
**FOM Hochschule für Oekonomie & Management**

university location Bonn

## Term paper

in the study course Business Informatics

as part of the course
**IT-Infrastructure**

on the subject

# Performance comparison between WASM and JavaScript based on code implementing computationally intensive operations on the clientside

by

Mads Fuchs, Nils Rüber, Janis Wiesen

Advisor: Christian Frank
Matriculation Numbers: 675314, 674514, 670300
Submission: 29.02.2024

# Table of Contents

# List of Figures

# List of Tables

# Listings

# 1 Introduction

The evolution of web technology has continuously pushed the boundaries of what can be achieved within web applications, particularly in terms of performance and computational capabilities. One of the latest advancements in this area is Web Assembly (WASM), a binary instruction format designed to enable high-performance code execution on web browsers. As the demand for sophisticated web applications increases, it is imperative to understand the comparative advantages of WASM (WASM) over traditional JavaScript (JavaScript), particularly in scenarios involving computationally intensive operations[1,2].

JavaScript has been the predominant language for web development for the last decades. However, the interpreted nature and runtime environment of JavaScript introduce performance bottlenecks, especially when dealing with complex computations. Therefore, it is important to consider the benefits of using WASM for implementing client-side logic. WASM offers a promising alternative by providing a low-level bytecode format that can be efficiently compiled and executed across different platforms[3,4].

The aim of this thesis is to investigate the performance differences between Web Assembly and JavaScript in client-side applications, with a specific focus on computationally intensive operations. Through rigorous experimentation and analysis, we aim to determine whether Web Assembly provides a distinct advantage over JavaScript in terms of efficiency.

To achieve this aim, the research will investigate various aspects of performance comparison, such as CPU utilization, memory usage, and DOM manipulation. We will conduct benchmark tests and empirical evaluations to provide concrete insights into the relative strengths and weaknesses of both approaches.

Furthermore, this research will not only clarify the technical aspects of Web Assembly and JavaScript but also investigate their practical implications for developers aiming to enhance performance in real-world situations. By explaining the trade-offs, limitations, and best practices of each technology, we aim to provide useful guidance for making informed decisions when choosing the most appropriate technology for client-side applications that require computationally intensive operations.

1 **reyes_webassembly_2018**.
2 **mozilla_webassembly_2024**.
3 **reyes_webassembly_2018**.
4 **mozilla_webassembly_2024**.

## 1.1 Personal Motivation

The motivation of this study is to analyse, clarify and evaluate the performance of WASM in comparison to JavaScript. The resulting findings are particularly interesting because this web standard is relatively new compared to JavaScript. For this reason, we believe that it is important to continue researching this standard in depth in order to enable meaningful use and in-depth future development. In addition, we would like to gain a deeper understanding of the performance characteristics of WASM in comparison to JavaScript. This requires both research into existing analyses and an independent analysis of the two web standards. Overall, we would like to make a contribution to the performance discussion between WASM and JavaScript with this work as part of a comprehensive analysis.

## 1.2 Research Aim

This paper aims to investigate the performance disparity between Web Assembly (WASM) and JavaScript (JavaScript) in the context of client-side applications, with a focus on computationally intensive algorithms. The objective is to determine whether WASM provides a distinct advantage over JavaScript in terms of efficiency through comprehensive experimentation and analysis.

## 1.3 Research Objectives

In order to completely cover the most important topics and to have a clear outline for the research process, we defined the following research objectives:

- Evaluate the performance of Web Assembly and JavaScript implementations in executing computationally intensive algorithms within client-side applications.
- Measure and compare CPU utilization between WASM and JavaScript implementations across various benchmark tests.
- Assess memory usage and management efficiency between WASM and JavaScript for tasks involving computationally intensive operations.
- Analyze the impact of DOM manipulation on the overall performance of WASM and JavaScript implementations within the context of client-side applications.

- Provide empirical evidence and insights to ascertain whether Web Assembly is advantageous over JavaScript from an efficiency perspective in the targeted application domain.
- Offer recommendations and guidelines for developers based on the findings to optimize performance when selecting between Web Assembly and JavaScript for client-side applications with computationally intensive algorithms.

# 2 Literature

## 2.1 WASM

WASM is a standardized bytecode that has been supported by modern browsers since 2017[5]. It is intended as a supplement to JavaScript in the browser, as the latter cannot deliver the necessary performance when performance requirements are high[6]. WASM was declared an official web standard by the World Wide Web Consortium. Apple, Google, Microsoft Mozilla and game engine manufacturers have also participated in the development of the new web standard[7]. In addition, WASM is executed in a so-called virtual machine and there are programming languages such as Rust, C/C++ or Go, which can be translated directly into the WASM bytecode. This means that WASM is not written directly in bytecode, but in one of the aforementioned programming languages and then automatically translated into bytecode[8]. WASM is not a programming language written by humans. WASM is much more a code that is to be written by a machine[9]. This WASM code is the result of compiled, conventional code, for example from a programming language such as Rust or Go. This compiled code is then available in a tightly packaged binary file and can be executed directly by the computer's processor. This directly executable type of code is called low-level code[10]. In order to be able to use web assembly in a browser that supports the technology, the compiled binary file must be executed in this browser using JavaScript. Execution creates a virtual instance within JavaScript, in which the WASM file is then executed[11].

## 2.2 Rust

Rust has emerged to become a very popular programming language in recent years, gathering attention for its innovative approach to addressing common pitfalls in software development, particularly those related to memory safety and concurrency. For the matter of this paper, it is important to take a deeper look into Rust, where it came from, what its main benefits are as well as Rusts problems. Rust has quite a unique history, as it was initially developed in 2006 by a young, 29 year old developer named Graydon Hoare in

[5] **niehoff_webassembly_nodate**.
[6] **zeroshope_was_2020**.
[7] **ct_heft_webassembly_2019**.
[8] **niehoff_webassembly_nodate**.
[9] **tovilo_intersim_nodate**.
[10] **augsten_was_2018**.
[11] **augsten_was_2018**.

his spare time. It was not until 2010 that the company he worked for, Mozilla Research, presented a project to create a new programming language. This new project was aimed at creating a safe and concurrent alternative to existing systems programming languages. Over the years, Rust has undergone significant development, resulting in its stable release in 2015[12]. Hoare took inspiration from existing programming languages such as C++, Haskell, and Erlang. Rust aimed to combine the performance of low-level programming languages with the safety, readability and "ease-of-use"of high-level programming languages. Early versions of Rust focused on refining its type system, borrowing rules, and ownership model, laying the groundwork for its distinctive approach to memory management and concurrency.

Now that the history on Rust is clear, it is more than useful to get a short summary on the functionalities of Rust. Therefore, ChatGPT has been asked to write a short summary of Rusts key features with the following output: Rust is distinguished by several key features that set it apart from traditional programming languages. Central to Rust's design is its ownership model, which ensures memory safety and prevents common issues such as data races and null pointer dereferences. In Rust, every value has a unique owner, and ownership can be transferred or borrowed through explicit rules enforced by the compiler. This approach eliminates the need for garbage collection while guaranteeing memory safety at compile time. Furthermore, Rust's expressive type system enables developers to write code that is both efficient and easy to reason about. The language supports static typing, generics, traits, and pattern matching, facilitating the creation of robust and scalable software solutions. Additionally, Rust's zero-cost abstractions enable developers to write high-level code without incurring runtime overhead, making it well-suited for performance-critical applications.

With the positive sides of Rust in mind the following section will focus on the drawbacks and limitations of Rust. One of the main hurdles for programmers is the steep learning curve associated with Rust's complex concepts like ownership, lifetimes, and borrowing. Although these features add to the language's safety and efficiency, they can pose difficulties for developers[13,14]. Another concern related to Rust the compilation time of its projects, which can be significantly longer compared to other languages. This is because Rust does not compile one file at a time but a whole package of files called "Crates", which can take a while to compile[15,16]. While optimizations and tooling improvements have mitigated this issue to some extent, it remains a point of discussions for developers working on

[12] **thompson_how_nodate**.
[13] **mukul_downsides_nodate**.
[14] **burkart_pros_2023**.
[15] **mukul_downsides_nodate**.
[16] **burkart_pros_2023**.

time-sensitive projects. Rust's limited codebase and lack of an elaborate library is another drawback. While efforts have been made to improve compatibility through tools such as the Rust FFI (Foreign Function Interface), seamless integration with legacy code remains an area of active research and development[17,18].

Overall Rust has become known to be one of the most beloved modern programming language in the past years. It becomes clear that Rust has potential to become even more popular and widely used looking at statements such as "It's enjoyable to write Rust, which is maybe kind of weird to say, but it's just the language is fantastic. It's fun. You feel like a magician, and that never happens in other languages"[19] or the fact that many large companies like Amazon, Dropbox or Microsoft already implemented it in their development process.

## 2.3 JavaScript

The programming JavaScript is without a doubt the most commonly used programming language on the planet, in fact, about 63% of developers are using JavaScript according to a 2023 study from Stackoverflow[20]. This marks JavaScripts 11th year in a row as the most popular programming language. Because of its popularity and importance for this research it is necessary to look further into the history of JavaScript and why it so important for web development.

The programming language JavaScript was initially invented in the year 1995 by Brendan Eich(w3). Two years later it was presented to the ECMA (European Computer Manufacturers Association) to become the ECMA-262 standard in the year 1997 (w3). After the first version (E1), which was solely supported by IE4 (Internet Explorer). After a lot of iterations, discussions and new major releases, the most important version in the history of JavaScript is version number 5 (E5), which was the first version to be supported by all major browser at the time.

It is also important to note that unlike that name suggests Java and JavaScript are not related and differ significantly from each other. The main difference is that JavaScript is a lightweight, dynamically interpreted scripting-language initially made only for browsers. Java on the other hand is a compiled, strongly and staticly typed, object-oriented programming language which is designed to be more robust and platform independent, leading with the

[17] **mukul_downsides_nodate**.
[18] **burkart_pros_2023**.
[19] **thompson_how_nodate**.
[20] **stackoverflow_stack_nodate**.

promise "Write Once, Run Anywhere". Both were released in 1995, with Java first in May and JavaScript following in December. At this point Java was already becoming popular, leading to the naming of JavaScript as a marketing ploy.[21]

In order grasp the complexity and importance of JavaScript, it is necessary to get a basic understanding of it. Therefore ChatGPT has been asked to summarize the available resources on JavaScript with the following result:

"JavaScript is a high-level, dynamic, interpreted programming language primarily utilized for client-side web development. It employs a prototype-based object-oriented paradigm, featuring first-class functions and asynchronous programming capabilities. JavaScript is commonly embedded within HTML documents and executed by web browsers to manipulate the Document Object Model (DOM) dynamically, enabling interactive user interfaces and dynamic content generation. Its versatility extends beyond web development, as it can also be employed for server-side scripting, mobile application development, and desktop application development through frameworks like Node.JavaScript. JavaScript's syntax is influenced by C and Java, offering a rich ecosystem of libraries, frameworks, and tools for application development. It serves as a fundamental technology underpinning modern web applications, facilitating the creation of dynamic and interactive experiences across various platforms."[22]

The main takeaway from ChatGPT's summary is that the programming language JavaScript is of utmost importance for the development and usage of modern websites. Because without JavaScript, many parts of a website would not be dynamic as we have gotten used to it by now but rather static in its code and interaction[23]. After all, without JavaScript "all you would have on the web would be HTML and CSS"[24] and commonly known website design-clues like a full-page drop down menu or content that is dynamically loaded into the websites body would be missing without the existence of JavaScript.

After those insights it is further necessary to take a look at common problems and setbacks of JavaScript. One main area of mistakes in JavaScript code is the syntax and data type handling, especially the lack of strong typing and type coercion. According to ChatGPT, the lack of strong typing in JavaScript "can make it prone to runtime errors and debugging difficulties, particularly in larger codebases where type safety is crucial"[25]. The other major source for errors in JavaScript is type coercion, which is a process where a value is converted from one data type to another. The process in itself is not the problem, it is

[21] **fin_js_brendan_2016**.
[22] **openai_chatgpt_nodate**.
[23] **oladele_what_2022**.
[24] **oladele_what_2022**.
[25] **openai_chatgpt_nodate**.

rather that JavaScripts loose typing and its therewith connected automatic type conversion can lead to errors. Another major source of errors is nesting multiple callbacks into a pyramid-like structure, which has become known as Callback Hell[26]. Callback Hell usually does not lead to technical errors but rather logical errors since the structure can become difficult to read and to understand. The last major error source is JavaScripts bottleneck in performance, caused by "Inefficient algorithms, excessive DOM manipulation, or poorly optimized code"[27]. Especially the excessiveness in which JavaScript allows you to manipulate the DOM seems to be a problem, since it does not provide any guidance on how to do DOM manipulation efficiently[28,29,30]. Combined with the common problem of memory leaks caused by "retaining references to objects that are no longer needed"[31]. Despite the many apparent performance problems and error sources within JavaScript, overall it remains a practical, lightweight and easy to learn language which continuously delivers good results.

## 2.4 DOM

Since this paper is going to do tests on the interaction of JavaScript and WASM with the DOM (Document Object Model), it is necessary to get a better understanding of the DOM, where it came from and what the benefits and drawbacks are.

The DOM originates from the need to dynamically interact with objects on a website through JavaScript. The standardization and documentation of the DOM was done by World Wide Web Consortium (W3C) as a platform-independent API, accompanied by a lot of stakeholders such as the Document Object Model Working Group and vendors of e.g. HTML or XML editors[32]. This first officially standardized version by the W3C is called DOM Level 1 and was released in 1998. From that point onward, DOM Level 2 was released in the year 2000 and DOM Level 3 was released in 2004, each with their own subsequent set of enhanced features and capabilities. The last iteration of the DOM, DOM Level 4, was released in the year 2015[33].

Due to its importance as the backbone structure of every modern website that needs dynamic interaction it is important to have a good general understanding of the DOM and

[26] **openai_chatgpt_nodate**.
[27] **openai_chatgpt_nodate**.
[28] **openai_chatgpt_nodate**.
[29] **peterson_10_nodate**.
[30] **mohit_common_nodate**.
[31] **openai_chatgpt_nodate**.
[32] **robie_what_nodate**.
[33] **oladele_what_2022**.

its functionality. Therefore ChatGPT has been asked to summarize the available resources on the DOM with the following result: "The Document Object Model (DOM) refers to a standardized, platform-independent application programming interface (API) utilized for representing and interacting with structured documents. Primarily employed in web development, the DOM provides a hierarchical representation of documents, enabling programmatic access and manipulation of their content, structure, and style. Characterized by its tree-like structure, the DOM organizes elements of an document into a logical arrangement, wherein each element is represented as a node possessing distinct properties and relationships with other nodes. This model facilitates dynamic manipulation of web content through scripting languages such as JavaScript, allowing for the modification of document elements, attributes, and event handling, thereby facilitating dynamic and responsive web experiences. As a fundamental component of web technologies, the DOM serves as an intermediary layer between web documents and scripting environments, facilitating seamless integration and manipulation of web content for diverse interactive applications."[34]

Of further importance for this paper are the potential bottlenecks developers face when working with the DOM. Performance bottlenecks, stemming from inefficient DOM manipulation and navigation, can diminish the responsiveness and user experience of web applications. Moreover, cross-browser inconsistencies and compatibility are major challenges for developers looking to create consistent experiences across different platforms and devices. Security vulnerabilities, such as cross-site scripting (XSS) attacks and DOM-based injection, underline the importance of implementing robust security measures to safeguard against malicious exploitation. Due to its steep learning curve and the general complexity of DOM manipulation, it is very difficult to learn for developers[35,36,37].

In conclusion, the HTML Document Object Model (DOM) represents a cornerstone of modern web development, enabling dynamic interactions between web content and scripting environments. Despite its transformative impact and widespread adoption, the DOM is not immune to limitations, which makes it a perfect candidate for the performance tests of this paper.

## 2.5 Assembler

Assemblers are programs that convert assembler code into machine language, i.e. binary code. They work directly with the respective processor architecture of the system in

[34] **openai_chatgpt_nodate-2**.
[35] **dohr_vanilla_0000**.
[36] **cody_arsenault_20_nodate**.
[37] **w3schools_javascript_nodate**.

question and are considered efficient and resource-saving. Assemblers translate code directly into binary code, which can be created manually or by machine. Some compilers first convert program code from higher programming languages into assembler code and then use an assembler to generate the final machine code. Assembler programs can use the entire instruction set of a processor, as each assembler instruction has a corresponding machine-level instruction. In contrast, higher programming languages are limited to a selection of these instructions and therefore sometimes offer the option of integrating assembly code into the source code if required. Each processor has its own architecture and its own instruction set, which means that each processor also requires its own assembler in order to process the instructions accordingly. Code for a specific processor can only be understood and translated in the specific assembly language. The assembly language uses mnemonic abbreviations for the internal instructions of the processor to enable logical and arithmetic operations, register accesses and control of the program flow. Assembly language programs are therefore highly platform-dependent and programs may have to be completely redeveloped in order to be transferred to another architecture or platform. One example of this is games that are developed for both desktop PCs with Intel processors and Playstation 5 with AMD processors[38]. Due to its proximity to the processor architecture, assembly code offers the advantage of being able to be quickly converted into binary code, i.e. machine code. Nowadays, pure assembly language programming is rarely used, as both system memory in various forms and basic computing power are available at low cost and the effort is mainly made to optimize time-critical systems or for educational purposes. Assembly language often seems limited and cumbersome compared to high-level languages, as complex operations are not part of the instruction set and longer programs are required. They are also difficult to manage, as the minimalist code is difficult to understand and requires very precise documentation in the source code[39].

## 2.6 Interpreter

An interpreter, in the context of programming, is a software that reads source code line by line and executes the source code directly at this point in time, the so-called runtime. The interpreter therefore analyzes the source code during execution and also only detects potential errors during execution. An example of a programming language that uses an interpreter is Java. The functionality of an interpreter therefore enables a program to execute the source code directly without having to translate it completely first. Although a translation takes place, this is not separate from the execution, unlike the compiler. For each line in

[38] **augsten_was_2018**.
[39] **augsten_was_2018**.

the source code, an immediate action takes place in the form of the execution of the underlying code. The sequence of these actions is determined by the instructions in the source code. The advantage of an interpreter is that errors that occur during programming can be detected immediately during execution. If there is an error in the source code, the interpreter stops further execution of the program and the programmer recognizes that there must be an error in the source code at precisely this point. At the same time, the interpreter also has disadvantages. Compared to a compiler, the interpreter works relatively slowly. This is because the interpreter reads each line individually. Even if lines of code are repeated, they are analyzed by the interpreter and only then successfully executed[40].

## 2.7 Compiler

A compiler, in the context of programming, is software that translates the source code completely into a form that can be processed by a machine. The compiler therefore analyzes the complete source code before execution and also detects potential errors in the source code before the first execution. An example of a programming language that uses a compiler is C. The procedure of the compiler differs not only in the separation of the analysis and execution of the code from the interpreter, but also in the general procedure. The compiler translates the source code by checking the syntax and then checking the semantics. If the source code passes this process without errors, it is translated into machine code, which can then be executed by the computer. The advantage of a compiler is that it only has to translate the source code once and can therefore be more performant than an interpreter[41].

[40] **xovi_was_2019-1**.
[41] **xovi_was_2019**.

# 3 Methodology

To investigate the research question, we decided to go for an explanatory research. An explanatory research is common for diving deeper into phenomena by describing and explaining the reasons for them, to give an answer why something is the way it is. It aims to understand the essence of what is being observed, focusing on explaining processes or structures and advancing knowledge about it. This type of research connects various factors and elements to formulate general statements and involves building, testing, or revising theories[42]. In our specific case, we want to improve our knowledge about something we do already have some information about. The objective we are aiming for is a mixture of providing information, to solve problems in the future and to test new services[43]. So in this chapter, we will describe how we have done our literature review at first. In the next step, we will describe the laboratory experiment and after that, the experimantel design. The next subchapter is about the setup, followed by the procedure and the research project. The last subchapter is about our GitHub, which we have used during our experiment.

## 3.1 Literature Review

In this chapter, we will describe what literature we have selected, how we have it selected and why we have selected our literature. The significance of the literature review lies not only in its ability to contextualize our study within existing scholarship but also in its multifaceted connections to our research questions, chosen methodologies and eventual findings. We followed the upcoming aspects to investigate our research question. At first we answered the question, whether our work has already been done or not. This is necessary in order to be able to make a statement about how and whether the selected topic is scientifically relevant[44]. There are already a lot of articles about what WebAssemby is, what it may can do or especially a performance comparison between WASM and JavaScript. In the world of software and hardware testing, it has become clear that the smallest differences can change or influence the result. The more tests are carried out, the more results can be compared. Also the more tests you make and the more results you have, the more can be ensured, that the results are valid or not. In the next step, we pointed out, what the main theoretical perspective is[45]. There are some cases in which WASM is faster than JavaScript and there are also cases, in which JavaScript is faster than WASM.

[42] **adams_research_2014**.
[43] **adams_research_2014**.
[44] **adams_research_2014**.
[45] **adams_research_2014**.

It depends on which explicit test you want to compare JavaScript and WASM[46]. WASM in general promisis a high and optimised performance[47]. On the other hand, JavaScript is the most common web development language with all its common issues[48]. WASM is build in a binary format, which makes it theoretically faster than Javascript. You can compare these two on different aspects like Runtime, Memory or cpu usage[49]. For example, the runtime of WASM is shorter than that of JavaScript in all WasmBoy benchmark tests. There are differences between the various browsers in which the WasmBoy benchmark test was run, with the differences being most noticeable in Mozilla's browser, Firefox[50]. There are also tests in which JavaScript performs better than WASM. In microbenchmark tests in the form of various sorting algorithms, JavaScript sometimes achieves shorter throughput times than WASM. Here too, the results differ depending on the browser used[51]. The memory usage of WASM also differs from JavaScript. WASM uses considerably more memory than JavaScript, both in Firefox and in Chrome. The difference between the memory usage of WASM and JavaScript is increasing as the input given to the two web standards increases. While the memory usage of JavaScript remains the same despite increasing input, the memory usage of WASM increases continuously[52]. After that, we want to show up the problems we got while researching the literature[53]. While gaining general information about the main theoretical perspectives, the problem was, that there was not that single literature which could answer our research question. It was also not easy to find out, how to make our specific tests between these two architectures. There are many different literatures which compared WASM and JavaScript on many different technical aspects but not pointed out, which method may be the best. The topic is also relatively new, which made it difficult to find suitable literature. Both quantitatively and qualitatively. In the following we will describe the major controversies on our topic[54]. WASM is not used in common yet. JavaScript is still the most common web development language. One reason for that can be the disadvantage, that a garbage collector is still missed in WASM, which means that storage management can not be done by itself. WASM can also not interact with the DOM of a Website without JavaScript and therefore not change the visualization of a website itself[55].

[46] **de_macedo_webassembly_2022**.
[47] **niehoff_webassembly_nodate**.
[48] **mohit_common_nodate**.
[49] **sunarto_systematic_2023**.
[50] **de_macedo_webassembly_2022**.
[51] **de_macedo_webassembly_2022**.
[52] **sunarto_systematic_2023**.
[53] **adams_research_2014**.
[54] **adams_research_2014**.
[55] **bigelow_was_nodate**.

## 3.2 Laboratory Experiment

### 3.2.1 Experimental Design

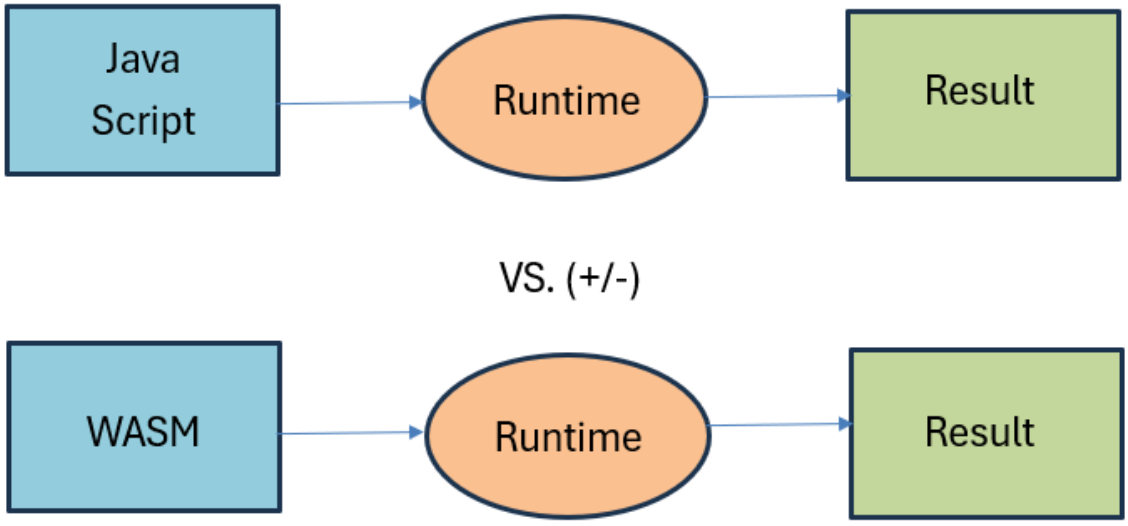
A scientific experiment is carried out in order to obtain empirical data and to test the hypotheses put forward. To distinguish it from an observation, independent variables are manipulated in an experiment under controlled conditions. The independent variable in turn influences a dependent variable. In the next step, the independent variable is deliberately manipulated and the dependent variable reacts to this event[56]. For our research, a laboratory experiment comes into question, as we create an artificial environment in which we can control all variables well. Due to the artificial environment we have created, assumptions can be made, but the results cannot be generalized with complete certainty[57]. In this chapter we describe our experimental design for our explanatory research. For our experimental design, we orientated on four key steps that help us to perform a theory based and structured experiment. The four key steps are:[58]

- Defining our variables
- Writing our hypothesis
- Designing our experimental treatments
- Measure our dependent variable

**Defining our Variables**

[56] **genau_experiment_2018**.
[57] **genau_experiment_2018**.
[58] **bevans_guide_2019**.

**Figure 1: Comparison Design, Source: Own depiction**




In that case our question is, if and how the runtime can be affected by whether using JavaScript or using WASM for different computationally intensive operations. The key independant variables are „JavaScript“ and „WASM“. The key dependant variable is „Runtime“. Abb.X is further called „Fixed testing part“. To specify different input operations, we will show them in the following figures, while there is the fixed test part as shown above and the öperation part"for the different test operations, which define the input given to each, JavaScript and WASM within the fixed test part.

**Figure 2: Advanced Comparison Design, Source: Own depiction**

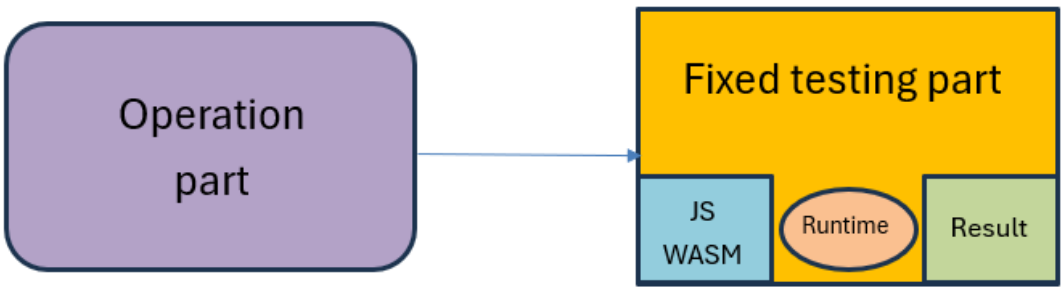


**Table 1: Differenciation of dependant and independant variables, Source: Own depiction**

| Operation part | Independent variable | Dependent variable |
|---|---|---|
| Sieve of Eratosthenes | JavaScript & WASM | Runtime |
| RAM read/write | JavaScript & WASM | Runtime |
| DOM manipulation | JavaScript & WASM | Runtime |

The first operation part is about the Sieve of Eratosthenes. The code is written in both, JavaScript and for WASM with Rust. Both programming languages are the independent variables, which are effecting the dependent variable runtime directly. The third operation part is about DOM manipulation. Here, the independent variables are JavaScript and WASM. The dependent variable, affected be the independent variables JavaScript and WASM, is runtime.

**Writing our hypothesis**

Now that we have defined our testing variables, we can put our hypothesis for each operation part in position[59].

**Table 2: Definition of hypothesis for each operation part, Source: Own depiction**

| Operation part | Null hypothesis($H_0$) | Alternate hypothesis($H_1$) |
|---|---|---|
| Sieve of Eratosthenes | WASM Runtime is faster than JavaScript Runtime. | JavaScript Runtime is faster than WASM Runtime. |
| RAM read/write | WASM Runtime is faster than JavaScript Runtime. | JavaScript Runtime is faster than WASM Runtime. |
| DOM manipulation | JavaScript Runtime is faster than WASM Runtime. | WASM Runtime is faster than JavaScript Runtime. |

With the execution of the Sieve of Eratosthenes algorithm it is assumed that WASM has a faster Runtime than JavaScript. For RAM read and write operations it is assumed that WASM has a faster runtime than JavaScript. For the third hypothesis it is assumed that JavaScript has a faster Runtime than WASM for DOM manipulation.

**Design our experimental treatments**

In order to carry out a realistic test experiment, it is necessary to make various changes to the variables. This allows a wide variety of conditions to be tested and the resulting findings to be recorded. Different variables ensure that the experiment is as valid as possible[60]. In

[59] **bevans_guide_2019**.
[60] **bevans_guide_2019**.

our case we simply change the operation part to manipulate our independant variables. By changing this, JavaScript and WASM can be compared on different ways in different use cases, so that we can collect as many results as possible and useful for our comparison and research goal.

**Measure our dependent variable**
In the end, we will collect all the data from the testing cases (operation part) and measure them, to answer our hypothesis.

### 3.2.2 Setup

In this chapter the experimental setup of this paper and the the corresponding code will be described. Referring to the chapter before, there are three different test that need to be conducted.

**Setup Website**
In order to do all tests in one place, a locally hosted website was created. The setup of this website is rather simple yet effective. It consists of a headline and six buttons.

**Figure 3: Benchmark Website, Source: Own depiction**


Each button executes one of the tests, three buttons are for the JavaScript tests and the other three for the WASM tests. Before each test is executed, an input window appears and asks the user to enter a number. What the number represents depends on the test that is executed.

The code of the main page of the website looks as follows:

**Listing 1: HTML Code of the main page of the website, source: self-coded**

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Benchmarks</title>
    </head>
    <body>
        <h1>Benchmarks</h1>
        <script type="module" src="./main.JavaScript"></script>
    </body>
</html>
```

This basic HTML Code defines the main structure of the website. In terms of styling, there is none. The main task of this code is to link the main.JavaScript file.

**Setup First Test - CPU Benchmark**

Next is the setup of the first test, which is a CPU Benchmark test based on the Sieve of Eratosthenes algorithm. In the field of computer science this algorithm is commonly used to measure computer performance. "The time complexity of calculating all primes below n in the random access machine model is $O(n \log \log n)$ operations, a direct consequence of the fact that the prime harmonic series asymptotically approaches $\log \log n$. It has an exponential time complexity with regard to input size, though, which makes it a pseudo-polynomial algorithm. The basic algorithm requires $O(n)$ of memory."[61]. The input parameter for this test determines the limit up to where the algorithm searches for prime numbers.

The following code was used to execute the algorithm with JavaScript:

**Listing 2: Sieve of Eratosthenes in JavaScript, source: self-coded**

```
function JavaScript_sieve(limit) {
    // Create an array to track whether each number is prime
    const sieve = new Array(Number(limit) + 1).fill(true);
    // 0 and 1 are not prime numbers, so mark them as false
    sieve[0] = sieve[1] = false;
    // Initialize an empty array to store prime numbers
    const primes = [];
    // Iterate through the numbers starting from 2 up to the square root of the limit
    for (let i = 2; i <= Math.sqrt(limit); i++) {
        // If the current number is prime (marked as true), proceed
        if (sieve[i]) {
            // Mark multiples of the current prime number as not prime
            // Starting from i*i, as all numbers below that would have been already
    marked
            for (let j = i ** 2; j <= limit; j += i) {
                sieve[j] = false;
            }
        }
    }
    // Iterate through the sieve to extract prime numbers
    for (let i = 0; i < sieve.length; i++) {
        if (sieve[i]) {
            primes.push(i);
        }
    }
    // Return the array of prime numbers
    return primes;
}
```

The following code was used to execute the algorithm with WASM, coded in Rust:

[61] **wikipedia_sieve_2024**.

**Listing 3: Sieve of Eratosthenes in Rust, source: self-coded**

```rust
#[wasm_bindgen]
pub fn wasm_sieve(limit: usize) -> Vec<usize> {
    // Create a boolean vector to track whether each number is prime
    let mut sieve: Vec<bool> = vec![true; limit + 1];
    let mut primes: Vec<usize> = Vec::new();
    // 0 and 1 are not prime numbers, so mark them as false
    sieve[0] = false;
    sieve[1] = false;
    // Iterate through the numbers starting from 2 up to the square root of the limit
    for i in 2..=(limit as f64).sqrt() as usize {
        // If the current number is prime (marked as true), proceed
        if sieve[i] {
            // Mark multiples of the current prime number as not prime
            // Starting from i*i, as all numbers below that would have been already
                marked
            for j in (i * i..=limit).step_by(i) {
                sieve[j] = false;
            }
        }
    }
    // Iterate through the sieve to extract prime numbers
    for i in 2..=limit {
        if sieve[i] {
            primes.push(i);
        }
    }
    // Return the vector of prime numbers
    primes
}
```

Both code snippets in essence implement the same algorithm, just in different languages. However, it was coded in such a way that the implementations are comparable. To give an example, the code written in Rust could be further optimized by iterating over the sieve vector with ‘sieve.iter()‘. Since this is not an option in JavaScript it was implemented similarly to JavaScript to enhance comparability. Another difference between both implementations is the usage of vectors in Rust. Because the neither the input nor the number of primes is not known beforehand, a growable object is required. In JavaScript an array is such a growable object but in Rust an array is not growable, therefore it is replaced by a vector.

### Setup Second Test - RAM Speed

The second test uses a multitude of read and write operations as a stress test for the RAM speed. The input parameter for this test determines the size of the iterable to perform the read and write operations on.

The following code was used to execute the stresstest with JavaScript:

**Listing 4: RAM stresstest in JavaScript, source: self-coded**

```javascript
function JavaScript_ram_test(size) {
```

```
    const array = new Array(size);
    // Write operation
    for (let i = 0; i < size; i++) {
        array[i] = i;
    }
    // Read operation
    for (let i = 0; i < size; i++) {
        array[i] += 1;
    }
}
```

The following code was used to execute the stresstest with WASM, coded in Rust:

**Listing 5: RAM stresstest in Rust, source: self-coded**

```
#[wasm_bindgen]
pub fn wasm_ram_test(size: usize) {
    let mut vec: Vec<usize>  = Vec::with_capacity(size);
    // Write operation
    for i in 0..size {
        vec.push(i);
    }
    // Read operation
    for i in 0..size {
        vec[i] += 1;
    }
}
```

As with the first test, the two code snippets, in essence, execute the same operations. The difference in both implementations are identical to the previous test and are also justified the same way.

### Setup Second Test - DOM manipulation

The final test was about DOM manipulation and was implemented because JavaScript "makes it relatively easy to manipulate the DOM (i.e., add, modify, and remove elements), but does nothing to promote doing so efficiently"[62]. Therefore this test investigates the DOM manipulation with JavaScript and WASM to see which operates more efficiently. The input parameter for this test determines the number of div elements that should be created and shown on the screen.

The following code was used to execute the DOM manipulations with JavaScript:

**Listing 6: DOM manipulation in JavaScript, source: self-coded**

```
function JavaScript_create_elements(count) {
    // Get a reference to the document body
    const body = document.body;
    // Iterate 'count' times to create elements
    for (let i = 0; i < count; i++) {
```

[62] **peterson_10_nodate**.

```
        // Create a new div element
        const div = document.createElement("div");
        // Set styles for the div element
        div.style.border = "2px solid #000";
        div.style.backgroundColor = "#e0e0e0";
        div.style.margin = "5px";
        // Set text content for the div element
        div.textContent = "Hello, JavaScript!";
        // Append the div element to the document body
        body.appendChild(div);
    }
}
```

The following code was used to execute the algorithm with WASM, coded in Rust:

**Listing 7: DOM manipulation in Rust, source: self-coded**

```
#[wasm_bindgen]
pub fn wasm_create_elements(count: usize) {
    // Access the document object from the web_sys crate
    let document = web_sys::window().unwrap().document().unwrap();
    // Iterate 'count' times to create elements
    for _ in 0..count {
        // Create a new div element
        let div = document.create_element("div").unwrap();
        // Set styles for the div element
        div.set_attribute("style", "border: 2px solid #000; background-color: #e0e0e0;
            margin: 5px;").unwrap();
        // Set text content for the div element
        div.set_text_content(Some("Hello, WASM!"));
        // Append the div element to the document body
        document.body().unwrap().append_child(&div).unwrap();
    }
}
```

In this case the two code snippet again implement the same procedure. Unlike in the previous tests, there are no performance-relevant differences between JavaScript and WASM. The difference in the code emerge from the syntax of the two languages. However, one difference needs to be highlighted. Rust encourages explicit error handling. Since this code will not be deployed to regular users, explicit error handling is not necessary and is avoided by using the '.unwrap()' function.

One aspect of all Rust-specific tests is the `#[wasm_bindgen]` command before every function which is described by mdn web docks as follows "wasm-pack uses wasm-bindgen, another tool, to provide a bridge between the types of JavaScript and Rust. It allows JavaScript to call a Rust API with a string, or a Rust function to catch a JavaScript exception"[63]. This command is essential for the main.JavaScript to access the Rust functions when a test is triggered, therefore it is another important of this chapter.

[63] **mozilla_compiling_nodate**.

### 3.2.3 Procedure

The structure of each test case was identical. The respective test is started via a button, which opens a pop-up window in which the input for the test is entered. The input is then confirmed and the following code is executed:

**Listing 8: Test execution, source: self-coded**

```
document.addEventListener("DOMContentLoaded", function () {
    init();
    console.log("Loaded");
    for (const [testName, testFunctions] of Object.entries(funcs)) {
        for (const [index, implementation] of testFunctions.entries()) {
            const button = document.createElement("button");
            button.textContent = `Run ${testName} with ${rust_JavaScript[index]}`;
            button.addEventListener("click", () => {
                const input = prompt("Input:");
                let totalTime = [];
                let output;
                for (let i = 0; i < runs; i++) {
                    const startTime = performance.now();
                    output = implementation(input);
                    const endTime = performance.now();
                    console.log(output);
                    const elapsedTime = endTime - startTime;
                    totalTime.push(elapsedTime);
                    console.log(`Run ${i + 1}: ${elapsedTime} milliseconds`);
                }
                const averageTime = totalTime.reduce((a,b) => a+b, 0) / runs;
                console.log('Total time:', totalTime)
                console.log(`Function ${testName}, ${rust_JavaScript[index]} with
    input ${input} took ${averageTime} miliseconds to execute`);
            });
            document.body.appendChild(button);
        }
    }
});
```

In order to allow a structured testing procedure, the tests were executed in the following order:

1. The first test was started with an input of 100.000.000 and was repeated 10 times for JavaScript and 10 times for WASM.
2. The second test was started with an input of 100.000.000 and was repeated 10 times for JavaScript and 10 times for WASM.
3. We started the third test with an input of 30.000 and ran it five times each for JavaScript and for WASM.

The result of each test is output via the command line and then recorded in a table. The table will be presented later on in the chapter results.

In addition, the average runtime, standard deviation and factor by which WASM is executed faster or slower than JavaScript, relative to the average runtime, was calculated and added to the table.

## 3.3 Research Project

The paper focuses on WASM. However, as Business Informatics students, we also aimed to explore professional collaboration in academic paper writing. We embraced the challenge of writing the paper using LaTeX, which required learning LaTeX syntax, setting up and structuring a LaTeX project, and understanding LaTeX source management.

Furthermore, the paper was collaboratively developed among the three researchers. We created a GitHub repository for the entire LaTeX project, just like for the experiment's code. This decision provided valuable insights and learnings which is important for our academic future since it is common practice in technical scientific papers to use LaTeX with a Git version control[64,65].

A consequence of these decisions was our agreement to work collectively in Microsoft Visual Studio Code using the 'LaTeX Workshop' plugin by James Yu, ensuring uniform technical capabilities among us. We considered an alternative, which was using Overleaf, an online LaTeX editor. However, this would have created costs for collaborative work, which we chose to avoid.

Subsequent chapters of this paper provide a more detailed analysis of LaTeX and GitHub, highlighting the challenges encountered during the process.

### 3.3.1 Latex Project

As mentioned earlier, we wanted to challenge ourselves by writing this paper in LaTeX. LaTeX is based on the markup language TeX, originally developed by the American computer scientist Donald Knuth in 1982. The LaTeX framework used for this project is "a macro package built on top of TeX", intended to "simplify the typesetting of TeX"[66]. Essentially,

[64] **laverny_using_2021**.
[65] **ranganath_git_2017**.
[66] **wikibooks_latex_nodate**.

a LaTeX document is nothing more than a text document enriched with commands and markup.

One of the main differences between LaTeX and conventional text editors such as Microsoft Word is that the author does not immediately see the result of what he is writing. Unlike MS Word, which operates on the principle of WYSIWYG (What You See Is What You Get), LaTeX operates on the principle of WYSIWYAF (What You See Is What You Asked For).

Despite the significant drawback of requiring authors to learn how to use LaTeX, the advantages outweigh the disadvantages. Wikibooks outlines the advantages of LaTeX as follows:

- You can concentrate purely on the structure and contents of the document. LaTeX will automatically ensure that the typography of your document—fonts, text sizes, line heights, and other layout considerations—are consistent according to the rules you set. - In LaTeX, the document structure is visible to the user, and can be easily copied to another document. In WYSIWYG applications it is often not obvious how a certain formatting was produced, and it might be impossible to copy it directly for use in another document. - Indexes, footnotes, citations and references are generated easily and automatically. - Mathematical formulae can be easily typeset. (Quality mathematics was one of the original motivations of TeX.) - Since the document source is plain text, - Document sources can be read and understood with any text editor, unlike the complex binary and XML formats used with WYSIWYG programs. - Tables, figures, equations, etc. can be generated programmatically with any language. - Changes can be easily tracked with version control software. - Some academic journals only accept or strongly recommend submissions in the form of LaTeX documents. Publishers offer LaTeX templates.[67].

We found these points to be significant advantages in our work with LaTeX, particularly the highly structured approach to document composition and the straightforward handling of sources.

However, a simple LaTeX document would not suffice. The most effective use of LaTeX is to set up an entire LaTeX project, especially when different sections of the paper are written simultaneously by different contributors. Such a LaTeX project allows each part of the structure of the paper, such as chapters including all subchapters, to be divided into individual files. These individual files can then be worked on independently, as they are simply referenced within the existing structure (in our case, from a parent chapter file). This provides the flexibility to create the structure of the whole document without content, and then modify the structure without having to consider the content layout. Furthermore, as mentioned

[67] **wikibooks_latex_nodate**.

in the advantages of Wikibooks, it is possible to change aspects such as layout and font size centrally.

That is why we decided to take on the challenge of writing this paper in LaTeX. Now we will go into the detailed structure of our LaTeX project.

Fortunately, we were able to adopt the general structure from Andy Grunwald, a former FOM student, who has published a "FOM LaTeX template "on GitHub[68]. This template has many contributors and is continuously improved and adapted to FOM-specific requirements.

In addition to providing the basic document structure, the template also provides various methods for compiling the associated PDF document for the project. This includes using the provided docker file or the compilation files together with the VS Code extension "LaTeX Workshop "by James Yu.

Now let's look at the basic structure of the LaTeX document itself. There is a main file called "thesis_main.tex"which describes the basic configurations for the document, such as font size, font, margins, etc., and defines the structural layout of the thesis. This is done using the "
input"method, where the path to the referenced file is enclosed in curly braces. This provides a concise and easy to understand method of defining the structure.

The content chapters are further encapsulated, each having its own "structure file"(e.g. chapterOverview.tex). This file references individual chapters, each of which can have its own respective structure. Here is an overview of the structure of the document:

(!Darstellung einfügen)

This ensures that the substantive parts (the chapters and subchapters) are separated from the more administrative parts, such as the table of contents. This clear separation is particularly helpful in academic papers where the style, such as numbering, may vary from section to section. With this clear separation, it is possible to apply different styles to each section.

Certainly, other methods would be sufficient for a term project and would achieve similar or identical results. However, as mentioned earlier, we wanted to overcome this technical challenge so that we could use it for future projects.

[68] **grundwald_andygrunwaldfom-latex-template_nodate**.

### 3.3.2 Github

GitHub is a web-based version control system for software development that allows developers to manage their code in private or public repositories. Although there are many alternatives, such as GitLab, Bitbucket or SourceForge, we chose GitHub mainly because of its seamless integration with Microsoft Visual Studio Code, the chosen editor for this project.

The benefits of a centralised version control are obvious. On the one hand, it allows the entire project to be managed in one place, ensuring that it is always the latest version. On the other hand, a GitHub repository, even if set to private, facilitates collaborative and platform-independent work on the project.

In theory, GitHub's ability to create project and organisational structures could make it much easier to scale up research projects. In addition to managing such projects through Git pull and push requests, GitHub offers effective use of built-in features such as issues and discussions to challenge others' contributions, leading to the development of better and more refined ideas.

However, to maintain the focus and scope of this paper, we have chosen to use GitHub exclusively as a version control system. There is a repository called "WASMvsJavaScript"for the development of the experiment[69], and another called "ITInfraPaper"for the LaTeX project[70]. Both repositories are publicly accessible and linked in the source directory.

[69] **fuchs_github_nodate**.
[70] **ruber_github_nodate**.

# 4 Results

This chapter is dedicated to the presentation and description of the test results. Together with the literature review this chapter will lay a foundation for the later discussion on the aim of this research. The following tables contain the different results of all test runs. As described in the previous chapter, test one and test two were carried out for both JavaScript and WASM with three different inputs of ten test runs each. Test three was also performed for both JavaScript and WASM. In contrast to tests one and two, the entire test was carried out with five different inputs and only once with five test runs. All result tables have an identical structure. The header row shows the number of test runs and the header column shows the respective input, categorised by JavaScript and WASM. In addition, there is an average and the standard deviation for each row. The first two tests were initially carried out with an input of 1,000,000. It was noticed that the results are not meaningful. Therefore the inputs have been gradually increased to reach an optimum in terms of runtime and credibility of the results. This approach identified 100,000,000 as the optimal input size for the first two tests. The same approach was used for the third test with a starting input of 100. This established optimum input for the third test at 30,000. From here on, we will therefore only deal with the results of the optimum input value.

**Table 3: Table for results of test 1, Source: Own depiction**

| Test 1: CPU Benchmark with Sieve of Eratosthenes | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Input | Run 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Avg | Std. Dev |
| JS | 100000000 | 4166.7500 | 4066.7480 | 3666.7400 | 3700.0740 | 3733.4080 | 4016.7470 | 3700.0740 | 4250.0850 | 4100.0820 | 4100.0820 | 3950.079 | 224.1627 |
| JS | 10000000 | 300.0060 | 266.6720 | 300.0060 | 333.3400 | 300.0060 | 283.3390 | 366.6740 | 300.0060 | 283.3390 | 333.3400 | 306.6728 | 29.6071 |
| JS | 1000000 | 16.6670 | 16.6670 | 16.6670 | 0.0000 | 16.6670 | 16.6670 | 16.6670 | 0.0000 | 16.6670 | 0.0000 | 11.6669 | 8.0509 |
| RUST/WASM | 100000000 | 1300.026 | 1200.024 | 1200.024 | 1216.691 | 1183.357 | 1200.024 | 1200.024 | 1183.357 | 1200.024 | 1200.024 | 1208.3575 | 33.5647 |
| RUST/WASM | 10000000 | 300.0060 | 266.6720 | 283.3390 | 266.6720 | 266.6720 | 283.3390 | 283.3390 | 266.6720 | 266.6720 | 283.3390 | 276.6722 | 11.6537 |
| RUST/WASM | 1000000 | 33.3340 | 33.3340 | 33.3340 | 16.6670 | 33.3340 | 16.6670 | 16.6670 | 33.3340 | 33.3340 | 16.6670 | 26.6672 | 8.6068 |

In the first test, the lowest value for the runtime of JavaScript is 3666.74 ms and the highest value is 4250.09 ms. The average value is 3950.08 ms and the standard deviation is 224.16. For WASM, the lowest value is 1183.36 ms and the highest value is 1300.03 ms. The average value is 1208.36 ms and the standard deviation is 33.56.

**Table 4: Table for results of test 2, Source: Own depiction**

| Test 2: RAM Stresstest | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Input | Run 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Avg | Std. Dev |
| JS | 100000000 | 2050.0410 | 1800.0360 | 1800.0360 | 1766.7020 | 1766.7020 | 1783.3690 | 1750.0350 | 1766.7020 | 1750.0350 | 1750.0350 | 1798.3693 | 90.4226 |
| JS | 10000000 | 116.6690 | 133.3360 | 150.0030 | 166.6700 | 166.6700 | 150.0030 | 166.6700 | 133.3360 | 116.6690 | 133.3360 | 143.3362 | 19.5635 |
| JS | 1000000 | 0.0000 | 0.0000 | 16.6670 | 0.0000 | 16.6670 | 0.0000 | 0.0000 | 16.6670 | 0.0000 | 0.0000 | 5.0001 | 8.0509 |
| RUST/WASM | 100000000 | 383.341 | 400.008 | 383.341 | 383.341 | 366.674 | 400.008 | 400.008 | 383.341 | 383.341 | 366.674 | 385.0077 | 12.2980 |
| RUST/WASM | 10000000 | 100.0020 | 100.0020 | 100.0020 | 116.6690 | 100.0020 | 100.0020 | 100.0020 | 116.6690 | 100.0020 | 100.0020 | 103.3354 | 7.0274 |
| RUST/WASM | 1000000 | 33.3340 | 0.0000 | 16.6670 | 0.0000 | 16.6670 | 16.6670 | 0.0000 | 16.6670 | 0.0000 | 16.6670 | 11.6669 | 11.2494 |

In the second test, the lowest value for the runtime of JavaScript is 1750.04 ms and the highest value is 2050.04 ms. The average value is 1798.37 ms and the standard deviation

is 90.42. For WASM, the lowest value is 366.67 ms and the highest value is 400.00 ms. The average value is 385.01 ms and the standard deviation is 12.30.

**Table 5: Table for results of test 3, Source: Own depiction**

| Test 3: DOM Manipulation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Input | Run 1 | 2 | 3 | 4 | 5 | Avg | Std. Dev |
| JS | 40000 | freeze | | | | | | |
| JS | 30000 | 14 | 14.1 | 13.4 | 13.5 | 13.3 | 13.66 | 0.36469165 |
| JS | 10000 | 3.27 | - | - | - | - | | |
| JS | 1000 | 330 | - | - | - | - | | |
| JS | 100 | 0 | - | - | - | - | | |
| RUST/WASM | 40000 | freeze | | | | | | |
| RUST/WASM | 30000 | 12.1 | 12 | 12.1 | 12.2 | 12.1 | 12.1 | 0.07071068 |
| RUST/WASM | 10000 | 2.9 | - | - | - | - | | |
| RUST/WASM | 1000 | 293 | - | - | - | - | | |
| RUST/WASM | 100 | 0 | - | - | - | - | | |

In the last test, the lowest value for the runtime of JavaScript is 13.3 s and the highest value is 14.1 s. The average value is 1798.37 ms and the standard deviation is 90.42. The average value is 13.66 s and the standard deviation is 0.36. For WASM, the lowest value is 12.0 s and the highest value is 12.2 s. The average value is 12.1 s and the standard deviation is 0.07.

# 5 Discussion

The following chapter will discuss the previously described test results and argument whether or not they confirm or refuse the in chapter 3.2.1 defined null hypothesis.

In general, however, it can be said in advance that the results of this paper clearly show that WASM offers clear performance advantages in certain scenarios. In particular, when executing computationally intensive algorithms, WASM was able to demonstrate a significant improvement in CPU utilization and overall performance compared to JavaScript. This observation confirms the perception of WASM as an advanced technology that enables more efficient execution of code on web browsers. A key aspect of our discussion is to analyze the underlying mechanisms that drive these performance differences. By examining CPU utilization patterns, memory consumption, and DOM manipulation efficiency, we were able to identify the specific strengths and weaknesses of WASM compared to JavaScript. These findings provide a deep insight into the technical aspects of the two platforms and help to make informed decisions when choosing the appropriate technology for specific use cases.

**Hypothesis Test 1**
The null hypothesis for the first comparison of JavaScript and WASM based on the CPU performance was "WASM Runtime is faster than JavaScript Runtime". Based on the results of the first test it can be said that WASM is about 3.27 times faster than JavaScript in executing the algorithm. Therefore the null hypothesis is confirmed and the alternate hypothesis is rejected. This result comes from the underlying technological advantage of WASM being low-level on the one hand and being compiled instead of interpreted on the other hand[71].

**Hypothesis Test 2**
The null hypothesis for the second test comparing JavaScript and WASM based on the speed of RAM read and write operations was "WASM Runtime is faster than JavaScript Runtime". Based on the results of the second test it can be said that WASM is about 4.67 times faster than JavaScript in executing the read and write operations. Therefore the null hypothesis is confirmed and the alternate hypothesis is rejected. As with the previous result, it again boils down to the underlying technological advantage of WASM being low-level on the one hand and being compiled instead of interpreted on the other hand[72].

**Hypothesis Test 3**
The null hypothesis for the last test comparing JavaScript and WASM based on the effi-

[71] **bigelow_was_nodate**.
[72] **bigelow_was_nodate**.

ciency in which they executed DOM manipulation commands was "JavaScript Runtime is faster than WASM Runtime". Based on the results of the last test it can be said that, again, WASM is about 1.13 times faster than JavaScript in manipulating the DOM by adding div elements. In contrast to the tests before, this time the null hypothesis is rejected and the alternate hypothesis, which is "WASM Runtime is faster than JavaScript Runtime", is confirmed. Since our null hypothesis was rejected, further explanation is necessary. Initially, the null hypothesis was based on the difficulty that WASM, regardless of its computational advantage, needs to 'pass' its result through JavaScript to the DOM in order to manipulate it. Therefore it is rather surprising that the results show that WASM is still faster than JavaScript in this discipline.

# 6 Conclusion

In conclusion, the findings of this paper provides evidence for the performance advantages of WASM over JavaScript, particularly when executing computationally intensive algorithms. Through a series of tests and respective analyses, we have confirmed that WASM offers notable improvements in CPU utilization, memory efficiency, and overall performance compared to JavaScript.

Two out of the three null hypothesis and one alternate hypothesis (which thought of WASM having a faster runtime) were confirmed, indicating that WASM runtime indeed outperforms JavaScript runtime across various performance metrics. This validation underscores the inherent technological advantages of WASM, including its low-level nature and compilation-based execution, which contribute to its superior performance characteristics. Furthermore, the unexpected result observed in the third test, where WASM showed faster DOM manipulation compared to JavaScript, highlights the complexity of performance evaluation in real-world scenarios. Despite WASM relying on JavaScript for DOM manipulation tasks, its inherent efficiencies shine through, suggesting subtle influences on performance outcomes. What determines those influence is matter to future research.

Our discussion of the underlying mechanisms driving these performance differences, including CPU utilization patterns, memory consumption, and DOM manipulation efficiency, has provided valuable insights into the technical nuances of both WASM and JavaScript platforms. These insights serve to inform developers and decision-makers when selecting the appropriate technology stack for specific use cases, enabling more informed choices and optimized performance outcomes. Overall, this study contributes to the growing body of research on Web Assembly and JavaScript performance, shedding light on their comparative advantages and limitations. By demonstrating the clear benefits of WASM in certain scenarios, we underscore its significance as an advanced technology that enables more efficient execution of code on web browsers, paving the way for enhanced web application performance and user experiences.

As the landscape of web development continues to evolve, further exploration and refinement of Web Assembly's capabilities will be essential, offering continued opportunities for innovation and optimization in the field of client-side application development.

# Appendices

## Appendix 1: Literature appendices

Together with the paper, the following literature appendices are submitted:

- Sunarto et al. - 2023 - A Systematic Review of WebAssembly VS Javascript P.pdf
- De Macedo et al. - 2022 - WebAssembly versus JavaScript Energy and Runtime.pdf
- javascript-issues-and-solutions.html
- 10-most-common-javascript-mistakes.html
- what-to-understand-callback-and-callback-hell-in-javascript.html
- introduction.html
- what-is-the-dom-explained-in-plain-english.html
- vanilla-javascript-libraries-quest-stateful-dom-rendering.html
- javascript-performance.html
- js_performance.html
- was-ist-ein-assembler-a-756636.html
- using-git-and-github-for-latex-writting.html
- a-git-workflow-for-writing-papers-in-latex-4cfb31be4b06.html
- _Einleitung.html
- ITInfraPaper.html
- WASMvsJS.html
- webassembly-solving-performance-problems.html
- experimental-design.html
- hitlist.html
- experiment.html
- Sieve_of_Eratosthenes.html

- webassembly-mehr-als-nur-ein-web-standard.html
- was-ist-webassembly-a-4243591d831cc5a12d425ede224f5e5b.html
- WebAssembly-Wasm.html
- javascript-vs-webassembly.html
- downsides-of-rust-programming-language.html
- rust-worlds-fastest-growing-programming-language.html
- 2023.html
- was-ist-ein-compiler.html
- was-ist-ein-interpreter.html
- Concepts.html
- FOM-LaTeX-Template.html
- Rust_to_wasm.html

They can also be found in the literature appendices folder of the GitHub repository.

# Appendix 2: GitHub Repositories

Here are the links to the GitHub repositories that contain the code and data from the LaTeXProject used in this paper:

- ITInfraPaper
- WASMvsJS

## Declaration of Authenticity

I hereby declare that I produced the submitted paper with no assistance from any other party and without the use of any unauthorized aids and, in particular, that I have marked as quotations all passages which are reproduced verbatim or near-verbatim from publications. Also, I declare that the submitted print version of this thesis is identical with its digital version. Further, I declare that this thesis has never been submitted before to any examination board in either its present form or in any other similar version. I herewith agree/disagree that this thesis may be published. I herewith consent that this thesis may be uploaded to the server of external contractors for the purpose of submitting it to the contractors' plagiarism detection systems. Uploading this thesis for the purpose of submitting it to plagiarism detection systems is not a form of publication.

Bonn, 29.2.2024

_____ & _____

&