern web browsers now allow for WASM module loading through JavaScript which unlocks the potential for close-to-native performance in the browser web application Virtual Machine (VM). Its sandboxed execution environment and portability made WASM also a perfect candidate for non-web use-cases and embeddings such as for IoT devices.

In order to support WASM execution outside web browsers, developers can rely on different *runtimes*. A runtime simply consists of an implementation of the WASM specification which also binds the execution to a specific system interface (e.g., POSIX or WASI) or JavaScript VM (e.g. Node) by providing basic services such as memory management and I/O. Being platform-agnostic, the WASM specification makes this technology extremely flexible and allows for runtime implementations that are tailored for specific use-cases.

Next to the choice of the runtime, source languages can also affect the compiled binary execution performance. In practice, the

runtime characteristics of the final executable might depend on several factors, such as the maturity of the compiler and support for language-specific optimizations or features such as garbage collection [6]. Therefore, while WASM ensures cross-compatibility for different languages, it is crucial for developers to be aware of the current limitations in the compilation process. Because so many factors are involved, it is challenging to reason about sustainability and performance implications of WASM-based implementations for typical IoT computing tasks. With this research, we aim to understand the viability of this technology in real-world applications with respect to energy efficiency and performance.

The **goal** of this study is to analyze WASM as a compilation target for different high-level programming languages for the purpose of evaluating differences in performance and energy consumption. We do so from the perspective of software developers, as the chosen programming language has a significant impact on the development process: based on personal experience or project requirements, developers might prefer a specific language in different situations.

This experiment involves: (i) four different programming languages (*i.e.,* Rust, Go, JavaScript, C), (ii) two different WASM runtime environments (*i.e.,* Wasmer, Wasmtime), and (iii) three third-party benchmarking algorithms. In terms of metrics, we measure the energy consumption, execution time, and memory usage of a real IoT device. The **results** of this experiment provide evidence that the choice of source programming language impacts energy, execution time, and memory consumption, and that a bad choice can result in performance drops even 5.4 times worse. We identified C and Rust as solid options for software projects working with low-powered hardware traditionally used in IoT devices, while Javy-compiled JavaScript should be avoided due to its high resource usage. We did not obtain statistical significance when considering the choice of runtime environment, since programming languages perform very differently when running on different runtime environments.

The **main contributions** of this study are: (i) an empirical investigation on the performance and energy consumption of WASM binaries compiled from 4 different programming languages (Rust, Go, JavaScript, C) and running on 2 different runtime environments (Wasmer and Wasmtime), (ii) a discussion of the obtained results, and (iii) the full replication package of the study containing raw data, source code, and scripts for data analysis. The results of this study are meant as (i) knowledge for helping IoT developers when choosing which programming language to use with WASM and (ii) a starting point for further research into language-specific differences within the WASM ecosystem in the context of IoT.

2 RELATED WORK

Yan et al. conducted a systematic investigation to compare the performance of WASM and JavaScript applications [7]. They observe that (i) the performance of WASM and JavaScript is highly dependent on the runtime environment and (ii) WASM has a significant performance advantage over JavaScript, but consumes three times more memory than JavaScript with Just-In-Time compilation because of different memory management. We analyze and compare high-level programming languages compiled to WASM for IoT devices using a single platform while Yan et al. conducted a study that compared WASM and JavaScript on various runtime environments and hardware platforms.

Herrera et al. [8] evaluated the performance of numerical computing on the web, including JavaScript, WASM, and server-side Node.js, across various devices, including IoT devices. Their study results indicate that WASM is slower to run in most browsers than native C and runs at least 30% slower than native C, regardless of the browser. Also, the WASM version of Node.js always outperforms the JavaScript version. Additionally, the WASM version of Node.js comes close to the performance of native C across all platforms. Compared to Herrera et al., our study focuses exclusively on IoT devices and examines energy consumption using non-browser runtime environments. We also chose benchmarks with 4 programming languages, while Herrera et al. only evaluate JavaScript compiled to WASM against a native C baseline.

Eriksson and Grunditz [9] looked into the performance of WASM as a containerization technology on IoT devices. For this comparison WASM runtimes, Wasmer and Wasmtime, are used and compared against Docker running C or Python. The focus is to find out if WASM can be deployed in IoT devices like network cameras. They find that Docker runs 69 times slower than Wasmtime on small tasks, but does well in medium to large tasks. In contrast, Wasmer is 3.5x slower on medium to large tasks compared to Docker. Overall, it is the slowest studied runtime because it optimized for a low start up time. They conclude that WASM is a very promising technology, but needs to mature before being ready for the main stage.

Oliveira and Mattos [10] investigate on how WASM can be used to improve the performance of JavaScript applications in IoT devices. Experiments were conducted on a Raspberry Pi to execute JavaScript, WASM, and C algorithms while collecting information on device resource consumption. Their results showed that using WASM improved JavaScript performance by 39.81% in terms of execution time, had a negligible impact on memory usage, and reduced battery consumption by 39.86%.

Macedo et al. [11] systematically analyzed the energy consumption and runtime performance of WASM and JavaScript on the web. According to their statistical analysis, WASM produces significant performance differences with up to 30% better energy efficiency than JavaScript. These differences vary between microbenchmarks and real-world benchmarks.

Hasselt et al. [12] examine the impact of using JavaScript compiled to WASM on the energy efficiency of browsers on Android devices. Their findings indicate that WASM and JavaScript have significantly different energy consumption rates.

Hampau et al. [13] investigated the effect of three different containerization strategies on the energy consumption and performance of AI-based computer vision models. Based on their findings, WASM uses the least memory and disk space, making it a good programming language for AI applications on the edge where memory and disk space are limited. In contrast, the ONNX runtime consumes less energy and executes the algorithms faster than Docker or WASM.

Our study is novel since we focus on comparing different programming languages in terms of both energy consumption and performance when they are compiled to WASM and run on IoT devices, whereas the above-mentioned studies either focus on comparing WASM against high-level languages, or on other quality properties (*e.g.,* performance or energy only), or on platforms different from IoT.

3 STUDY DESIGN

3.1 Research Questions

This study aims at answering the following research questions (RQs):

- **RQ1:** How does the source programming language affect the energy consumption of WASM binaries?
- **RQ2:** How does the source programming language affect the performance of WASM binaries?
- **RQ3:** How does the runtime environment affect the energy consumption of different programming languages in WASM?
- **RQ4:** How does the runtime environment affect the performance of different programming languages in WASM?

Our RQs focus either on performance or energy consumption in order to contribute meaningfully to our goal. However, the questions can be divided into two categories: RQ1 and RQ2 are concerned with the impact of programming languages, while RQ3 and RQ4 focus on the influence of runtime environments. For RQ1 and RQ2, we aim to provide empirical evidence for real-life differences in performance and energy consumption. This evidence can help to guide developers in their choice of a programming language when building IoT-based software with WASM. Therefore, a deeper analysis of the impact of language-specific properties, such as dynamic typing or automatic memory management, on the performance of WASM could be of great value. Similarly, runtimes can have a significant impact on performance and energy [7, 12]. However, RQ3 and RQ4 are not designed to find the best runtime for the IoT domain, but rather intended to understand whether different runtimes favor different high-level programming languages. Once again, this would allow to guide developers in their choices, and indicate that further research is necessary.

3.2 Subjects Selection

Benchmarking algorithms for this experiment are taken from *The Computer Language Benchmarks Game* [14] (CLBG). It is designed for the comparison of runtime performance across programming languages, and has been used as a benchmark for energy efficiency by others [15–18]. In its current version, the CLBG implements 10 computational problems in up to 26 languages, as well as a framework to run, compare, and test different implementations. For this work, we rely exclusively on the problem implementations in selected languages and compile them to WASM with custom scripts. Due to the time limitations for this project, we selected a subset of 3 problems. A short description of each benchmark is provided in Table 1.

Table 1: Explanation of the benchmarks.

ID	Full name	Description
NBY	nbody	Models the orbiting of Jovian planets.
BNT	binarytrees	Creates a perfect binary tree with the given depth.
SPN	spectral-norm	Calculates the maximum value in a given matrix.

In order to execute our benchmarks we use the following input values and settings. For NBY we pass 55000000 as the number of iterations for the bodies' simulation. For BNT we select a depth of 16. Since the available stack memory space in our execution environment (see Section 4) would not allow for a depth larger than 16, in order to obtain a sufficiently long execution time (*i.e.,* in the order of minutes) we fixed a number of repetitions for each run of this benchmark to 80. For SPN we pass a matrix size of 8100 as input.

3.3 Experimental Variables

For RQ1 and RQ2, this independent variable is the programming language, to which we assign four treatments: *C*, *Rust*, *JavaScript*, and *Go*. To select the four treatments, we considered three factors: the overall popularity, the support of WASM, and the balance between interpreted and compiled languages. To evaluate a language's popularity, we used the *Stack Overflow Developer Survey 2022* [19] with over 71,000 responses, the *Voice of IoT engineer* [20] which is related to our domain, and *The State of Webassembly* [21] which reflects the popularity of languages within the WASM community. By analyzing the top five languages of each survey, we found JavaScript, Python, C, and Java to be most popular. But since the support for WASM is currently unstable for Python and Java [22], we chose the next most popular languages: Go and Rust. With JavaScript, we include one interpreted language.

For RQ3 and RQ4, we have the WASM runtime environment as an independent variable. Here, we decided to have two treatments: *Wasmer* and *Wasmtime*. Both were identified by looking at the repositories with the most stars for WASM runtimes on GitHub [23].

The dependent variables of this study are the following:

- **Energy Consumption** (RQ1 and RQ3): the energy consumption will be measured in Joules (J).
- **Execution Time** (RQ2 and RQ4): the execution time for each workload of the benchmark will be measured in milliseconds (ms).
- **Memory Usage** (RQ2 and RQ4): the memory usage is the mean memory usage during each run in %.

3.4 Experimental Hypotheses

For this study, we intend to reason about the impact of our previously defined independent variables (programming language and runtime environment) on dependent variables such as energy consumption, execution time, and memory usage.
To answer RQ1 and RQ2, we define the following null hypothesis:

$$H_0^{pl,\,d}: \mu^d_{Rust} = \mu^d_{Go} = \mu^d_{JavaScript} = \mu^d_{C}$$
$$d \in \{energy\ usage,\ execution\ time,\ memory\ usage\}$$

where pl represents the independent variable for the selected programming language, while d stands for a dependent variable such that $d \in \{energy\ usage,\ execution\ time,\ memory\ usage\}$. Consequently, $H^{pl,\,d}$ then determines the effect of the chosen programming language pl on our dependent variable d. Furthermore, μ^d_p represents the average measurement result of variable d with programming language p selected as treatment.

This null hypothesis states that no meaningful difference for any of our dependent variables d can be detected when executing our benchmark with different programming languages. This leads to the following alternative hypothesis, stating that for each dependent variable d a statistically relevant difference can be observed between programming languages:

$$H_a^{pl,\,d}: \exists (p1, p2) \mid \mu^d_{p1} \neq \mu^d_{p2}$$
$$\forall p1,\ p2 \in \{Rust,\ Go,\ JavaScript,\ C\}$$
$$d \in \{energy\ usage,\ execution\ time,\ memory\ usage\}$$

To answer *RQ3* and *RQ4* we construct the following hypotheses.

Consider *re* as the independent variable for the selected runtime environment. Once again, *d* refers to our dependent variables. Then $H^{re,\,d}$ determines the effect of the chosen runtime environment *re* on our dependent variable *d*. Also, β_r^d represents hereby the median measurement result of variable *d* for a selected runtime environment, which are Wasmer and Wasmtime as treatments. The null hypothesis $H_0^{re,d}$ declares that the chosen runtime environment does not introduce any meaningful difference for any of our dependent variables, energy usage, execution time, and memory usage. The null hypothesis is formulated as:

$$H_0^{re,\,d}: \beta^d_{Wasmtime} = \beta^d_{Wasmer}$$
$$d \in \{\textit{energy usage, execution time, memory usage}\}$$

The alternative hypothesis $H_a^{re,d}$ states that there is at least one statistically meaningful difference between any of our runtime environment and our dependent variables. This is the formulation of the alternative hypothesis:

$$H_a^{re,\,d}: \beta^d_{Wasmtime} \neq \beta^d_{Wasmer}$$
$$d \in \{\textit{energy usage, execution time, memory usage}\}$$

Finally, our two independent variables might interact with each other. The effect of such interaction is modelled via a third set of hypotheses. The following two hypotheses are defined for the interaction of programming languages and runtime environments, where μ_p is the effect of treatment *pl* (Rust/C/Go/JavaScript) of the programming language factor from RQ1 and RQ2, and β_r is the effect of treatment *re* (Wasmer and Wasmtime) of the runtime environment factor from RQ3 and RQ4.

The null hypothesis $H_0^{(pl,\,re),\,d}$ states that there is no statistically significant difference between programming languages and runtime environments interactions and our dependent variables.

$$H_0^{(pl,\,re),\,d}: \mu\beta^d_{(p,\,r)} = 0$$
$$\exists p \in \{\textit{Rust, Go, JavaScript, C}\}$$
$$\exists r \in \{\textit{Wasmer, Wasmtime}\}$$
$$d \in \{\textit{energy usage, execution time, memory usage, storage}\}$$

The alternative hypothesis $H_0^{(pl,\,re),\,d}$ asserts that at least one statistically meaningful difference exists between programming language and runtime environment interactions and our dependent variables. This is the formulation of the alternative hypothesis:

$$H_a^{(pl,\,re),\,d}: \mu\beta^d_{(p,\,r)} \neq 0$$
$$\exists p \in \{\textit{Rust, Go, JavaScript, C}\}$$
$$\exists r \in \{\textit{Wasmer, Wasmtime}\}$$
$$d \in \{\textit{energy usage, execution time, memory usage}\}$$

3.5 Experiment Design

After identifying our subjects, variables, and constructing the related hypotheses for our experiment, we now have to design the experiment. Based on our two independent variables and their corresponding treatments, we will use a "two factors - multiple treatments" design type (2F-MT). The first factor results from research questions RQ1 and RQ2, and defines the programming language used to approach a given subject. Possible treatments are Go, JavaScript, C, and Rust, requiring us to use a multi-treatment design type. RQ3 and RQ4 introduce the WASM runtime as a second factor, with possible treatments being Wasmer and Wasmtime.

To generate configurations to test within our experiment, we use factorial design to include all combinations of treatments across independent variables. Furthermore, we will test each combination of treatments on all three benchmarking algorithms, resulting in 24 different configurations (4 *programming languages*×2 *runtimes*× 3 *algorithms*). Each of these configurations is then executed 10 times to account for slight variations in-between runs due to background activities of the OS and other potentially uncontrolled factors. Taking into consideration that we repeat each combination 10 times, the dataset produced in this study contains 240 data points for each dependent variable, for a total of 720 individual data points.

To answer the first two research questions, RQ1 and RQ2, all three benchmark problems will be executed in all four programming languages and they will be executed using Wasmer, as it is the most popular WASM runtime. Therefore, all four were chosen to be easily compatible with WASM, either via native language features, e.g. Rust, or 3rd-party compilers, e.g. Emscripten for C. For JavaScript, we use Javy[1], a WASM embedded JavaScript runtime. Using these compilers, the benchmark problems from the CLBG will be compiled to WASM bytecode. For RQ3 and RQ4, we will also execute all combinations of benchmarking problems and programming languages, but this time they will be executed on two different runtime environments (namely, Wasmer and Wasmtime).

3.6 Data Analysis

Data Exploration To get a first overview of the characteristics of the captured data, we use descriptive statistics and box plots.

Check for Normality To be able to apply the correct statistical tests to our data, we test our data for normality. We do this via the *Shapiro-Wilk* test with $\alpha = 0.05$ to our data. Because for some hypothesis our sample size can be as small as 10, the Shapiro-Wilk test can be unreliable. As a consequence, we also visually confirm our findings using *histograms*. Due to the amount of metrics, we only show a selection of the plots and include the rest in the replication package.[2] If we find that our data is not normal, we transform it using the following normalization techniques: *logarithm*, *square root*, $1/x$, *standardized Sepal width*, *standard scaling*, and *min-max scaling* [24]. To confirm/reject a successful normalization, we repeat the Shapiro-Wilk test. The tests results are included in the replication package.

Hypothesis Testing In the case that we find normalized data, we will use the *two way ANOVA test*. However, if we have non-normal data, we will use a variety of non-parameterized tests. For hypothesis 1 we use the *Kruskal-Wallis* test. The Kruskal-Wallis test is a rank-based test applicable to one-way data containing more than two groups. Since there are more than two groups, we choose this statistical test to evaluate our first hypothesis. For hypothesis 2 we use the *Wilcoxon Signed-Rank* test. The unpaired two-samples Wilcoxon test is a non-parametric alternative to the unpaired two-samples t-test for comparing two independent sample groups. Our last hypotheses

[1] https://github.com/Shopify/javy
[2] https://github.com/S2-group/ease-2023-wasm-iot-rep-pkg

set features both of our factors, programming language and WASM runtime, and both our performance and energy related metrics. If our data is not normally distributed, we test the statistical significance of our hypothesis using the *Aligned Rank Transformation (ART) ANOVA* test. An adjustment of the null hypothesis is hereby not necessary, as the transformation is used to make the data more amenable, however, it does not change the underlying assumptions of the ANOVA test [25]. If our data was initially not suitable for parametric tests, we relied on the Aligned Rank Transformation and the minimal statistical impact of normalization on ANOVA's robustness [26]. ANOVA with aligned ranks transformation is a non-parametric method that allows for multiple independent variables, interactions, and repeated measures.

Effect Size Estimation Whenever our hypothesis tests identify a statistically significant effect of a treatment or combination of such, we will use *Cliff's Delta* [27] to estimate its magnitude. We chose this non-parametric test as it can be used even in case our data is not normally distributed. This way, we can identify whether the influence of any treatment pair is not only of statistical significance, but also of practical relevance. We will interpret Cliff's Delta as proposed by Vargha and Delaney [28].

4 EXPERIMENT EXECUTION

As shown in Figure 1, we run our experiments on a Raspberry Pi 3 Model B Rev 1.2 with the following characteristics: 1GB RAM, Broadcom BCM2835 64-bit System on Chip, Processor sub-architecture ARM1176JZF-S at 1.2GHz. The Raspberry Pi is a popular choice for experiments in various domains [29–31] and allows for customization at a low price [32]. We run it with Raspberry Pi OS Lite 64-Bit (released 22.09.2022) and we use the Experiment-Runner, "a generic framework to automatically execute measurement-based experiments on any platform"[33], to orchestrate our experiment. We make both our code for setting up the infrastructure and the experiment orchestrator available in our replication package.

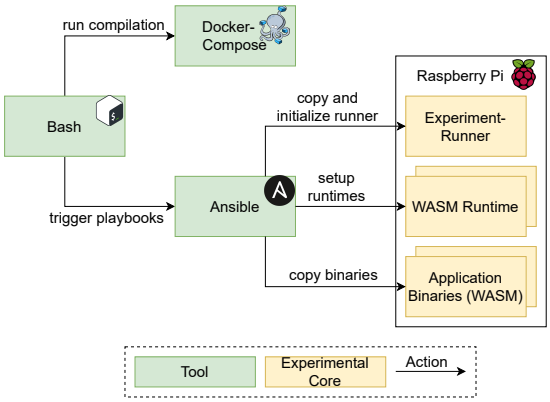


Figure 1: Experimental setup. Docker and Ansible are used to automatically set up a Raspberry Pi 3B with WASM runtimes and applications.

To compile our benchmark, we defined a Docker image for each language included in our experiment. We exposed each image as a docker-compose service, which can be configured to run on a selected set of benchmark algorithms, and stores the WASM executables in a mounted volume, in our case a Raspberry Pi. To make our experiment easy to reproduce, we have also automated the setup process of the Raspberry Pi using Ansible. This includes the installation of runtimes, the Experiment-Runner, and other configurations. As Ansible operates without an agent, no additional software needs to be installed on the device itself, reducing background noise during the experiment. Similarly, the separation of compilation and execution allows us to reduce the number of tasks that have to be performed on the Raspberry Pi itself, reserving it for the exclusive purpose of running our experiments. Docker, on the other hand, improves reproducibility, as it is a platform-independent technology and provides self-documentation of the performed steps as a byproduct of creating the corresponding images. It is important to note that in this experiment the WASM binaries are executed bare-metal on the Raspberry Pi; Docker is used exclusively for the compilation step, which is not part of the measurement phase of the experiment.

A single run of a configuration is repeated 10 times to account for statistical errors. Furthermore, we introduce a pseudo-random execution order for our configurations, where we validate that similar configurations are not executed consecutively. This allows us to minimize the risk of temporary background tasks affecting a specific configuration. In order to minimize the impact of CPU heating and general system throttle, we introduce a break of one minute after each individual benchmark run.

We collect measures via three tools: *PowerJoular*, *PS*, and *time*. PowerJoular is a tool for monitoring the power consumption of software and hardware in realtime [34], while recording the energy. We chose PowerJoular over other tools like pTop [35], because it is designed specifically with the Raspberry Pi in mind. *PS* is a Linux command-line tool that provides information about selected processes, captures the memory usage of our benchmark programs. It is included in Raspberry Pi OS, and allows for easy application in an automated setting. We measure the runtime with the Linux command-line tool called *time*.

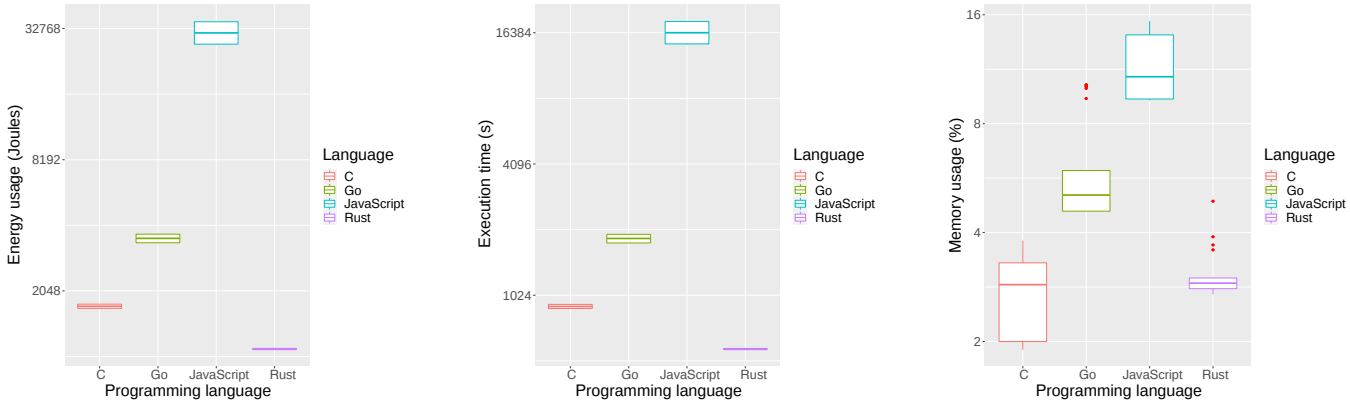
5 RESULTS

5.1 Impact of programming languages

5.1.1 Data Exploration. As shown in Figure 2a, Go exceeds the other compiled languages and uses 3.22x more energy than Rust, and 2.05x more energy than C. Compared with the interpreted language JavaScript, the energy usage of Go is low. JavaScript needs 8.83x more energy than Go. We notice a similar pattern for the execution time, which appears to be proportional to the energy consumption (notice that only the scale has changed). For the memory usage, Go continues to outrank C and Rust by, on average, a factor of at most 2.20. However, while Rust previously performed best, we observe that C is now the most performant language in terms of memory usage with an average usage of 2.774% of the total memory. As we can infer from Figure 2c, the memory usage of C varies significantly, while Rust is the most stable of all languages. While JavaScript is still the most resource-demanding language, the differences are now considerably smaller, with only a factor of 4.20 between the memory usage of C and JavaScript.

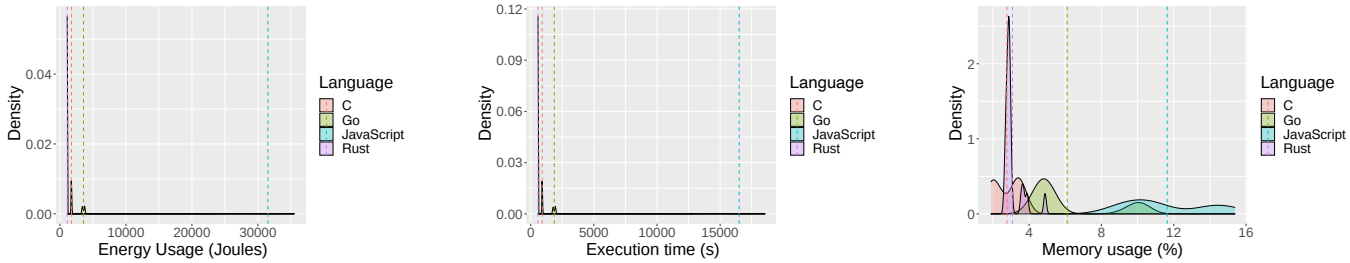
5.1.2 Normality and Data Transformation. Our analysis of the histograms, which can be found in the replication package, suggest a

[3] Raw numbers of each of the presented plots are present in the replication package.



(a) Energy usage. **(b) Execution time.** **(c) Memory usage.**

Figure 2: Box-plots for H^{pl}. The dependent variables are scaled logarithmically for better visibility [3].



(a) Energy usage. **(b) Execution time.** **(c) Memory usage.**

Figure 3: Density curves for H^{pl}.

non-normal distribution. We use the Shapiro-Wilks test to confirm this for each of our sub-dataset, which did not yield p-values above 0.05. Therefore, we can reject the normal distribution null hypothesis. We applied the normalization techniques mentioned in Section 3.6, but this did not yield normalized data.

5.1.3 Hypothesis Testing. Our first hypothesis set features a single factor, the programming language, and both our performance and energy metrics. Since we apply multiple treatments to the programming language and that our data is not normally distributed even after transformation (see Section 5.1.2), we chose to test the statistical significance of our hypothesis using the Kruskal-Wallis test. Our results in Table 2 indicate that the mean for our metrics across the language treatments is significantly different, with p-values very close to 0. Therefore, we can reject the null hypothesis H_0^{pl}. This means that there is a significant difference among at least two programming languages regarding both energy, execution time, and memory.

Table 2: Test results for hypothesis H^{pl} across all variables.

	Chi-Square	DF	P-value
Energy Usage	74.074	3	5.722e-16
Execution Time	74.075	3	5.719e-16
Memory Usage	64.078	3	7.899e-14

5.1.4 Effect Size Estimation. As the Kruskal-Wallis test provides us with significant evidence that the choice of programming language has an effect on all of our dependent variables, we will now use Cliff's Delta to uncover the magnitude of this effect. By looking at the density curves in Figure 3, we expect a large effect size given the spread of the memory usage and also the different peaks of energy usage and execution time. The results can be found in Table 3.

Table 3: Cliff's Delta values for H^{pl}.

		C	Rust	Go	JavaScript
Energy Usage	**C**	-	1.000	-1.000	-1.000
	Rust	-1.000	-	-1.000	-1.000
	Go	1.000	1.000	-	-1.000
	JavaScript	1.000	1.000	1.000	-
Execution Time	**C**	-	1.000	-1.000	-1.000
	Rust	-1.000	-	-1.000	-1.000
	Go	1.000	1.000	-	-1.000
	JavaScript	1.000	1.000	1.000	-
Memory Usage	**C**	-	-0.175	-1.000	-1.000
	Rust	0.175	-	-0.960	-1.000
	Go	1.000	0.960	-	-0.850
	JavaScript	1.000	1.000	0.850	-

Interestingly, Cliff's Delta takes a value of 1 or -1 in most cases. This represents the maximum and minimum value possible respectively, and thus a very large effect size confirming our expectations. In particular, JavaScript is significantly slower and consumes far more energy than any other language, while Rust is seemingly faster and uses less energy. Only for the memory usage we find values other

than 1 and -1. Here, the effect size differs from small, with a value of 0.175/-0.175 for C and Rust, to large for all other cases. Overall, this result is also in line with the box-plots in Figure 2, which present a similar trend between the different languages.

5.2 Impact of WASM runtime environments

5.2.1 Data Exploration. In Figure 4 we compare both runtimes, with Figure 4a showing energy usage. Based on the mean, Wasmer uses about 18.59% less energy than Wasmtime. Both runtimes are positively skewed. When comparing both medians, Wasmer only uses 4.57% less energy compared to Wasmtime.

Figure 4b compares execution time, which seems to be proportional to the energy usage.

Finally, Figure 4c shows that Wasmer also has a lower memory usage than Wasmtime. Compared to the energy usage, the difference in memory usage is rather high, with a deviation of 14.95% between both medians. Both distributions are negatively skewed, with higher rather than lower memory usage being the norm. Contrary to the energy usage and execution time, where Wasmer has a smaller interquartile range, for the memory usage Wasmtime has the smallest of both runtimes.

5.2.2 Normality and Data Transformation. The histograms in the replication package suggest a non-normal distribution. This result is also confirmed by the Shapiro-Wilks test. We applied the normalization techniques mentioned in Section 3.6, but this did not give us normalized data.

5.2.3 Hypothesis Testing. Our second hypotheses set features a single factor, the WASM runtime, and both our performance and energy related metrics. Given that we only apply two different treatments to our single factor, Wasmer and Wasmtime, and that our data is not normally distributed even after transformation (see Section 5.2.2), we chose to test the statistical significance of our hypothesis using the Wilcoxon Rank-Sum test.

Table 4: Wilcoxon Rank-Sum test results for hypothesis H^{re} across all metrics.

	W	P-value
Energy Usage	700	0.3383
Execution Time	700	0.3383
Memory Usage	646	0.1393

As shown in Table 4, there is no significant difference for all our metrics across the different runtime environments, with p-values as large as 0.3383. Therefore, we cannot reject the null hypothesis H_0^{re}. Since the statistical test did not yield significant results, we do not perform an effect size estimation.

5.3 Interaction between programming languages and runtime environments

5.3.1 Data Exploration. In Figure 5 we compare the interaction of runtimes with different languages. The Figure 5a shows the energy usage of each programming language among the two single runtime environments. When JavaScript is run in Wasmtime, it uses the most energy of all and consumes 21.29% more energy than it does in Wasmer. Running Rust in Wasmer uses the least amount of energy of all treatments. Compared to its run in Wasmtime, it uses 1.41% less energy on average and 35.34% less energy than the third least energy-consuming combination C in Wasmtime.

Similarly to the two previous hypotheses, execution time exhibits a similar pattern as energy consumption (see Figure 5b).

Finally, Figure 5c shows the memory usage of a single language in a single runtime. While JavaScript is still the most demanding language, it now performs worst in Wasmer and consumes 0.83% more memory than in Wasmtime. C in Wasmer is the programming language consuming the lowest amount of memory (*i.e.,* 2.099%).

5.3.2 Normality and Data Transformation. Based on the histograms and on the results of the Shapiro-Wilk test (both reported in the replication package), we conclude that our data does not follow a normal distribution.

5.3.3 Hypothesis Testing. Since the data does not follow a normal distribution, we apply the ART ANOVA test (see Table 5).

Table 5: Interactions for energy usage, execution time, and memory usage.

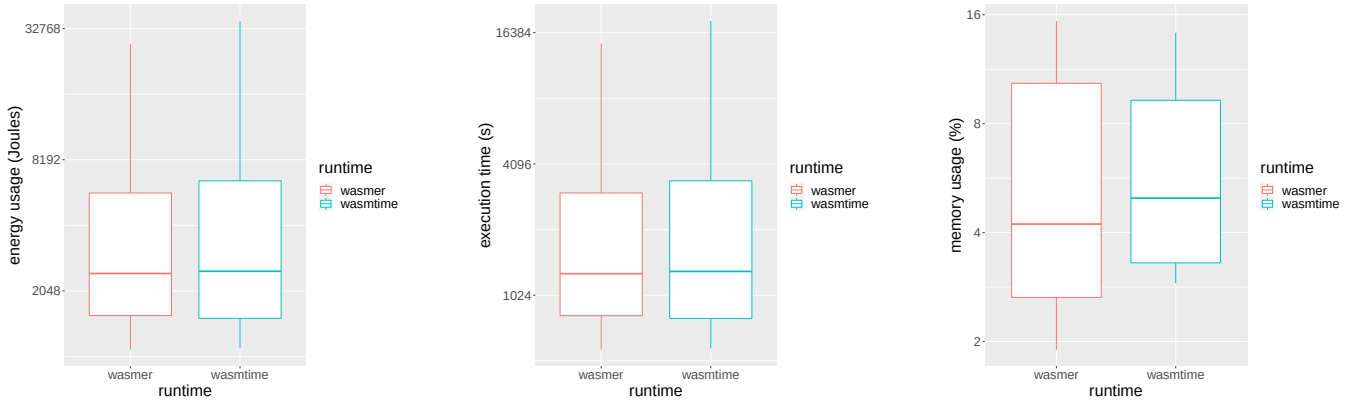
	Variable	DF	Df.res	F.value	P-value
Energy Usage	Language	3	72	369.49	<2.22e-16
	Runtime	1	72	237.85	<2.22e-16
	Language:Runtime	3	72	1509.09	<2.22e-16
Exec. Time	Language	3	72	376.16	<2.22e-16
	Runtime	1	72	242.46	<2.22e-16
	Language:Runtime	3	72	1510.24	<2.22e-16
Mem. Usage	Language	3	72	129.4181	<2e-16
	Runtime	1	72	2.8784	0.094098
	Language:Runtime	3	72	3.5444	0.018691

Hereby, we can identify an effect for the programming languages on all dependent variables, as the corresponding p-values are all smaller than 0.05. Meanwhile, the runtime has no effect on memory usage with a p-value of 0.094, which is in line with our previous results using the Wilcoxon Rank-Sum. However, with the ANOVA test, we identify a statistically significant effect for both, the energy usage and execution time, with corresponding p-values approaching 0. This is likely due to the fact that our data was not fully normalized, and the Aligned Rank Transformation only provides an approximation. Consequently, the results are less reliable than the previous results based on non-parametric tests.

The interaction of runtimes and programming languages has a statistically significant impact, as shown by the corresponding p-values for all three dependent variables. Thus, we can reject the null hypothesis $H_0^{(pl,re)}$.

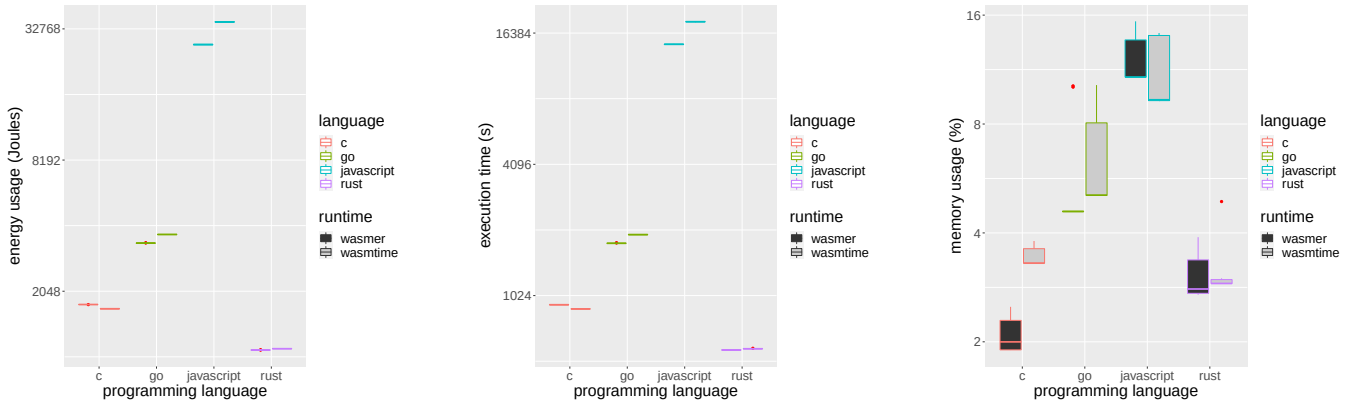
5.3.4 Effect Size Estimation. As we were able to reject the null hypothesis $H_0^{(pl,re)}$ for all three dependent variables, we have statistically significant evidence that the combination of different treatments for our two factors influences the resulting energy usage, execution time, and memory usage. As a result, we use Cliff's Delta to verify that this effect is also of realistic impact. The delta values for the memory usage can be found in Table 6. For the other metrics, all delta values are either 1.000 or −1.000, so we omit them here for readability.

This means that the effect size for all dependent variables is very large, for all possible interactions of language and runtime. This is in large part due to the large influence of the selected programming



(a) Energy usage. **(b) Execution time.** **(c) Memory usage.**

Figure 4: Box-plots comparing Wasmer and Wasmtime.



(a) Energy usage. **(b) Execution time.** **(c) Memory usage.**

Figure 5: Box-plots comparing the interaction of runtime and language.

Table 6: Cliff's Delta values for $H^{(pl,re)}$ - Memory Usage.

	Wasmer C	Wasmer Rust	Wasmer Go	Wasmer Javascript	Wasmtime C	Wasmtime Rust	Wasmtime Go	Wasmtime Javascript
Wasmer C	-	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000
Wasmer Rust	1.000	-	-1.000	-1.000	-0.500	-	-1.000	-1.000
Wasmer Go	1.000	1.000	-	-1.000	1.000	0.840	-0.640	-0.760
Wasmer Javascript	1.000	1.000	1.000	-	1.000	1.000	1.000	0.440
Wasmtime C	1.000	0.500	-1.000	-1.000	-	0.800	-1.000	-1.000
Wasmtime Rust	1.000	-	-0.840	-1.000	-0.800	-	-1.000	-1.000
Wasmtime Go	1.000	1.000	0.640	-1.000	1.000	1.000	-	-0.640
Wasmtime Javascript	1.000	1.000	0.760	-0.440	1.000	1.000	0.640	-

language. However, the interaction of both factors plays a similarly important role: When comparing the configuration "Wasmtime - C" to "Wasmer - Rust", the delta value for memory usage is 0.500, however, when comparing "Wasmer - C" to "Wasmer - Rust", the delta value moves to −1.000. Consequently, the impact of Wasmer in combination with C is much larger than when paring C with Wasmtime, even when comparing to the same configuration of Wasmer and Rust.

6 DISCUSSION

We analyzed our results based on our three hypothesis pairs, as this allowed for a meaningful grouping and interpretation of our data. In the following section, we will translate these results to our four research questions, and expand on our interpretation by taking the perspective of a IoT software developer.

For our first two research questions, *RQ1* and *RQ2*, we focused on comparing the energy consumption and performance respectively of different source programming languages compiled to WASM. Based on our results for the corresponding hypothesis H^{pl}, we were able to gather statistically significant evidence that the chosen programming language does in fact have a meaningful influence on all selected dependent variables. Furthermore, the effect size is large enough to result in real-world impact.

However, this is not true for our research questions *RQ3* and *RQ4*, which were analyzed by the means of hypothesis H^{re}. Focused on the impact of the selected runtime on energy consumption and performance, we investigated data generated using several programming languages. However, we found no statistical significance related to an impact on any of our dependent variables, and consequently, we cannot assume any real-world difference of the runtime on energy consumption or performance.

Finally, we analyzed the interaction of both, programming language and runtime, as part of our third and final hypothesis, $H^{(pl,\, re)}$. The resulting data is relevant for all four research questions, as it impacts both, the choice of programming language and runtime. Hereby, we found statistical evidence that the interaction of our two factors has in fact an effect on all of our dependent variables. Furthermore, this effect is also relevant in a realistic setting, indicating that developers should carefully pair the right runtime with a given programming language to observe a real-world impact.

Interpreting these results from the perspective of an IoT software developer, we have several recommendations. IoT devices often limit available resources, and consequently any reduction in power or resource usage is welcome.

In particular, we discourage developers from using WASM in combination with Javy-compiled JavaScript, when possible. Our data clearly shows that it performs worse than any other programming language, independently of the chosen runtime. Lin Clark from the Bytecode Alliance explains a possible reason[4]: To run JavaScript, an engine is bundled with a WASM instance. However, WASM does not allow JIT compilation, a mechanism which speeds JavaScript engines up. To work around that, they recommend pre-initialization, which reduces the startup time, and inline caches, which increase the throughput. Javy uses pre-initialization to reduce its startup time, but does not use inline caches to increase the throughput. Javy needs to be optimized further to deliver similar results to the other languages.

When looking for a particularly fast and energy-efficient language, Rust is a viable option, outperforming other languages across all test scenarios. However, it is important to note that benchmark algorithms used during our tests were designed to push the system to its limits. As a consequence, the CPU usage during our tests stayed at around 98%-100%, leading to similar energy consumption per second for all tested programming languages. Therefore, the improved energy efficiency is a direct consequence of the lower execution time, and more research is needed to investigate efficiency for less intense loads. For memory usage, both C and Rust perform reasonably well. While C has instances where it uses less memory than a comparable Rust program, results for C tend to spread over a larger range. However, on any reasonably modern system, neither of both should cause problems. In terms of runtime impact, we were not able to identify a statistically significant effect, thus we cannot make any general recommendations. However, the data found in Figures 5a and 5b reveals that Go tends to be faster when running on Wasmer, while C is more efficient when using Wasmtime. Consequently, the correct choice of runtime is still important, when a certain language has to be used. Nonetheless, the impact of the chosen language is by far greater.

Because we focussed on the availability in multiple programming languages when we selected our benchmarks, our findings might not be generalizable to WASM as a whole. More research with different benchmarks and more samples is needed for clarifying this aspect.

[4]https://bytecodealliance.org/articles/making-javascript-run-fast-on-Webassembly

7 THREATS TO VALIDITY

Internal Validity. The execution of the whole experiment is completely automatized via dedicated orchestration scripts (which are available in the replication package), thus limiting the possible influence of manual steps and allowing for independent replication and verification of the experiment. Several precautions have been taken while setting up the measurement infrastructure in order to collect metrics in a reliable and precise manner, such as the 1-minute waiting time between runs, the repetition of each experimental trial, the stopping of unnecessary background processes at the OS level, and the randomization of the order of the runs of the experiment.

External Validity. The external validity of our experiments is threatened by three factors: the compile chain, the immature ecosystem, and the use of micro-benchmarking algorithms.

JavaScripts bytecode needs to run in a dedicated runtime. At the moment, JavaScript applications are therefore interpreted by a native JavaScript runtime wrapped into WASM. Because there are many ways to do this and many JavaScript runtimes to choose, our selection Javy can significantly influence the generalizability of our findings for the energy consumption and performance of JavaScript in WASM. Because we chose the most popular compile chain, we deem this an acceptable result for practitioners, but call for more research that helps to generalize the interaction of WASM and JavaScript.

The immaturity of WASM supporting tools for JavaScript also reflects in the whole WASM ecosystem. Many programming languages are either not or poorly supported. This has impacted our choice of the programming language in a way that we chose programming languages by taking their maturity in WASM into account. Therefore, the findings in this paper are highly language specific and largely show the mature parts of WASM. Nevertheless, due to the selection of programming languages mostly on popularity, we are confident to draw relevant conclusions for practitioners.

Finally, because we target multiple programming languages, we had to use micro-benchmarks in our experiments, which only exercise individual characteristics of realistic workloads. This makes the results less tailored to the IoT-domain. However, we are not aware of realistic IoT workload that are available in multiple programming languages and creating our own would be both time-consuming and challenging to show correctness.

Construct Validity. In terms of construct design, our research was partially affected by the large range of metrics we opted for. While striving for a setup that would prevent mono-operational biases, the inclusion of some of our measures made part of our analysis redundant. Specifically, metrics such as the CPU usage in our initial analysis did not yield interesting results, because the processor was always fully utilized as a result of the used benchmarks. Additionally, the strong correlation between the energy consumption expressed in Joules and our execution time makes one of them redundant.

Conclusion Validity. As discussed in Section 3.6, we mitigated the risks of having erroneous statistical results, we carefully checked the assumptions of the applied statistical tests. Moreover, our replication package allows to verify each phase of our data analysis.

8 CONCLUSIONS AND FUTURE WORK

In this report, we analyzed the impact of selected programming languages and runtimes on the energy consumption and performance of WASM binaries in the context of IoT devices. We tested four different programming languages across two different runtimes by the means of three benchmarking algorithms, measuring energy usage, execution time, and memory usage. The results of this experiment provide evidence, that the source programming language can have a major impact on all the selected metrics, and that a bad choice can result in performance up to 5.4 times worse. We identified C and Rust as solid options for software projects working with low-powered hardware traditionally used in IoT devices, while Javy-compiled JavaScript should be avoided due to its high resource usage. We did not arrive at conclusive results considering the choice of runtimes, as different languages can favor different runtimes. We recommend further research in this area to allow for a deeper understanding of the relation between programming language and runtime in WASM. Furthermore, the severe performance penalty experienced when using JavaScript compiled with Javy should be further investigated, as it might hint to a compatibility issue of WASM with interpreted languages.

For future work, we plan to use more than 3 benchmarks to increase the generalizability and confidence in our results. We also plan to study the used benchmarks independently to understand if WASM favors certain kinds of computations. Finally, to see, if our findings also extend to real-world workloads, we want to extend our work with realistic benchmarks.

ACKNOWLEDGMENT

This project is partially supported by (i) the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS" and (ii) the Individual Travel Grant of the Amsterdam University Fund.

We would like to thank Saúl Cabrera for supporting and informing us on the current state of Javy.

REFERENCES

[1] O. Vermesan and P. Friess, *Internet of things: converging technologies for smart environments and integrated ecosystems.* River Publishers, 2013.

[2] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate IoT Edge Computing with Lightweight Virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, 2018.

[3] F. Al-Turjman, *Edge Computing From Hype to Reality*, ser. EAI/Springer Innovations in Communication and Computing. Springer Cham, 2019.

[4] W3C Community Group. [Online]. Available: https://webassembly.org/

[5] WebAssembly Working Group, "WebAssembly Core Specification," https://www.w3.org/TR/wasm-core-2/.

[6] B. Spies and M. Mock, "An Evaluation of WebAssembly in Non-Web Environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10.

[7] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the Performance of Webassembly Applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 533–549.

[8] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, "Numerical Computing on the Web: Benchmarking for the Future," in *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 88–100.

[9] F. Eriksson and S. Grunditz, "Considering WebAssembly Containers on IoT Devices as Edge Solution," Bachelor's Thesis, Linköping University, 2021. [Online]. Available: https://liu.diva-portal.org/smash/get/diva2:1575228/FULLTEXT01.pdf

[10] F. Oliveira and J. Mattos, "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.* Porto Alegre, RS, Brasil: SBC, 2020, pp. 133–138.

[11] J. De Macedo, R. Abreu, R. Pereira, and J. Saraiva, "WebAssembly versus JavaScript: Energy and Runtime Performance," in *2022 International Conference on ICT for Sustainability (ICT4S)*, 2022, pp. 24–34.

[12] M. van Hasselt, K. Huijzendveld, N. Noort, S. de Ruijter, T. Islam, and I. Malavolta, "Comparing the Energy Efficiency of WebAssembly and JavaScript in Web Applications on Android Mobile Devices," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, ser. EASE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 140–149. [Online]. Available: https://doi.org/10.1145/3530019.3530034

[13] R. M. Hampau, M. Kaptein, R. van Emden, T. Rost, and I. Malavolta, "An Empirical Study on the Performance and Energy Consumption of AI Containerization Strategies for Computer-Vision Tasks on the Edge," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, ser. EASE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 50–59.

[14] D. Bagley, B. Fulgham, I. Gouy, and D. Alioth, "The Computer Language Benchmarks Game." [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[15] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[16] W. Oliveira, R. Oliveira, and F. Castor, "A Study on the Energy Consumption of Android App Development Approaches," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 42–52.

[17] L. Koedijk and A. Oprescu, "Finding Significant Differences in the Energy Consumption when Comparing Programming Languages and Programs," in *2022 International Conference on ICT for Sustainability (ICT4S)*, 2022, pp. 1–12.

[18] M. Denti and J. K. Nurminen, "Performance and Energy-Efficiency of Scala on Mobile Devices," in *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*, 2013, pp. 50–55.

[19] Stack Overflow, "Stack overflow developer survey 2022," https://survey.stackoverflow.co/2022/.

[20] R. Murdock, S. Hoffenberg, and C. Rommel, "Voice of the IoT Engineer 2017: Survey Dataset and Analysis," https://www.jasoncassel.com/assets/files/White-papers/Voice-of-the-IoT-Engineer.pdf, 2018.

[21] C. Eberhardt, "The State of WebAssembly 2022," https://blog.scottlogic.com/2022/06/20/state-of-wasm-2022.html, Jun. 2022.

[22] "Awesome WebAssembly Languages," https://github.com/appcypher/awesome-wasm-langs, 2022.

[23] GitHub, "WASM Runtimes," https://github.com/search?q=WASM+Runtimes&type=repositories&s=stars&o=desc.

[24] S. G. K. Patro and K. K. Sahu, "Normalization: A Preprocessing Stage," 2015.

[25] J. O. Wobbrock, L. Findlater, D. Gergle, and J. J. Higgins, "The Aligned Rank Transform for Nonparametric Factorial Analyses Using Only Anova Procedures," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 143–146.

[26] J. P. S. Keenan A. Pituch, *Applied Multivariate Statistics for the Social Sciences: Analyses with SAS and IBM's SPSS*, 6th ed. Routledge, 2015.

[27] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, pp. 494–509, 1993.

[28] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[29] A. Kumar, G. Chattree, and S. Periyasamy, "Smart Healthcare Monitoring System," *Wireless Personal Communications*, vol. 101, no. 1, pp. 453–463, Jul. 2018.

[30] A. K. Saha, S. Sircar, P. Chatterjee, S. Dutta, A. Mitra, A. Chatterjee, S. P. Chattopadhyay, and H. N. Saha, "A raspberry Pi controlled cloud based air and sound pollution monitoring system with temperature and humidity sensing," in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, 2018, pp. 607–611.

[31] S. A. Umarghanis, F. Darari, and A. Wibisono, "A Low-Cost IoT Platform for Crowd Density Detection in Jakarta Commuter Line," in *2020 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, 2020, pp. 121–128.

[32] M. Maksimovic, V. Vujovic, N. Davidović, V. Milosevic, and B. Perisic, "Raspberry Pi as Internet of Things hardware: Performances and Constraints," in *Proceedings of the 1st International Conference on Electrical, Electronic and Computing Engineering*, Jun. 2014.

[33] N. Chalkiadakis and I. Malavolta, "Experiment-Runner," https://github.com/S2-group/experiment-runner, 2022.

[34] A. Noureddine, "PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools," in *2022 18th International Conference on Intelligent Environments (IE)*, 2022, pp. 1–4.

[35] T. Do, S. Rawshdeh, and W. Shi, "ptop: A process-level power profiling tool," in *Proceedings of the 2nd Workshop on Power Aware Computing and Systems (HotPower'09)*, Oct. 2009.