On the Runtime and Energy Performance of WebAssembly

Is WebAssembly superior to JavaScript yet?

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

2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW) | 978-1-6654-3583-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASEW52652.2021.00056

***Abstract*—In the early days of the world wide web, browsers were developed to navigate through (static) HTML web page documents. This has changed dramatically, and nowadays web pages are dynamic, expressed by programs written in regular programming languages. As a result, browsers are almost operating systems, having to interpret/compile such programs and execute them within the browser itself.**

Currently, while JavaScript is the main de facto language to express web pages, it does have various short comings and performance inefficiencies. WebAssembly, a new portable and size/load efficient alternative developed by major IT powerhouses, is seen as the future substitute. As WebAssembly aims to be more performance efficient than JavaScript, we aim to look at this current status and present a preliminary study on the performance of these two, based on their runtime and energy efficiency. Preliminary results show that WebAssembly, while still in its infancy, is starting to already challenge JavaScript, with much more room to grow. Additionally, our benchmarking framework is also made available to allow further research and replication.

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

I. Introduction

The Internet is one of the 20th century's most groundbreaking inventions. However, it was already in this century, with the advent of the mobile smartphone and its widespread usage, that ordinary people noticed its impact: *everyone* uses a computer and/or smartphone to perform everyday tasks, such as viewing emails, browsing news, chatting with friends and playing games. Most of these tasks are performed via web browsers: the most widely used software tool to access internet [1]. While in the very beginning, web browsers were used to navigate via static web (HTML) documents, soon dynamic features were included in web pages to make them more expressive. In fact, already in 1995, Netscape and Sun reported JavaScript (JS) as an "easy-to-use object scripting language designed for creating live online applications that link together objects and resources on both clients and servers". Thus, JavaScript became the *de facto* standard client-side web scripting language for over 20 years. As a consequence, browsers need to co-evolve to support the dynamic behaviour of web pages expressed by JavaScript programs. In fact, browsers are now very much like operating systems: they need to read/parse web documents with embedded JavaScript programs, which define their behaviour, and to interpret/compile/execute such programs to provide the desired dynamic behaviour. Because JavaScript source code is on the web (downloaded together with the static part of the webpage), its security and efficiency are of major concern. Internet attacks are often performed by injecting malicious code into the JavaScript component of the webpage being downloaded/executed [2].

Although JavaScript technology has improved by using advanced Virtual Machines (VM) offering both Just-In-Time (JIT) compilation and GPU support, JavaScript is also known to have poor performance as shown in recent surveys on the performance of 27 programming languages implementing the same 10 software problems: JavaScript is in position 15 in the reported ranking and 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]–[5]. For numerical computations JavaScript is within a factor of 2 of C [6].

To overcome the limitations of JavaScript, software developers from the four major browsers designed WebAssembly (Wasm) as a fast, safe, portable low-level bytecode tailored for Web-based applications [7]. Like many bytecode formats, WebAssembly aims to be the compilation target for the Web, which like the code produced by a C compiler is highly optimized at code generation time. By being a binary format, Wasm is transmitted over the network much faster than JavaScript and thus reducing load times. Moreover, Wasm is designed such that function bodies are placed after all declarations, thus allowing browsers to minimize pageload latency by starting streaming compilation as soon as functions arrive over the wire. This feature also allows parallel function body decoding and compilation.

As a consequence of this modern design, when a C program is compiled to Wasm instead of JavaScript, it runs 34% faster on the Google Chrome browser [7]. Although speed is a key aspect of an application, the widely use of powerful mobile computing devices is making the energy consumption another relevant aspect of software [8]. Unfortunately, there is no work on analysing the energy efficiency of browsers

Authorized licensed use limited to: University of Zagreb. Downloaded on January 18,2025 at 17:43:46 UTC from IEEE Xplore. Restrictions apply.

running Wasm applications. In this paper, we analyse the performance of Wasm, both in terms of it execution time and energy efficiency. Moreover, we compare the performance of 10 Wasm applications to their JavaScript counterparts. Thus, we aim to answer the following research questions:

RQ1: Is Wasm currently more run-time efficient than JS?
RQ2: Is Wasm currently more energy efficient than JS?
RQ3: Does performance scale differently for Wasm and JS depending on the input size?

Although WebAssembly is still in a very early phase, our preliminary results show that there is already a slight difference in performance, with WebAssembly being faster and more energy-efficient most of the time. In fact, Wasm is consistently more energetically efficient and faster than JS when programs are executed with small inputs. Our results also show that with larger inputs, Wasm programs are still faster and greener but the gap is minor.

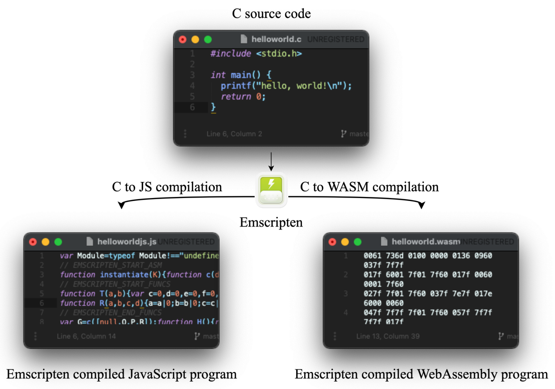
This paper is organized as follows: Section II presents WebAssembly and supporting tools/compilers. Section III details the methodology followed to design and execute our benchmarks and preliminary study. In Section IV we analyze and discuss in detail the performance of Wasm and JS benchmark programs. We discuss the threats to validity of our study in Section V. Finally, we present the related work in Section VI and our conclusions in Section VII.

II. WebAssembly's Ecosystem

WebAssembly is the result of a pioneer collaboration between software engineers from the four most widely used Web browsers, namely Chrome, Edge, Firefox and Safari. WebAssembly was developed with two main goals: speed and safety [7]. By being a binary/bytecode format, Wasm is design as a target format[1] . Wasm programs are typically produced by compiling existing programs written in high level programming languages (such as C/C++, Rust and Haskell) to Wasm binaries. Very much like a binary program produced by a C compiler, the produced Wasm program is highly optimized at compilation time. Such an optimized binary program offers other web application features: compact format (saving bandwidth), reduced load time, and streaming and parallel decoding/compilation.

There are several compiler frameworks that already target Wasm: Cheerp[2] is an enterprise C/C++ compiler for the Web, based and integrated into the LLVM/Clang infrastructure. It allows compiling virtually any C/C++ code to Wasm, JavaScript, and asm.js. Emscripten[3] is a complete open source compiler toolchain for WebAssembly. Using Emscripten it is possible to compile C/C++ code, or any other language that uses LLVM, into Wasm, and run it on the Web, Node.js, or other Wasm runtimes. Another WebAssembly compiler is Asterius[4]: a Haskell to WebAssembly compiler based on GHC. It compiles Haskell source code to WebAssembly and JavaScript code, which can be executed in Node.js or Web browsers. Figure 1 shows a simple example of Emscripten compiling the well-know *"hello world"* C program into JavaScript (left) and Wasm (right).

[1]Wasm also includes a human understandable textual format (WAT) to help program comprehension and debugging.
[2]Cheerp: https://leaningtech.com/cheerp/
[3]Emscripten: https://emscripten.org/
[4]Asterius: https://github.com/tweag/asterius


C source code
helloworld.c UNREGISTERED
Line 6, Column 2
C to JS compilation
C to WASM compilation
Emscripten
helloworldjs.js UNREGISTERED
Line 6, Column 14
helloworld.wasm UNREGISTERED
Line 13, Column 39
Emscripten compiled JavaScript program
Emscripten compiled WebAssembly program

```c
#include <stdio.h>

int main() {
  printf("hello, world!\n");
  return 0;
}
```

```javascript
var Module=typeof Module!=="undefine
// EMSCRIPTEN_START_ASM
function instantiate(K){function c(d
// EMSCRIPTEN_START_FUNCS
function T(a,b){var c=0,d=0,e=0,f=0,
function R(a,b,c,d){a=a|0;b=b|0;c=c|
// EMSCRIPTEN_END_FUNCS
var G=c([null,O,P,R]);function H(){r
```

```
0061 736d 0100 0000 0136 0960
037f 7f7f
017f 6001 7f01 7f60 017f 0060
0001 7f60
027f 7f01 7f60 037f 7e7f 017e
6000 0060
047f 7f7f 7f01 7f60 057f 7f7f
7f7f 017f
```

Fig. 1. Emscripten: compiling C to JavaScript and Wasm.

III. Benchmark Design and Execution

WebAssembly was recently designed to bring the Web up to speed [7]. Although the language and its supporting tools, namely compilers and virtual machines, are still very novel, it is important to assess how they are keeping up with their promise. Thus, the primary motivation of this work is to understand how Wasm *currently* compares to JavaScript, both in terms of energy consumption and run-time efficiency.

To generate equivalent Wasm and JavaScript programs, we need a supported source program. Thus, we used C programs for our solutions, which will also be included in this performance analysis so that we can compare Wasm with native code, as studies have shown it to be the golden standard for programming language efficiency [4], [5].

Our benchmarking framework consists of four steps: 1) Embedding input data, 2) Compilation to Wasm/JS, 3) Energy and run-time measuring, and 4) Data Collection. The following sub-sections will describe each of these steps, beginning with a description of our chosen benchmark problems and solutions. Additionally, our benchmarking framework is publicly available[5] for both researchers and practitioners to both replicate and build upon. Finally, Figure 2 presents the overview of our benchmarking framework.

A. Benchmark Programs

WebAssembly was designed to be used in compute-intensive cases such as compression, encryption, image processing, games, and numeric computations. For this preliminary study, we focused on computational heavy operations where performance is a concern.

One such operation was `sorting`, which we obtained a total of 8 different sorting algorithm solutions from Rosetta

[5]Github page: https://github.com/greensoftwarelab/WasmBenchmarks

Authorized licensed use limited to: University of Zagreb. Downloaded on January 18,2025 at 17:43:46 UTC from IEEE Xplore. Restrictions apply.

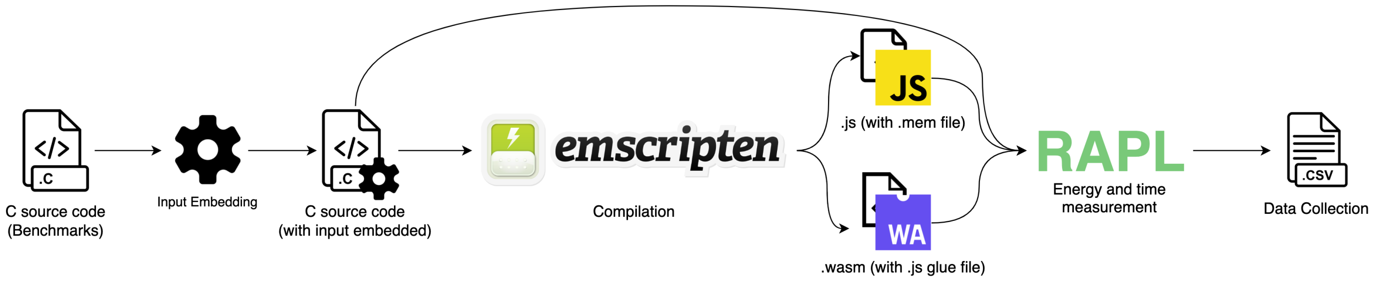

C source code (Benchmarks)
.C
Input Embedding
C source code (with input embedded)
emscripten
Compilation
JS
.js (with .mem file)
WA
.wasm (with .js glue file)
RAPL
Energy and time measurement
.csv
Data Collection

Fig. 2. Benchmark framework overview

Code[6]. Rosetta Code is a programming chrestomathy repository that presents solutions to over a thousand programming tasks in as many different languages as possible. Additionally, we also included two Wasm compatible[7] intensive benchmark problems from the Computer Language Benchmarks Game (CLBG): `fannkuch-redux` and `fasta`. The CLBG[8] is a website competition aimed at comparing the performance of several programming languages, with heavily optimized solutions written by corresponding language experts. The collected source code of all solutions were written in the C language, to be then compiled into their respective WebAssembly and JavaScript versions in a future stage.

Both Rosetta Code and CLBG have been previously used for comparing the performance of programming languages and/or analyze their energy efficiency [3]–[5], [9]–[14]. Finally, shown in Table I is the list of our benchmarks and their brief description, totalling 10 unique solutions.

B. Embedded Inputs

The performance of a program can vary depending on its complexity and the effort required for its execution. Thus, in our preliminary study, we have categorized three sizes of input data for each benchmark: `Small`, `Medium`, and `Large`. While the input size and data varies between the different benchmarks, they are consistent between the three languages under test (C, Wasm, and JS) within a given benchmark.

For each of the 8 sorting benchmarks, the inputs were randomly generated unsorted lists of values. The size of `Large` inputs was chosen so that they could be processed by all three languages without running out of memory. Our `Medium` input was half the size of our `Large`, and the `Small` input was half the size of our `Medium`. The `Large` size varied across the different benchmarks. For the remainder 2 benchmarks from CLBG, we based the sizes off their competition's input sizes, with small modifications applied when needed to execute with no errors, and have sizeable (yet not overly sizeable) inputs. Full details on the input sizes/data for all benchmarks can be seen on the framework's GitHub page.

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

[7]Most of the benchmark problems used base libraries which are yet not compatible with Wasm.

[8]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 OIDs, 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.

Emscripten, while an LLVM based tool, does not currently support a few libraries that traditional LLVMs do, such as those for defining file input/output for a *normal commandline experience*. As a workaround, instead of passing input/datasets during execution, all the input datasets, for all languages, are in a C header file (where more inputs can be easily added). When compiling the benchmarks to Wasm, JavaScript, or C, macros are used to specify the size of the input data, which will be compiled directly into the program. An example of such a header file is shown in Listing 1.

```
#ifdef SMALL_UOList
    #define INPUT {5,52,..,21} //of size 250
#endif
#ifdef MEDIUM_UOList
    #define INPUT {929,124,...,491} //of size 500
#endif
  ...
```

Listing 1. Example input header file.

C. Compiling to WebAssembly and JavaScript

In order to compile our C based benchmarks into the respective Wasm and JavaScript versions, we used Emscripten

Authorized licensed use limited to: University of Zagreb. Downloaded on January 18,2025 at 17:43:46 UTC from IEEE Xplore. Restrictions apply.

for the compilation process. We chose Emscripten as it is open source, already used by researchers and practitioners [7], [15], [16], and offers extensive documentation[9].

When Emscripten compiles a C program to Wasm, it creates two files, .wasm and .js, that work together. The .wasm file contains the translated code from the C benchmark, and the .js file (denominated as glue code) is the main target of compilation that will load and set up the Wasm code. Similarly, compiling to JavaScript creates the .js file containing the translated code, and creates a .mem file containing the static memory initialization data.

Currently, a `makefile` automatically compiles each benchmark solution (in C, Wasm, and JS) with each one of the corresponding input sizes. Each `makefile` contains a series of `command` variables containing the compilation and execution string for the corresponding language and input size. However, it is trivial to add new or modify input and testing scenarios within our benchmarking framework. After fully compiling the program artefacts for each benchmark, each compiled program was executed and verified to produce the correct output/result.

Finally, considering our 3 input sizes (defined in the previous sub-section), with our 3 languages (C, Wasm, and JS), across the 10 benchmarking problems, we have a final total of 90 unique compiled programs which will be analyzed.

D. Measuring Energy and Run-time

To accurately measure the energy consumption of each benchmark solution, we used Intel's Running Average Power Limit (RAPL)[10] to measure the energy consumed by the system's Package, CPU, GPU, and DRAM in Joules. RAPL has already been shown to provide very accurate energy measurements [17], high sampling rate (10ms), and has been used in several studies on energy consumption and software [4], [5], [10], [18]–[21].

As to guarantee no register overflow[11] occurs when measuring with RAPL, a C based thread is executed alongside the benchmark execution, continuously sampling the energy consumption using RAPL. Additionally, this thread also registers the start and end run-time of the executed benchmark. In order to collect consistent data and reduce effects from cold starts, warm-ups, and cache effects, each benchmark was executed twenty times [22], with a sleep of five seconds between each execution.

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.

E. Data Collection

The final step of our benchmarking framework is the data collection. We have created a Python script, *cleanresults.py*, that automatically aggregates all the RAPL energy samples per benchmark-input-language-execution. The final result is a CSV file for each benchmark-size pair containing the results of all three languages, with their RAPL samplings combined. Each CSV presents the results for each execution run and final results of our measured metrics (both median and mean): Package (Joules), CPU (Joules), DRAM (Joules), GPU (Joules) and Time (Seconds).

[9]MDN Emscripten: https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_wasm

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

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

IV. Analysis and Discussion

This section presents the results collected by our preliminary study. The main focus is to understand if WebAssembly is already outperforming JavaScript when it comes both to energy consumption and run-time execution, considering that WebAssembly is still in a very early phase.

A. Results

Shown in Figure 3 are the results collected from the ten benchmarks executed in our study. Each individual chart represents one of the specific benchmarks. In each chart, the results are ordered (from left to right) by the input size of `Small`, `Medium`, and `Large`, and within each size are the three languages.

The left most axis represents the amount of energy consumed (Joules) by the CPU (blue bars) and DRAM (green bars). The right most axis represents the run-time (Seconds), and corresponds to the orange line. Finally, the red dots represent the ratio between the energy consumed (here we are considering the sum of CPU and DRAM) and the time spent. In essence, this ratio can be seen as the average power (Watts) used. In terms of both energy and run-time efficiency, the lower the bars and orange line, the more efficient.

Figure 4 is a Heat map that shows the proportion between JavaScript and Wasm. The left axis represents the ten benchmarks and, in the last row, is the average of the above values. All types of results obtained are on the upper axis, including the tree input sizes, the Energy (Joules) and Time (ms) values, with the color scale shown on the right. If a value is greater than 1, it means that Wasm is better, in other words, more efficient. For example, for the `Fannkuch-redux` row and `Small Time` column, Wasm was shown to be 1.173x more efficient.

Finally, available in the benchmark's online page[5] are also a set of violin plots for each benchmark and each metric. This allows us to display the entire density of our collected data, for both energy consumption and run-time, including outliers, median, and quartiles. These plots allow us to understand whether a language is consistent or has very inconsistent performances.

B. Discussion

The main focus of this study is to compare the energy and run-time performance between Wasm and JavaScript. As we needed C source code to generate equivalent Wasm and JavaScript programs, we included it as a reference point to

Authorized licensed use limited to: University of Zagreb. Downloaded on January 18,2025 at 17:43:46 UTC from IEEE Xplore. Restrictions apply.

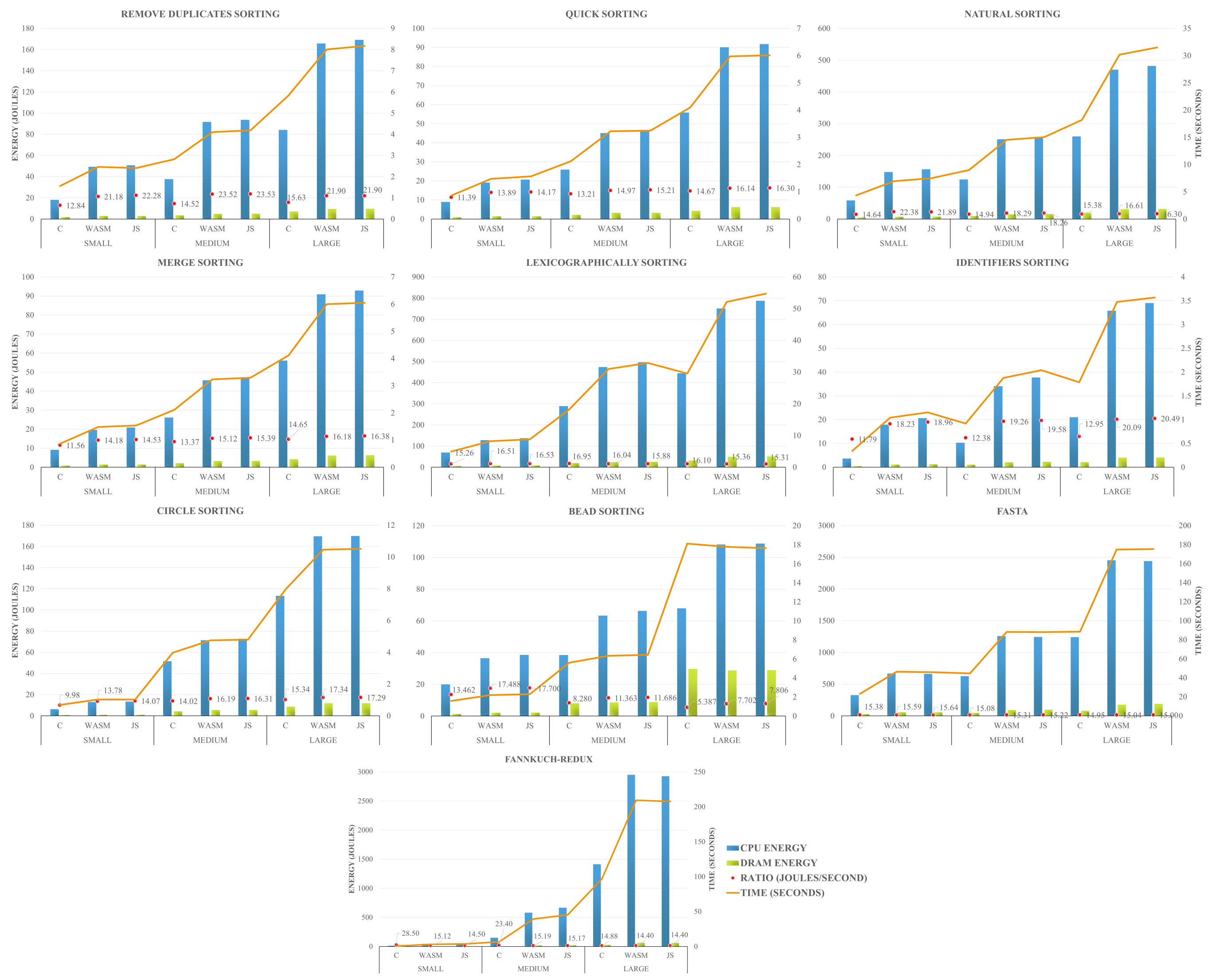


Fig. 3. Energy consumed by the CPU and DRAM for each benchmark with the three input sizes and the respective execution time.

compare the performance of Wasm and JavaScript to one of the most run-time and energy efficient languages [4].

As expected, C is by far more efficient than Wasm and JavaScript, both in terms of energy and run-time performance. The only exception is in the `Bead Sorting` benchmark, where the execution time ends up being higher than Wasm and JavaScript with a `Large` input. However, its energy consumption is significantly lower. Another essential detail of the C programming language performance is that when the input size increases, the performance difference between C and the other two languages (Wasm and JS) decreases. This may be due to *emcc* optimization flags, similar to *gcc, clang*, and other compilers, which further apply additional optimization specifically for WebAssembly.

The results of Wasm and JavaScript are very similar, both energy efficient and run-time performance. The smallest and largest performance difference between them is 0.959x and 1.173x, respectively. Nevertheless, if we look closely, it is possible to notice several differences.

Beginning with CPU energy consumption, in all thirty tests (ten benchmarks, each one with three different inputs), Wasm is more energetically efficient than JavaScript across twenty-six cases (87%). It is less efficient in the cases of `Small` input `Fasta`, `Medium` input `Fasta`, and twice with `Large` input `Fasta and Fannkuch-redux`. The same happens with DRAM consumption, but not in the same benchmarks, which may be due to different algorithms using more or less memory. With DRAM, it happens two times with `Small` input `Sorting-circle` and `Fasta`, and with `Large` input `Sorting-quick` and `Fannkuch-redux`. With Medium input, Wasm is always more efficient **ARQ2**. Wasm is faster in twenty-five of thirty cases when it comes to run-time performance **ARQ1**. By observing the ratio values (or, in other words, the average Power in kW), Wasm only uses more Power in ten cases. It is a consequence of taking less time to execute the programs. Even so, it is more efficient two-thirds of the

Authorized licensed use limited to: University of Zagreb. Downloaded on January 18,2025 at 17:43:46 UTC from IEEE Xplore. Restrictions apply.

time. WebAssembly's compact code format and its low-level nature means that it can load, parse, and compile the code faster than JavaScript.

Figure 4 presents how efficient each benchmark was for a given performance metric and within each benchmarked case. For example, in the `Fannkuch-redux` benchmark, Wasm had both its best and worst performance when compared to JavaScript. Using the `Small` and `Medium` inputs, Wasm presented the results but, when looking at the `Large` input, Wasm had the worst results across all thirty tests. The worst benchmark for Wasm was the `Fasta` benchmark, where JavaScript was more energetically efficient with the three inputs, and faster with `Small` and `Medium` input. Thus, the results show that currently Wasm is in general both more run-time (**ARQ1**), and energy efficient (**ARQ2**).

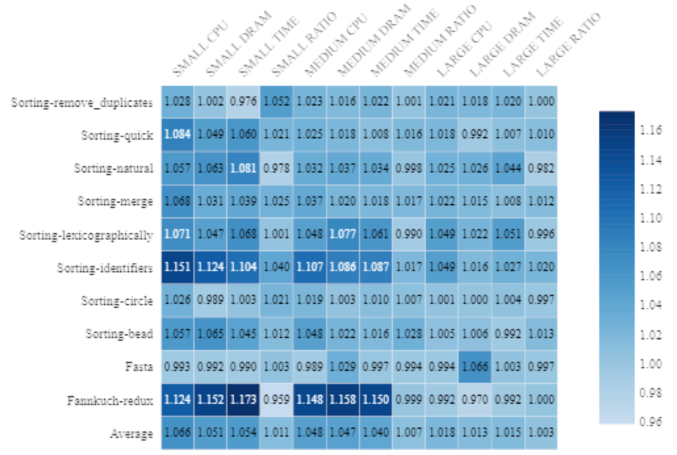
When comparing the three inputs, is clear that Wasm is more energetically efficient and faster than JavaScript with smaller inputs. However, with larger inputs, Wasm does continue to be more energy and run-time efficient but with a much smaller difference. Additionally, while the performance gap between Wasm and JavaScript ends up being narrower with larger inputs, all three languages tend to maintain the same pattern ranking and ratio between each regardless of input scale within each individual case (**ARQ3**). The smaller difference (with larger inputs) for Wasm and JavaScript may be due to the Node.js virtual machine, which applies dynamic optimizations to improve the JavaScript speed, particularly the JIT compilation. Note that JIT optimization is more aggressive when the engine identifies hot loops. By contrast, the current WebAssembly virtual machine is very recent, thus optimization techniques for WebAssembly may not be mature yet.

Finally, we can say, looking at the last row (`Average row`) of Figure 4, that on average, Wasm is always more energetically efficient and faster than JavaScript (further repeating **ARQ1/ARQ2**). Considering the average of all tests and all inputs, Wasm is 1.044x more CPU energy efficient than JavaScript, 1.037x more DRAM energy efficient, and 1.036x more run-time efficient. While the results seem small, as 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 may over time far surpass JavaScript.

V. Threats to Validity

The goal of this preliminary study is to both measure and understand the energetic and run-time behaviour between the novel Wasm and matured JavaScript languages. This section presents some threats to the validity of our study, separated into four categories [23].

Conclusion: While the difference between Wasm and JavaScript is small, there is a clear consistency in favor of Wasm. However, analyzing the impact of other hardware components (such as memory usage) deserve further analysis. Finally, while we mainly focused on sorting based benchmarks, further benchmarks should also be considered. However, all our data and benchmarking framework is made available and can be very easily extended to include further benchmarks.



	SMALL CPU	SMALL DRAM	SMALL TIME	SMALL RATIO	MEDIUM CPU	MEDIUM DRAM	MEDIUM TIME	MEDIUM RATIO	LARGE CPU	LARGE DRAM	LARGE TIME	LARGE RATIO
Sorting-remove_duplicates	1.028	1.002	0.976	1.052	1.023	1.016	1.022	1.001	1.021	1.018	1.020	1.000
Sorting-quick	1.084	1.049	1.060	1.021	1.025	1.018	1.008	1.016	1.018	0.992	1.007	1.010
Sorting-natural	1.057	1.063	1.081	0.978	1.032	1.037	1.034	0.998	1.025	1.026	1.044	0.982
Sorting-merge	1.068	1.031	1.039	1.025	1.037	1.020	1.018	1.017	1.022	1.015	1.008	1.012
Sorting-lexicographically	1.071	1.047	1.068	1.001	1.048	1.077	1.061	0.990	1.049	1.022	1.051	0.996
Sorting-identifiers	1.151	1.124	1.104	1.040	1.107	1.086	1.087	1.017	1.049	1.016	1.027	1.020
Sorting-circle	1.026	0.989	1.003	1.021	1.019	1.003	1.010	1.007	1.001	1.000	1.004	0.997
Sorting-bead	1.057	1.065	1.045	1.012	1.048	1.022	1.016	1.028	1.005	1.006	0.992	1.013
Fasta	0.993	0.992	0.990	1.003	0.989	1.029	0.997	0.994	0.994	1.066	1.003	0.997
Fannkuch-redux	1.124	1.152	1.173	0.959	1.148	1.158	1.150	0.999	0.992	0.970	0.992	1.000
Average	1.066	1.051	1.054	1.011	1.048	1.047	1.040	1.007	1.018	1.013	1.015	1.003

Fig. 4. Heat map representing the proportion between WebAssembly and JavaScript results.

Internal: To avoid internal factors which may interfere with our results, all benchmarked scenarios execute in the same manner with the same input, to which we additionally verified the produced output of each case. In addition, measurements were repeated 20 times, to which we calculated both median and mean values, with each being executed with the recommended Emscripten flags and commands. This allowed us to minimize uncontrollable system processes and software within the tested machine. Finally, the used energy measurement tool has been proven to be very accurate [4], [5], [10], [17]–[21].

Construct: We analyzed 10 different benchmark scenarios across 3 languages, each with 3 input sizes, totalling 90 different measured cases. The original C solutions were obtained from two commonly used programming language repositories, with the Wasm and JavaScript solutions being generated by the Emscripten compiler tool. This guaranteed that the algorithms are identical, and there is no basis to suspect that these solutions are better or worse than any other.

External: This threat concerns itself with the generalization of the results. At the time of our study, the novel WebAssembly language only appeared 4 years prior. As such, it is still in its infancy, with a large possibility of further evolution. While there seems to be a slight favor in performance of Wasm over JavaScript, the differences are small. Thus, results might not be considered completely stable and may vary throughout its early evolution. Nevertheless, considering the development team behind this language (W3C, Mozilla, Microsoft, Google, and Apple), and one principal objective being performance, we expect a continued improvement, and thus the performance differences we have observed in this study to be further highlighted and distanced.

VI. Related Work

For the past two decades, JavaScript has been the most commonly supported scripting language. With the introduction

Authorized licensed use limited to: University of Zagreb. Downloaded on January 18,2025 at 17:43:46 UTC from IEEE Xplore. Restrictions apply.

of WebAssembly in 2017, its dominance is being tested.

There have been studies on analyzing comparing JavaScript's performance with numerical computations with that of native code [24], showing that in certain cases it can actually be run-time efficient. Other researchers have studied JavaScript performance issues through analyzing popular client-side and server-side projects [25]. They found that inefficient usage of APIs was the most leading cause for such problems, which could be solved with very minor optimizations.

Finally, a study on the performance of JavaScript versus C++, based on common searching and sorting algorithms, has been previously carried out showing how JavaScript falls off whenever greater processing power is needed [26]. However, these studies did not compare JavaScript performance with a direct substitute language, and only focused on run-time performance.

Similar studies analyzing WebAssembly performance have been carried out. An independent analysis looked at the performance between WebAssembly and JavaScript in the context of browser based gaming, specifically emulating a Gameboy [27]. The study showed that WebAssembly performed much better, in terms of run-time, than JavaScript. Additionally, its resource efficiency allowed the game to be played on budget mobile devices and Chromebooks. WebAssembly's own engineers have performed preliminary studies on run-time performance [7], but these were small performance micro-benchmarks, and file size comparison against JavaScript. Researchers have also carried out a large-scale analysis on the run-time performance of Wasm vs. native code [6] within two different browsers (Chrome and Firefox). Their results showed that native code is much more run-time efficient (as expected). Surprisingly enough, WebAssembly performed very differently across the two browsers, being much more inefficient with the former. While these studies looked at Wasm and its run-time performance, they did not consider its energy efficiency performance.

Finally, several researchers have looked at the performance, both run-time and energy efficiency, of several programming languages in contexts such as mobile [9], desktop and server [3]–[5], [10], [11], and embedded systems [12], [13]. However, none of these studies considered the run-time and energy efficiency of the WebAssembly language.

VII. Conclusions and Future Directions

In this paper, we present a preliminary study and its results on the run-time and energy efficiency performance between three languages: the web's primary language (JavaScript) its newer and promising competitor (WebAssembly), and a native language (C). We monitored the energy consumed and execution time of 10 computer programs, when executed with three different input sizes.

Unsurprisingly, C continues to be the fastest and greenest programming language. However, between WebAssembly and JavaScript, the results show that there is already a very slight difference in performance, with WebAssembly being quicker and more energy efficient in most cases, albeit not with any huge margins. The results also show that the bigger the program input size, the smaller the performance gap between WebAssembly and JavaScript.

In this study, we only executed the WebAssembly and JavaScript programs on Node.js. As future work, we plan to extend our study to include other computational intensive benchmarks and web based applications, while also extending the framework to allow measuring the performance within a browser based environment (i.e. Chrome, Firefox, and Edge), and with strong statistical tests. We also plan to study memory usage alongside energy consumption and run-time execution. Finally, our benchmarking framework is open source[5], 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] V. Anand and D. Saxena, "Comparative study of modern web browsers based on their performance and evolution," *2013 IEEE International Conference on Computational Intelligence and Computing Research, IEEE ICCIC 2013*, 2013.

[2] C. Yue and H. Wang, "Characterizing insecure javascript practices on the web," in *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, (New York, NY, USA), p. 961–970, Association for Computing Machinery, 2009.

[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*, 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 Int. Conference on Software Language Engineering, co-located with SPLASH 2017*, (New York, New York, USA), Association for Computing Machinery, Inc, oct 2017.

[5] 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.

[6] 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)*, pp. 107–120, 2019.

[7] 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*, PLDI 2017, (New York, NY, USA), p. 185–200, Association for Computing Machinery, 2017.

[8] 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.

[9] 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)*, pp. 42–52, IEEE, 2017.

Authorized licensed use limited to: University of Zagreb. Downloaded on January 18,2025 at 17:43:46 UTC from IEEE Xplore. Restrictions apply.

[10] 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)*, pp. 517–528, IEEE, 2016.

[11] 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, 2019.

[12] 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*, pp. 303–313, 2018.

[13] S. Georgiou and D. Spinellis, “Energy-delay investigation of remote inter-process communication technologies,” *Journal of Systems and Software*, vol. 162, p. 110506, 2020.

[14] S. Nanz and C. A. Furia, “A comparative study of programming languages in rosetta code,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 778–788, IEEE, 2015.

[15] 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.

[16] A. Zakai, “Fast physics on the web using c++, javascript, and emscripten,” *Computing in Science Engineering*, vol. 20, no. 1, pp. 11–19, 2018.

[17] 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.

[18] W. Júnior, R. dos Santos, F. de Lima, B. de Neto, and G. Pinto, “Recommending energy-efficient java collections,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 160–170, IEEE, 2019.

[19] 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)*, pp. 15–21, 2016.

[20] J. a. de Macedo, J. a. Aloísio, N. Gonçalves, R. Pereira, and J. a. Saraiva, “Energy wars - chrome vs. firefox: Which browser is more energy efficient?,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ASE ’20, (New York, NY, USA), p. 159–165, Association for Computing Machinery, 2020.

[21] 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.

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

[23] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-experimentation: Design & analysis issues for field settings*, vol. 351. Houghton Mifflin Boston, 1979.

[24] 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.

[25] K. Stefanoski, A. Karadimche, and I. Dimitrievski, “Performance Comparison of C++ and JavaScript (Node.js – V8 Engine),” no. September, 2019.

[26] M. Selakovic and M. Pradel, “Performance issues and optimizations in javascript: An empirical study,” in *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, (New York, NY, USA), Association for Computing Machinery, 2016.

[27] A. Turner, “Webassembly is fast: A real-world benchmark of webassembly vs. es6,” 2018.

Authorized licensed use limited to: University of Zagreb. Downloaded on January 18,2025 at 17:43:46 UTC from IEEE Xplore. Restrictions apply.