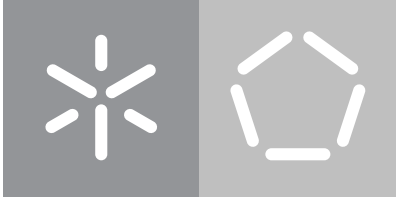
University of Minho
Engineering School

João Gonçalves De Macedo

On the performance of WebAssembly

February, 2022



University of Minho
Engineering School

João Gonçalves De Macedo

On the performance of WebAssembly

Master's Dissertation
Integrated Master's in Informatics Engineering

Work supervised by
João Alexande Baptista Vieira Saraiva
Rui Alexandre Afonso Pereira

February, 2022

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Acknowledgements

First of all, I wish to express my deep appreciation to my supervisor, Professor João Saraiva, who has the substance of a genius: he convincingly guided and supported me to be professional and do the right thing even when the road got tough. Without his relentless aid, my aim of finishing this thesis would not have been accomplished.

I would also like to thank my co-supervisors, Rui Pereira and Rui Maranhão, for their significant guidance throughout my dissertation. The help and advice they gave me - almost weekly -, were crucial to finish this journey with two written research papers and a lot more accomplished dissertation.

Aaron Turner, a person from across the planet that I met while developing this thesis and whose work helped me develop this dissertation. Thank you for your availability, for your dedication to helping me, for the meetings you made available to have with me, and for the advice.

My 38 fellows! What a journey! What memories! What stories! All the hours we spent teaching each other, all the hours spent in the 24 rooms, all the nights we stayed awake finishing our works. I would never be finishing this chapter of my life without you. Bolt and Korine, special thanks for helping me whenever I needed extra help doing this dissertation. Thank you all Bestas!

Finally, I must express my very profound gratitude to my parents and to my brother for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

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 dissertation, 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 real applications to have more realistic results. The 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 show 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. Our benchmarking framework is also available to allow further research and replication.

Keywords: Energy Efficiency, Green Software, Web Browsers, WebAssembly

Resumo

A Web evoluiu dramaticamente em todo o mundo nos últimos anos. As páginas Web são dinâmicas, expressas por programas escritos em linguagens de programação comuns, dando origem a aplicativos Web sofisticados. Assim, os navegadores Web são quase como sistemas operacionais, tendo que interpretar/compilar tais programas e executá-los. Embora o JavaScript seja amplamente usado para expressar páginas Web dinâmicas, ele tem várias deficiências e ineficiências de desempenho. Para superar tais limitações, as principais potências de TI estão a desenvolver uma nova linguagem portátil e eficiente em tamanho/carregamento: WebAssembly.

Nesta dissertação, conduzimos o primeiro estudo sistemático sobre o desempenho da energia e do tempo de execução do WebAssembly e JavaScript na Web. Usamos micro-benchmarks e aplicações reais para obter resultados mais realistas. Os resultados mostram que WebAssembly, embora ainda esteja na sua infância, já está começa a superar o JavaScript, com muito mais espaço para crescer. Uma análise estatística indica que WebAssembly produz diferenças de desempenho significativas em relação ao JavaScript. No entanto, essas diferenças diferem entre micro-benchmarks e benchmarks de aplicações reais. Os nossos resultados também mostram que o WebAssembly melhorou a eficiência energética em 30%, em média, e mostram como o comportamento do WebAssembly é diferente entre três navegadores Web populares: Google Chrome, Microsoft Edge e Mozilla Firefox. As nossas descobertas indicam que o WebAssembly é mais rápido que o JavaScript e ainda mais eficiente em termos de energia. A nossa benchmarking framework está disponível para permitir pesquisas adicionais e replicação.

Palavras-chave: Eficiência Energética, Navegadores Web, Software Verde, WebAssembly

Contents

List of Figures

List of Tables

Listings

Acronyms

CLBG	Computer Language Benchmarks Game
COVID-19	Coronavirus Disease 2019
CT-Wasm	Constant-Time WebAssembly
GPU	Graphics processing unit
GREENS	International Workshop on Green and Sustainable Software
ICT4S	International Conference on ICT for Sustainability
IT	Information technology
JIT	Just-In-Time
JS	JavaScript
NaCl	Native Client
PNaCl	Portable Native Client
RAPL	Running Average Power Limit
RE4SuSy	International Workshop on Requirements Engineering for Sustainable Systems
SANER	IEEE International Conference on Software Analysis, Evolution and Reengineering
SEISMIC	Cryptomining detector
SFI	Software fault isolation
SPL	Software Product Lines
SpMV	Sparse matrix-vector multiplication
SUSTAINSE	International Workshop on Sustainable Software Engineering
TS	TypeScript

VM	Virtual Machine
W3C	World Wide Web Consortium
Wasm	WebAssembly
WAT	WebAssembly Text Format

Chapter 1

Introduction

1.1 Context

Imagine a world where you could build software with C, C++, Rust, Python, Go, or even Cobol, then deliver that software to the end-user in a Web browser without any installation and achieving near-native performance. That would be cool, right? That world became a reality in December 2019, when WebAssembly became an official World Wide Web Consortium (W3C) standard (W3C, 2019). But first, let's go back in time.

The Internet is one of the 20^{th} century's most groundbreaking inventions. In this century, with the advent of the mobile smartphone and its widespread usage, those ordinary people noticed its impact: *everyone* uses a computer and 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 Anand and Saxena, 2013. While in the very beginning, people used web browsers to navigate via static web (HTML) documents, developers included dynamic features in web pages to make them more expressive.

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". JS has been the *de facto* standard client-side Web scripting language, for more than two decades (Paolini, 1994). Consequently, browsers need to co-evolve to support the dynamic behaviour of web pages expressed by JS programs. Browsers are now like operating systems: they need to read/parse web documents with embedded JS programs, which define their behaviour and interpret/compile/execute such programs to provide the desired dynamic behaviour. Because JS source code is on the web (downloaded together with the static part of the webpage), its security and efficiency are of significant concern. Internet attacks are often performed by injecting malicious code into the JS component of the webpage being downloaded/executed (Yue & Wang, 2009).

Although JS technology has improved by using advanced Virtual Machines (VM) offering both Just-In-Time (JIT) compilation and GPU support, JS is also known to have poor performance. Recent surveys show that the performance of 27 programming languages implementing the same 10 software problems: JS 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) (Couto, Pereira, et al., 2017; Pereira, Couto, Ribeiro, et al., 2017, 2021). For numerical computations, JS is within a factor of 2 of C (F. Khan et al., 2015).

As the only embedded language on the Web, JS falls short in efficiency and security, especially as a compilation target. Major IT powerhouses are developing a new portable and size/load efficient bytecode language: WebAssembly (abbreviated Wasm, pronounced waz-um) to overcome such limitations (Haas et al., 2017).

1.2 Motivation and Research Questions

According to W3C, Wasm is the fourth language for the Web, which allows code to run in the browser (W3C, 2019). The other three languages - HTML, CSS, and JS - were developed in the previous century, already! The introduction of Wasm is a vital contribution aiming at improving both the run-time performance (Haas et al., 2017) and the security of Web applications (Watt, Renner, et al., 2019a). 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 (Haas et al., 2017). However, being a pretty new language/system, it is essential to assess its impact on our daily websites and browsers fully. 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 (Pinto & Castor, 2017). Unfortunately, there is no work analysing in detail the energy consumption of Wasm applications when compared to an equivalent JS alternative.

In this paper, we present the first detailed study on the impact on the energy efficiency of Wasm. We consider two real-world Wasm applications developed with benchmark goals: the Game-boy console emulator (Turner, 2018) and the *PSPDFKit* portable document format (PDF) viewer/editor (Spiess & Gurgone, 2018). The Game-boy emulator was directly written in JS (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 JS (*asm.js*) and into Wasm. Thus, we can compare the JS and Wasm implementations of both benchmarks. Moreover, we also analyse the performance of ten Wasm/JS micro-benchmarks.

To get a better idea of how Wasm looks like, Figure 1.1 shows a simple example of Emscripten compiling a factorial program written in C into Wasm (left) and JS (right).

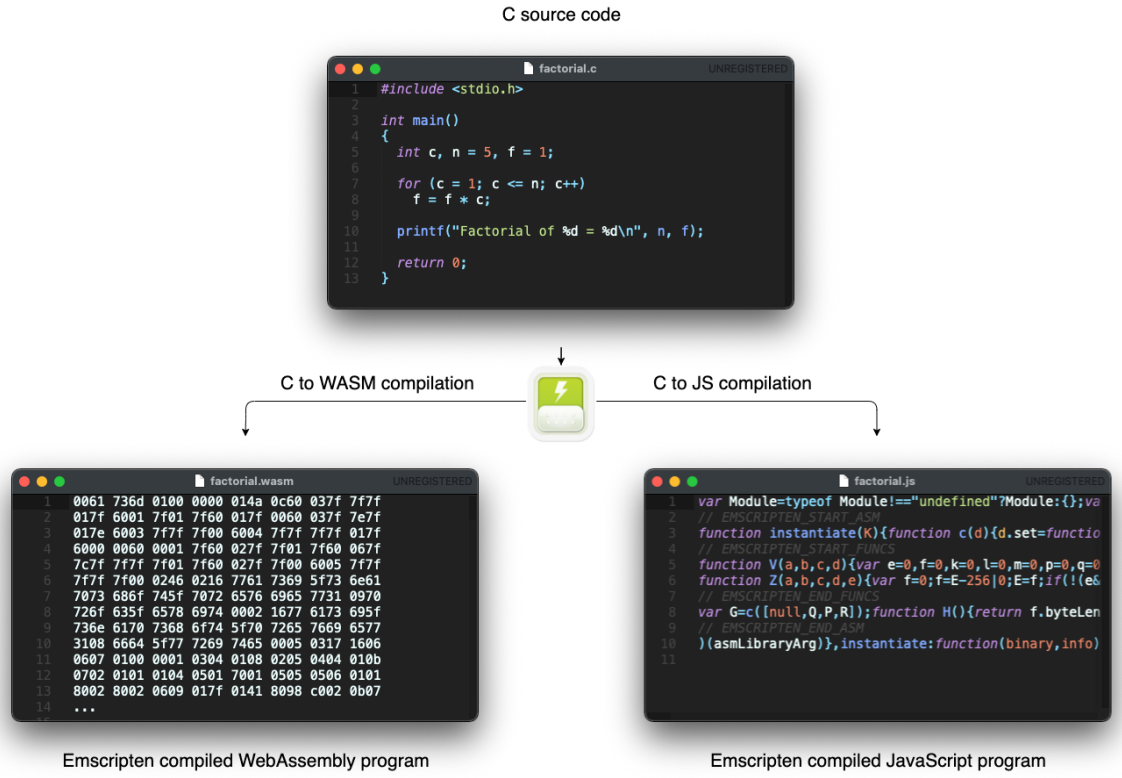


Figure 1.1: Emscripten: compiling C to Wasm and JS.

Although Wasm is mainly a generated language, it should also support the direct development of programs because it makes (slightly) easy for software developers to understand, debug, and evaluate the Wasm module. For this, there is a textual representation of the Wasm binary format, called WebAssembly Text Format (WAT) (Mozilla, 2021). Writing codes using the text format and compiling them to the binary format is possible, but the text format is only intended for last-resort debugging. Figure 1.2 corresponds to the textual format of the Wasm factorial program shown in Figure 1.1.

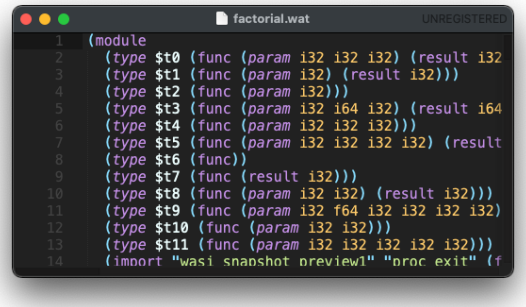


Figure 1.2: *factorial.wat*: equivalent textual format of *factorial.wasm*.

To translate these languages to each other exist a suite of tools called *The WebAssembly Binary Toolkit* that includes, among others tools, *wat2wasm* and *wasm2wat*. *Wat2wasm* translates from Wasm text format to the Wasm binary format and *wasm2wat* translates from the binary format back to the text format (also known as a .wat) (WebAssembly, 2015).

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, obsoleting existing JS solutions such as asm.js, it is crucial to assess whether it is also already more energy-efficient than the existing solutions. It is also essential 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?*

  There are several micro-benchmarks that test Wasm versus JS run-time performance (Haas et al., 2017). We want to consider such micro-benchmarks and larger real-world benchmarks to have more real and trustworthy performance measurements.

- **RQ4:** *Does Wasm present the same performance between different browsers?*

  Web browsers are the most widely used software tool to access internet. Since the developers of all major browsers are also involved in the creation of Wasm, it is important to study which browser offers the best performance when running Wasm bytecode.

1.3 Contributions

During the development of this dissertation, I have provided various contributions of scientific knowledge to the research area in which this thesis is involved, such as research publications and presentations. In addition to the contributions of this thesis, this section also lists additional contributions I have made during my Master's degree, which are related to the research area.

1.3.1 Talks

In the context of this thesis I delivered two research talks at international scientific events:

- *How Green is WebAssembly?*, talk at the CERCIRAS Workshop, Cost Action CA19135 , Novi Sad, Serbia, September 2nd 2021.

- *On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet?*, talk at SUSTAIN-SE Workshop, co-located with 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Australia (virtual event), November 21th 2021.

1.3.2 Research Papers

I am the main author of the following research papers in the context of Wasm energy and run-time performance using different benchmarks and real applications:

- De Macedo, J., Abreu, R., Pereira, R., & Saraiva, J. (2021, November). *On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet?*. Paper accepted at the 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW).

- De Macedo, J., Abreu, R., Pereira, R., & Saraiva, J. (2021, November). *WebAssembly: The Game Changer of Energy and Runtime Performance*. Paper submitted at the 29th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).

1.3.3 Others Research Papers

I am also the first author of the following research paper on green software:

- De Macedo, J., Aloísio, J., Gonçalves, N., Pereira, R., & Saraiva, J. (2020). Energy Wars - Chrome vs. Firefox: Which browser is more energy efficient?. *35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)* (pp. 159-165). doi: 10.1145/3417113.3423000.

1.4 Document Structure

This section explains how the dissertation is organized. All of the chapters begin with a quick overview of the topic that will be covered during the thesis. The structure of this dissertation is organized as follows:

- **Chapter 2 - State of the Art** focuses on two aspects: Wasm, and Green Software. It contains information on Wasm invention, its current exponential growth, explaining what it is and why it is so crucial to the Web's future. It also includes knowledge on green computing evolution and the emergence area of green software.

- **Chapter 3 - Benchmark Design and Execution** presents the methodology used. It shows which benchmarks were used, how we created the framework used to run all programs solutions, how we measured the results and how we collected them.

- **Chapter 4 - Analysis and Discussion** contains the results obtained in our research. It also includes graphics to help understand the results and explain why we reached the conclusions we

did. This chapter also includes the answers to our research questions and the threats to validity of our study.

- **Chapter 5 - Conclusions and Future Directions** is the last chapter of this dissertation. It contains our final considerations, a summary of the results obtained, as well as future work ideas to provide the study's continuity and progress.

Chapter 2

State of the Art

This chapter describes the state of the art of Wasm and Green Software. Firstly, we focus on the reasons why Wasm was invented and why this language can be so crucial to the Web's future. Furthermore, we describe Wasm, its primary concerns, and the studies already conducted on it. Secondly, we introduce Green IT, specifically Green Software, and the study's involving its research area. We also explore the works using the same tool used in this study to measure energy consumption, called RAPL.

2.1 WebAssembly's Ecosystem

Over the last decades, the Web has been a significant and powerful resource, it has transformed how we work, play, communicate, and socialize (Brügger, 2010; Leiner et al., 2009).

Since its appearance, in 1995, officially, Internet has become fundamental for the digital networked communicative infrastructure (Brügger, 2012; Castells & Chemla, 2001). Indeed, nowadays, even in the face of a global pandemic - Coronavirus Disease 2019 (COVID-19) - the Web has the potential to revolutionize the world. This virus forced us all to a lockdown, however, the world couldn't stop, leading to adjusting to the new laws and constraints. The urgent need to adapt to this new reality led, inevitably, to the rise in the use of digital technologies (De' et al., 2020): in education, for example, Internet proved to be an asset, particularly in distance education (Bergdahl & Nouri, 2021; Castaman & Rodrigues, 2020); in business, organizations tried to maintain their sales, even with the stores closed, through Web platforms, which allowed them to show their products, sell them and deliver them to customers' residences (Bhatti et al., 2020; Kim, 2020); in companies, working from home was the alternative adopted to deal with labor impositions related to COVID-19 using several Web applications to don't lose contact, having online meetings, keep contact with colleagues and work as a team (Bonacini et al., 2021; Purwanto et al., 2020).

The majority of these operations are performed by Web applications using Web browsers - the most frequently used software tools for accessing Internet -, they can do much more than rendering a Web

page (Butkiewicz et al., 2013; Laperdrix et al., 2016; H. J. Wang et al., 2007). 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.

To understand the rationale behind Wasm, and why it works the way it does, it helps to remember just how far JS has come. For more than two decades, the *de facto* standard client-side Web scripting language has been JS. According to Netscape and Sun, 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"(Paolini, 1994).

There is no doubt that JS is the King of Web development and the most widely used language among Web developers. It's also the only language that allows you to develop both *frontend* and *backend* Web applications, as well as mobile applications. The strength of JS is not just that it can run on both browser and server using *Node.js*, but also that it offers some frameworks and tools for Web and app development (Wirfs-Brock & Eich, 2020). Even though JS technology has improved by utilizing sophisticated Virtual Machines (VMs) providing both JIT compilation and Graphics processing unit (GPU) support, JS was not designed with performance in mind. Recent studies analysing 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) (Couto, Pereira, et al., 2017; Pereira, Couto, Ribeiro, et al., 2017, 2021). For numerical computations, JS is 2 times less efficient compared to C (F. Khan et al., 2015). Compared to C++, JS is not as fast, mainly because of source code run-time parsing, JIT compilation and its dynamic nature (Stefanoski et al., 2019).

Prior attempts at low-level code on the Web have fallen short of properties that a low-level compilation target should have (Haas et al., 2017). ActiveX was a Microsoft technology for code-signing x86 binaries to run on the Web. It was reliant on code signing, and as a result, it did not ensure safety through technological construction but rather through a trust paradigm (Microsoft, 2017).

Native Client (NaCl) was the first solution to use a Sandboxing technique on the Web to allow machine code to execute at near-native speeds (Ansel et al., 2011; Yee et al., 2009). It works by requiring code generators to follow specific patterns, such as bitmasks before memory accesses and jumps, in order to validate x86 machine code. While the sandbox architecture allowed NaCl code to interact with sensitive data in the same process, the Chrome browser's limitations forced NaCl code to use an out-of-process approach that prevents NaCl code from synchronizing accessing JS or Web APIs. NaCl is fundamentally not portable because it is a subset of a certain architecture's machine code. Portable Native Client (PNaCl) builds upon NaCl's sandboxing techniques by using a stable subset of LLVM Bitcode as an interchange format, allowing for portability (Donovan et al., 2010; Lattner & Adve, 2004). However, it isn't a huge improvement in compactness, because it still reveals compiler or platform-specific details like the call stack layout. Because NaCl and PNaCl are only available in Chrome, their applications are inevitably limited in their portability.

Java and Flash introduced managed run-time plugins to the Web, but neither enabled high-performance low-level code, therefore their use is now decreasing thanks to security and performance concerns (Adobe,

2017).

JS has become a compilation target for other languages due to its ubiquity, quick speed increases in newer VMs, and probably simple need (Haas et al., 2017). In 2012, Mozilla introduced a "strict subset of JS that can be used as a low-level, efficient target language for compilers, called *asm.js* (Herman et al., 2014). Even C and C++ applications can be converted to *asm.js* - a subset of JS where all the operations are clearly statically typed -, using a compiler called Emscripten (Zakai, 2011). So, why create a new standard when there is already *asm.js*? The native decoding of Wasm binary format is substantially faster than parsing JS (Group, 2017b). Experiments have revealed that it can be up to 20 times faster (Group, 2017a). Large compiled codes, for example, can take 20 to 40 seconds to parse on mobile. Therefore, native decoding is essential for a decent cold-loader experience. A new standard makes it much easier to add the functionality required to reach native performance by eliminating the *asm.js* limitations of AOT-compilability and good performance even on engines without specific *asm.js* optimizations. Every new standard comes with costs, such as maintenance, attack surface, and code size, which must compensate with advantages. Wasm reduces costs by allowing a browser to implement Wasm inside its current JS engine. As a result, Wasm is a new and vital JS feature rather than an extension to the browser model. When comparing the two, the benefits outweigh the expenses, even for engines that already optimize *asm.js* (Group, 2017b).

With this in mind, addressing the problem of safe, fast, portable low-level code on the Web, major IT powerhouses are developing a new portable and size/load efficient bytecode language: Wasm (Haas et al., 2017).

Introduced in 2015, Wasm is described as,

> *"A safe virtual instruction set architecture that can be embedded into a range of host environments, such as Web browsers, content delivery networks, or cloud computing platforms. It is represented as a byte code designed to be just-in-time-compiled to native code on the target platform. Wasm is positioned to be an efficient compilation target for low-level languages like C++."* (Watt, Rossberg, et al., 2019, p. 1)

The Wasm ecosystem was developed by the companies that offer the four most widely used Web browsers, namely Google, Microsoft, Mozilla, and Apple. According to Mozilla Tech, *"Wasm 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 (Bryant, 2017).

The primary concern of Wasm is to be fast, safe, portable, and compact. These goals are extremely important and difficult to achieve.

- **Fast**

  Low-level code, such as that produced by a C/C++ compiler, is usually optimized before being executed. Native machine code, whether written by hand or generated by an optimizing compiler, can use the machine's full capabilities. However, low-level code has usually been burdened by managed run-times and sandboxing approaches (Haas et al., 2017).

- **Safe**

  On the Web, security is critical for mobile code since it comes from untrustworthy sources. A managed language run-time, such as the browser's JS VM or a language plugin, has typically been used to protect mobile code. Memory safety is enforced in managed languages, preventing programs from jeopardizing user data or system state. On the other hand, managed language run-times haven't always provided much portable low-level code (Haas et al., 2017).

- **Portable**

  The Web includes not just a wide range of device types, but also a variety of computer architectures, operating systems, and browsers. To execute programs across various browsers and hardware types with the same behavior, Web-targeted code must be hardware and platform-independent. Previous low-level code solutions were either architecture-specific or had additional portability issues (Haas et al., 2017).

- **Compact**

  To reduce load times, conserve potentially expensive bandwidth, and increase overall responsiveness, the code transferred over the network should be as compact as possible. Even when minified and compressed, code on the Web is often delivered as JS source, which is significantly less compact than a binary format (Haas et al., 2017).

Wasm already provides low-level performance that isn't achievable with JS, but there are even more exciting features on the way: streaming compilers and more efficient binary encodings improving loading speed; by permitting integration with the engine's garbage collection, garbage-collected languages will improve performance; improved speed when accessing browser APIs (such as the DOM) by not needing a JS translation layer (Eberhardt & Price, 2018).

Wasm's influence is limited in the immediate term due to the lack of mature tools and the restricted nature of the MVP, which limit its application to a few specialized use cases. However, as the Wasm run-time and tools mature, their influence may be significant. Thus, various studies and researches on Wasm have been conducted, with a particular focus on its efficiency and security.

2.1.1 Performance

The Wasm stack machine is intended to be encoded in a binary format that is both small and fast to load. Wasm aspires to run at native speed by using standard hardware capabilities available on a wide range of platforms. While there are JIT compilers that will compile a Wasm module into native code, certain run-times, such as Web browsers, can be extremely fast if relying only on interpretation (WebAssembly, 2017a).

Wasm's own developers conducted early run-time performance tests, however, these were tiny performance micro-benchmarks and file size comparisons with JS (Haas et al., 2017). They tested micro-benchmarks written in *asm.js* and Wasm. Wasm was 33.7% faster, on average, than *asm.js*. Validation, in

particular, was substantially more efficient.

Previous work compared Wasm and JS on desktop and mobile devices on both client-side Web browsers and the server-side Node.js using numerical benchmarks (Herrera et al., 2018). According to this study, Wasm performs efficiently, particularly with Firefox approaching native C performance. All browsers that used Wasm technology achieved significant improvements over existing JS engines.

Another study evaluated the efficiency and choice of an optimal sparse matrix storage format for sequential Sparse matrix-vector multiplication (SpMV) in JS and Wasm, compared to native languages like C for Firefox and Chrome (Sandhu et al., 2018). They looked at the performance differences between native C, JS, and Wasm. Their findings showed that the highest performing browser had a slowdown of just 2.2x to 5.8x compared to C when it came to JS. Surprisingly, they saw equivalent or superior performance for Wasm when compared to C. Second, they looked at how single-precision vs. double-precision SpMV performed. Unlike C, they showed that double-precision is typically more efficient than single-precision in JS and Wasm. Finally, they looked at selecting the best storage format. Surprisingly, the ideal format options for C vary significantly from those for JS and Wasm, and even across the two browsers.

On the other hand, some researchers indicate the contrary, i.e., to the poor performance of Wasm. For example, previous work shows that the average slowdown of Wasm vs. native was 1.55x in Chrome and 1.45x in Firefox across SPEC benchmarks, with peak slowdowns of 2.5x in Chrome and 2.08x in Firefox (Jangda et al., 2019). To investigate Wasm's performance and, consequently, obtain these findings, they built BROWSIX-WASM, a substantial extension of BROWSIX, and BROWSIX-SPEC, a harness that provides precise performance analysis in order to run the SPEC CPU2006 and CPU2017 benchmarks as Wasm in Chrome and Firefox.

Researchers conducted a study in the sense of better understanding the performance of Wasm applications alongside JS (Yan et al., 2021). In pursuit of this goal, they tested a variety of topic programs, including compiler-generated programs, manually written programs, and real-world applications. As a result, their findings concluded that compiler optimizations for Wasm are often inefficient, resulting in unexpected outcomes. They also observed that JIT did not significantly improve Wasm speed and that the performance of Wasm and JS differ considerably depending on the execution environment. Finally, the researchers noticed that Wasm consumes substantially more memory than JS.

Even though it's recent, Wasm is ready to use. Not only by programmers but also by huge companies and projects. Startups like Figma and Zoom are already embracing Wasm to offer new and better experiences on the Web.

Zoom is the leader in modern workplace video communications, offering an intuitive, reliable cloud platform for video and audio conferencing, chat, and webinars[1]. Zoom already utilizes Wasm SIMD (single instruction, multiple data) to improve audio, video and image processing. When Zoom produces a virtual background or decodes audio, it utilizes Wasm, and this is why Zoom looks to perform so much smoother than earlier video conferencing (Zoom, 2021).

[1]Zoom: https://zoom.us

Figma is a cloud-based design tool that works totally in the Web browser[2]. It has been using *asm.js* for years. The application was initially written in C++ and exported to *asm.js* using Emscripten. But after Wasm was released in 2017, they converted to Wasm, and the results are astounding. The load time is three times smaller. It is essential to remember that the performance boost is compared to *asm.js* and not JS. This subset is already an atypical-performance optimization. Compared to conventional JS, Wasm performs even better in this use case (Wallace, 2017).

2.1.2 Security

Wasm is an increasingly popular compilation target meant to run code in browsers and other platforms safely and securely by rigidly separating code and data, enforcing types, and restricting indirect control flow (WebAssembly, 2017b). It is a sandboxed, memory-safe execution environment that may be used inside current JS virtual machines. Wasm will enforce the browser's same-origin and permissions security standards when embedded in the Web.

Given the more widespread adoption of Wasm, researchers addressed the first in-depth security analysis of Wasm binaries and compares the level of security offered by Wasm with native platforms (Lehmann et al., 2020). These researchers observed that susceptible source programs result in binaries that allow many attacks, including attacks that have not been possible on native platforms for decades. Their results represent a call for action for further hardening the Wasm language, its compilers, and ecosystem, making the promise of a safe platform a reality.

MS-Wasm proposal explicitly targets memory safety enabling developers to capture low-level C/C++ memory semantics such as pointers and memory allocation in Wasm at compile time. MS-Wasm provides a spectrum of security-performance trade-offs and allows users to transition to increasingly better models of memory safety as hardware improves (Disselkoen et al., 2019).

Among the early adopters of Wasm, websites, employ the computing resources of visitors to mine cryptocurrency. A study revealed that over 50% of all sites using Wasm utilize it for malicious deeds, such as mining and obfuscation (Musch et al., 2019).

Various dynamic analyses for Wasm have already been suggested, including two taint analyses - that can also be used to enforce security policies on untrusted Wasm applications - and a Cryptomining detector (SEISMIC) (Fu et al., 2018; Szanto et al., 2018; W. Wang et al., 2018). Developers of the taint tracking system - built by implementing a new Wasm virtual machine in JS -, demonstrate that their system is accurate, safe, and relatively efficient, benefiting from the native speed of Wasm while keeping exact security assurances of more mature software paradigms (Szanto et al., 2018). Other researchers described a taint tracking engine for interpreted Wasm - built by changing the V8 engine - and concluded that their changes to the V8 engine do not impose substantial cost concerning vanilla V8's interpreted (Fu et al., 2018). SEISMIC provides a semantic-based cryptojacking detection technique for Wasm scripts that is more powerful than traditional static detection defenses deployed by antivirus programs and browser plugins (W.

[2]Figma: https://www.figma.com/

Wang et al., 2018). These studies can be done in top of Wasabi - a general dynamic analysis framework for Wasm - with substantially less effort (Lehmann & Pradel, 2019). This framework faithfully preserves the original program behavior, imposes an overhead that is reasonable for heavyweight dynamic analysis, and makes it straightforward to implement various dynamic analyses, including instruction counting, call graph extraction, memory access tracing, and taint analysis.

Investigators from two industry White Papers demonstrate example attacks against vulnerable Wasm binaries (Bergbom, 2018; McFadden et al., 2018). Simple memory safety vulnerabilities and exploits from the '90s might have an impact on today's Wasm applications. Thus, researchers reported how vulnerability classes may affect Wasm Web applications developed in memory-unsafe languages (Bergbom, 2018). Other study explored the actual security risks that a developer may take on by adopting Wasm and also gave a general description of recommended practices and security considerations for developers intending to incorporate Wasm into their products (McFadden et al., 2018).

Wasm's host security features may also act as a base for Software fault isolation (SFI). By compiling specific libraries to Wasm and embedding a run-time into the main application, memory issues in the library are separated from the main program (Lehmann et al., 2020). Wasm has also been used as a compilation target for formally validated cryptography. Developers described how to produce verified implementations of cryptographic primitives deployed both within platform libraries and within pure JS programs, through a Wasm implementation. They also explained how to develop a validated implementation of the Signal protocol - as a Wasm module - and use it to produce a replacement for LibSignal (Protzenko et al., 2019).

Researchers from University of Cambridge and California presented the design and implementation of Constant-Time WebAssembly (CT-Wasm), a low-level bytecode language that extends Wasm to enable developers to build verifiably safe crypto algorithms (Renner et al., 2018). CT-Wasm offers a principled approach to improving the quality and auditability of Web platform cryptographic libraries while preserving the convenience that made JS so popular. These developers claim that CT-Wasm is fast and flexible enough, meant to be use-able as a development language for current, based on Wasm environments (Watt, Renner, et al., 2019b).

To protect the Wasm code, investigators presented a framework called SELWasm (Sun et al., 2019). Three primary mechanisms were offered in this framework: Environment Self-checking, Encryption&Decryption, and Lazy-loading. The Environment Self-checking technique can prevent unauthorized Websites from reusing the source code. The Encryption&Decryption mechanism can prevent plain text distribution of Wasm code. The lazy-loading mechanism can enhance page loading performance while reducing the overhead caused by prior mechanisms. Their findings suggest that Web attacker activities can be countered and that the overhead needed is minimal.

Other researchers suggested Swivel, a system that protects Wasm modules against Spectre attacks while providing robust in-memory isolation (Narayan et al., 2021). They described two Swivel designs: Swivel-SFI, a software-only method that offers mitigation's compatible with current CPUs, and Swivel-CET, which uses Intel® CET and Intel® MPK, were detailed. Swivel versions that use ASLR have negligible performance costs, proving that Swivel can offer strong security assurances for Wasm modules while

preserving the performance advantages of in-process sandboxing.

A study of the security features, languages, and use cases of a varied set of real-world Wasm binaries was described in a paper, in which the results confirm several previously held assumptions regarding real-world Wasm and point to issues that need more investigation (Hilbig et al., 2021). They reveal that vulnerabilities propagated from insecure source languages can impact a vast array of binaries and show that 29% of all binaries on the Internet are minified, requiring decompilation and reverse engineering methods for Wasm.

2.1.3 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. In fact, 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 (Contributors., 2021). 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) - it's only really intended for last-resort debugging -, there are compilers for most languages that produce the low level Wasm code, such as Cheerp (L.Technologies, 2021), Emscripten (Zakai, 2011), AssemblyScript (AssemblyScript, 2021), and Asterius (Cheng et al., to appear). 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[3]. The hand written code of the Game-boy 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. In fact, 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.

2.2 Green Software

The third decade of the 21^{th} century has begun and climate change is a serious environmental issue that humanity needs to face. Over the years, Information technology (IT) is being both a problem and solution to environmental sustainability. On the one hand, IT manufacturing, use and disposal, and other IT equipment require a lot of energy, which overloads power networks and contributes to greenhouse

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

gas emissions. On the other hand, IT has a beneficial influence by increasing energy efficiency, reducing greenhouse gas emissions and harmful materials consumption, and promoting reuse and recycling (Hilty et al., 2006; Köhler & Erdmann, 2004). Green IT is responsible for this IT's beneficial influence on environmental sustainability (Chetty et al., 2008; Melville, 2010). In 2007 was released the first report on Green IT (Mingay, 2007). Since then, the term *"Green IT"* became widely used.

Green IT is described as

> *"the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated subsystems - such as monitors, printers, storage devices, and networking and communications systems - efficiently and effectively with minimal or no impact on the environment. Green IT also strives to achieve economic viability and improved system performance and use, while abiding by our social and ethical responsibilities."* (Murugesan, 2008, p. 25)

The software side of Green It - known as Green software - is one of the most critical parts. Green software aims to use energy more intelligently and decrease energy consumption through analysis, transformations, and optimizations techniques in a wide range of computing systems such as mobile, programs, databases, etc (Calero & Piattini, 2015).

Research in Green software is growing, and as a result, software engineering experts are increasingly focusing on it as a significant challenge area. The growing number of publications in important events such as the International Workshop on Green and Sustainable Software (GREENS) [4], International Workshop on Requirements Engineering for Sustainable Systems (RE4SuSy) [5], International Workshop on Sustainable Software Engineering (SUSTAINSE) [6], International Conference on ICT for Sustainability (ICT4S) [7], and IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) [8] are examples of the growing and widespread interest in this research area.

Studies on Green software have grown with several purposes and in different areas, intending to understand how development factors may affect energy consumption in a variety of software systems. Due to the exponential growth of the use of powerful mobile devices such as smartphones, tablets, smartwatches, and laptops, the need to reduce energy consumption is becoming, more than ever, widely recognized. As a result, there are works focused on monitoring how energy consumption evolves in mobile devices, on techniques to identify anomalous energy consumption in Android applications, or to detect energy-inefficient fragments in the source code of a software system (Couto et al., 2014; Ding et al., 2011; Pereira, Carção, et al., 2017, 2020). Other studies provide detailed information about the energy consumption of mobile applications, techniques and tools to explain the energy consumption of all

[4]GREENS: https://greens.cs.vu.nl/
[5]RE4SuSy: http://birgit.penzenstadler.de/re4susy/
[6]SUSTAINSE: https://sites.google.com/view/sustainse2021/home
[7]ICT4S: https://conf.researchr.org/home/ict4s-2022
[8]SANER: https://saner2022.uom.gr/index

products in a Software Product Lines (SPL) or by providing conclusions on the energy impact of different implementation decisions (Couto, Borba, et al., 2017; Cruz & Abreu, 2017; Li et al., 2013).

Several other studies have found that coding practices have a considerable impact on the energy efficiency of software. For mobile devices, for example, researchers have studied the application-level impact of refactorings and analyzed the impact of some of the most popular development approaches on the energy consumption of Android applications (Couto et al., 2020; Oliveira et al., 2017). For embedded systems, investigations measured the energy consumption of programming tasks and performed a study to evaluate the energy consumption of Remote Inter-Process communication technologies (Georgiou et al., 2018; Georgiou & Spinellis, 2020). For desktop and server, there are studies analyzing the energy behavior of programs written in Haskell and analyzing the run-time, memory usage, and energy consumption of twenty seven well-known software languages (Couto, Pereira, et al., 2017; Lima et al., 2016; Lima et al., 2019; Pereira, Couto, Ribeiro, et al., 2017, 2021). In fact, this recent research provides a popular programming language ranking that reflects - among 27 well-known languages - which ones have the best energy consumption, run-time performance, and memory usage. According to these researchers, they were able to relate execution time and memory usage with energy consumption. In addition, they understood how memory usage affects energy consumption, and how energy and time relate. In order to help developers to become more energy-aware when programming, they also clarify that a faster language is not always the most energy-efficient. Figure 2.1 shows the ranking of the 27 programming languages for Energy, Time and Memory.

2.2.1 RAPL

Intel's Running Average Power Limit (RAPL) is a tool that most modern processors have that provides power limiting features and allows to monitor energy consumption of the CPU package and its components, including the DRAM memory that the CPU is managing with high sampling rate (Weaver et al., 2012). This functionality was introduced in Intel's Sandy Bridge architecture and has evolved in the later versions of Intel's processor architecture, documented in the Intel Software Developer's Manual (Intel, 2009). Furthermore, the latest version of RAPL allows it - via jRAPL - to be called from any application written not only in C but also in Java (Liu et al., 2015). Therefore, RAPL is a proper tool to measure, monitor, and respond to the energy consumption of computing (K. N. Khan et al., 2018).

Several previous research works have used RAPL and it has been proved that RAPL is capable of collecting accurate energy calculations at a very fine-grained level (Hähnel et al., 2012; Rotem et al., 2012). For example, a study used RAPL on an analysis, comparing the energy consumed by two of the most popular Web browsers, Google Chrome and Mozilla Firefox. They were able to see which browser tends to be the most energy-efficient, how various Websites behave differently, and observed the energy consistency of the two Web browsers (de Macedo et al., 2020). Other research aimed to establish a method to help non-specialist developers create energy-aware software for mobile and desktop environments. They measured energy consumption on desktop apps using the jRAPL library and found that some of the

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Figure 2.1: Ranking of 27 programming languages results in terms of Energy, Time and Memory performance (Pereira, Couto, Ribeiro, et al., 2017).

most common collection implementations aren't always the most energy-efficient (de Oliveira Júnior et al., 2019). JRAPL was also used by researchers in an analysis of the energy consumption of the Java Collection Framework implementations when handling different amounts of data and in a technique to detect energy-inefficient fragments in the source code of a software system. (Pereira et al., 2020; Pereira et al., 2016).

Chapter 3

Benchmark Design and Execution

Here we present the methodology used in building a benchmarking framework capable of running, measuring, and collecting the results needed to perform a sustained study. We start by describing the micro-benchmarks used in this study, where we got them from, and the source code adjustments that we needed to do. Moreover, we explain how we compile the benchmark programs in WebAssembly and JS. Apart from micro-benchmarks, this study also includes two real applications with benchmarking purposes within a browser-based environment. Finally, we described how this framework measures energy consumption and collects data.

3.1 Micro-Benchmarks

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 JS 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

Micro-benchmarks is one of the principal ways to measuring the performance of a software system, thus, Wasm is no exception. To generate equivalent Wasm and JS 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 (Pereira, Couto, Ribeiro, et al., 2017, 2021).

Our benchmarking framework consists in 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[1] for both researchers and practitioners to both replicate and build upon. Finally, Figure 3.1 presents the overview of our benchmarking framework.

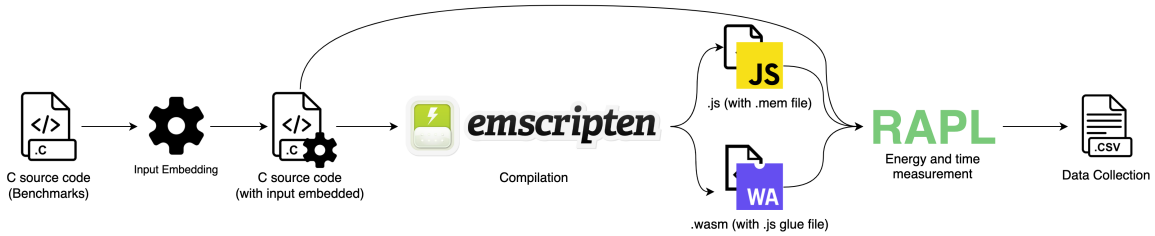


Figure 3.1: Benchmark framework overview.

3.1.1 Micro-Benchmark Programs

Wasm was designed to be used in compute-intensive cases such as compression, encryption, image processing, games, and numeric computations. For this study, we firstly 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 Code[2]. 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[3] intensive benchmark problems from the Computer Language Benchmarks Game (CLBG): *fannkuch-redux* and *fasta*. The CLBG[4] 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 Wasm and JS 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 (Couto, Pereira, et al., 2017; Georgiou et al., 2018; Georgiou & Spinellis, 2020; Lima et al., 2016; Lima et al., 2019; Nanz & Furia, 2015; Oliveira et al., 2017; Pereira, Couto, Ribeiro, et al., 2017, 2021). Finally, shown in Table 3.1 is the list of our benchmark programs and their brief description, totalling 10 unique solutions.

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

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

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

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

Table 3.1: Benchmark programs 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.

3.1.2 Embedded Inputs

The performance of a program can vary depending on its complexity and the effort required for its execution. Thus, in this study, to have different complexities of each benchmark solution, 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[1].

Emscripten, 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 , JS , 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 3.1.

```
#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 3.1: Example input header file.

3.1.3 Compiling to WebAssembly and JavaScript

In order to compile our C based benchmarks into the respective Wasm and JS versions, we used Emscripten for the compilation process. We choose Emscripten as it is open source, already used by researchers and practitioners (Haas et al., 2017; Zakai, 2018; Zakai, 2011), and offers extensive documentation[5].

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 JS creates the .js file - more specifically *asm.js*, a subset of JS - 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.

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.

This approach, 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. Therefore, we also developed a framework to measure the performance within a browser-based environment, namely within the Google Chrome, Mozilla Firefox, and Microsoft Edge browsers. Figure 3.2 illustrates the benchmark framework in a similar way to Figure 3.1, but with the addition of a browser-based environment.

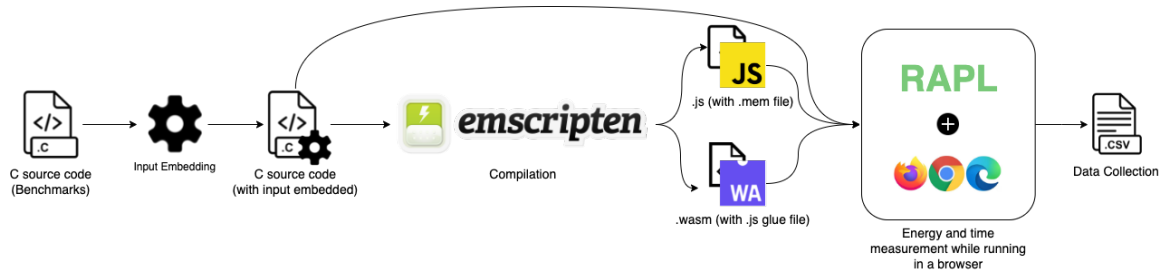


Figure 3.2: Benchmark framework overview within a browser-based environment.

To run benchmark programs in Web browsers, we create an HTML file for each solution. Listing 3.2 shows an example of a *Fasta* HTML file in Wasm language with a *Small* input size (*fasta_WASM_SMALL.js*). Thereafter, we built a local server with Python - with the command `"python -m SimpleHTTPServer"` -

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

where each HTML file locates. Finally, we opened the respective browser in *incognito mode* and entered *"localhost:8000/file.html"* in the address bar, where *file.html* is the HTML file we want to execute.

```
<script defer src="fasta_WASM_SMALL.js"></script>
```

Listing 3.2: Example of an HTML file.

On the other hand, micro-benchmarks aren't the best fit for benchmarking Wasm because the primary goal of Wasm is to enable high-performance real-world applications on web pages. In order to simulate a more realistic test case, we also benchmark the performance of two Wasm web-based applications developed with benchmarking purposes when executed within a browser-based environment. To monitor such Web browsers, we developed a framework to measure the energy consumption and run-time while such browsers are executing these benchmark applications.

3.2 Real-World Wasm Applications

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

3.2.1 WasmBoy Benchmark

WasmBoy is a GameBoy/GameBoy Color Emulator, written in TypeScript (TS) to benchmark WebAssembly created by Aaron Turner (Turner, 2018). WasmBoy is written in JS/TS and it was created with the main goal of comparing the run-time performance between Wasm - produced by the AssemblyScript compiler - and the ES6 latest version of JS as produced by the TS compiler. WasmBoy is organized into two sections: the "lib" (JS API Interface) and the "core" (GameBoy Emulation "Backend"). This benchmark is open source[6] and ready to use[7]. The WasmBoy benchmarking program works by loading each of the possible WasmBoy core configurations and then executing a defined number of frames of an input Game/ROM. The number of frames chosen for our study was based on the number of frames until the respective Game achieves the well-known "Main Menu"of each Game. Each core is then imported by the benchmarking application using standard ES6 imports and built into an IIFE using rollup.js.

WasmBoy contains various open-source Games that run from the tool, but it's possible to upload other GameBoy Games to be tested. Every frame of a Game is different, and so is every Game! Our setup of this game console includes six different open source games that can be executed by the console.

The WasmBoy framework, as shown in Figure 3.3, consists in five steps: 1) choose a Game, 2) select the language and the number of frames to run, 3) generate the *index* and HTML files, 4) open it in a browser and start measuring, and 5) Data Collection.

[6]WasmBoy benchmark source code: https://github.com/torch2424/wasmBoy/tree/master/demo/benchmark
[7]WasmBoy benchmarking tool: https://wasmboy.app/benchmark/

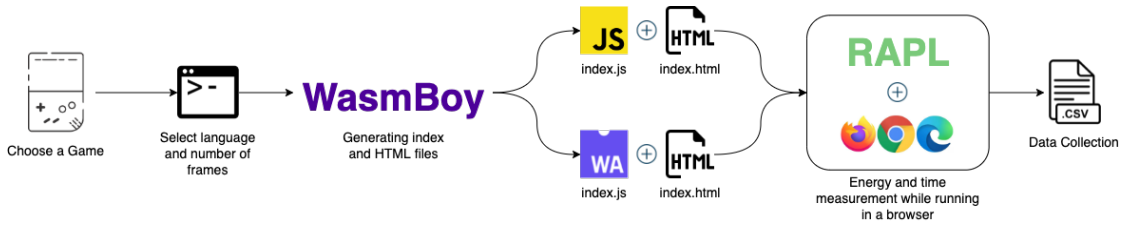


Figure 3.3: Benchmark framework overview of WasmBoy.

Before generate each *index* and HTML files, we updated WasmBoy source code in order to specify the Game we want to run, the language and the number of frames the Game have to execute. Then, a makefile executes each HTML solution in all three browsers and collect the results. Finally, considering the 6 different games, with 2 languages (Wasm and JS), across the 3 chosen browsers, we have a total of 36 unique samples.

3.2.2 PSPDFKit Benchmark

PSPDFKit software allows to view, annotate, and fill in forms in PDF documents on any platform. In order to 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. The PSPDFKit benchmark is a JS application that runs in the browser that measures the time of various activities on a PDF document, using the Web Performance API. This benchmark is also open source[8] and ready to use[9].

To execute this benchmark with realistic and different 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).

The PSPDFKit framework, as shown in Figure 3.4, consists in five steps: 1) choose a PDF and the language to run, 2) run the benchmark server, 3) open it in a browser and start measuring, and 5) Data Collection.

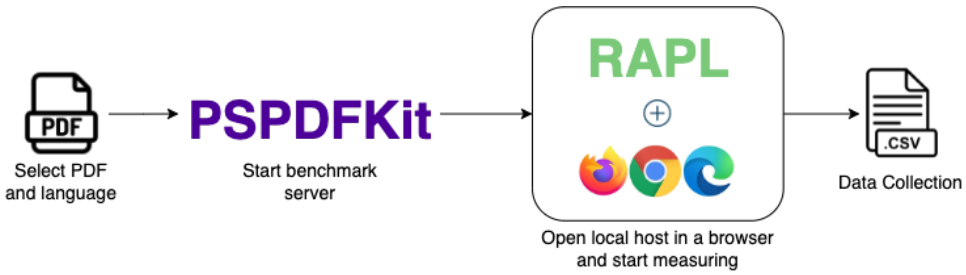


Figure 3.4: Benchmark framework overview of PSPDFKit.

[8]PSPDFKit benchmark source code: https://github.com/PSPDFKit-labs/pspdfkit-webassembly-benchmark
[9]PSPDFKit benchmarking tool: https://pspdfkit.com/webassembly-benchmark/

Before we ran the benchmark server, we had to update the source code whenever we wanted to specify the PDF and language we wanted to run. Then, we started the benchmark server with `"npm start"` and, with a makefile, we opened a specific browser in the local host (available at `"http://localhost:3000"`). While the browser is open, this makefile starts measuring and collecting energy usage results. Considering our 5 various PDF 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 JS 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.

3.3 Measuring Energy and Run-time

To monitor the energy consumption of our benchmarks, we rely on RAPL[10]. 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 (de Macedo et al., 2020; de Oliveira Júnior et al., 2019; Lima et al., 2016; Pereira et al., 2020; Pereira, Couto, Ribeiro, et al., 2017, 2021; Pereira et al., 2016), and has been proven to give highly accurate energy measurements (Hähnel et al., 2012).

We have developed a C-based thread that runs alongside the benchmark execution, constantly sampling the energy usage to ensure no register overflow[11] happens while measuring using RAPL. This thread also records the start and finish run-times of the benchmark being performed. Each micro-benchmark program running outside the Web browser was executed twenty times and all others benchmark was run five times (Hogg et al., 2010), with a five-second sleep between each execution, to gather consistent data and minimize cold start, warm-up, and cache effects.

In order to have more accurate results, we created two separate files named *raplServer* (written in C++) and *raplClient* (written in C). The file *raplServer* is always waiting to receive a message from *raplClient*. When it gets a message from *raplClient* - with the right message -, it launches a thread responsible for start gathering energy information from RAPL. At the same time, when *raplClient* sends a message to *raplServer* to start measuring, *raplClient* starts counting the execution time and is also responsible for running the respective benchmark program using the system *function*. The overall process of this framework is described in Listings 3.4 and 3.3.

[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

```
int main(int argc, char **argv)
{
    ...
    // Create start message to send to server
    char * startmsg = malloc(10 + strlen(argv[1]) + strlen(argv[2]));
    sprintf(startmsg, "start %s %s", argv[1], argv[2]);
    printf(argv[2]);
    ...
    /* Open file */
    FILE * fp = fopen(timefile, "w+");
    /* Write header line */
    fprintf(fp, "Time\n");
    ...
    // connect the client socket to server socket
    if (connect(sockfd, ainfo->ai_addr, ainfo->ai_addrlen) != 0) {
        printf("Connection with the server failed...\n");
        exit(0); }
    else
        printf("Connected to the server.\n");

    // Send message to start measuring
    write(sockfd, startmsg, 256);

    // get start time
    struct timeval  tv1, tv2;
    gettimeofday(&tv1, NULL);

    // Run command
    system(command);

    // get end time
    gettimeofday(&tv2, NULL);
    ...

    // write exectime in time file
    sprintf(timefinal,"%f", exectime);
    fprintf(fp,timefinal);

    // Send message to stop measuring
    write(sockfd, "end", 3);
    ...
    // close the socket
    close(sockfd);
}
```

Listing 3.3: Overall process of *raplClient.c*.

```
void * runRAPL(){
    int core = 0;

    /* Open file */
    FILE * fp = fopen(raplOutput, "w+");

    /* Write header line */
    fprintf(fp, "Package,CPU,GPU,DRAM\n");

    rapl_init(core);

    while(run){
        rapl_before(fp, core);
        usleep(usecs);
        if(run) rapl_after(fp, core);
        }

    fclose(fp);
    pthread_exit(0);
    return NULL;
}
```

Listing 3.4: Overall process of *raplServer.cpp*.

As we can see, both the execution energy and time are measured for every execution of *command*. This variable holds a string responsible for defining which benchmark program to run, what size of the input when running micro-benchmarks and the sample rate. For example, using micro-benchmarks, to run the *Fasta* program with Large input in WebAssembly, the value of the command variable will be `"sudo .raplClient 10000 fastaLARGE "node ../fasta.gcc-2.gcc_runWASM_LARGE.js""`. For *Wasm-Boy*, to run *Pokemon* Game, the value of the command variable will be `"sudo raplClient 10000 pokemon "sudo -u diguest xdg-open ../../index.html""`. When using Web browsers to run the programs, we use the command `"xdg-settings set default-web-browser"`, to set the default Web browser that we want to run the benchmark solution. For example, to set Google Chrome as the default browser, we run `"xdg-settings set default-web-browser google-chrome.desktop"`. As a result, the *html* file that runs *Pokemon* Game will be launched with Google Chrome.

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.

3.4 Data Collection

We evaluate Wasm and JS programs' performance in three of the four most popular and used browsers according to several statistics websites[12,13]: 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, *cleanresults.py*, with three versions depending on the benchmark used. It automatically aggregates all the RAPL energy and run-time samples per benchmark-input-language-execution or benchmark-input-language-browser-execution for micro-benchmarks and benchmark-language-browser-execution for real-world applications. 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 (also containing both median and mean): Package (Joules), CPU cores (Joules), DRAM (Joules), GPU (Joules) and Time (Seconds). A short sample of the file *cleanresults.py* used to collect PSPDFKit results is shown in Listing 3.5.

```
text = sys.argv[1]
finalfile = open(text+"all.csv", "w")
finalfile.write("Browser,Language,N,Package,CPU,GPU,DRAM,Time\n")
finalfile.close()
cwds= [os.getcwd(),os.getcwd()]
cwds[0]+= "/TS/Results/"
cwds[1]+= "/Wasm/Results/"
for cwd in cwds:
    for filename in os.listdir(cwd):
        f = os.path.join(cwd, filename)
        if os.path.isdir(f):
            caminho = f.rsplit('/', 3)
            browser = caminho[3]
            language = caminho[1]
            for i in range(1,6):
                    urltime = f+"/"+text+str(i)+".time"
                    urlrapl = f+"/"+text+str(i)+".rapl"
                    if(path.exists(urlrapl)):
                        if(i==1):
                            m1 = {}
                            m1["Package"] = []
                            m1["CPU"] = []
                            m1["GPU"] = []
                            m1["DRAM"] = []
                            m1["Time"] = []
                        raplclean(urlrapl)
                        crp(urltime,urlrapl,text,m1,i,browser,language)
            printmediana(text,m1,browser,language)
            printmedia(text,m1,browser,language)
```

Listing 3.5: Overall process of *cleanresults.py*.

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

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

Chapter 4

Analysis and Discussion

This chapter presents the benchmark results collected by running the two Wasm benchmarks and the micro-benchmarks presented in the previous chapter. The main focus is to understand if Wasm is already outperforming JS regarding 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. Finally, we end this chapter by presenting the threads to validity of this research.

4.1 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 lot of speculation 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, which means, $Joules$ per $Second$. 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.

4.1.1 Micro-Benchmark Programs

Shown in Figure 4.1 are the results collected from six of the ten benchmarks executed in our study. This section does not include all the graphs. It only contains those that can illustrate the widest diversity of findings. All graphics are in the appendix at the end of the dissertation.

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 (C, Wasm and JS).

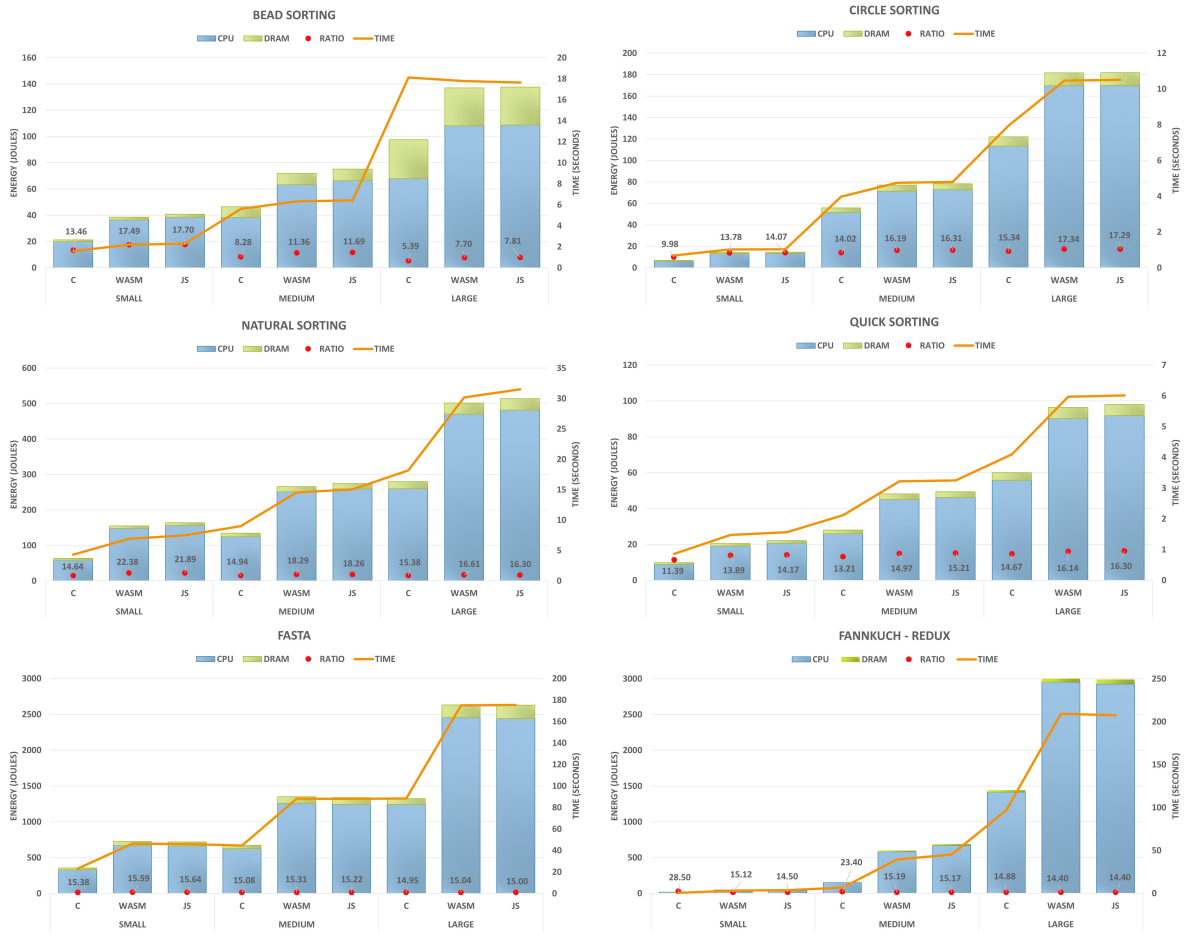


Figure 4.1: Energy consumed by the CPU and DRAM by six benchmarks solutions with the three input sizes and the respective execution times and power values.

In Figure 4.2 are a set violin plots of three benchmarks. 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. Each pair of violin plots represents the respective energy consumed (left plot) and run-time (right plot) of each benchmark program. In each violin plot, the results are also ordered (from left to right) by the input size of *Small*, *Medium*, and *Large*, and within each size are the three languages (C, Wasm and JS).

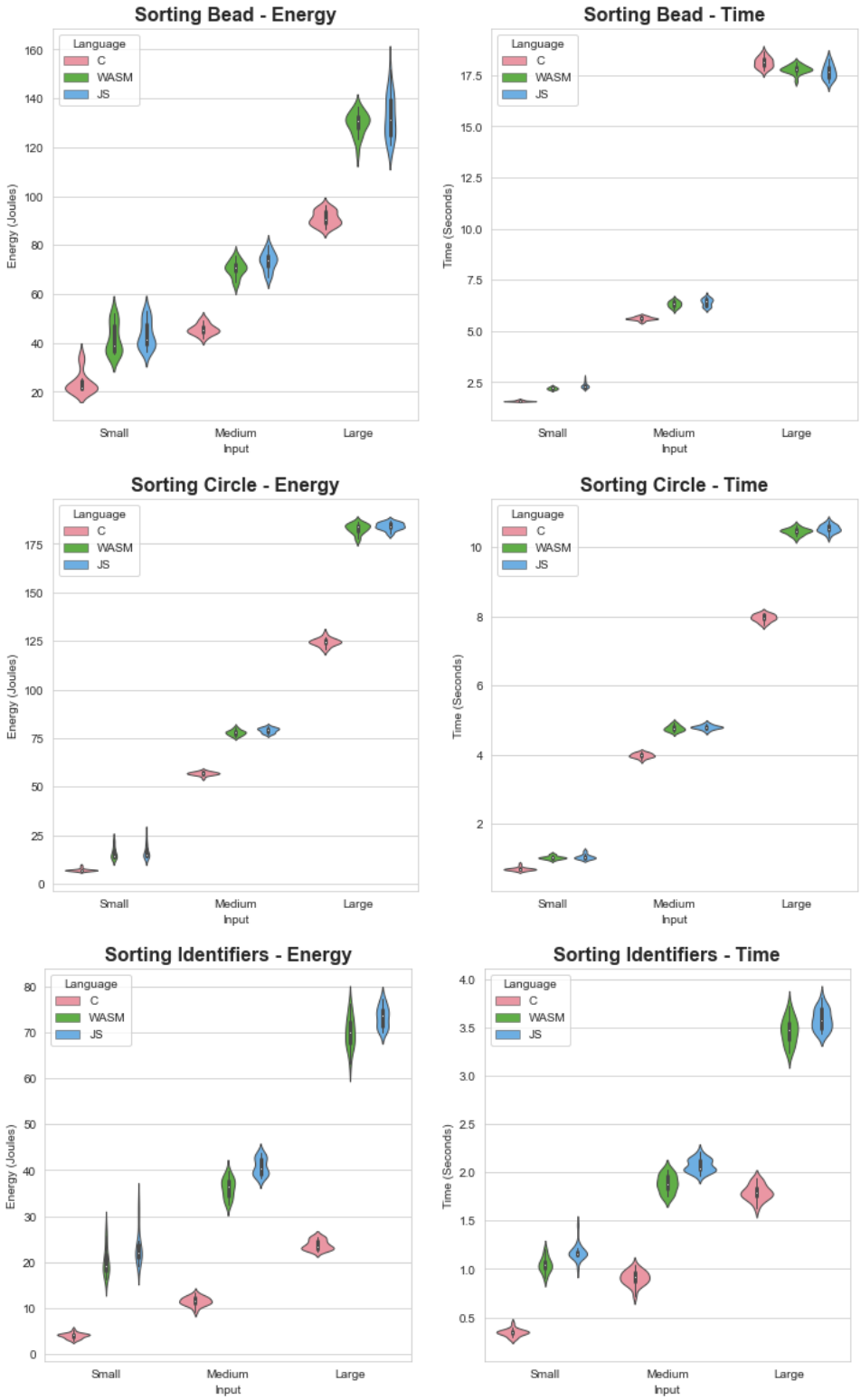


Figure 4.2: Violin plots of energy consumed by a benchmark execution with the three input sizes and the respective execution times.

Figure 4.3 is an Heat map that shows the proportion between JS 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 lower axis, including the tree input sizes, the Energy (Joules), Time (ms) and Ratio (J/s) 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 than JS. On the other hand, in the same row but for the *Small Ratio* column, Wasm was less efficient than JS, which means that Wasm spent more power with the *Fannkuch-redux* program with a *Small* input size.

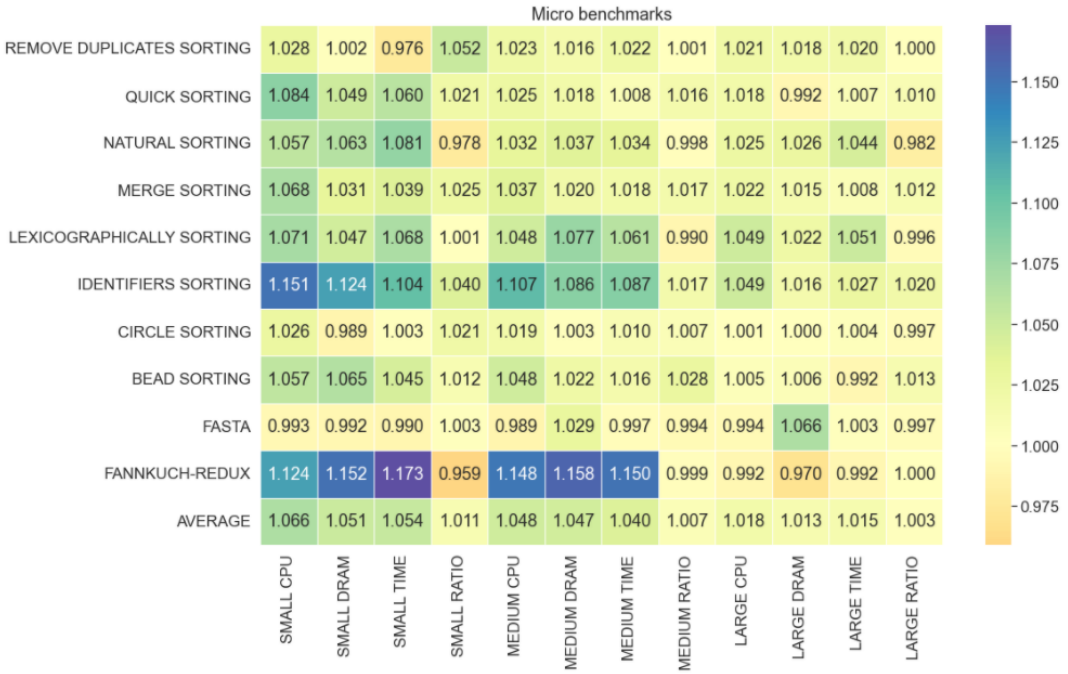


Figure 4.3: Heat map representing the proportion between Wasm and JS results with the three input sizes.

Since this previous approach only executes both Wasm and JS micro-benchmarks in their virtual machines, we added Web browsers to the test cases. Figure 4.4 shows five benchmarks, but, this time, executed within a browser-based environment in order to have more realistic outcomes with the possibility to compare Web browsers performances. Each line of plots represents a single benchmark with the three different input sizes. The difference between the three plots of each line is the input size. On the left we have the results using a *Small* input size, on the middle a *Medium* input size and on the right a *Large* input size. In each chart, the results are ordered (from left to right) by the the browser used (Chrome, Edge and Firefox) and within each browser are the two languages (JS and Wasm).

Figure 4.4: Energy consumed by each browser for each benchmark with the three input sizes and the respective execution times and ratio values.

4.1.2 Real-World Wasm Applications

In this subsection we show the results obtained by the real-world applications, *WasmBoy* and *PSPDFKit*. Firstly, with *WasmBoy* benchmark, we tested six different open source Games:

- Back To Color
- Dinos Offline Adventure
- Pokemon
- Super Mario
- Super Mario Land

- Tobu Tobu Girl

and, Figure 4.5 shows the average of the energy consumed and run-time results of each Game, using different Web browsers. For example, in the top left-most chart (Back To Color Game), we can see that the most inefficient performance was on Mozilla Firefox, using JS. In terms of energy consumed, it used, on average, almost 3000 $Joules$ (CPU plus DRAM) per execution and, in terms of run-time, it took nearly 400 seconds. Also, this Game had a power (or ratio) of 7.92 $Watts$, which means that, per $second$, it spent 7.92 $Joules$.

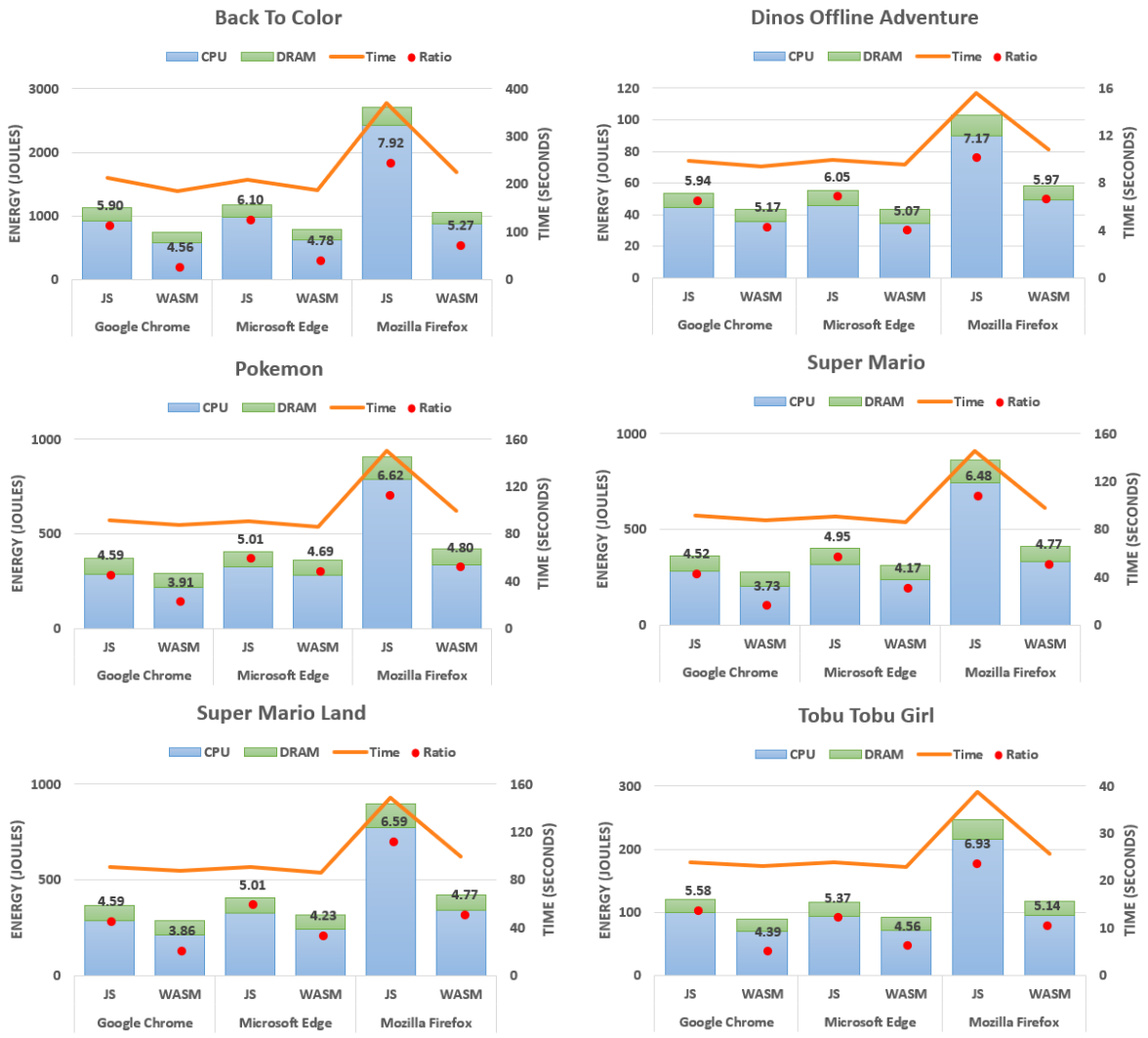


Figure 4.5: WasmBoy: Energy consumed and run-time by each program in each Web browser and the respective ratio values.

Unlike bar plots, which only show the average of all executions, violin plots can display the entire distribution of the data, e.g., all results from all tests. As a result, in Figure 4.6 we can verify whether a language is consistent or has very inconsistent performance. In each violin plot, the results are also ordered (from left to right) by the Web browser used - Chrome, Edge and Firefox -, and within each browser are the two languages (JS or TS in blue, and Wasm in green). For example, in the top left-most violin plot, it is possible to conclude that, when compared to Wasm, JS energetically performed much more

inconsistently using Mozilla Firefox. JS had nearly 2500 $Joules$ and more than 3000 $Joules$ in different executions, while Wasm had consistent results. Each pair of violin plots represents the respective energy consumed (left plot) and run-time (right plot) of each Game. In Figure 4.6 we show the results of three different Games.

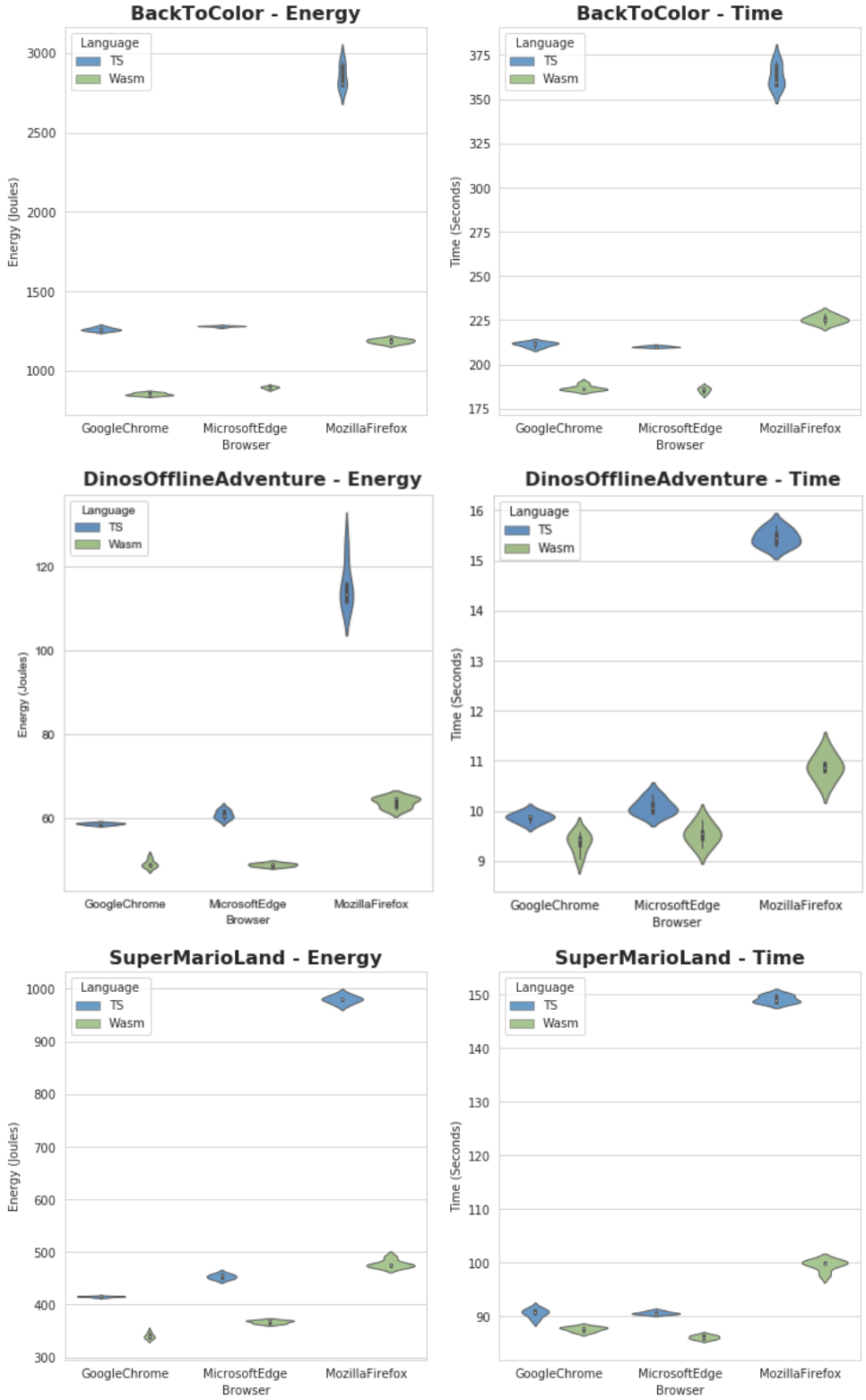


Figure 4.6: WasmBoy: Violin plots of energy consumed and run-time by each program execution in each Web browser and the respective ratio values.

Figure 4.7 is an Heat map that shows the proportion between JS and Wasm, e.g., how many times Wasm was more efficient than JS with *WasmBoy* benchmark. The left axis represents the six Games and, in the last row, is the average of the above values. All types of results obtained are on the lower axis, including the tree browsers, the Energy (Joules), Time (seconds) and Ratio (J/s) 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 *Back To Color* row and *Firefox Energy* column, Wasm was shown to be 2.46 times more efficient than JS.

Figure 4.7: WasmBoy: Heat map representing the proportion between Wasm and JS results in the three browsers.

Secondly, with *PSPDFKit* benchmark, we test five different PDFs:

- 20 pages of a book (Book - 20 Pages)
- 40 pages of the same book (Book - 40 Pages)
- 80 pages of the same book (Book - 80 Pages)
- 10 pages of a Scientific paper (Research Paper)
- 20 slides of a slide presentation (PPT)

and, Figure 4.8 shows all the average results obtained using these PDFs files. These bar plots have the same format as the previous *WasmBoy* bar plots. For example, in the top left-most chart (Book - 20 Pages), we can see that the most inefficient performance was on Mozilla Firefox, using JS. In terms of energy consumed, it used, on average, about 2000 $Joules$ (CPU plus DRAM) per execution and, in terms

of run-time, it took about 120 seconds per execution. Also, this PDF had a power (or ratio) of 14.41 $Watts$, which means that, per $second$, it spent 7.92 $Joules$.

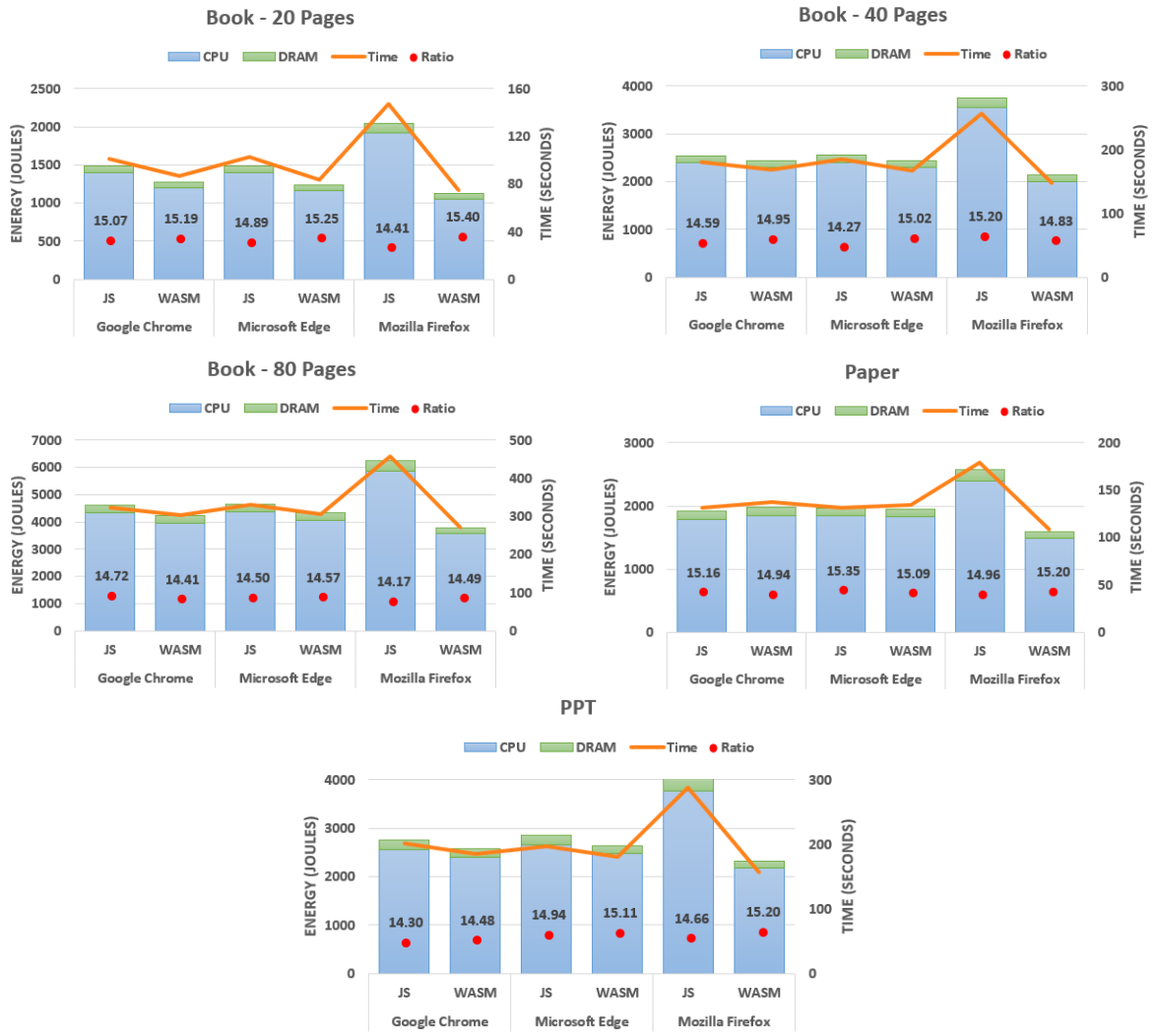


Figure 4.8: PSPDFKit: Energy consumed and run-time by each program in each Web browser and the respective ratio values.

For the same reason that micro-benchmarks and *WasmBoy* violin plots were shown, Figure 4.9 shows all data collected from three different PDFs using *PSPDFKit* benchmark of all tests executed. Each pair of violin plots represents the respective energy consumed (left plot) and run-time (right plot) of each PDF. For example, we can see that Wasm performance was not so good using the Research Paper PDF (middle violin plots), compared to the others two PDFs examples shown (Book - 80 pages and PPT). With this two PDFs, Wasm was always more energy efficient and faster than JS. However, using the Research Paper PDF, Wasm only had a better performance than JS using Firefox browser.

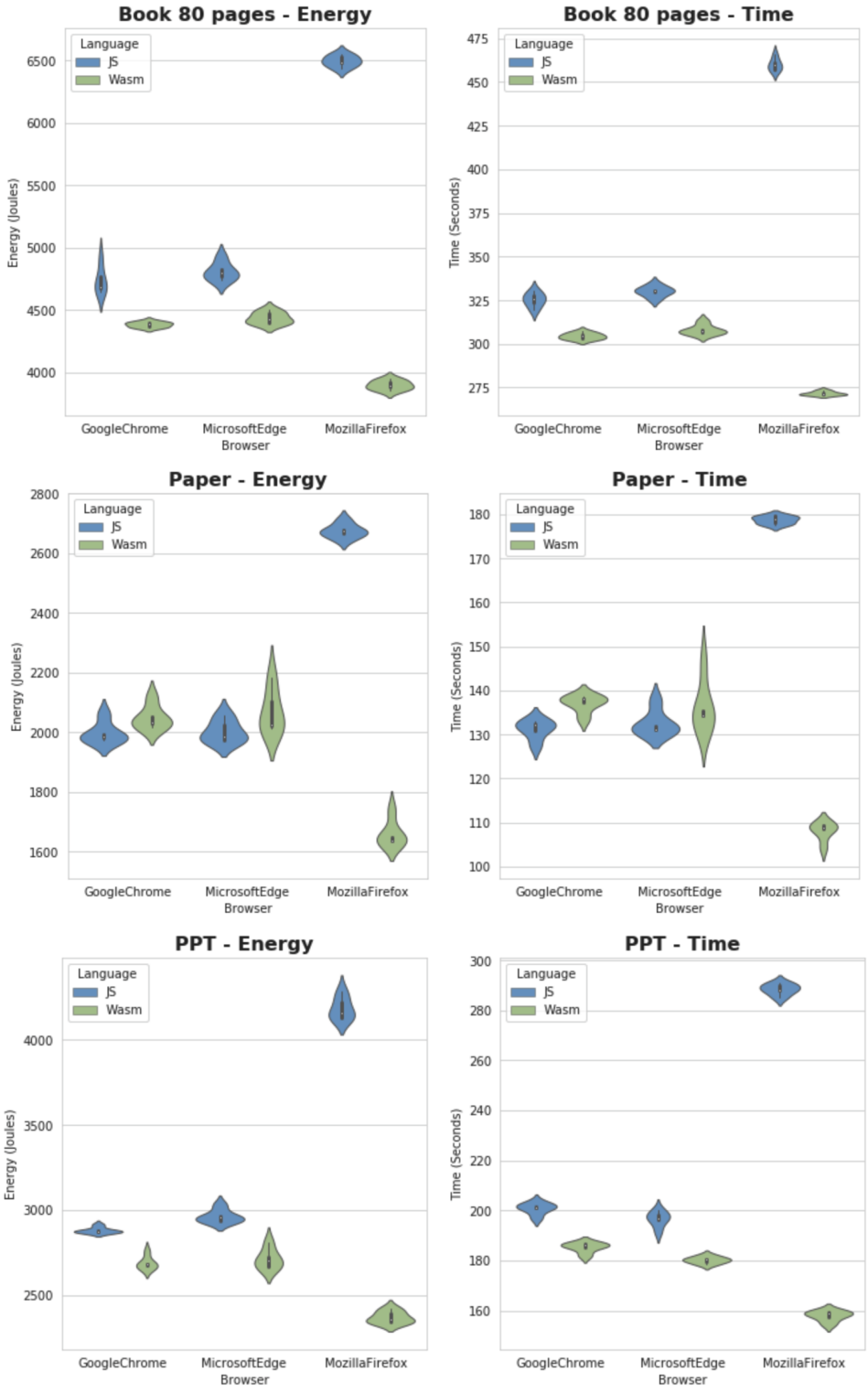


Figure 4.9: PSPDFKit: Violin plots of energy consumed and run-time by each program execution in each Web browser and the respective ratio values.

With Figure 4.10, it is possible to see where and how much Wasm was more energy and run-time efficient than JS using the *PSPDFKit* benchmark. For example, Wasm was almost two times faster (1.96x) than JS when rendering 20 pages of the Book using Mozilla Firefox. On the other hand, Wasm had poor results with the *Paper*, for example using Chrome, JS was more energy and run-time efficient. However, JS spent more energy than Wasm with Chrome, as can be seen in the *Chrome Ratio* row, with the *Paper* PDF (value of 1.02).

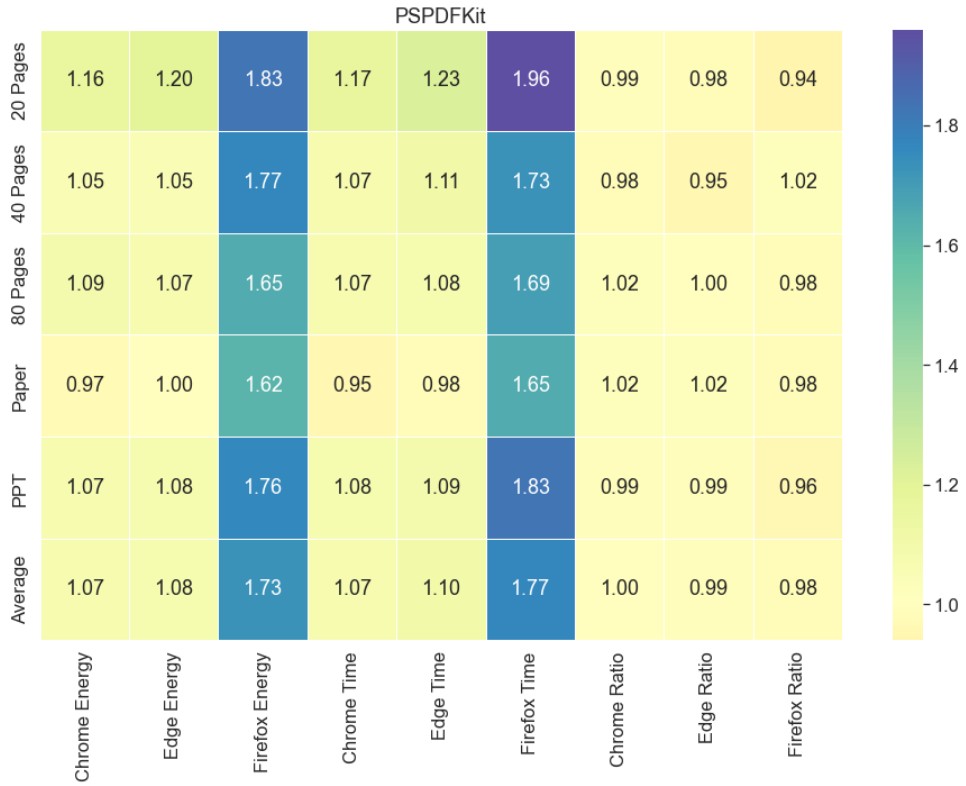


Figure 4.10: PSPDFKit: Heat map representing the proportion between Wasm and JS results in the three browsers.

Figure 4.11 shows the average percentage of energy gains between all JS and Wasm performances with *WasmBoy* (in blue) and *PSPDFKit* (in green) on each Web browser. The higher the bars, the more energetically efficient the system was using Wasm. For example, in *WasmBoy*, Wasm had an average energy increase of 20.88% using Google Chrome, compared to JS.

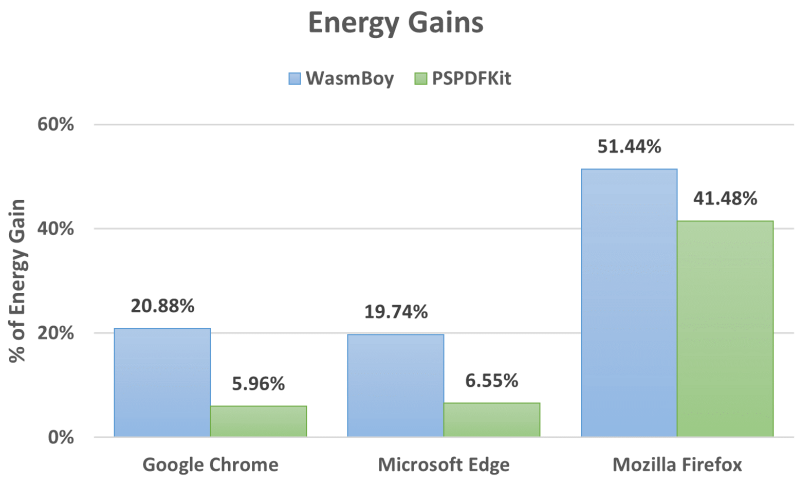


Figure 4.11: Average percentage of energy gains between JS and Wasm performances on real-world applications.

Figure 4.12 is the average Power (or ratio), in Watts, utilized by JS (in blue) and Wasm (in green) 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 consumed more energy per second (7 J/s). The lower the bars, the more energetically efficient the system is.

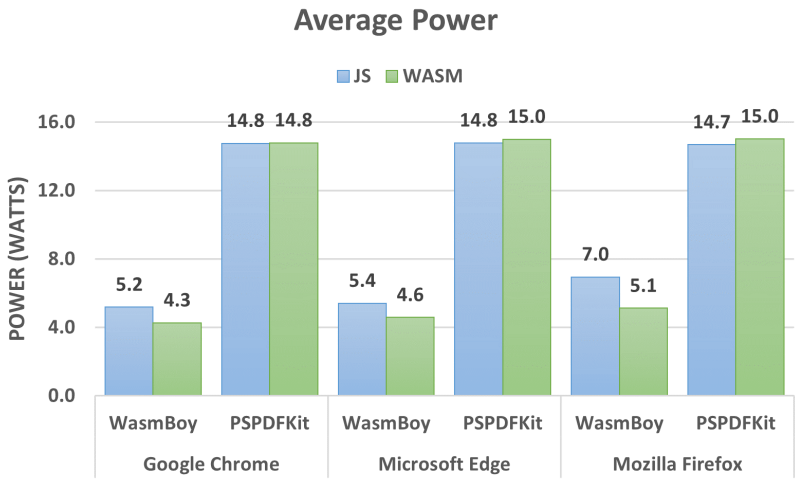


Figure 4.12: Average power, in Watts, used by JS and Wasm on real-world applications.

4.2 Discussion

The main focus of this study is to compare the energy and run-time performance between Wasm and JS. As we needed C source code to generate equivalent Wasm and JS programs, we included it as a reference point to compare the performance of Wasm and JS to one of the most run-time and energy efficient languages (Pereira, Couto, Ribeiro, et al., 2017). We also examine how its behavior changes between different environments, in which the Wasm has the best performance and how better that performance is.

As expected, with micro-benchmarks running in their virtual machines, C is by far more efficient than Wasm and JS, 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 JS 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 Wasm.

The results of Wasm and JS are very similar, both energy efficient and run-time performance. The smallest and largest performance difference between them is $0.959x$ and $1.173x$ (Figure 4.3), respectively. Nevertheless, if we look closely, it is possible to notice several performance differences.

Beginning with CPU energy consumption, in all thirty tests (ten benchmarks, each one with three different input sizes), Wasm was more energetically efficient than JS across twenty-six cases (87% of cases). It was 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 was always more efficient. Wasm was faster in twenty-five of thirty cases when it comes to run-time performance.

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

Figure 4.2 show us various violin plots allowing us to observe the consistency of each language. Although the results are very similar, it is possible to observe that, for example with `Sorting-Bead` and `Large` input, Wasm had a more consistent performance. On the other hand, with `Sorting-Identifiers`, JS was more consistent. This similarity is due to micro-benchmarks being very small programs that don't have the complexity to test the consistency of the languages.

Figure 4.3 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 JS. Using the *Small* and *Medium* inputs, Wasm presented the best 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 JS was more energetically efficient with the three inputs, and faster with *Small* and *Medium* input. Thus, the results show that currently, using micro-benchmarks Wasm is in general both more energy (**ARQ1**), and run-time efficient (**ARQ2**), however, the difference is not significantly.

When comparing the three inputs, is clear that Wasm is more energetically efficient and faster than JS 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 JS 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. The smaller difference (with larger inputs) for Wasm and JS may be due to the Node.js virtual machine, which applies dynamic optimizations to improve the JS speed, particularly the JIT compilation. Note that JIT optimization is more aggressive when the engine identifies hot loops, which frequently happens in micro-benchmarks. By contrast, the current Wasm virtual machine is very recent, thus optimization techniques for Wasm may not be mature yet.

With the micro-benchmarks approach without using Web browsers, we can say, looking at the last row (*Average row*) of Figure 4.3, that on average, Wasm is always more energetically efficient and faster than JS (further repeating **ARQ1/ARQ2**). Considering the average of all tests and all inputs, Wasm is $1.044x$ more CPU energy efficient than JS, $1.037x$ more DRAM energy efficient, and $1.036x$ more run-time efficient.

While the previous results seem small, we included Web browsers to micro-benchmarks solutions to provide additional outcomes, as shown in Figure 4.4. Yet, including Web browsers in the micro-benchmark framework didn't allow us to acquire different conclusions about the performance between Wasm and JS. The performance of these two languages continued similarly. Nonetheless, incorporating browsers helped us to comprehend the behavior of the three Web browsers with Wasm and JS. Concerning Web browsers, Google Chrome is always the more stable browser, both in energy and run-time efficiency. Moreover, using Chrome, Wasm is more efficient than JS most of the times, while Microsoft Edge and Mozilla Firefox have mixed results. For example, 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 `Large` input where it decreases from $22.08\ W$ to $21.60\ W$). Mozilla is always more efficient running JS except in the *Fasta* example. In the *Sorting-Circle* 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 *Sorting-Normal* solution is the only program where JS is always more efficient, except energetically using Edge. In *Sorting-Lexicographic*, JS and Wasm have equal performances except with Firefox, where Wasm had a poor performance.

In real-world benchmarks, the results collected are vastly different than micro-benchmarks. However, the real-world benchmark results aren't similar between the two (Figure 4.5 and Figure 4.8). It is possible to notice that the energy gap between JS and Wasm is more significant on *WasmBoy*. Also, the ratio differs between *WasmBoy* and *PSPDFKit*. *WasmBoy* solutions use less power to run the programs while,

in *PSPDFKit*, the ratio is similar.

To understand if there is an overall significant relevance 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 using Web browsers (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, non-normal 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* (Field, 2009) suggests using Rosenthal's formula (Rosenthal, 1991; Rosenthal & Rosnow, 2008) to compute a correlation and compare the correlation values against (Cohen, 2013) 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, 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 important relevance 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, we can affirm that Wasm is more efficient than JS on real-world applications, *WasmBoy* and *PSPDFKit*. We have shown that there are both significant improvements and a large effect size when using Wasm to increase the energy efficiency of programs **(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. Besides, Wasm has a predictable run-time performance. With JS, the performance generally increases with each iteration as it

is further optimized, however it can also decrease due to se-optimization. This scenario never occurs with Wasm and this is a big improvement.

Looking at run-time results, we can say that the outcomes are very different between micro and real-time applications, as occurred on energy results. While Wasm, on average, is 9.84% *slower* than JS in micro-benchmarks within a browser-based environment, the opposite occurs on real-world applications, where Wasm, on average, is 17.24% *faster* than JS **(ARQ2)**.

In order to understand the differences between benchmarks, we also observed that there is a significant difference between micro and real-world benchmarks performance, with an average energy gain of -5.18% and 24.34% for micro and real-world benchmarks, respectively. It shows 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, there is an overhead in calling Wasm from JS, which affects 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 realistic benchmarks, with an average energy gain - of the three browsers - of 30.69% and 18% for *WasmBoy* and *PSPDFKit*, respectively (Figure 4.11). 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 humanly writable, unlike ES6 that is the standardization of JS.

Additionally, we calculated the average percentage of energy gains between JS and Wasm to understand if the behavior was the same within the three browsers. As shown in Figure 4.11, *WasmBoy*, achieved energy gains 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 **(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 (de Macedo et al., 2020), 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 4.12, Wasm can be faster using less power per second with 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 JS over time by a large margin. Likely, Wasm is here to stay and revolutionize the Web.

4.3 Threats to Validity

The goal of this dissertation is to both measure and understand the energetic and run-time behaviour between the novel Wasm and matured JS languages. This section presents some threats to the validity of our study, separated into four categories (Cook et al., 1979).

Conclusion: While the difference between Wasm and JS is small in micro-benchmarks - even so, there is a evident consistency in favor of Wasm - 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. 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.

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 using micro-benchmarks outside Web browsers, and 5 times using micro-benchmarks with Web browsers and real-world applications. We calculated both median and mean values for each benchmark solution, 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 (de Macedo et al., 2020; de Oliveira Júnior et al., 2019; Hähnel et al., 2012; Lima et al., 2016; Pereira et al., 2020; Pereira, Couto, Ribeiro, et al., 2017, 2021; Pereira et al., 2016).

Construct: Firstly, 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 JS 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. Additionally, We analyzed the previous ten distinct micro-benchmarks scenarios across the two main languages of this research (Wasm and JS), three browsers, each with three input sizes, totaling 180 different measured cases. 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 concerns itself with the generalization of the results. 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 and exponential improvement, and thus the performance differences we have observed in this study to be further highlighted and distanced.

Chapter 5

Conclusions and Future Directions

In this thesis, we present a 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. We also considered two real applications: a Gameboy console emulator, *WasmBoy*, and a portable document format (PDF) viewer/editor, *PSPDFKit*. 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. Unsurprisingly, C continues to be the fastest and greenest programming language. However, between WebAssembly and JavaScript using micro-benchmarks, 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. 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.

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. For real applications, Wasm outperforms JS in all cases with a significant difference. With WasmBoy solutions, Wasm can be faster and, still, use less power, which means using less energy per second. What more could we possibly ask? Thus, we can say that Wasm, in its proper environment (Web browsers), is greener and faster than JS for a significant margin.

So, will Wasm change the web? Most likely yes. Wasm is still in its infancy and only time could tell us how it will evolve. In our wildest predictions, we see Wasm completely pushing out native apps from operating systems and crowning the Web browser, as the operating system of the twenty first century.

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[1] for researchers and practitioners to replicate and build upon.

Bibliography

Adobe. (2017). *End of life for adobe shockwave*. Retrieved November 10, 2021, from https://get.adobe.com/shockwave/

Anand, V., & Saxena, D. (2013). 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*. https://doi.org/10.1109/ICCIC.2013.6724273

Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D. L., Sehr, D., Biffle, C. L., & Yee, B. (2011). Language-independent sandboxing of just-in-time compilation and self-modifying code. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 355–366.

AssemblyScript. (2021). *Assembly script - a typescript-like language for webassembly.* Retrieved April 29, 2021, from https://www.assemblyscript.org/

Bergbom, J. (2018). Memory safety: Old vulnerabilities become new with webassembly.

Bergdahl, N., & Nouri, J. (2021). Covid-19 and crisis-prompted distance education in sweden. *Technology, Knowledge and Learning*, *26*(3), 443–459.

Bhatti, A., Akram, H., Basit, H. M., Khan, A. U., Raza, S. M., & Naqvi, M. B. (2020). E-commerce trends during covid-19 pandemic. *International Journal of Future Generation Communication and Networking*, *13*(2), 1449–1452.

Bonacini, L., Gallo, G., & Scicchitano, S. (2021). Working from home and income inequality: Risks of a 'new normal'with covid-19. *Journal of population economics*, *34*(1), 303–360.

Brügger, N. (2010). *Web history* (Vol. 56). Peter Lang.

Brügger, N. (2012). When the present web is later the past: Web historiography, digital history, and internet studies. *Historical Social Research / Historische Sozialforschung*, *37*(4 (142)), 102–117. http://www.jstor.org/stable/41756477

Bryant, D. (2017). *Why webassembly is a game changer for the web — and a source of pride for mozilla and firefox*. Retrieved April 29, 2021, from https://medium.com/mozilla-tech/why-webassembly-is-a-game-changer-for-the-web-and-a-source-of-pride-for-mozilla-and-firefox-dda80e4c43cb

Butkiewicz, M., Madhyastha, H. V., & Sekar, V. (2013). Characterizing web page complexity and its impact. *IEEE/ACM Transactions on Networking*, *22*(3), 943–956.

Calero, C., & Piattini, M. (2015). *Green in software engineering* (Vol. 3). Springer.

Castaman, A. S., & Rodrigues, R. A. (2020). Educação a distância na crise covid-19: Um relato de experiência. *Research, Society and Development*, *9*(6), e180963699–e180963699.

Castells, M., & Chemla, P. (2001). La galaxia internet.

Cheng, S., Karachalias, G., & Hoeglund, H. (to appear). Asterius: Bringing haskell to webassembly. *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming*.

Chetty, M., Tran, D., & Grinter, R. E. (2008). Getting to green: Understanding resource consumption in the home. *Proceedings of the 10th international conference on Ubiquitous computing*, 242–251.

Cohen, J. (2013). *Statistical power analysis for the behavioral sciences*. Academic press.

Contributors., W. (2021). *Webassembly use cases*. Retrieved October 19, 2021, from https://webassembly.org/docs/use-cases/

Cook, T. D., Campbell, D. T., & Day, A. (1979). *Quasi-experimentation: Design & analysis issues for field settings* (Vol. 351). Houghton Mifflin Boston.

Couto, M., Borba, P., Cunha, J., Fernandes, J. P., Pereira, R., & Saraiva, J. (2017). Products go green: Worst-case energy consumption in software product lines. *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, 84–93. https://doi.org/10.1145/3106195.3106214

Couto, M., Carção, T., Cunha, J., Fernandes, J. P., & Saraiva, J. (2014). Detecting anomalous energy consumption in android applications. *Brazilian Symposium on Programming Languages*, 77–91.

Couto, M., Pereira, R., Ribeiro, F., Rua, R., & Saraiva, J. (2017). Towards a green ranking for programming languages. *Proceedings of the 21st Brazilian Symposium on Programming Languages*.

Couto, M., Saraiva, J., & Fernandes, J. P. (2020). Energy refactorings for android in the large and in the wild. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 217–228. https://doi.org/10.1109/SANER48275.2020.9054858

Cruz, L., & Abreu, R. (2017). Performance-based guidelines for energy efficient mobile applications. *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 46–57.

De', R., Pandey, N., & Pal, A. (2020). Impact of digital surge during covid-19 pandemic: A viewpoint on research and practice [Impact of COVID-19 Pandemic on Information Management Research and Practice: Editorial Perspectives]. *International Journal of Information Management*, *55*, 102171. https://doi.org/https://doi.org/10.1016/j.ijinfomgt.2020.102171

de Macedo, J., Aloísio, J., Gonçalves, N., Pereira, R., & Saraiva, J. (2020). Energy wars - chrome vs. firefox: Which browser is more energy efficient? *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 159–165. https://doi.org/10.1145/3417113.3423000

de Oliveira Júnior, W., dos Santos, R. O., de Lima Filho, F. J. C., de Araújo Neto, B. F., & Pinto, G. H. L. (2019). Recommending energy-efficient java collections. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 160–170.

Ding, F., Xia, F., Zhang, W., Zhao, X., & Ma, C. (2011). Monitoring energy consumption of smartphones. *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, 610–613. https://doi.org/10.1109/iThings/CPSCom.2011.122

Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A., & Stefan, D. (2019). Position paper: Progressive memory safety for webassembly. *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 1–8.

Donovan, A., Muth, R., Chen, B., & Sehr, D. (2010). Pnacl: Portable native client executables. *Google White Paper*.

Eberhardt, C., & Price, C. (2018). *White paper: The web assembles*. Retrieved April 29, 2021, from https://blog.scottlogic.com/2018/04/24/the-web-assembles.html

Field, A. (2009). *Discovering statistics using spss*. Sage publications.

Fu, W., Lin, R., & Inge, D. (2018). Taintassembly: Taint-based information flow control tracking for webassembly. *arXiv preprint arXiv:1802.01050*.

Georgiou, S., Kechagia, M., Louridas, P., & Spinellis, D. (2018). What are your programming language's energy-delay implications? *Proceedings of the 15th International Conference on Mining Software Repositories*, 303–313.

Georgiou, S., & Spinellis, D. (2020). Energy-delay investigation of remote inter-process communication technologies. *Journal of Systems and Software*, *162*, 110506.

Group, W. C. (2017a). *Binary format - webassembly*. Retrieved December 13, 2021, from https://webassembly.github.io/spec/core/binary/index.html#why-a-binary-encoding-instead-of-a-text-only-representation

Group, W. C. (2017b). *Faq - webassembly*. Retrieved December 13, 2021, from https://webassembly.org/docs/faq/

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. (2017). Bringing the web up to speed with webassembly. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 185–200. https://doi.org/10.1145/3062341.3062363

Hähnel, M., Döbel, B., Völp, M., & Härtig, H. (2012). Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Performance Evaluation Review*, *40*(3), 13–17.

Herman, D., Wagner, L., & Zakai, A. (2014). *Asm.js*. Retrieved November 10, 2021, from http://asmjs.org/spec/latest/

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

Hilbig, A., Lehmann, D., & Pradel, M. (2021). An empirical study of real-world webassembly binaries: Security, languages, use cases. *Proceedings of the Web Conference 2021*, 2696–2708.

Hilty, L. M., Arnfalk, P., Erdmann, L., Goodman, J., Lehmann, M., & Wäger, P. A. (2006). The relevance of information and communication technologies for environmental sustainability–a prospective simulation study. *Environmental Modelling & Software*, *21*(11), 1618–1629.

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

Intel, I. (2009). Intel architecture software developer's manual volume 3: System programming guide.

Jangda, A., Powers, B., Berger, E. D., & Guha, A. (2019). Not so fast: Analyzing the performance of webassembly vs. Native code. *arXiv*.

Khan, F., Foley-Bourgon, V., Kathrotia, S., Lavoie, E., & Hendren, L. (2015). Using JavaScript and WebCL for numerical computations: A comparative study of native and web technologies. *ACM SIGPLAN Notices*, *50*(2), 91–102. https://doi.org/10.1145/2661088.2661090

Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K., & Ou, Z. (2018). Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, *3*(2), 1–26.

Kim, R. Y. (2020). The impact of covid-19 on consumers: Preparing for digital sales. *IEEE Engineering Management Review*, *48*(3), 212–218.

Köhler, A., & Erdmann, L. (2004). Expected environmental impacts of pervasive computing. *Human and Ecological Risk Assessment*, *10*(5), 831–852.

Laperdrix, P., Rudametkin, W., & Baudry, B. (2016). Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. *2016 IEEE Symposium on Security and Privacy (SP)*, 878–894.

Lattner, C., & Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 75–86.

Lehmann, D., Kinder, J., & Pradel, M. (2020). Everything old is new again: Binary security of webassembly. *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 217–234.

Lehmann, D., & Pradel, M. (2019). Wasabi: A framework for dynamically analyzing webassembly. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1045–1058.

Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J., Roberts, L. G., & Wolff, S. (2009). A brief history of the internet. *ACM SIGCOMM Computer Communication Review*, *39*(5), 22–31.

Li, D., Hao, S., Halfond, W. G. J., & Govindan, R. (2013). Calculating source line level energy information for android applications. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 78–89. https://doi.org/10.1145/2483760.2483780

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

Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., & Fernandes, J. P. (2019). On haskell and energy efficiency. *Journal of Systems and Software*, *149*.

Liu, K., Pinto, G., & Liu, Y. D. (2015). Data-oriented characterization of application-level energy optimization. *International Conference on Fundamental Approaches to Software Engineering*, 316–331.

L.Technologies. (2021). *Cheerp — c/c++ to webassembly compiler*. Retrieved January 26, 2021, from https://leaningtech.com/pages/cheerp.html

McFadden, B., Lukasiewicz, T., Dileo, J., & Engler, J. (2018). Security chasms of wasm. *NCC Group Whitepaper*.

Melville, N. P. (2010). Information systems innovation for environmental sustainability. *MIS quarterly*, 1–21.

Microsoft. (2017). *Activex controls*. Retrieved November 10, 2021, from https://msdn.microsoft.com/en-us/library/aa751968(v=vs.85).aspx

Mingay, S. (2007). Green it: The new industry shock wave. *Gartner RAS Research Note G*, *153703*(7).

Mozilla. (2021). *Understanding webassembly text format*. Retrieved January 26, 2021, from https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

Murugesan, S. (2008). Harnessing green it: Principles and practices. *IT professional*, *10*(1), 24–33.

Musch, M., Wressnegger, C., Johns, M., & Rieck, K. (2019). New kid on the web: A study on the prevalence of webassembly in the wild. *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 23–42.

Nanz, S., & Furia, C. A. (2015). A comparative study of programming languages in rosetta code. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, *1*, 778–788.

Narayan, S., Disselkoen, C., Moghimi, D., Cauligi, S., Johnson, E., Gang, Z., Vahldiek-Oberwagner, A., Sahita, R., Shacham, H., Tullsen, D., et al. (2021). Swivel: Hardening webassembly against spectre. *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

Oliveira, W., Oliveira, R., & Castor, F. (2017). A study on the energy consumption of android app development approaches. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 42–52.

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

Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Helping programmers improve the energy efficiency of source code. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 238–240.

Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. P., & Saraiva, J. (2020). Spelling out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software*, *161*, 110463.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Energy efficiency across programming languages: How do energy, time, and memory relate? *SLE 2017 -*

Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2017, 256–267. https://doi.org/10.1145/3136014.3136031

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2021). Ranking programming languages by energy efficiency. *Science of Computer Programming*, *205*, 102609. https://doi.org/10.1016/j.scico.2021.102609

Pereira, R., Couto, M., Saraiva, J., Cunha, J., & Fernandes, J. P. (2016). The influence of the Java collection framework on overall energy consumption. *Proceedings - International Conference on Software Engineering*, 15–21. https://doi.org/10.1145/2896967.2896968

Pinto, G., & Castor, F. (2017). Energy efficiency: A new concern for application software developers. *Communications of the ACM*, *60*(12), 68–75.

Protzenko, J., Beurdouche, B., Merigoux, D., & Bhargavan, K. (2019). Formally verified cryptographic web applications in webassembly. *2019 IEEE Symposium on Security and Privacy (SP)*, 1256–1274.

Purwanto, A., Asbari, M., Fahlevi, M., Mufid, A., Agistiawati, E., Cahyono, Y., & Suryani, P. (2020). Impact of work from home (wfh) on indonesian teachers performance during the covid-19 pandemic: An exploratory study. *International Journal of Advanced Science and Technology*, *29*(5), 6235–6244.

Renner, J., Cauligi, S., & Stefan, D. (2018). Constant-time webassembly. *Principles of Secure Compilation*.

Rosenthal, R. (1991). Meta-analytic procedures for social research. 1984, beverly hills.

Rosenthal, R., & Rosnow, R. L. (2008). *Essentials of behavioral research: Methods and data analysis*.

Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., & Rajwan, D. (2012). Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, *32*(2), 20–27. https://doi.org/10.1109/MM.2012.12

Sandhu, P., Herrera, D., & Hendren, L. (2018). Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in javascript and webassembly. *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, 1–13.

Spiess, P., & Gurgone, G. (2018). *A real-world webassembly benchmark*. Retrieved October 20, 2021, from https://pspdfkit.com/blog/2018/a-real-world-webassembly-benchmark/

Stefanoski, K., Karadimche, A., & Dimitrievski, I. (2019). PERFORMANCE COMPARISON OF C ++ AND JAVASCRIPT (NODE . JS – V8 ENGINE). (September).

Sun, J., Cao, D., Liu, X., Zhao, Z., Wang, W., Gong, X., & Zhang, J. (2019). Selwasm: A code protection mechanism for webassembly. *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 1099–1106.

Szanto, A., Tamm, T., & Pagnoni, A. (2018). Taint tracking for webassembly. *arXiv preprint arXiv:1807.08349*.

Turner, A. (2018). *Webassembly is fast: A real-world benchmark of webassembly vs. es6*. Retrieved January 26, 2021, from https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193

W3C. (2019). *World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation.* Retrieved January 26, 2021, from https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en

Wallace, E. (2017). *Webassembly cut figma's load time by 3x*. Retrieved December 13, 2021, from https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/

Wang, H. J., Fan, X., Howell, J., & Jackson, C. (2007). Protection and communication abstractions for web browsers in mashupos. *ACM SIGOPS Operating Systems Review*, *41*(6), 1–16.

Wang, W., Ferrell, B., Xu, X., Hamlen, K. W., & Hao, S. (2018). Seismic: Secure in-lined script monitors for interrupting cryptojacks. *European Symposium on Research in Computer Security*, 122–142.

Watt, C., Renner, J., Popescu, N., Cauligi, S., & Stefan, D. (2019a). Ct-wasm: Type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, *3*(POPL). https://doi.org/10.1145/3290390

Watt, C., Renner, J., Popescu, N., Cauligi, S., & Stefan, D. (2019b). Ct-wasm: Type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages*, *3*(POPL), 1–29.

Watt, C., Rossberg, A., & Pichon-Pharabod, J. (2019). Weakening webassembly. *Proceedings of the ACM on Programming Languages*, *3*(OOPSLA), 1–28.

Weaver, V. M., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., & Moore, S. (2012). Measuring energy and power with papi. *2012 41st international conference on parallel processing workshops*, 262–268.

WebAssembly. (2015). *Wabt: The webassembly binary toolkit*. Retrieved January 26, 2021, from https://github.com/WebAssembly/wabt

WebAssembly. (2017a). *Webassembly*. Retrieved April 29, 2021, from https://webassembly.org/

WebAssembly. (2017b). *Webassembly security*. Retrieved April 29, 2021, from https://webassembly.org/docs/security/

Wirfs-Brock, A., & Eich, B. (2020). Javascript: The first 20 years. *Proceedings of the ACM on Programming Languages*, *4*(HOPL), 1–189.

Yan, Y., Tu, T., Zhao, L., Zhou, Y., & Wang, W. (2021). Understanding the performance of webassembly applications. *Proceedings of the 21st ACM Internet Measurement Conference*, 533–549.

Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., & Fullagar, N. (2009). Native client: A sandbox for portable, untrusted x86 native code. *2009 30th IEEE Symposium on Security and Privacy*, 79–93.

Yue, C., & Wang, H. (2009). Characterizing insecure javascript practices on the web. *Proceedings of the 18th International Conference on World Wide Web*, 961–970.

Zakai, A. (2018). Fast physics on the web using c++, javascript, and emscripten. *Computing in Science Engineering*, *20*(1), 11–19. https://doi.org/10.1109/MCSE.2018.110150345

Zakai, A. (2011). Emscripten: An LLVM-to-JavaScript compiler. *SPLASH'11 Compilation - Proceedings of OOPSLA'11, Onward! 2011, GPCE'11, DLS'11, and SPLASH'11 Companion*, 301–312. https://doi.org/10.1145/2048147.2048224

Zoom. (2021). *Zoom web sdk*. Retrieved December 13, 2021, from https://marketplace.zoom.us/docs/sdk/native-sdks/web

Appendix A

All Benchmark Results

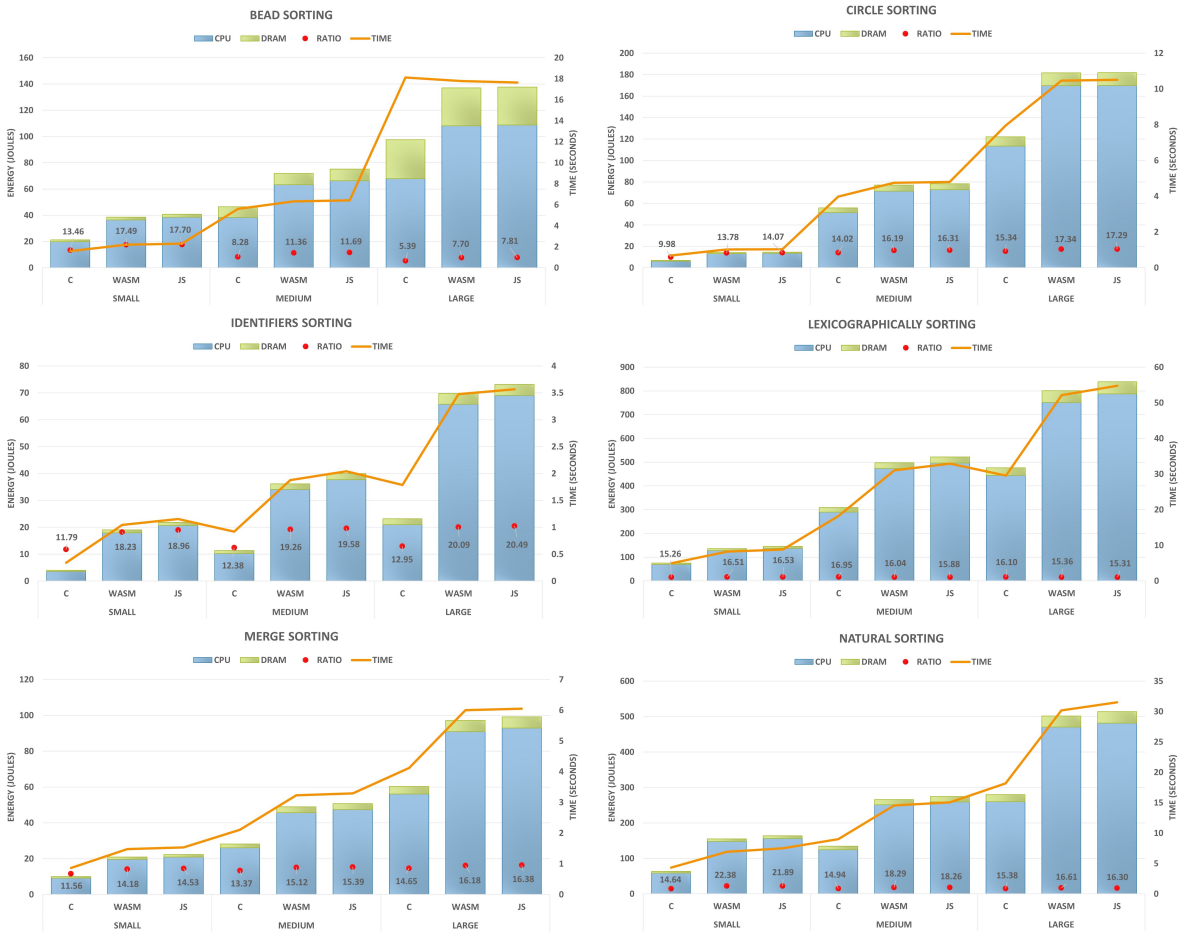


Figure A.1: Energy consumed by the CPU and DRAM for each benchmark with the three input sizes and the respective execution times and power values.

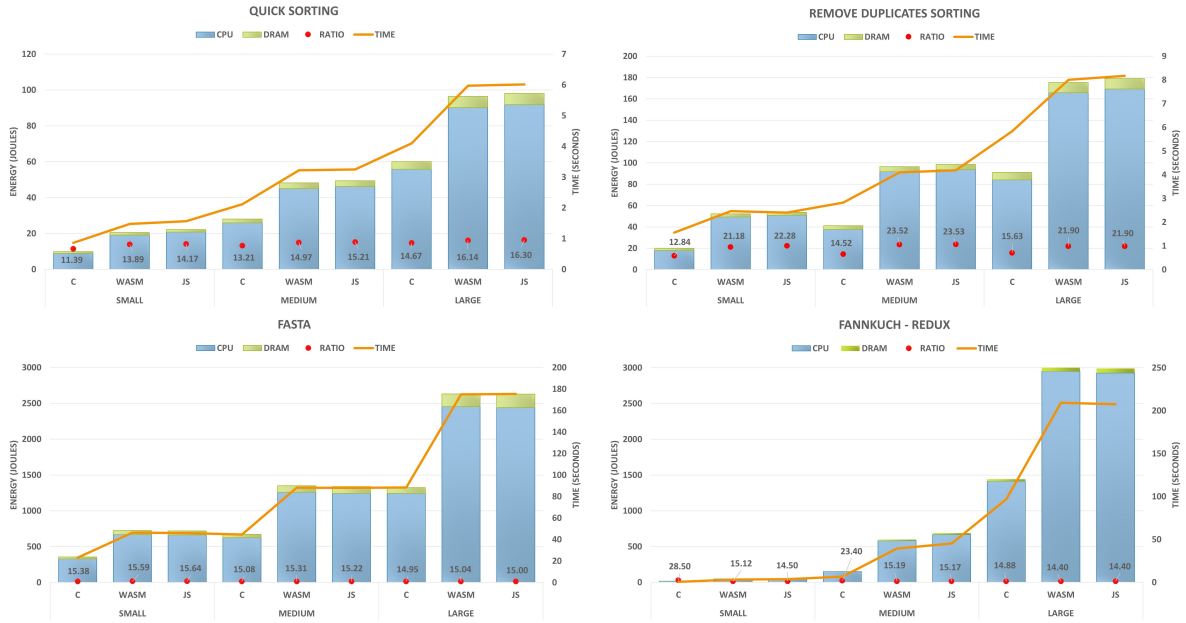


Figure A.1: Energy consumed by the CPU and DRAM for each benchmark with the three input sizes and the respective execution times and power values.

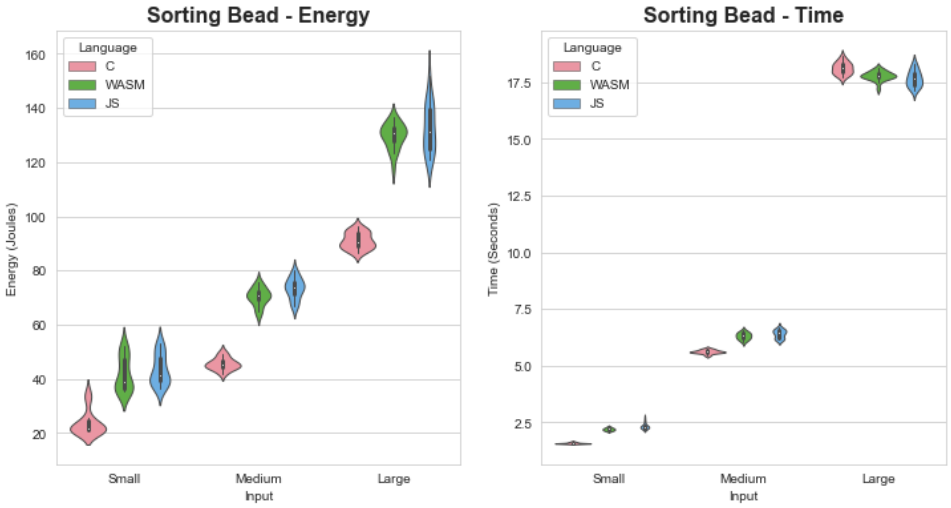


Figure A.2: Violin plots of energy consumed by each benchmark execution with the three input sizes and the respective execution times.

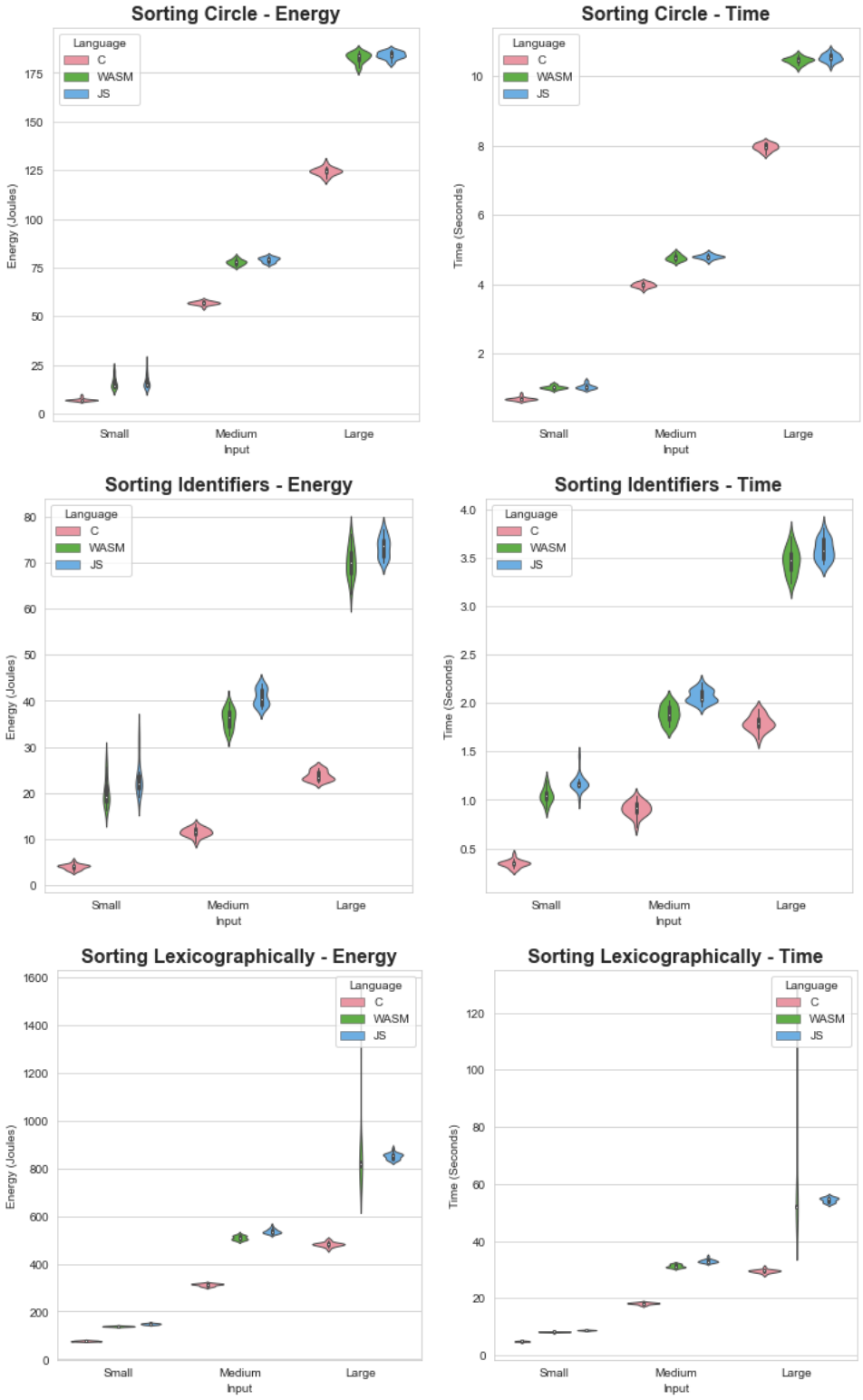


Figure A.2: Violin plots of energy consumed by each benchmark execution with the three input sizes and the respective execution times.

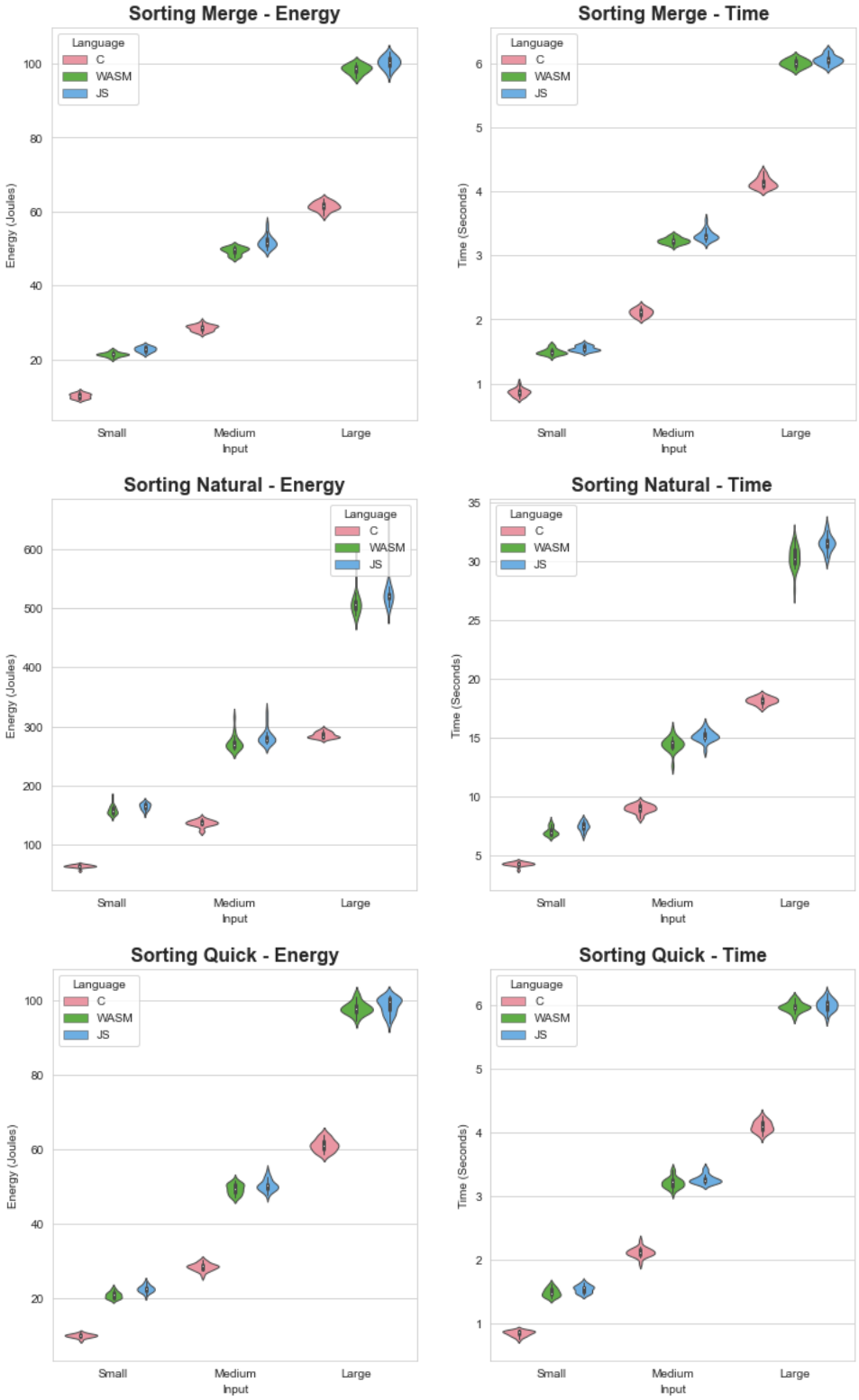


Figure A.2: Violin plots of energy consumed by each benchmark execution with the three input sizes and the respective execution times.

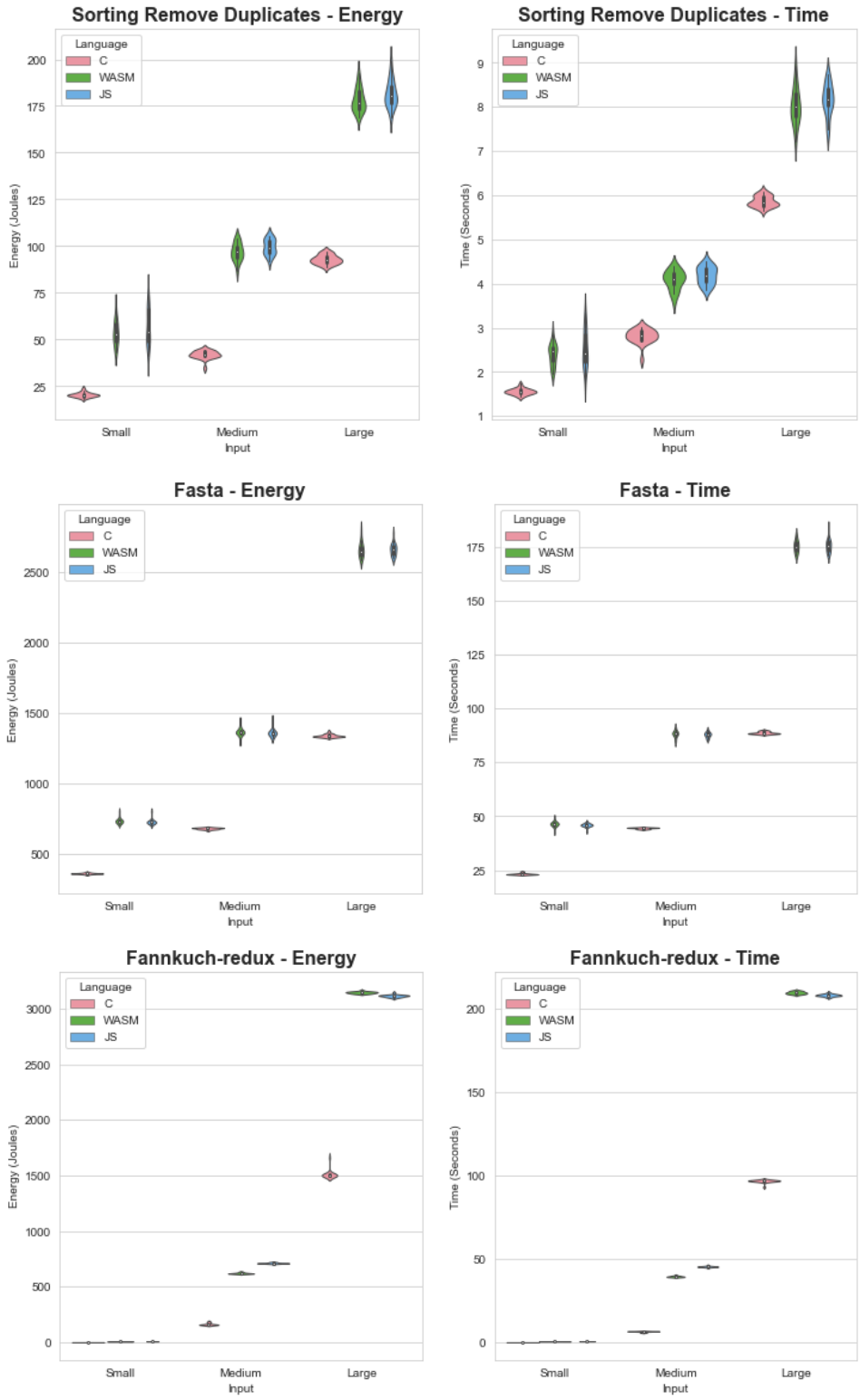


Figure A.2: Violin plots of energy consumed by each benchmark execution with the three input sizes and the respective execution times.

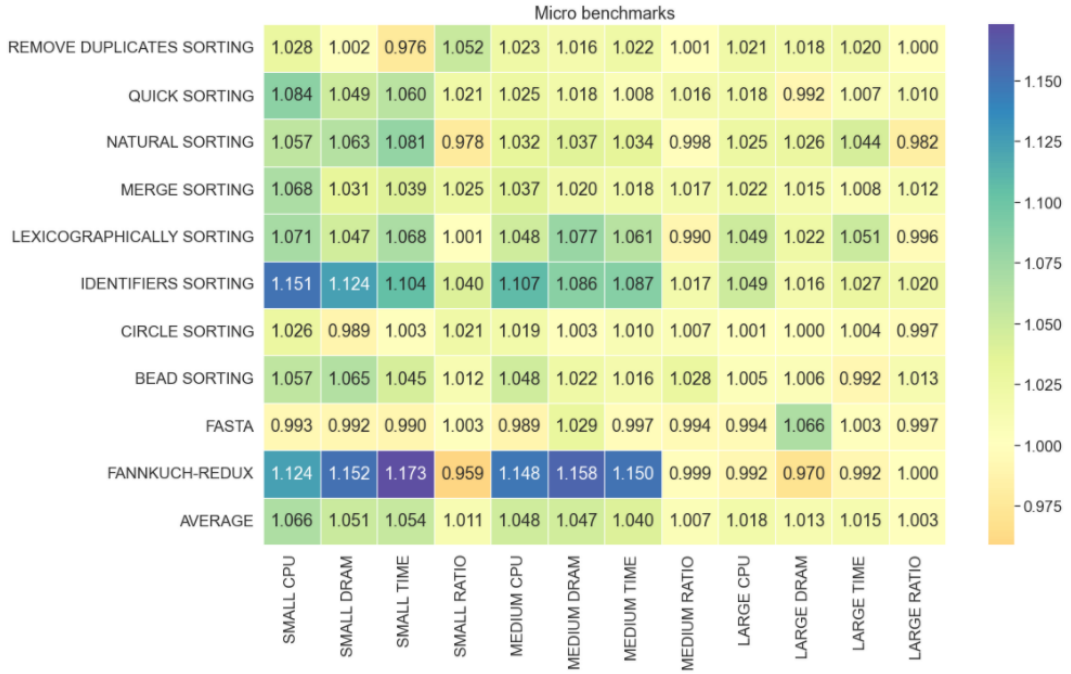


Figure A.3: Heat map representing the proportion between WebAssembly and JavaScript results with the three input sizes.

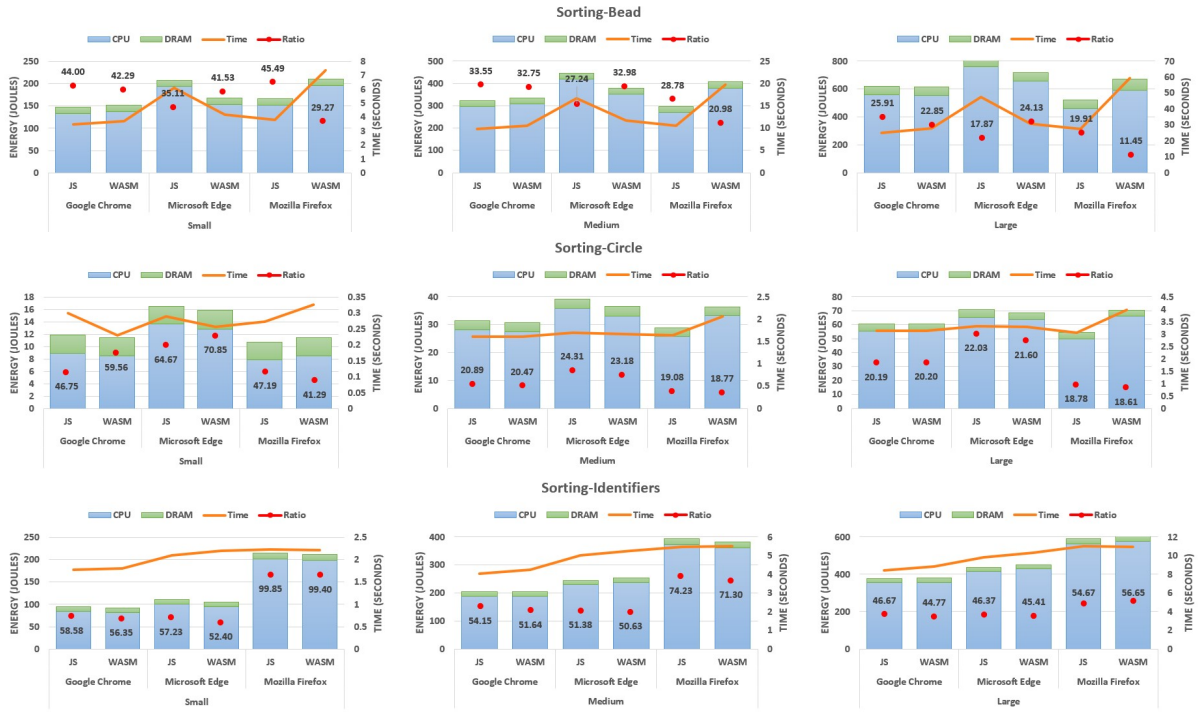


Figure A.4: Energy consumed by each browser for each benchmark with the three input sizes and the respective execution times and ratio values.

Figure A.4: Energy consumed by each browser for each benchmark with the three input sizes and the respective execution times and ratio values.

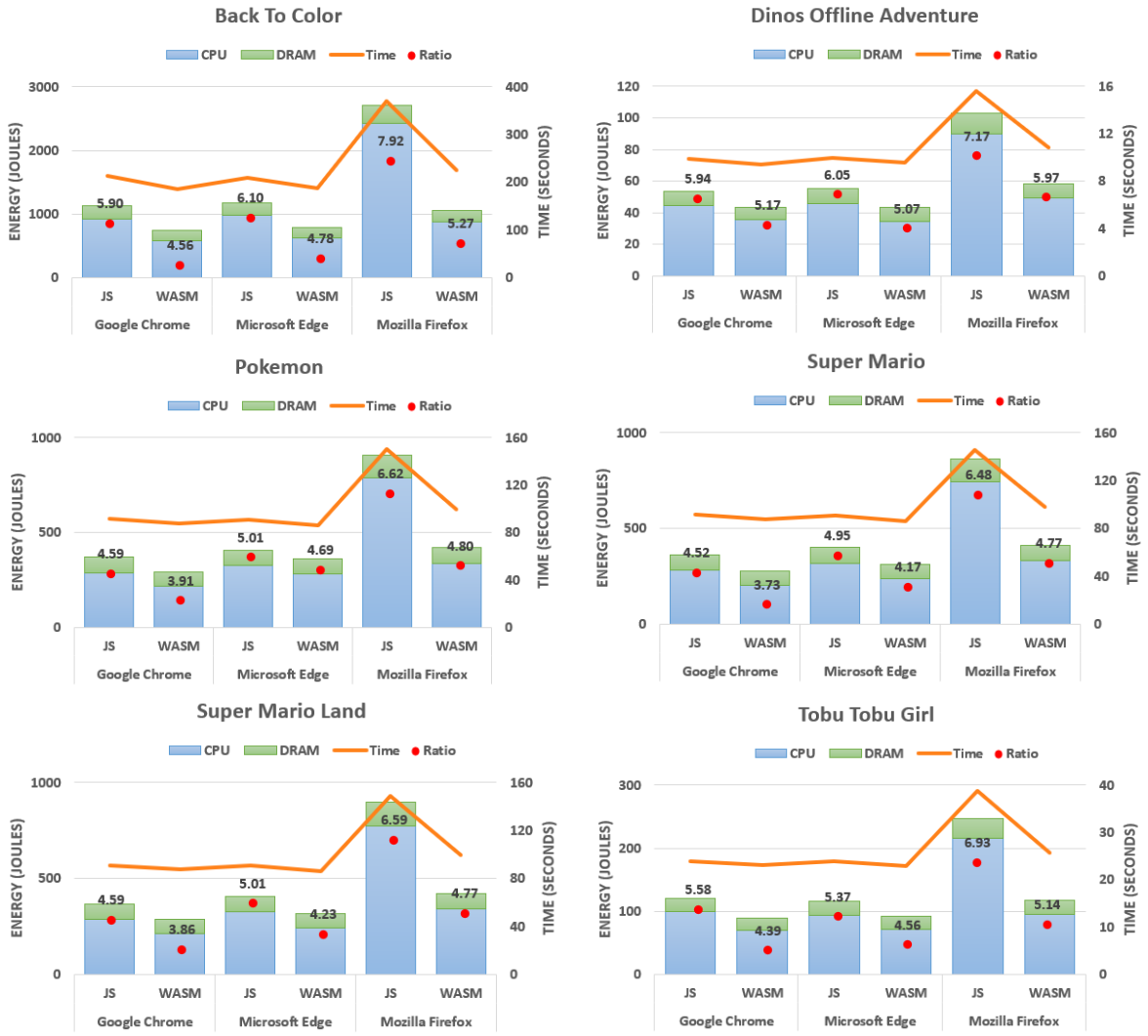


Figure A.5: WasmBoy: Energy consumed and run-time by each Game in each Web browser and the respective ratio values.

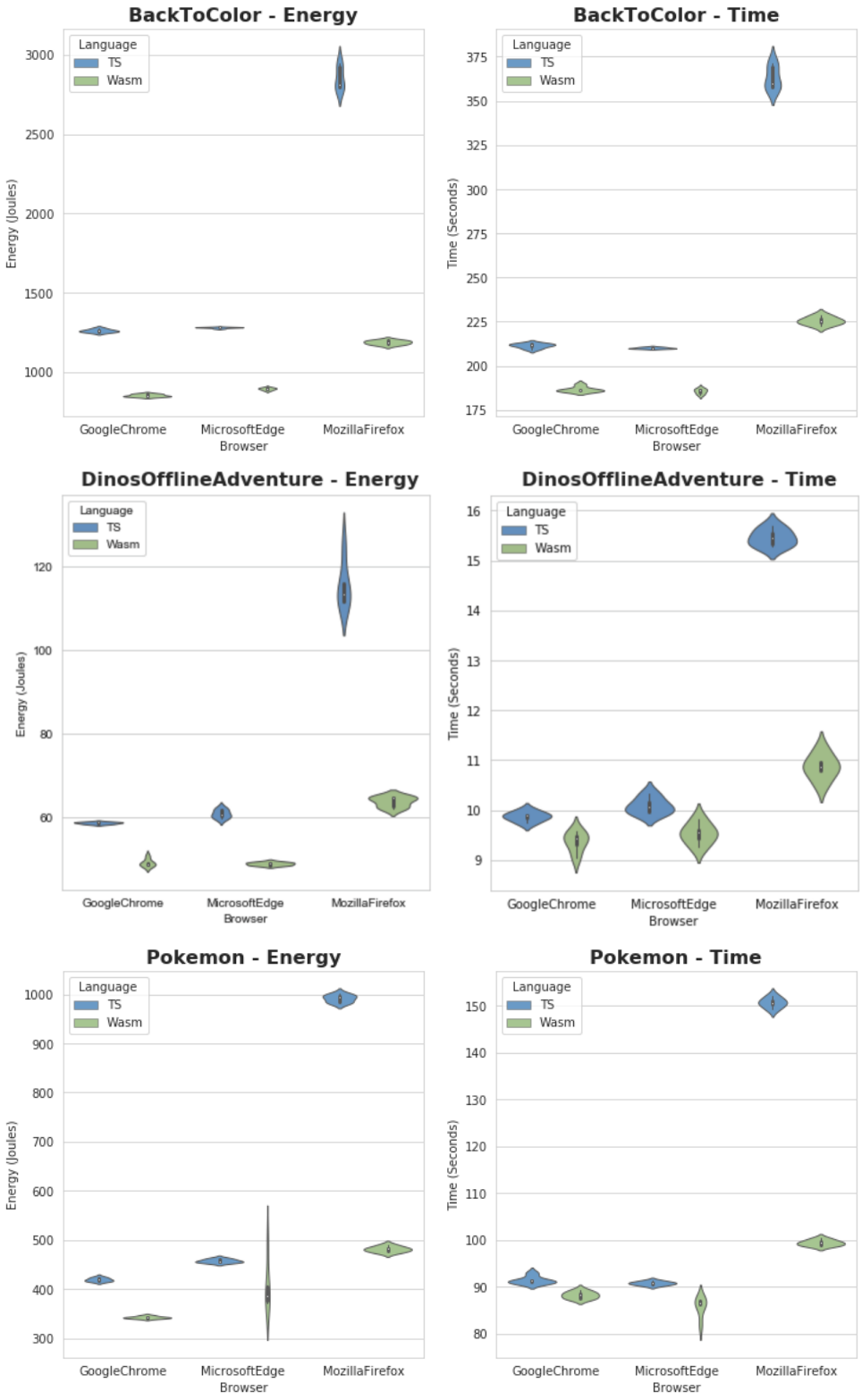


Figure A.6: WasmBoy: Violin plots of energy consumed and run-time by each Game execution in each Web browser and the respective ratio values.

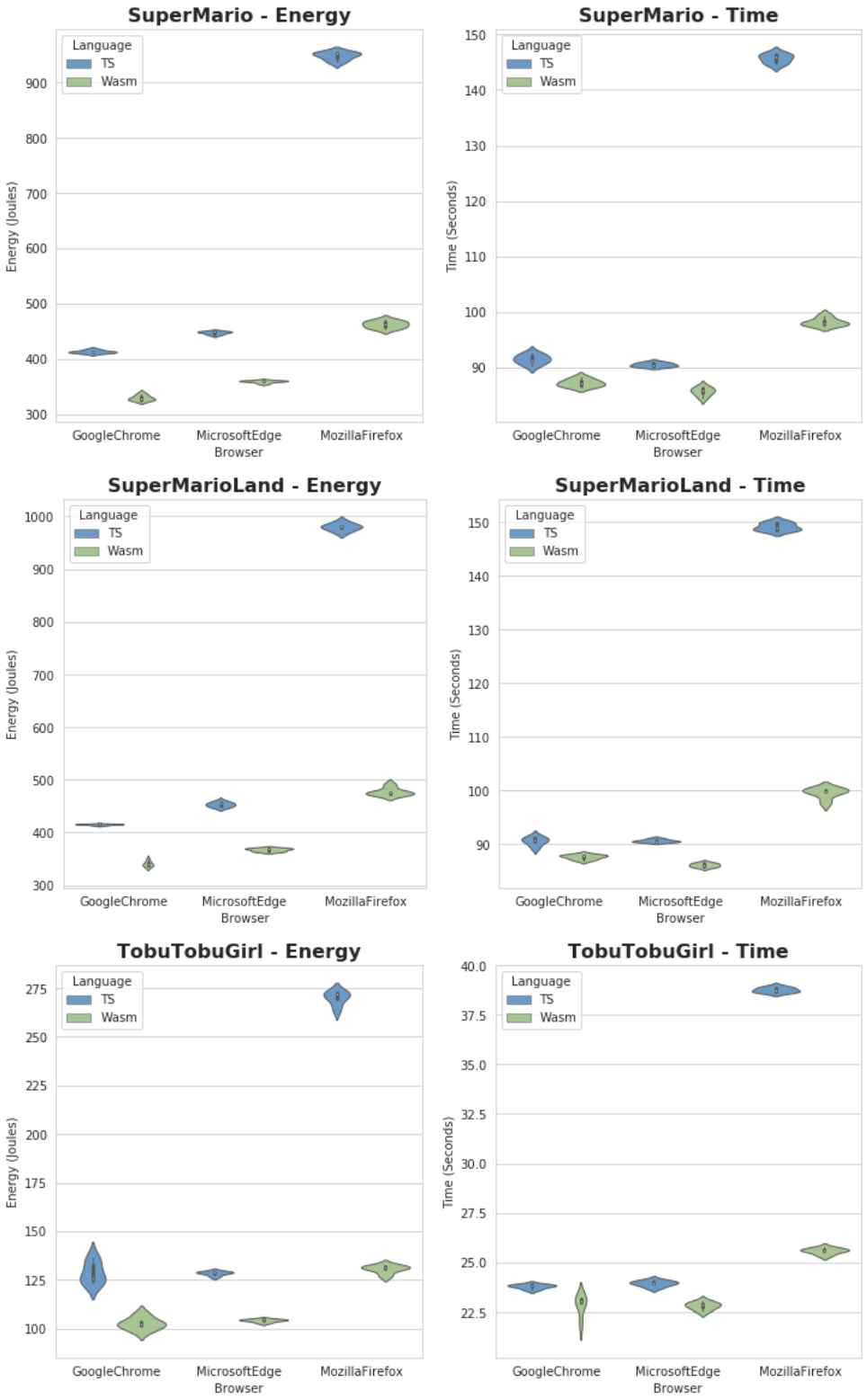


Figure A.6: WasmBoy: Violin plots of energy consumed and run-time by each Game execution in each Web browser and the respective ratio values.

Figure A.7: WasmBoy: Heat map representing the proportion between WebAssembly and JavaScript results in the three browsers.

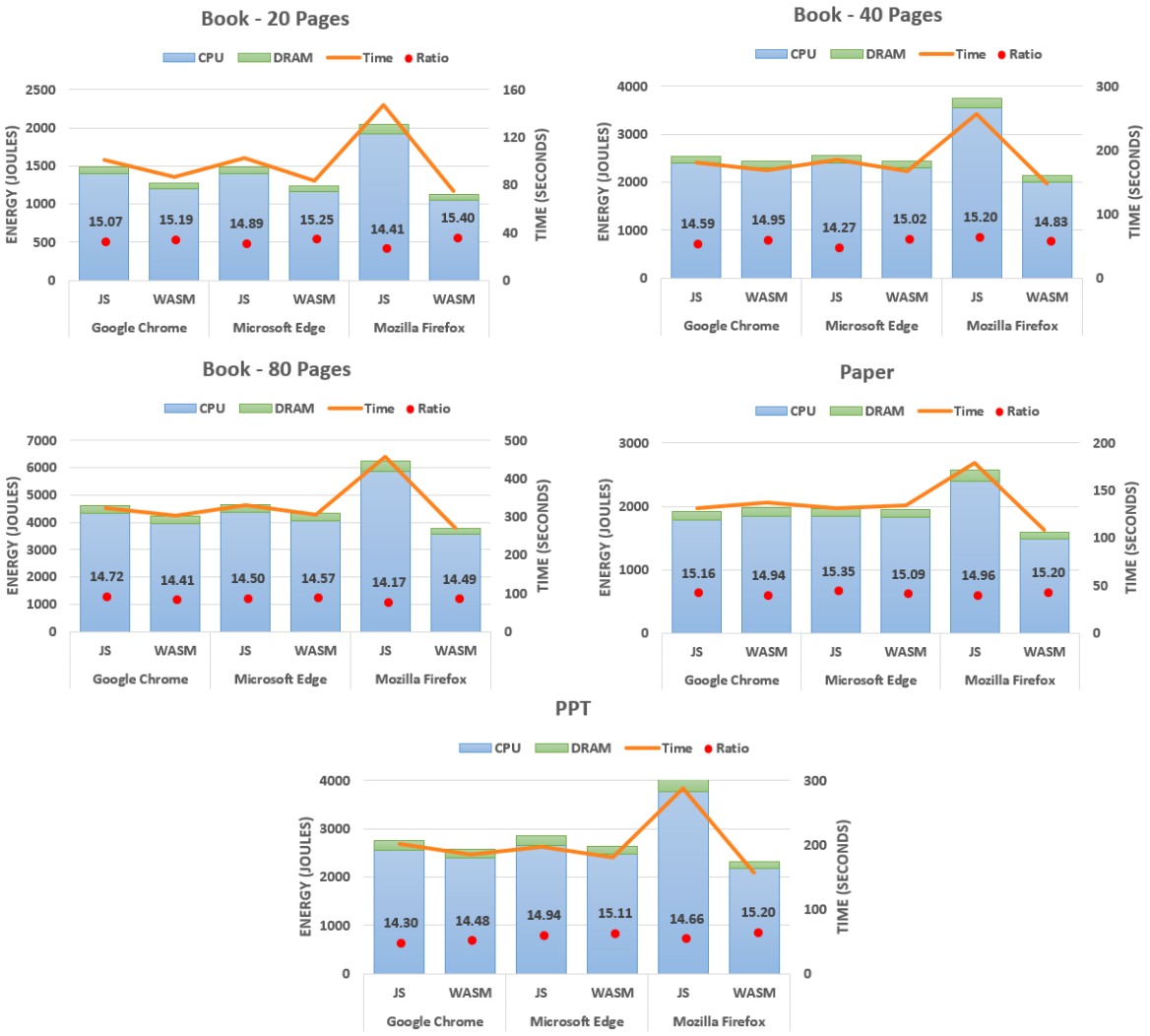


Figure A.8: PSPDFKit: Energy consumed and run-time by each PDF in each Web browser and the respective ratio values.

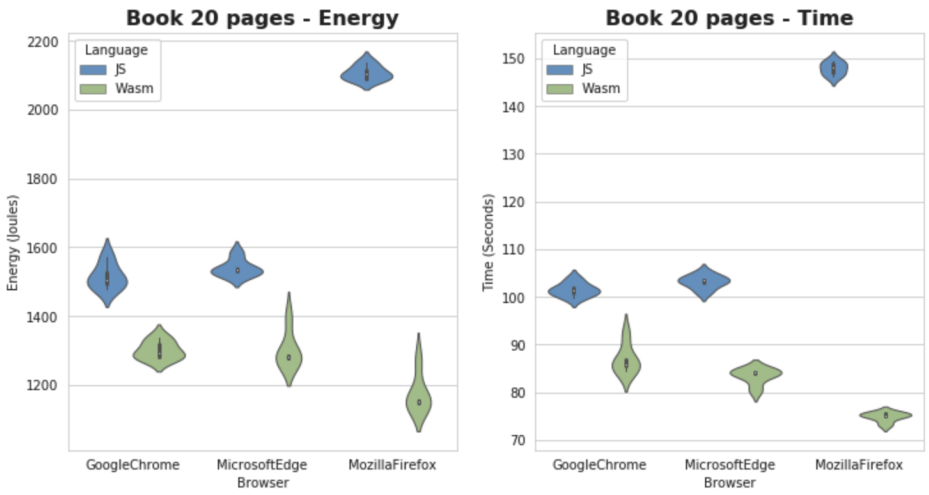


Figure A.8: PSPDFKit: Violin plots of energy consumed and run-time by each PDF execution in each Web browser and the respective ratio values.

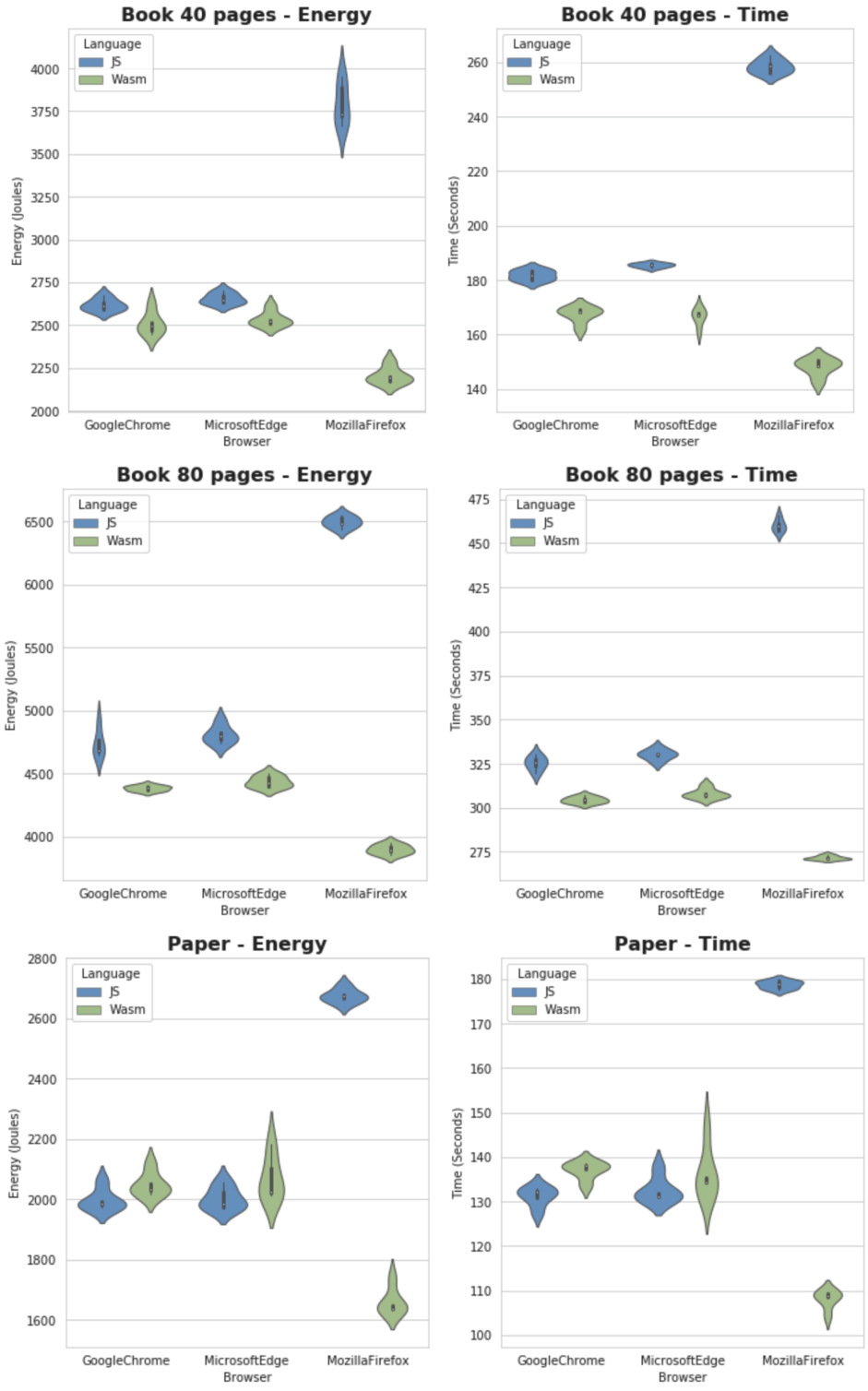


Figure A.8: PSPDFKit: Violin plots of energy consumed and run-time by each program execution in each Web browser and the respective ratio values.

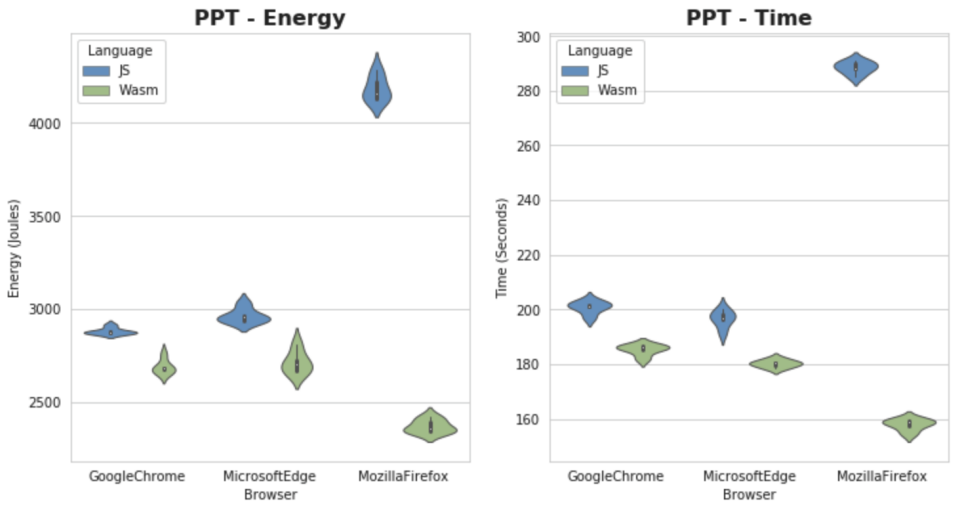


Figure A.8: PSPDFKit: Violin plots of energy consumed and run-time by each program execution in each Web browser and the respective ratio values.

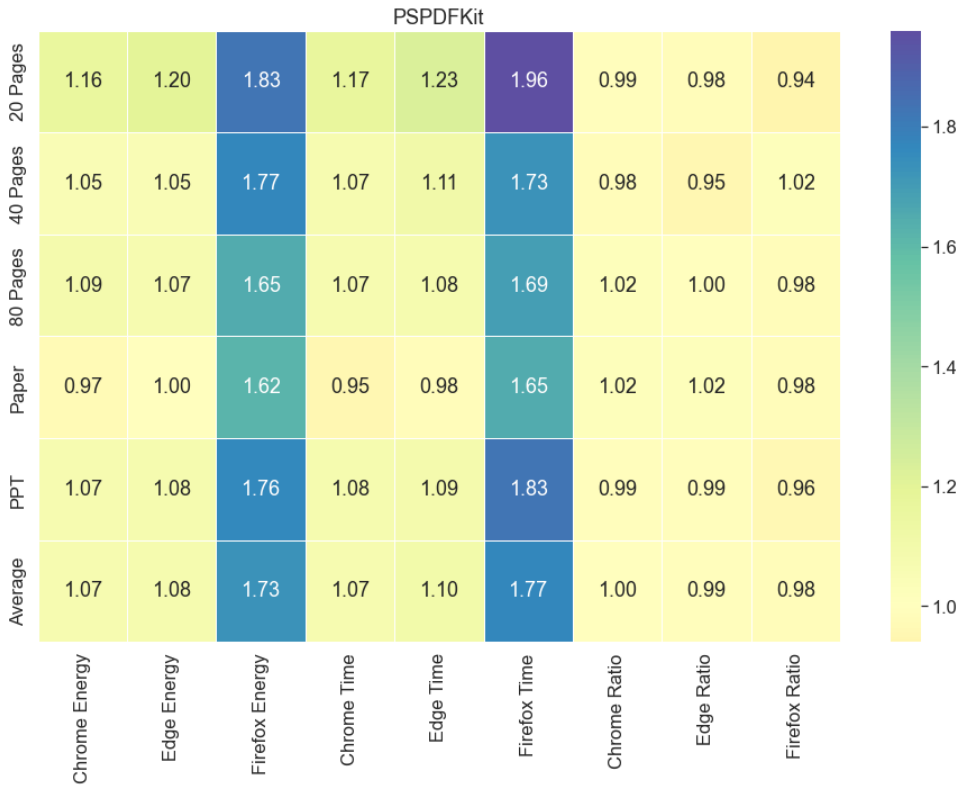


Figure A.9: PSPDFKit: Heat map representing the proportion between WebAssembly and JavaScript results in the three browsers.

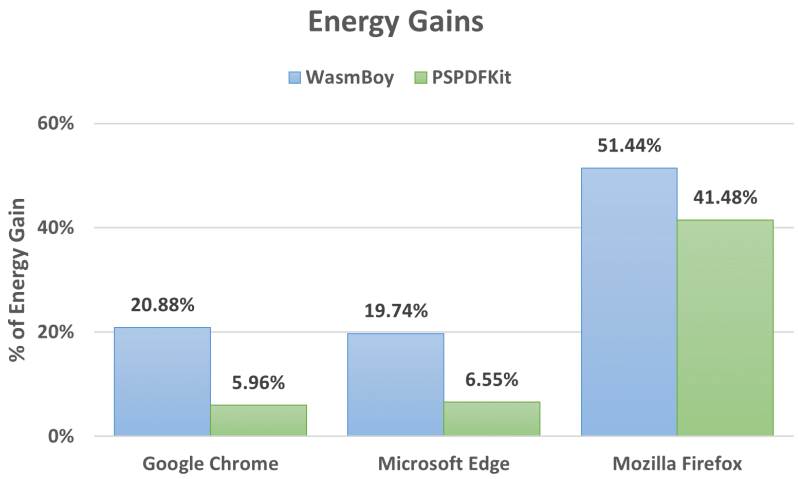


Figure A.10: Average percentage of energy gains between JS and Wasm performances on real-world applications.

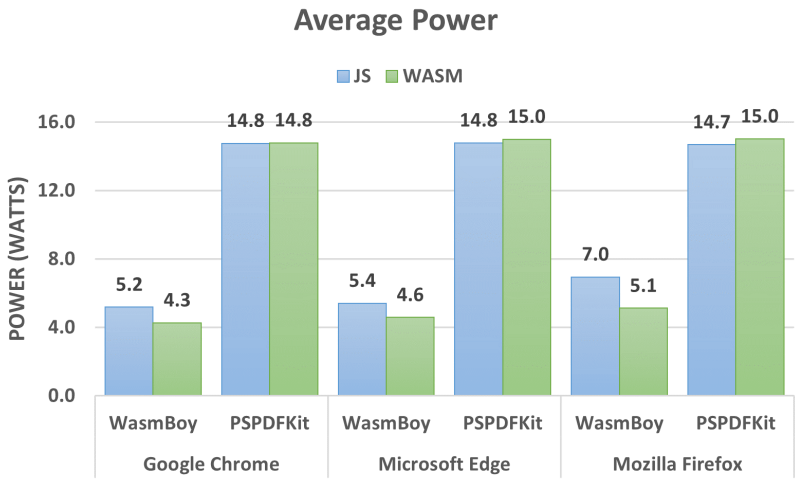


Figure A.11: Average power, in Watts, used by JS and Wasm on real-world applications.