WebAssembly versus JavaScript: Energy and Runtime Performance

João De Macedo*†, Rui Abreu‡, Rui Pereira†, João Saraiva*†
*University of Minho
†HASLab/INESC Tec
‡Faculty of Engineering of University of Porto & INESC-ID
a76268@alunos.uminho.pt, rui@computer.org, rui.a.pereira@inesctec.pt, saraiva@di.uminho.pt

2022 International Conference on ICT for Sustainability (ICT4S) | 978-1-6654-8286-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICT4S55073.2022.00014

***Abstract*—The worldwide Web has dramatically evolved in recent years. Web pages are dynamic, expressed by programs written in common programming languages given rise to sophisticated Web applications. Thus, Web browsers are almost operating systems, having to interpret/compile such programs and execute them. Although JavaScript is widely used to express dynamic Web pages, it has several shortcomings and performance inefficiencies. To overcome such limitations, major IT powerhouses are developing a new portable and size/load efficient language: WebAssembly.**

In this paper, we conduct the first systematic study on the energy and run-time performance of WebAssembly and JavaScript on the Web. We used micro-benchmarks and also real applications in order to have more realistic results. Preliminary results show that WebAssembly, while still in its infancy, is starting to already outperform JavaScript, with much more room to grow. A statistical analysis indicates that WebAssembly produces significant performance differences compared to JavaScript. However, these differences differ between micro-benchmarks and real-world benchmarks. Our results also show that WebAssembly improved energy efficiency by 30%, on average, and showed how different WebAssembly behaviour is among three popular Web Browsers: Google Chrome, Microsoft Edge, and Mozilla Firefox. Our findings indicate that WebAssembly is faster than JavaScript and even more energy-efficient. Additionally, our benchmarking framework is also available to allow further research and replication.

***Index Terms*—Energy Efficiency, Green Software, Runtime, WebAssembly, Web Browsers**

I. Introduction

Over the last decades, the internet has been a significant and powerful resource. It has transformed how we work, play, communicate, and socialize. The majority of these operations are performed using Web browsers, the most frequently used software tools for accessing the internet. They can do much more than rendering a Web page [1]. As the Web platform has matured, complex and demanding Web applications, such as music editing and streaming, video editing, encryption, and game development, have emerged. The number of web applications has been growing drastically both for desktop and mobile devices. In fact, they can have a considerable and non-negligible impact on sustainability.

JavaScript (JS) is an "easy-to-use object scripting language designed for creating live online applications that link together objects and resources on both clients and servers", according to Netscape and Sun in 1995 [2]. For more than two decades, JS has been the *de facto* standard client-side Web scripting language [2]. Even though JS technology has improved by utilizing sophisticated Virtual Machines (VM) providing both Just-In-Time (JIT) compilation and GPU support, JS was not designed with performance in mind. Recent studies analyzed the performance of 27 programming languages performing the same ten software problems: JS ranks 15 in the reported ranking. It is 6.5 times slower and 4.45 times more energy greedy than the C language (the fastest and greenest in that ranking) [3], [4], [5]. In mobile devices, results are a bit more conflicting when JS is compared against Java and C++ [6]. For numerical computations, JS is within a factor of 2 compared to C when it comes to runtime performance. [7].

As the only embedded language on the Web, JS falls short in efficiency and security, especially as a compilation target. To overcome such limitations, major IT powerhouses are developing a new portable and size/load efficient bytecode language: WebAssembly (abbreviated Wasm) [8]. According to W3C, Wasm is the fourth language for the Web which allows code to run in the browser [9]. The other three languages - HTML, CSS, and JavaScript - were developed in the previous century, already! In fact, the introduction of Wasm is an important contribution aiming at improving both the run-time performance [8] and the security of Web applications [10]. As any bytecode format, Wasm is not a new language to directly write our applications. Instead, it is a compilation target that allows C/C++, Rust or TypeScript developers to build their applications, compile to Wasm and execute it on a browser.

As a consequence of its modern design, the Wasm developers outline an expected run-time performance gain of around 30% on the Google Chrome browser compared to JS performance [8]. Being a pretty new language/system, however, it is important to fully assess its impact on the websites and browsers we use daily. Because internet browsing is one of the main tasks performed in non-wired devices (smartphones/tablets/laptops) the impact of Wasm on the energy consumption of applications/Web browsers is also a critical aspect that may influence its adoption/success [11]. Unfortunately, there is no work analysing in detail the energy consumption of Wasm applications when compared to an equivalent JavaScript alternative.

In this paper we present the first detailed study on the impact on the energy efficiency of Wasm. We consider two real-

978-1-6654-8286-8/22/$31.00 ©2022 IEEE
DOI 10.1109/ICT4S55073.2022.00014

world Wasm applications developed with benchmark goals: the WasmBoy console emulator [12] and the *PSPDFKit* portable document format (PDF) viewer/editor [13]. The WasmBoy emulator was directly written in JavaScript (Typescript) and compiled into Wasm. The *PSPDFKit* editor was developed in C/C++ and this code was compiled both into a low level and optimized subset of JavaScript (*asm.js*) and into Wasm. Thus, we are able to compare the JS and Wasm implementations of both benchmarks. Moreover, we also analyse the performance of ten Wasm/JS micro-benchmarks.

We designed an empirical study to understand the performance of Wasm, both in terms of its execution time and energy consumption. With this study we wish to answer the following research questions:

- **RQ1** *Is Wasm currently more energy efficient than JS, and if so, are they significantly different?*
  Since Wasm is designed to become the universal compilation target for the web, it is important to assess whether it is also already more energy efficient than the existing solutions. It is also important to understand how significant the difference is, since it may influence its early adoption, or not.
- **RQ2** *Is Wasm currently faster than JS?*
  Wasm is a low-level language whose instructions are intended to compile almost directly to hardware. Collecting run-time results from real-world applications let us know if its advanced architecture makes it faster than the JIT JS compilation.
- **RQ3** *Does Wasm present the same performance between micro-benchmarks and real-world applications, in terms of energy consumption and runtime?*
  There are several micro-benchmarks that test Wasm versus JS run-time performance [8]. We would like to consider not only such micro-benchmarks, but also larger real-world benchmarks, in order to have more real and trustworthy performance measurements.
- **RQ4** *Does Wasm present the same performance between different browsers, in terms of energy consumption and runtime?*
  Web browsers are the most widely used software tool to access the internet. Since the developers of all major browsers are also involved in the creation of Wasm, it is important to study which browser spends less energy and has the best runtime when running Wasm bytecode.

This paper is organized as follows: Section II briefly presents the Wasm ecosystem. In Section III we describe the design and execution of our study. Section IV analyses in detail the energy consumption and runtimes of our benchmarks. In Section V we present the threats to validity, while in Section VI we discuss related work. Finally, in Section VII we give our conclusions.

II. WebAssembly's Ecosystem

The Wasm ecosystem was developed by the companies that offer the four most widely used Web browsers, namely Google, Microsoft, Mozilla, and Apple. Wasm is a low-level, compiled assembly-like language that can run with near-native performances on all major browsers. According to Mozilla Tech, *"WebAssembly is one of the biggest advances to the Web platform over the past decade."* and, over time, many current productivity applications (e.g., email, social networks, word processing) and JS frameworks will likely adopt Wasm to substantially decrease load times and increase run-time performance [14].

The primary concern of Wasm is speed, safety, and portability. Wasm has significant outgrowths for the Web platform since it allows applications written in most major programming languages to efficiently run on the web. Wasm is intended to be the *de facto* Web compilation target for languages such as C/C++, Rust, Haskell, etc. Thus, it increases Web software portability while significantly improving speed [15]. Moreover, Wasm is developed to operate alongside JS, making this combination a powerful tool because JS can focus on DOM manipulation, while Wasm can handle CPU-intensive tasks. Although Wasm included a human readable format (WAT), there are compilers for most languages that produce the low level Wasm code, such as Cheerp [16], Emscripten [17], AssemblyScript [18], and Asterius [19]. Cheerp allows companies to preserve critical legacy applications written in Java, Flash and C/C++, and automatically migrate them to HTML5 and Wasm, making their application accessible from any modern browser. Emscripten is a complete open source compiler toolchain for Wasm. It compiles C/C++ code (or any other language that uses LLVM) into Wasm. The *PSPDFKit* benchmark was compiled into Wasm with Emscripten. AssemblyScript compiles a variant of TypeScript (basically JS with types) to Wasm using Binaryen[1]. The hand written code of the Gameboy benchmark was compiled to Wasm with this compiler. Asterius is an Haskell to Wasm compiler based on GHC. It compiles Haskell source files or Cabal executable targets to Wasm+JS code. There are many more compilers within the Wasm ecosystem, which is changing the way developers build Web applications: they are not limited to the JS realm and are able to use their favourite programming language.

III. Benchmark Design and Execution

The development of a new language and its supporting tools, namely compilers and virtual machines, is a complex and time consuming task. Moreover, during its development, the language and its tools need to be tested and compared to the state-of-the-art competitors to fully assess the advantages of such new language. Wasm is no exception, and although its ecosystem is still in its infancy, it is crucial to compare it to the state-of-the-art, fully optimized JavaScript environment. Because one of the main goals to develop Wasm is the improvement of the performance of Web applications, it is particularly relevant to compare the run-time and energy performance of Wasm and JS.

A preliminary study of the energetic and run-time behavior of the novel Wasm and the matured JS languages has been

[1]Binaryen is a compiler and toolchain infrastructure library for Wasm, written in C++. It is available at https://github.com/WebAssembly/binaryen

reported in [20]. This study, however, considers only a set of micro-benchmarks of heavy computational operations which do not represent typical Web applications. Furthermore, the compiled Wasm programs are executed directly by the node.js virtual machine and not by a browser as usual in Web applications. In this paper we consider two Wasm web-based applications developed with benchmarking purposes. Moreover, we also benchmark the performance of these applications when executed within a browser-based environment, namely within the Google Chrome, Mozilla Firefox, and Microsoft Edge browsers. To monitor such Web browsers, we developed a framework to measure the energy consumption and run-time when such browsers are executing the benchmark applications. Finally, we extend the previously reported micro benchmark study so that the micro benchmarks are executed in the (monitored) browsers.

A. Real-World Wasm Applications

To study the performance of Wasm we consider two real-world applications developed with benchmark goals, namely WasmBoy and *PSPDFKit*.

a) ***The WasmBoy Benchmark****:* WasmBoy is a Gameboy/Gameboy Color Emulator, written in TypeScript to benchmark Wasm. WasmBoy is written in JavaScript/TypeScript and it was created with the main goal of comparing the run-time performance between Wasm, that is produced by the AssemblyScript compiler and the ES6 latest version of JavaScript as produced by the TypeScript compiler. This game console includes six different open source games that can be executed by the console. We updated the WasmBoy source code in order to specify the browser where the games have to be executed. Thus, considering the 6 games, with 2 two languages we benchmark (Wasm and JS), across the 3 chosen browsers, we have a total of 36 unique samples.

b) ***The PSPDFKit Benchmark****:* This software allows to view, annotate, and fill in forms in PDF documents on any platform. In order assess the possibility of porting this software to the Wasm ecosystem, the company that developed it created the *PSPDFKit* benchmark: a real-world, open-source benchmark aiming to compare its Wasm and JS implementations. To execute this benchmark with realistic inputs we considered five different pdf documents: one book divided into three parts (with 20, 40, and 80 pages, respectively), one scientific paper (10 pages long), and a slide presentation (containing 20 slides). We also made a few changes in the application's source code in order to execute several inputs in the two different languages considered (Wasm and asm.js). Similar to the WasmBoy benchmark, we developed `makefiles` to automate the execution in the different browsers. Considering our 5 various examples, with the 2 languages, across the 3 chosen browsers, we have a final total of 30 unique program executions.

It should also be noticed that there is a difference between the JavaScript implementations of the two benchmarks. The *PSPDFKit* benchmark uses asm.js, a low-level and fast subset of JS (that is not particularly human writable). In contrast, WasmBoy uses ES6, the first significant update to the JS language. Moreover, the JS (asm.js) implementation of *PSPDFKit* is a highly optimized implementation as it was produced by the C/C++ compiler, while the JS implementation of WasmBoy was hand-written with no advanced optimizations.

B. Micro-Benchmark Programs

Micro-benchmarks is one of the principal ways to measuring the performance of a software system. Wasm is no exception, and researchers have presented a preliminary study and its results on the run-time and energy efficiency performance of Wasm versus JS [20]. The considered benchmark programs are originality written in C and compiled into both Wasm and JS.

In order to compile C-based benchmarks into their respective Wasm and JS variants, they used the Emscripten compiler. Emscripten compiles a C program to Wasm and generates two files (.wasm and .js) that operate together. The .wasm file includes the C translated source code, while the .js file (also known as glue code) is the compilation's primary target for loading and setting up the Wasm code. Likewise, when compiling to JS, a .js file is created with the translated code and a .mem file with the static memory initialization data.

Wasm was designed to be used in compute-intensive cases such as compression, encryption, image processing, games, and numeric computations. Thus, they monitored the energy consumed and execution time of 10 heavy computational (Table I) programs from two benchmarks repositories: Roseta Code[2] and Computer Language Benchmarks Game (CLBG)[3]. From Rosetta Code, they obtained a total of 8 different `sorting` algorithm solutions, and from CLBG, they used two intensive benchmark problems: **Fannkuch-redux** and **Fasta**. Both Rosetta Code and CLBG have been used to evaluate programming language performance and/or assess their energy economy. Finally, below is the list of the benchmarks utilized, along with a short explanation of each, for a total of 10 distinct solutions.

One of the aspects that they wanted to investigate was if performance scale differently for Wasm and JS depending on the input size. Thus, they divided the input data into three categories: `Small, Medium,` and `Large` for each benchmark. While the input size and data change across benchmarks, they are consistent among the three languages tested (C, Wasm, and JS) within each benchmark. This gave a total of 90 distinct compiled programs to evaluate, based on three input sizes and three languages (C, Wasm, and JS) used throughout the ten benchmarks.

That previous study, however, has an important limitation since it executes both Wasm and JS micro-benchmarks in their virtual machines without considering the use of web browsers. In order to simulate a more realistic test case, we developed

[2]Rosetta Code: http://www.rosettacode.org/wiki/Rosetta_Code

[3]The Computer Language Benchmarks Game: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

TABLE I: Benchmarks details.

Benchmark	Description
Fannkuch-redux	Indexed access to tiny integer sequence.
Fasta	Generate and write random DNA sequences.
Bead Sorting	Sort an array of positive integers using the Bead Sort Algorithm.
Circle Sorting	Sort an array of integers into ascending order using Circlesort.
Identifier Sorting	Sort a list of IDs, in their natural sort order.
Lexicographic Sorting	Given an integer n, return n in lexicographical order.
Merge Sorting	The merge sort is a recursive sort of order n*log(n).
Natural Sorting	Sort a list of strings, in their natural sort order.
Quick Sorting	Sort an array of elements using the quicksort algorithm.
Remove Duplicates and Sort	Remove all duplicates of a given array and sort.

the framework to measure the performance within a browser-based environment.

C. Measuring Energy and Run-time

To monitor the energy consumption of our benchmarks, we rely on Intel's Running Average Power Limit (RAPL)[4]. RAPL monitors the energy consumed by the system's Package, CPU cores, GPU, and DRAM with a high sample rate (10ms). In fact, RAPL has previously been used in several research works on energy consumption and software [21], [22], [23], [24], [25], [4], [5], and has been proven to give highly accurate energy measurements [26].

We have developed a C-based thread that runs alongside the benchmark execution, constantly sampling the energy usage to ensure no register overflow[5] happens while measuring using RAPL. This thread also records the start and finish run-times of the benchmark being performed. Each benchmark was run five times [27], with a five-second sleep between each execution, to gather consistent data and minimize cold start, warm-up, and cache effects.

All measurements were performed on a Linux Ubuntu 20.04.2.0 LTS operating system, with 16GB of RAM, Intel® Core™ i7 8750H 1.80 GHz Maximum Boost Speed 1.99 GHz, with a Coffee Lake micro-architecture. The versions used of Chrome, Firefox and Edge, were: 92.0.4515.107 (Official Build) (64-bit), 90.0 (64-bit) and 92.0.902.55 (Official Build) beta (64-bit), respectively.

To reduce the overhead caused by other tasks running on the computer, we limited the number of processes running by the Linux OS to the minimum, and the browsers used a single tab to execute the benchmark.

[4]Intel® Power Governor: https://software.intel.com/content/www/us/en/develop/articles/intel-power-governor.html

[5]A known possible occurrence when using RAPL longer than 60s

D. Data Collection

We evaluate Wasm and JS programs' performance in three of the four most popular and used browsers according to several statistics websites[6,7]: Google Chrome, Mozilla Firefox, and Microsoft Edge. We did not include Safari in our study because it does not have a stable version for the operating system of Linux.

The final step of our benchmarking framework is the data collection. We have created a Python script, *clean-results.py*, with three versions depending on the benchmark used. It automatically aggregates all the RAPL energy and run-time samples per benchmark-input-language-browser-execution or benchmark-language-browser-execution for micro-benchmarks and real-world applications, respectively. The final result is a csv file for each benchmark-program pair containing the results of all three browsers and languages, with their RAPL samplings combined. Each csv file includes the results for each execution and final results of our measured metrics (both median and mean): Package (Joules), CPU cores (Joules), DRAM (Joules), GPU (Joules) and Time (Seconds).

IV. Analysis and Discussion

This section presents the benchmark results collected by running the two Wasm benchmarks and the micro-benchmarks presented in the previous section. The main focus is to understand if Wasm is already outperforming JavaScript when it comes both to energy consumption and run-time execution, considering that Wasm is still in a very early phase. Furthermore, we answer the research questions presented in the introduction and its possible justifications.

A. Results

Wasm has only been available for a few years, yet it's already in all of our browsers, whether we realize it or not. Given the hype surrounding Wasm, we want to see if it is already outperforming JS in terms of energy usage and run-time execution.

To better understand the results, our graphics include blue and green bars that represent the energy consumed $(Joules)$ by CPU and DRAM, respectively (left axis). The orange line corresponds to the right axis, which indicates the run-time in seconds. Finally, the red dots represent the relationship between the total energy used (we consider the sum of CPU and DRAM) and the amount of time spent. This ratio may be considered as the average power $(Watts)$ utilized. In each chart, the results are ordered (from left to right) by the browser used (Chrome, Edge and Firefox). The lower the bars and the orange line, the more efficient the system is in terms of both energy and run-time, and the lower the red dots, the less Power the language spend.

Figures 1 and 2 show the energy and runtime results of the *WasmBoy* game and *PSPDFKit* viewer/editor. For example in Figure 1, the top-left most chart represents the energy

[6]statista: https://www.statista.com/

[7]statcounter: https://gs.statcounter.com/

consumed and run-time by game *Back To Color*. In Figure 2, the top-left most chart shows the information about the performance of the book with 20 pages.

Figure 3 shows results collected from five different micro-benchmarks with all input sizes. Each chart represents one benchmark example with a respective input size of `Small`, `Medium`, or `Large`, and within each size are the three browsers in Wasm and JS. For example, the top left most chart represents the **Sorting-Natural** benchmark with `Small` input.

In Figure 4, it is possible to see the average improvements (or not) of micro-benchmarks and real-world applications using Wasm. The higher the bars, the more energetically or runtime efficient the benchmarks were. For example, in terms of energy (left bar), the micro-benchmarks (blue bars) spent 5.18% more energy using Wasm than JS. On the other hand, the real apps spent 24.34% less energy using Wasm. This means that micro-benchmarks use less energy using JS, unlike real apps that are more energy efficient using Wasm.

Figures 5 and 6 show the average percentage of energy and runtime gains between all JS and Wasm performances with *WasmBoy* and *PSPDFKit* on each Web browser. The higher the bars, the more efficient the system is using Wasm. For example in Figure 5, with *WasmBoy*, Wasm was 20.88% more energetically efficient, on average, using Google Chrome. In other words, Wasm had an energy consumption reduction of 20.88%, on average. In Figure 6 with *WasmBoy* using Google Chrome, Wasm was 5.15% faster than JS, on average.

Figure 7 is the average power, in $Watts$, utilized by JS and Wasm in all executions of *WasmBoy* and *PSPDFKit*. For example, in *WasmBoy*, using Firefox, Wasm had a power of 5.1 $Watts$, which means that, on average, Wasm consumed 5.1 $Joules\ per\ second$, while JS used more energy per second (7 J/s). The lower the bars, the more energy energetically the system is.

Finally, each real-world benchmark and metric has a set of violin plots accessible on the benchmark's online page[8] in jupyter notebook files. This information allows us to see the total density of the gathered data, including outliers, medians, and quartiles, for both energy consumption and run-time. These graphics help in understanding when a language is consistent or has a lot of inconsistencies.

B. Discussion

The main focus of this study is to compare the energy and run-time performance between Wasm and JS. We analyze if there is difference between the two and how significant the difference is. We also examine how its behavior changes between different environments, in which the Wasm has the best performance and how better that performance is.

Micro-benchmarks solutions give us a diversity of different outcomes, as shown in Figure 3. Google Chrome is always the more stable browser, both in energy and run-time efficiency. Moreover, Wasm is more efficient than JS most of the times, while Microsoft Edge and Mozilla Firefox have mixed results. For example, while in the **Fannkuch-redux**, **Fasta**, and **Sorting-Circle** programs, Edge has better runtime performance using Wasm, although it uses more power (the exception being the **Sorting-Circle** when executed with a `Medium` and `Large` input). Mozilla is always faster running JS except in the **Fasta** example. In the **Circle Sorting** program, the results differ with different input sizes, Small and Large. With Small input, Chrome and Edge have better performances with Wasm. However, with Large input size, the gap between JS and Wasm is significantly smaller. The **Normal Sorting** solution is the only program where JS is always more runtime efficient. In **Lexicographic Sorting**, JS and Wasm have equal performances except with Firefox, where Wasm had a poor performance.

In real-world benchmarks (Figures 1 and 2), the graphics are similar. Nevertheless, the power differs between *WasmBoy* and *PSPDFKit*. *WasmBoy* solutions use less power to run the programs while, in *PSPDFKit*, the power is similar. It is possible to notice that the energy gap between JS and Wasm is more significant on *WasmBoy*.

To understand if there is an statistically-significant difference between Wasm and JS we performed a statistic analysis on the obtained results. Thus, we tested the following hypothesis:

$$H_0 : P(A > B) = 0.5$$
$$H_1 : P(A > B) \neq 0.5$$

where $P(A > B)$ represents, when we randomly draw from both A and B, that the probability of a draw from A is larger than the one from B is 50% in the case of our null hypothesis, and different than 50% in our alternative hypothesis.

The data from all measured samples were grouped according to their type (micro-benchmarks (MB), *WasmBoy* (WB), and *PSPDFKit* (PDF) and each of the analyzed browser (Google Chrome, Microsoft Edge, and Mozilla Firefox). Additionally, for micro-benchmarks, we also grouped each by input size (`Small`, `Medium`, and `Large`). Thus, we obtained 12 (A, B) pairs, such as, ($^{MB}\text{JS}_{Chrome/Small}$ vs. $^{MB}\text{Wasm}_{Chrome/Small}$), ($^{WB}\text{JS}_{Edge}$ vs. $^{WB}\text{WASM}_{Edge}$), ($^{WB}\text{JS}_{Chrome}$ vs. $^{WB}\text{WASM}_{Chrome}$), ($^{PDF}\text{JS}_{Firefox}$ vs. $^{PDF}\text{WASM}_{Firefox}$), etc.

We considered the samples independent, normally distributed and ran the Wilcoxon signed-rank test with a two-tail p-value with confidence level of 5%. To calculate a non-parametric effect size, *Field* [28] suggests using Rosenthal's formula [29], [30] to compute a correlation and compare the correlation values against [31] proposed thresholds of 0.1, 0.3, and 0.5 for small, medium, and large magnitudes, respectively.

The micro-benchmarks improvements were not very considerable for Chrome and all input sizes, with p-values > 0.05. Firefox and Edge had significant differences with p-values < 0.05. Thus we can say that these two browsers had meaningful differences between JS and Wasm performance. However, these differences mean opposite things because, on Edge, the difference is related to the better efficiency of Wasm,

[8]Github page: https://github.com/OnThePerformanceofWebAssembly/PerformanceOfWebAssembly

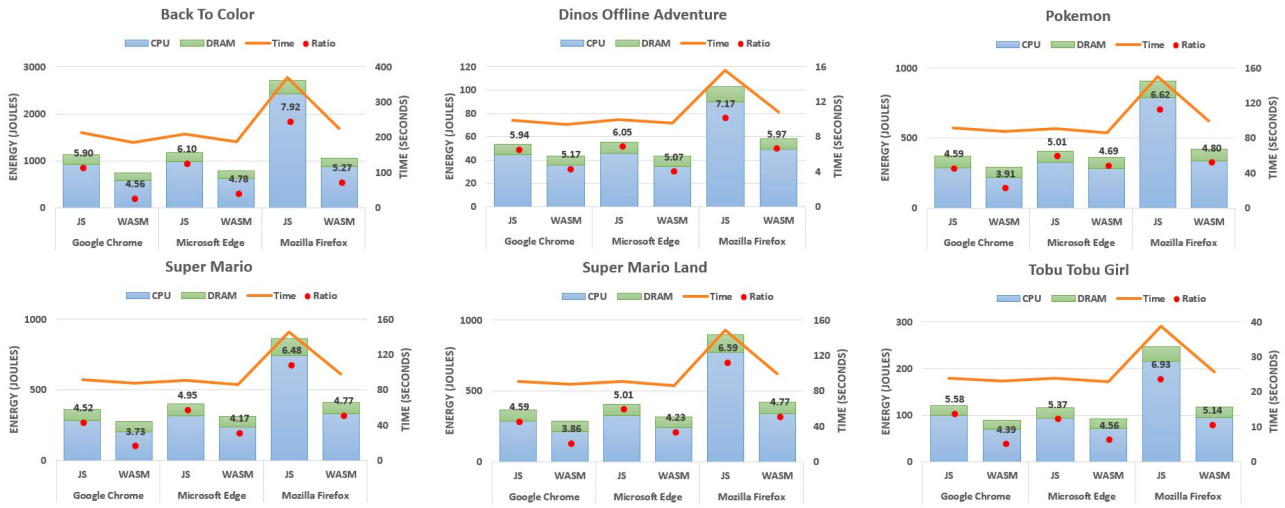

Fig. 1: *WasmBoy*: Energy consumed and run-time by each program in each Web browser.

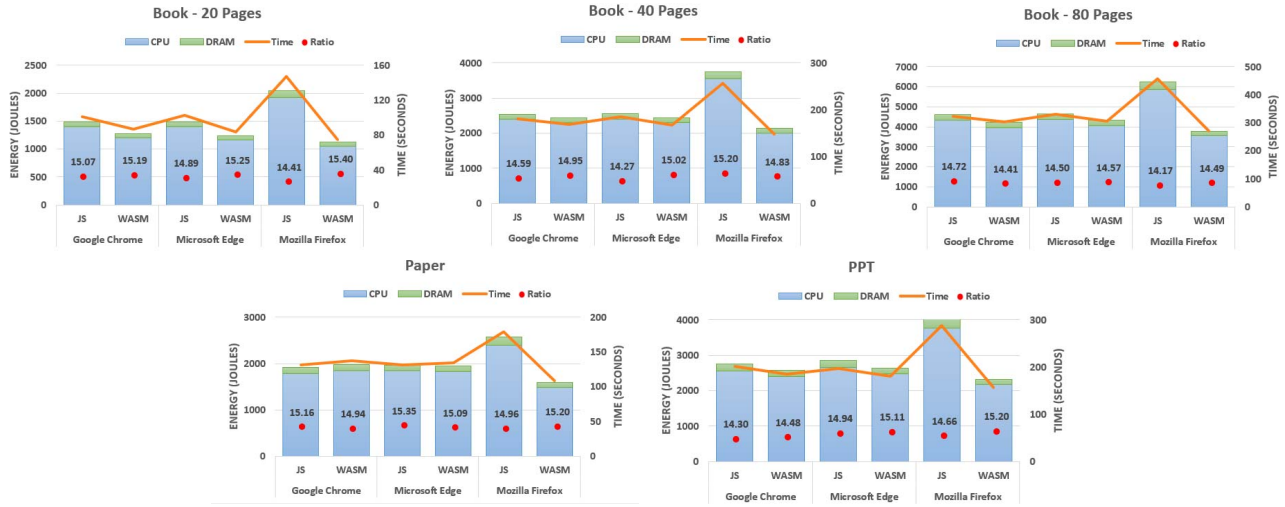

Fig. 2: *PSPDFKit*: Energy consumed and run-time by each program in each Web browser.

but on Firefox is due to JS be better. When calculating the effect size of these two browsers, we obtained the values of 0.41 and 0.5 for the respective Edge and Firefox. It means that the improvement of Wasm on Edge had a medium effect while, on Firefox, JS had a large effect.

WasmBoy and *PSPDFKit* had improvements completely different than micro-benchmarks. The differences were indeed very significant, producing statistically-significant results in all browsers, with all p-values < 0.0001. The same happens with the effect size, with all values > 0.8 (large effect). Thus, this shows that JS and WASM performances are significantly different, with a very large magnitude of discrepancy.

Returning to our research questions and looking at Figure 4, we can claim that the results between micro-benchmarks and real apps are significantly distinct. Unlike micro-benchmarks that spent, on average, more 5.18% energy using Wasm, real-world applications, had a average reduction of energy consumption of 24.34%, compared to JS. We have shown that there are both significant improvements and a large effect size when using Wasm to increase the energy efficiency of real applications **(ARQ1)**. Its compact binary format and low-level nature mean the browser can load, parse and compile the code faster than JS. It is anticipated that Wasm can be compiled faster than browsers can download it. With JS, the performance generally increases with each iteration as it is further optimized, however it can also decrease due to re-optimization.

Looking at run-time results, we can say that the outcomes are also very different between micro and real-time applications, as occurred in energy results. While Wasm, on average, is 9.84% slower than JS in micro-benchmarks, the opposite happens in real-world applications, where Wasm, on average,

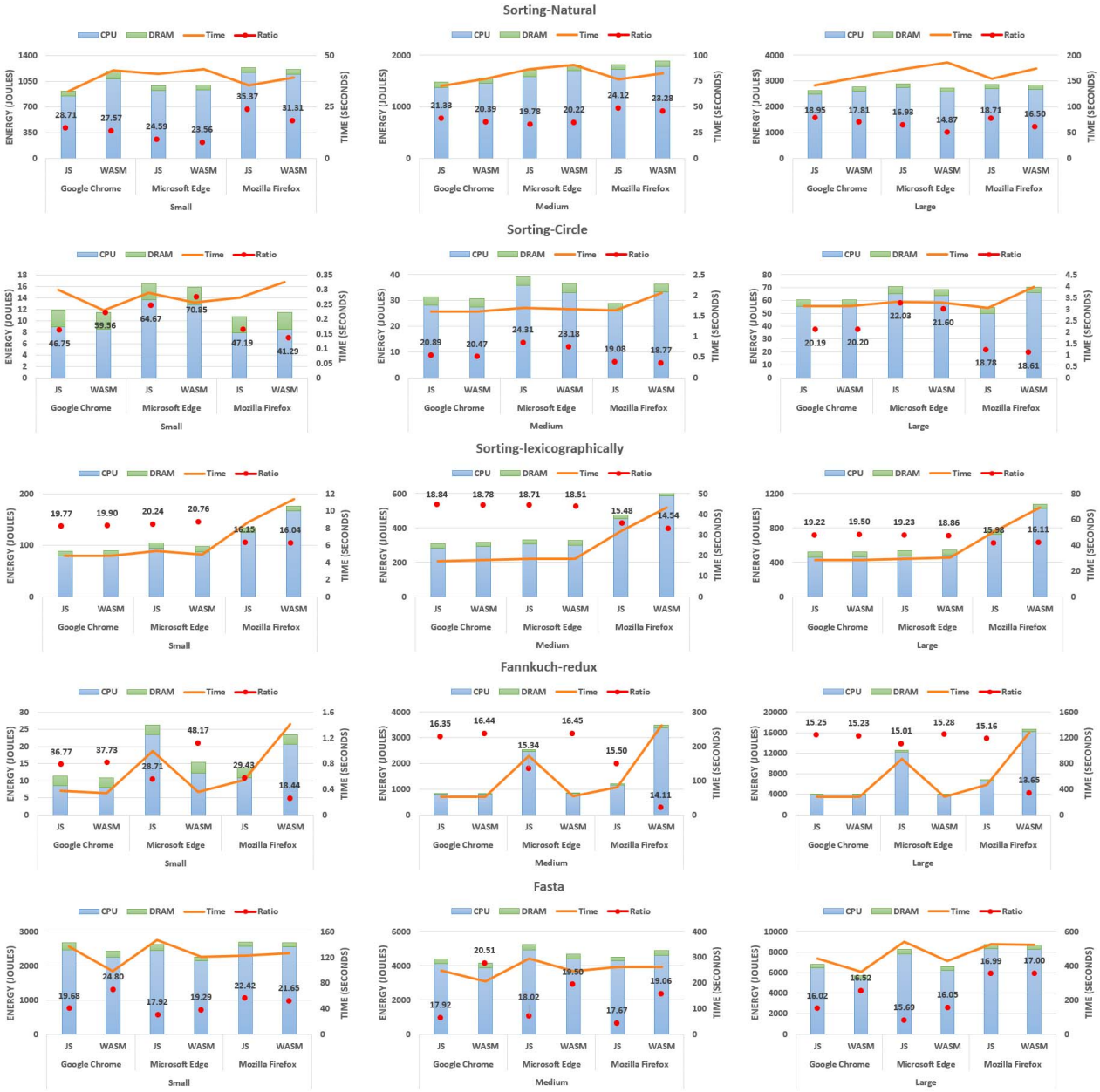

Fig. 3: Micro-Benchmarks: Energy consumed by each browser for each benchmark with the three input sizes and the respective execution times and ratio values.

is 17.24% faster than JS **(ARQ2)**.

These significant differences between benchmarks lead us to try to understand the different benchmarks' behaviors. Figure 4 shows clearly that Wasm's behavior is not the same between micro and real applications **(ARQ3)**. JS has much better results on micro-benchmarks because of its optimization over time through the browser engine's Just-in-Time compiler. Engines like V8 and SpiderMonkey optimize JS until getting a near-native performance. These optimizations only happen if it's doing the exact same small piece of code over and over in a loop. Also, for Wasm, it is assumed that the producing (offline) compiler has already performed relevant optimizations, so a Wasm JIT tends to be less aggressive than one for JS, where static optimization is impossible. Another reason is that the code testing is so small that overheads within the test loop are a significant factor. For example, the smallest overhead in calling Wasm from JS can affect the results. Consequently, micro-benchmarks are not the best and most realistic fit to measure Wasm performance.

There are also some relevant differences between the two

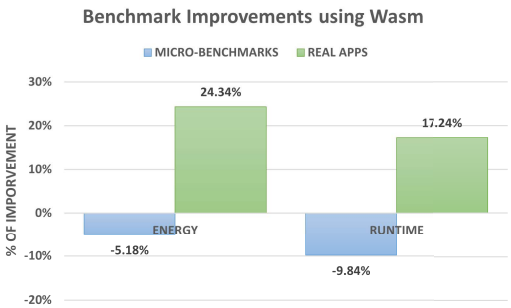

Fig. 4: Percentage of energy and runtime improvements between micro-benchmarks and real-world applications using Wasm.

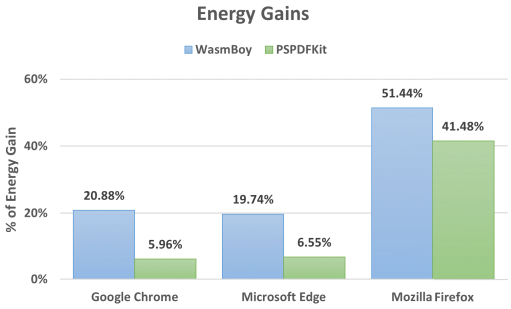

Fig. 5: Average percentage of energy gains between JS and Wasm performances on real-world applications.

realistic benchmarks, with an average energy gain/reduction of 30.69% and 18% for *WasmBoy* and *PSPDFKit*, respectively (averages of Figure 5). Regarding the runtime performance, *WasmBoy* and *PSPDFKit* were faster using Wasm, with speed increases of 14.92% and 19.25%, respectively (averages of Figure 6). This is due to their different language type because asm.js is a very small, strict subset of JS highly optimized in many JS engines using Just-In-Time (JIT) compiling techniques. The performance characteristics of asm.js are closer to native code than that of standard JS. However, asm.js is not

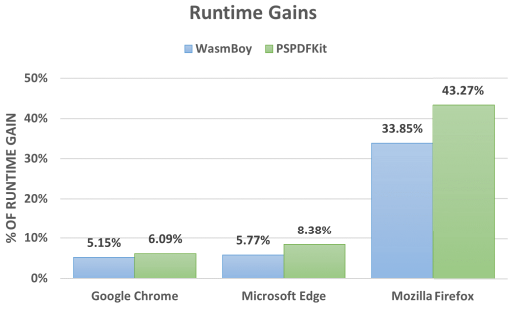

Fig. 6: Average percentage of runtime gains between JS and Wasm performances on real-world applications.

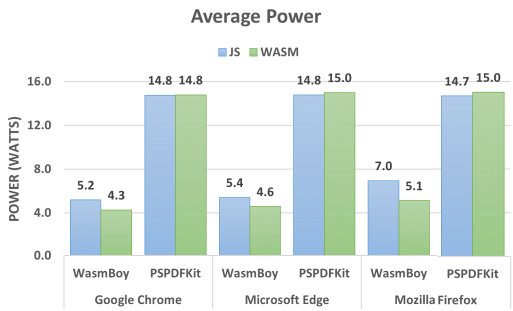

Fig. 7: Average power, in Watts, used by JS and Wasm on real-world applications.

humanly writable, unlike ES6 that is the standardization of JS.

Additionally, looking only at real applications, we calculated the average percentage gains (in terms of energy consumption and runtime) between JS and Wasm to understand if the behavior was the same within the three browsers. As shown in Figure 5, *WasmBoy*, achieved energy reductions using Wasm of 20.88%, 19.74%, and 51.44% for Google Chrome, Microsoft Edge, and Mozilla Firefox, respectively. In *PSPDFKit*, the energy improvements were not so attractive for Chrome and Edge, having gains of 5.96%, 6.55%, and 41.48% for Chrome, Edge, and Firefox, respectively. Figure 6 shows that all three browsers were faster using Wasm. *WasmBoy*, using Wasm, was 20.88%, 19.74%, and 51.44% faster with Google Chrome, Microsoft Edge, and Mozilla Firefox, respectively. *PSPDFKit*, had runtime gains of 5.96%, 6.55%, and 41.48% for Chrome, Edge, and Firefox, respectively **(ARQ4)**. Edge had similar results to Chrome on both applications because it is based on the open-source Chromium browser to run on Linux OS. While both Chrome and Edge have similar improvements, Firefox had a considerable percentage of gains, reaching more than 40% performance energetically in both applications. These results can be related to the weaker Firefox performance with JS [24], but now, with Wasm, Firefox appears to compete with the competition. The remaining question here is: *Will be, in the future, Mozilla Firefox the best Web-browser with Wasm evolution?*

Finally, we would like to know if Wasm improvements affect energy efficiency and run-time performance in the same way. With *PSPDFKit*, the average gain of energy and run-time using Wasm were similar, with 18% and 19.41%, respectively. Nevertheless, with the most practical application in this study, *WasmBoy*, the results were slightly different. The run-time performance had a 15.06% average improvement, while energy performance was two times more efficient, with 30.69% of average gains. Even so, with *WasmBoy* solutions, as shown in Figure 7, Wasm can be faster using less power per second using all three browsers. Therefore, these outcomes can show that Wasm can be faster than JS and, even so, utilize less energy. Wasm is very novel and has much more room to grow. We expect that with the continued development and support that the language has, it will surpass JavaScript over time by

a large margin. Likely, Wasm is here to stay and revolutionize the Web.

V. Threats to validity

This research aims to measure and understand the energetic and run-time behavior in micro-benchmarks and real applications between the new Web language Wasm and matured JS. This section presents some threats to the validity of our study, separated into four categories.

Conclusion: While the difference between Wasm and JS performance is irrelevant in micro-benchmarks, there is a clear and significant gain in both energy and run-time efficiency of Wasm using real applications. However, analyzing the impact of other hardware components (such as memory usage) deserve further analysis. All our data and benchmarking framework are available and can be easily extended to include additional benchmarks.

Internal: The PSPDFKit editor is arguably not really a JavaScript application, at least not one any person would write, since it is generated from C/C++ code. Therefore, the PSPDFKit benchmark is great if you are a developer with a large C/C++ application, and were wanting to know if moving from asm.js to Wasm is a great idea (which it is). All micro-benchmarks programs run with the same input, and we double-checked each case's generated output to prevent factors interfering with the results. Also, real applications and micro-benchmarks run in the same circumstances. In addition, each solution was performed five times, with median and mean values determined for each. This enabled us to reduce the number of uncontrolled system activities and software in the machine under test. Finally, the used energy measurement tool has been proven to be very accurate [26], [21], [22], [23], [24], [25], [4], [5].

Construct: We analyzed ten distinct micro-benchmarks scenarios across two languages, three browsers, each with three input sizes, totaling 180 different measured cases. These Wasm and JS solutions were produced using the Emscripten compiler tool, with the original C solutions coming from two widely known programming language sources. As a result, the algorithms are guaranteed to be similar, and there is no reason to believe that these solutions are better or worse than others. We also analyze two real applications: a Game-boy emulator called WasmBoy and a rendering and parsing of PDF documents, PSPDFKit. We measure the performance of six and five different Games and PDFs in two languages across three browsers, totaling 66 distinct solutions.

External: This threat is related to the generalization of the findings. Since Wasm allows C/C++, Rust or TypeScript developers to build their applications and we didn't measure the performance of WASM code generated from other languages for the micro-benchmarks, we don't know if the code written in all these languages is equally efficient when compiled to WASM. The new Wasm language has only been around for four years at the time of our research. Thus it is still in its infancy, with a lot of room for growth. However, we show that Wasm already outperforms JS in terms of energy and run-time performance. Consequently, the findings may not be entirely stable and may change during the early stages of development. Nonetheless, given the development team behind this language (W3C, Mozilla, Microsoft, Google, and Apple), and one of the primary goals being performance, we expect a continued improvement. Thus the performance differences we have observed in this study to be further highlighted and distanced.

VI. Related Work

JavaScript has been the most widely used scripting language for the last two decades, however, its supremacy is being challenged with the release of Wasm in 2017.

Given the more widespread adoption of Wasm, researchers have been studying it increasingly, such as its binary security [32], on hardening Wasm against Spectre attacks [33], and how to speed it up with Dynamic Linking [34].

Researchers compared the speed of JavaScript with C++ using standard searching and sorting algorithms revealed that JavaScript suffers when more processing power is required. Other study compares JavaScript's efficiency using numerical computations with native code, demonstrating that it can be run-time efficient in certain scenarios [7]. However, these studies only looked at run-time performance and did not compare JavaScript to a direct replacement language [35].

Similar evaluations of Wasm performance have been conducted. Wasm's own developers conducted early run-time performance tests, however these were tiny performance micro-benchmarks and file size comparisons with JavaScript [8]. Researchers also conducted a large-scale comparison of Wasm's run-time speed against native code in two distinct browsers (Chrome and Firefox). Native code is significantly more run-time efficient, according to their findings (as expected). Surprisingly, Wasm behaved significantly differently in the two browsers, with Mozilla being far more inefficient [36]. The performance of JavaScript and Wasm on both the client-side Web browsers and the server-side Node.js was already studied but only using numerical computing [37].

Previous studies already looked at Wasm and its run-time performance, but they didn't take into account its energy efficiency and didn't measure Wasm performance using real applications [8] .

Finally, various researchers have investigated the performance of several programming languages in various contexts, including mobile [6], desktop and server [23], [38], [3], [4], [5], and embedded systems [39], [40], in terms of both run-time and energy efficiency. None of these studies, however, took into account the Wasm language's run-time and energy efficiency.

VII. Conclusions and Future Directions

This paper presented a study a on the energy efficiency and run-time performance between the Web's primary language, JavaScript, and its newer and promising competitor, Wasm. We considered two real applications: a Gameboy console emulator, *WasmBoy*, and a portable document format (pdf) viewer/editor,

PSPDFKit. We also monitored the energy consumed and execution time of 10 micro-benchmarks, executed with three different input sizes. We executed all benchmarks in three popular Web browsers: Google Chrome, Mozilla Firefox, and Microsoft Edge.

Our findings show that Wasm performance differs when we consider the real-world benchmarks and micro-benchmarks. While JS can be, in some cases, more energy-efficient and faster than Wasm in micro-benchmarks, in real applications, Wasm outperforms JS with a significant difference. Using micro-benchmarks, Wasm is more energy and run-time efficient than JS in Google Chrome and Microsoft Edge. In Mozilla Firefox, JS has better performance results than Wasm, with a significant difference most of the time. For real applications, however, Wasm outperforms JS in all cases with a significant difference. With WasmBoy solutions, Wasm can be faster and, even so, use less power, which means using less energy per second. Thus, we can say that Wasm, in its proper environment (Web browsers), is greener and faster than JS for a significant margin.

As future work, we plan to extend our study to include other Web-based applications while also studying memory usage alongside energy consumption and run-time execution. Finally, our benchmarking framework is open source[8] for researchers and practitioners to replicate and build upon.

ACKNOWLEDGEMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT -Fundação para a Ciência e a Tecnologia within project UIDB/50014/2020. Additionally, this paper acknowledges the support of the Erasmus+ Key Action 2 project No. 2020-1-PT01-KA203-078646: "SusTrainable - Promoting Sustainability as a Fundamental Driver in Software Development Training and Education".

REFERENCES

[1] M. Butkiewicz, H. V. Madhyastha, and V. Sekar, "Characterizing web page complexity and its impact," *IEEE/ACM Transactions on Networking*, vol. 22, no. 3, pp. 943–956, 2013.

[2] G. Paolini, "Netscape and sun announce javascript, the open cross-platform object scripting language for enterprise networks and the internet," *Press Release]. Sun Microsystems, Inc*, 1994.

[3] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. a. Saraiva, "Towards a green ranking for programming languages," in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ser. SBLP 2017. New York, NY, USA: Association for Computing Machinery, 2017.

[4] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *SLE 2017 - Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2017*. New York, New York, USA: Association for Computing Machinery, Inc, oct 2017, pp. 256–267.

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

[6] 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)*. IEEE, 2017, pp. 42–52.

[7] F. Khan, V. Foley-Bourgon, S. Kathrotia, E. Lavoie, and L. Hendren, "Using JavaScript and WebCL for numerical computations: A comparative study of native and web technologies," *ACM SIGPLAN Notices*, vol. 50, no. 2, pp. 91–102, 2015.

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

[9] W3C. (2019) World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation. [Online]. Available: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en

[10] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290390

[11] G. Pinto and F. Castor, "Energy efficiency: a new concern for application software developers," *Communications of the ACM*, vol. 60, no. 12, pp. 68–75, 2017.

[12] A. Turner. (2018) Webassembly is fast: A real-world benchmark of webassembly vs. es6. [Online]. Available: https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193

[13] P. Spiess and G. Gurgone. (2018) A real-world webassembly benchmark. [Online]. Available: https://pspdfkit.com/blog/2018/a-real-world-webassembly-benchmark/

[14] D. Bryant. (2017) Why webassembly is a game changer for the web — and a source of pride for mozilla and firefox. [Online]. Available: https://medium.com/mozilla-tech/why-webassembly-is-a-game-changer-for-the-web-and-a-source-of-pride-for-mozilla-and-firefox-dda80e4c43cb

[15] W. Contributors. (2021) Webassembly use cases. [Online]. Available: https://webassembly.org/docs/use-cases/

[16] L.Technologies. (2021) Cheerp — the enterprise-grade c/c++ compiler for the web. [Online]. Available: https://leaningtech.com/pages/cheerp.html

[17] A. Zakai, "Emscripten: An LLVM-to-JavaScript compiler," *SPLASH'11 Compilation - Proceedings of OOPSLA'11, Onward! 2011, GPCE'11, DLS'11, and SPLASH'11 Companion*, pp. 301–312, 2011.

[18] AssemblyScript. (2021) Assembly script - a typescript-like language for webassembly. [Online]. Available: https://www.assemblyscript.org/

[19] S. Cheng, G. Karachalias, and H. Hoeglund, "Asterius: bringing haskell to webassembly," in *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming*, unpublished.

[20] J. de Macedo, R. Pereira, J. Saraiva, and R. Abreu, "On the runtime and energy performance of webassembly: Is webassembly superior to javascript yet?" in press.

[21] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto, "Recommending energy-efficient java collections," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 160–170.

[22] R. Pereira, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "The influence of the java collection framework on overall energy consumption," in *2016 IEEE/ACM 5th International Workshop on Green and Sustainable Software (GREENS)*, 2016, pp. 15–21.

[23] L. G. Lima, G. Melfe, F. Soares-Neto, P. Lieuthier, J. P. Fernandes, and F. Castor, "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language," in *Proc. of the 23rd IEEE Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER'2016)*. IEEE, 2016, pp. 517–528.

[24] 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.

[25] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "Spelling out energy leaks: Aiding developers locate energy inefficient code," *Journal of Systems and Software*, vol. 161, p. 110463, 2020.

[26] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using RAPL," *SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.

[27] R. V. Hogg, E. A. Tanis, and D. L. Zimmerman, *Probability and statistical inference*. Pearson/Prentice Hall Upper Saddle River, NJ, USA:, 2010.

[28] A. Field, *Discovering statistics using SPSS*. Sage publications, 2009.

[29] R. Rosenthal, "Meta-analytic procedures for social research. 1984, beverly hills," 1991.
[30] R. Rosenthal and R. L. Rosnow, *Essentials of behavioral research: Methods and data analysis*, 2008.
[31] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.
[32] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 217–234.
[33] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen *et al.*, "Swivel: Hardening webassembly against spectre," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.
[34] N. Mäkitalo, V. Bankowski, P. Daubaris, R. Mikkola, O. Beletski, and T. Mikkonen, "Bringing webassembly up to speed with dynamic linking," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1727–1735.
[35] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: An empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 61–72.
[36] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of webassembly vs. native code," in *2019 {USENIX} Annual Technical Conference (USENIX ATC'19)*, 2019, pp. 107–120.
[37] 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.
[38] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, "On haskell and energy efficiency," *Journal of Systems and Software*, vol. 149, pp. 554–580, 2019.
[39] S. Georgiou, M. Kechagia, P. Louridas, and D. Spinellis, "What are your programming language's energy-delay implications?" in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 303–313.
[40] S. Georgiou and D. Spinellis, "Energy-delay investigation of remote inter-process communication technologies," *Journal of Systems and Software*, vol. 162, p. 110506, 2020.