Tampere University

MohammadJavad Moshiri

BENCHMARKING NATIVE CODE AGAINST WEBASSEMBLY IN WEBASSEMBLY SYSTEM INTERFACE COMPLIANT ENVIRONMENTS

Faculty of Information Technology and

Communication Sciences

Bachelor's Thesis

March 2024

ABSTRACT

MohammadJavad Moshiri: Benchmarking Native Code Against WebAssembly in WebAssembly System Interface Compliant Environments
Bachelor’s Thesis
Tampere University
Bachelor's Programme in Science and Engineering
March 2024

WebAssembly is increasingly recognized, with its ecosystem experiencing rapid growth due to significant investment from companies and ongoing research efforts. The introduction of the WebAssembly System Interface (WASI) potentially transforms WebAssembly runtimes and environments into standalone platforms, a development whose implications have yet to be extensively explored.

This thesis investigates the performance of WebAssembly in comparison to native code across three programming languages: C, Go, and Rust, in scenarios likely to be affected by WASI, including system calls, memory management, and input/output operations. Tests were conducted in three environments: native code on Raspberry Pi Model 4B, and the WebAssembly code on the Wasmtime and Wasmer runtimes, focusing exclusively on execution time.

Results indicate significant variability, ranging from WebAssembly outperforming native code by up to 100 times in system directory access—a result that warrants further investigation—to experiencing a slowdown of 375 times in I/O operations on the Wasmer runtime for C.

This study highlights the case-by-case nature of WebAssembly's performance and underscores the potential of WASI in diversifying the computational landscape.

Keywords: WebAssembly, IoT, Wasmer, Wasmtime, CPU Benchmark, Performance Comparison

PREFACE

I want to deeply thank my supervisor, Professor Kari Systä. His amazing guidance and support helped me stay on track, even when things got hard or time was short. His advice and belief in me made this thesis possible.

I'm also so grateful to my best friends, Sina Saeedi and Saeed Keramati, and everyone in our "Thug Life" group. You all kept me going with your friendship, ideas, and support throughout this journey.

Thank you to everyone at Tampere University, especially my advisors, Kaisu Kalliomäki and Tiina Riihelä. Your willingness to help and your expert advice made my time here better and helped me finish my thesis.

I also want to express my sincere gratitude to Finland, a country that provided me with an incredible life experience and the chance to receive a top-quality education, which wouldn't have been available to me otherwise.

I couldn't have done this without all of you.

Tampere, March 8, 2024

MohammadJavad Moshiri

CONTENTS

1. Introduction **7**

1.1 Background 7

1.2 Motivation 8

2. Related Work **9**

2.1 Compiler Optimizations and WebAssembly 9

2.2 Performance Discrepancies and System Calls 9

2.3 Potential Beyond the Web 10

2.4. Research Gap and Thesis Contribution 10

3. Methodology **10**

3.1 Approach 11

3.2 Tools and Technologies 11

3.2.1 Hardware 12

3.2.2 Runtimes 12

3.2.3 Languages 12

3.2.4 Compiler 13

3.3 Benchmarking 14

3.4 Tests 15

4. Results **17**

4.1 Loop 18

4.2 Fibonacci 19

4.3 Dynamic Memory Allocation 20

4.5 Time (System Call) 23

4.6 Get Working Directory (System Call) 24

5. Future Work **25**

6. Conclusion **26**

7. Appendices **28**

8. References **29**

LIST OF FIGURES

Figure 1: Pseudocode of Benchmarking method. 14
Figure 2: Legends for the result plots used throughout the study. 17
Figure 3: Average (Noise-Trimmed) Time Taken, Loop Module 18
Figure 4: Average (Noise-Trimmed) Time Taken, Fibonacci Module 19
Figure 5: Adjusted Average (Noise-Trimmed) Time Taken, DMA Module 21
Figure 6: Adjusted Average (Noise-Trimmed) Time Taken, Standard Output Module 22
Figure 7: Adjusted Average (Noise-Trimmed) Time Taken, Time Module 23
Figure 8: Adjusted Average (Noise-Trimmed) Time Taken, Get_WD Module 25

LIST OF SYMBOLS AND ABBREVIATIONS

ARM: Acorn RISC (Reduced Instruction Set Computer) Machine

API: Application Programming Interface

BROWSIX: Browser-based Unix

CPU: Central Processing Unit

GB: Gigabyte

WD: Working Directory

IoT: Internet of Things

I/O: Input/Output

RAM: Random Access Memory

WASI: WebAssembly System Interface

WASM: WebAssembly

1. Introduction

1.1 Background

WebAssembly, a modern binary instruction format, represents a significant milestone in the evolution of web technologies. Developed through the collaborative efforts of major web browsers, it aimed to revolutionize web applications by enabling the execution of code written in languages other than JavaScript at near-native speeds. This innovation has marked the beginning of a new era in web development, initially aimed at overcoming the performance limitations of JavaScript.[1]

Offering portability, safety, and an experience close to native applications, WASM's benefits quickly extended its appeal beyond web applications to diverse software domains [2], [3]. Just to give a few examples, as of the date of publishing this thesis, CloudFlare, integrates WASM in its Workers for efficient server-side operations [4] Similarly, Siemens explores WASM's potential in IoT at Carnegie Mellon University's WebAssembly Research Center [5] Moreover, HotG, an IoT startup[6] leverages WASM for deploying machine learning models on edge devices, demonstrating its broad applicability and performance advantages across various domains.

The ecosystem around WebAssembly has seen rapid expansion, with increased support for various programming languages through compilers like Emscripten [7], and the introduction of new languages such as AssemblyScript [8]. Additionally, runtimes including Wasmtime [9] and Wasmer [10] provide the infrastructure needed to execute WASM modules across different environments.

The introduction of the WebAssembly System Interface has been a key development in broadening WASM's application scope. WASI was created to extend WebAssembly's capabilities beyond the browser, offering a standardized system interface that enables WASM modules to interact with the underlying system regardless of the operating system. This initiative supports WASM's principles of portability and security, facilitating the execution of applications across various platforms without compromising safety. [11]

WebAssembly's influence now extends beyond web browsers into sectors such as the Internet of Things, gaming, and more, as exemplified by the cases mentioned above and

as highlighted on the official WebAssembly website [12]. Its versatility and potential to reshape software development landscapes underscore the importance of WASM in the programming domain.

1.2 Motivation

The growth of the WebAssembly ecosystem has ignited research across various domains, exploring its applications and implications. Investigations have covered areas such as performance[13], application scopes [14], security aspects [15], and even energy consumption [16], reflecting the dynamic evolution spurred by ongoing developments and the contributions of the WebAssembly Working Group [17]. The introduction of new features and enhancements, alongside the identification and resolution of limitations, signifies the maturing landscape of WASM and the deepening understanding within the developer and research communities.

Although prior research has extensively compared the performance of WebAssembly to native code, primarily focusing on theoretical and technical dimensions, there remains a gap in practical, application-oriented analyses. This is particularly true for studies that consider WASM code in environments enabled by the WASI, which extends WASM's capabilities beyond traditional web settings.

This study aims to fill this gap by examining the performance of various code segments compiled for WASI-compliant environments relative to their native executions across different system functionalities. Specifically, it will focus on quantifying performance differences in critical operations, such as I/O operations, system calls, and memory management, within environments that support these features. The significance of this research lies in its potential to offer practical insights into the efficiency and applicability of WASM in real-world applications, especially those leveraging WASI to broaden WASM's utility beyond web browsers.

1.3 Objectives

This thesis aims to analyze CPU performance differences between WASI-compliant WebAssembly and native code execution, specifically focusing on C, Go, and Rust. The objectives are:

Comparative Performance Analysis: Measure the execution times of identical code implementations compiled to WebAssembly and their native counterparts. This analysis will highlight CPU performance overheads potentially introduced by the WebAssembly environment.

Language-Specific Compilation Impacts: Evaluate the efficiency of compiling C, Go, and Rust into WebAssembly. Analyze performance variations arising from different language constructs and their translation into WebAssembly instructions.

Important Note: This study will establish a performance baseline with optimizations disabled. Memory usage, file sizes, and other metrics are outside the thesis's scope.

2. Related Work

WebAssembly has been identified as a pivotal technology for enhancing web application performance. The foundational work by Haas et al. set the stage by discussing the motivations and challenges WebAssembly aims to overcome [1] Since then, research has broadened into assessing WebAssembly's performance across various environments and identifying factors affecting its efficiency.

2.1 Compiler Optimizations and WebAssembly

A significant study by Yan Y. et al. focused on WebAssembly's behavior in web applications, comparing it with JavaScript to evaluate the effects of compiler optimizations and execution environments on memory usage, CPU utilization, and code size. They found that optimizations typically applied to JavaScript do not translate effectively to WebAssembly, suggesting a misalignment with WebAssembly's distinctive properties. Their analysis, concentrating on web-centric compilers like Cheerp [18] and Emscripten, provides insights into how WebAssembly performs in different browsers and runtime environments [13]

2.2 Performance Discrepancies and System Calls

Jangda A. et al.'s research challenges the assumption that WebAssembly achieves near-native performance, pointing out that initial studies, including Haas et al.'s, may have

overlooked the complexity of real-world applications. They introduced BROWSIX-WASM, a tool for running Unix applications in the browser via WebAssembly, to conduct a comprehensive analysis using SPEC CPU benchmarks. Their findings reveal a more considerable performance gap than previously reported, attributed to various technical limitations and the absence of standard Unix APIs in browsers. This study emphasizes the need for further optimization and development to close the performance gap in web environments. [15]

2.3 Potential Beyond the Web

The exploration of WebAssembly's potential is not limited to web applications. Research by Spies B. and Mock M. illustrates its capacity to match, and occasionally surpass, native code performance, significantly when combined with the WebAssembly System Interface (WASI). This combination highlights WebAssembly's emerging role outside traditional browser applications, signaling its adaptability and utility in broader contexts.[19]

2.4. Research Gap and Thesis Contribution

Despite extensive examination of WebAssembly within web browsers, its application in conjunction with WASI, especially regarding IoT devices, remains underexplored. This thesis positions itself to bridge this gap by conducting a preliminary performance analysis of WebAssembly and WASI within the IoT domain. The aim is to establish a foundational understanding of WebAssembly's CPU efficiency and potential as a secure, lightweight, and portable execution environment for IoT applications.

3. Methodology

This section outlines the systematic approach and procedural steps undertaken to achieve the objectives of this research. It describes the foundational principles guiding methodological choices, the selection of tools and technologies, and the criteria for their use in the study. By delineating the method, this section provides a blueprint of the research process, detailing how the study was designed, executed, and analyzed to explore the performance of WebAssembly and WASI in IoT applications.

3.1 Approach

The formulation of this thesis was driven by three main goals, reflecting a strategic methodological framework designed to optimize the research process under specific constraints:

- **Simplicity:** Recognizing the constraints imposed by a strict timeline, the approach was to simplify the methodological framework wherever possible. This involved the selection of straightforward tools and techniques, aiming to reduce complexity for more efficient implementation and to facilitate a clearer understanding of the research process.
- **Common Tools:** To enhance accessibility, reproducibility, and potential for adoption of the research findings, there was a deliberate focus on employing popular and widely recognized tools and technologies. This strategy ensures broader compatibility and lessens the learning curve for individuals seeking to apply or expand upon the thesis work.
- **Stability:** The research aimed at producing dependable results using established operating systems, runtimes, programming languages, and code libraries. By prioritizing stability, the research minimizes the risk of encountering unforeseen errors or compatibility issues, thus fostering a more reliable and consistent research environment.

These guiding principles shaped the design and execution choices made throughout the research. The subsequent sections will elaborate on the specific tools, technologies, and procedures adopted in alignment with these goals.

3.2 Tools and Technologies

This section delineates the specific hardware, runtimes, and programming languages employed in this research, detailing the rationale behind their selection and their roles in the study's methodology. The choice of tools and technologies is integral to the research design, influencing both the feasibility and the accuracy of the experimental results. Each category reflects a deliberate choice aimed at aligning with the study's emphasis on simplicity, commonality, and stability.

3.2.1 Hardware

Central to the experimental setup is the selection of appropriate hardware that matches the IoT environment's constraints and requirements. The Raspberry Pi 4 Model B was chosen as the primary device for this study due to its robust performance specifications and widespread adoption in IoT projects. Equipped with a Quad-core Cortex-A72 (ARM v8) 64-bit CPU, this model offers a powerful computing platform conducive to evaluating WebAssembly's performance. Additionally, the version with 8 gigabytes of RAM, complemented by an 8GB Kingston industrial microSD card, was selected to ensure ample processing and storage capabilities for the research tasks. [20],[21]

3.2.2 Runtimes

The software environment is equally critical to the research's success, with specific attention given to the operating system and WebAssembly runtime choices. RaspiOS Lite 64-bit,[22] based on Debian and running the Linux kernel version 6.1.0 as of November 24, 2023, was selected for its lightweight nature and optimization for Raspberry Pi hardware. Its minimalistic setup is ideal for IoT projects, where resource efficiency is paramount. The operating system's broad support and popularity within the Raspberry Pi community further justify its selection, as highlighted by a 2018 IoT survey. [23][24]

For executing WebAssembly, Wasmer 4.2.5 and Wasmtime 18.0.2 were chosen. These runtimes are recognized for their industry-standard support for WebAssembly and the WASI, offering a versatile and reliable platform for general-purpose use. The latest stable versions available during the testing phase were selected to ensure the research utilized the most up-to-date and supported technology. [9], [10]

3.2.3 Languages

The programming languages chosen for compiling to WebAssembly were C, Go, and Rust. These languages were identified as WebAssembly product-ready during the research period, supported by their listing in the Awesome WebAssembly Languages GitHub repository. This repository is a curated resource tracking the development of WebAssembly-compatible languages, including their compilers and virtual machines.[25] The popularity of C, Go, and Rust, reinforced by rankings such as the IEEE Spectrum's top programming languages of 2023, substantiates their selection for this study. The choice of these languages aims to explore the compilation efficiency and runtime

performance of WebAssembly across different programming paradigms and language features [26]

3.2.4 Compiler

In this study, compiler optimizations were intentionally disabled to establish a level playing field for comparing the performance of native execution and WASM. This approach focuses on evaluating the inherent efficiency of the code and runtime environments, free from the potential distortions introduced by compiler optimization techniques. By doing so, the research aims to isolate and measure the performance overhead and characteristics inherent to WASM compared to native execution, thereby providing a clear baseline for understanding the fundamental differences between these execution contexts. This strategy not only aids in producing reproducible results but also simplifies the debugging process by ensuring a closer correlation between the source code and its compiled output. Ultimately, the decision to disable compiler optimizations ensures that any observed performance discrepancies stem directly from the execution environments themselves, not from the varied impact of compiler optimizations.

C:

For the compilation of C language code, clang version 17.0.6,[27] in conjunction with the WASI SDK version 21.0, was chosen.[28] These versions represent the most up-to-date stable releases available during the testing phase. In the native compilation process, the settings were tuned for the Cortex-A72 [29] architecture to enhance code performance on this specific hardware setup. This included specifying the target (aarch64-linux-gnu) and machine architecture (ARMv8-A), optimizing the compiled output for Cortex-A72 CPU-equipped devices. Both native and WebAssembly compilations were conducted with optimization level set to 0, maintaining the focus on the raw performance characteristics of each execution environment without the influence of compiler optimizations.

Rust:

For Rust, the compiler version used was rustc 1.75.0, with optimizations disabled, debug information omitted, and debug assertions turned off.[30] For native compilation, the Clang compiler (version 17.0.6, the same as used for C) was specified as the linker, with flags to target the aarch64-linux-gnu architecture, aligning with the Cortex-A72 CPU

characteristics. This setup ensured that Rust code could be compiled to both native and WASM targets under comparable conditions.

Go:

The Go language compilation utilized the TinyGo compiler[31], a variant of the Go compiler designed specifically for embedded systems and devices with constrained resources. TinyGo was chosen for its advantages in optimizing Go code for such environments, making it more suitable for embedded systems compared to the standard Go compiler [32] For native compilation, settings were adjusted to target ARM64 architecture and Linux operating system. Additionally, optimizations were disabled, scheduler were turned off, and garbage collection was turned off as well. The effect of garbage collection on the language is outside the scope of this program.

3.3 Benchmarking

To assess the performance of C, Go, and Rust in relation to WebAssembly, a straightforward and replicable benchmarking approach was employed. The methodology follows a simple yet effective pattern to measure execution time accurately:

1. Initialization of a Timer: At the beginning of the main function, a high-resolution timer is initiated to mark the start time.
2. Execution of Test Code: The specific operations or computations meant to be benchmarked are placed here.
3. Final Timer Check: Immediately following the test code, the timer is checked again to mark the end time.

```
main()
{
    timer T;
    T.run();

    TestCase();

    T.stop();
    print(T.duration);
}
```

Figure 1: Pseudocode of Benchmarking method.

This structure ensures that only the test code's execution time is measured, excluding initial setup or subsequent operations that are not part of the benchmark. Furthermore, the timing method uses the internally available calculation of time provided by the standard libraries of each language [33], [34], [35]

To ensure the accuracy and consistency of our benchmarking results, each test is executed 500 times. This repetition is crucial for mitigating the variability in execution times due to transient system or environmental factors. To give the benchmarked operations the highest priority and ensure minimal interference from other processes, each test execution is configured to run with the highest possible scheduling priority. This is achieved by utilizing the nice command, a tool that adjusts the scheduling priority of a process in the operating system [36] The results from these executions are collected for further statistical analysis.

While testing, the CPU power management is set to 'performance' mode. This configuration, combined with the prioritization provided by the nice command, ensures that the benchmarking process is conducted under optimal conditions for accurate measurement of the test functions' execution times. By implementing these measures, we aim to minimize external variables that could skew the results, thus providing a reliable assessment of the performance.

3.4 Tests

The initial plan was centered on evaluating dynamic memory allocations, I/O operations, filesystems, threading, system calls, and networks. However, adjustments were made to the scope of the study due to several factors. Threading was excluded from the investigation as the proposal was only in Phase 1, indicating that the feature was not yet available for implementation.[37], [38] Similarly, filesystems and networks were omitted due to the challenges associated with implementing a fair assessment within the constraints of technical feasibility and time.

To ensure more accurate and meaningful results, we've implemented a strategy where each test case, excluding the Fibonacci sequence test, is wrapped within a loop that

iterates 50,000 times. For instance, when testing memory allocation by adding nodes to a linked list, the operation is repeated in a loop like so:

```
// timer starts
for (i := 0; i < 50000; i++) {
    // Add node to linked list
}
// timer stops
```

This method allows us to precisely measure the actual time each test takes by repeatedly executing the test cases, thereby enhancing the reliability of our measurements.

To assess performance aspects, Dynamic memory allocation involved the addition of a new node to a linked list during each loop iteration, I/O operations were evaluated by printing a line to the standard output within each cycle, and System call impacts were assessed through two distinct tests: the acquisition of system time and the retrieval of the current working directory, both performed within each loop iteration.

Finally, attention was given to a time-intensive calculation function that refrains from utilizing any WASI related features. This function served as a benchmark for comparing the performance of the runtime environment against native execution when code is compiled using the WASI Software Development Kit. The function chosen for this comparison was the Fibonacci sequence without intermediate caching, represented by recursive function $F(n) = F(n-1) + F(n-2)$ which exhibits an exponential time complexity of $O(2^n)$. This selection aimed to provide a reference point for evaluating the time performance of the functions.

It is also noteworthy that all tested source codes, their compiled versions, available binaries, and a guide on how to run them are accessible in the GitHub repository, which is detailed in appendix A. This inclusion ensures that the research is transparent and reproducible, allowing others to verify the findings and conduct further investigations based on the provided resources.

3.5 Statistical Analysis

Following the completion of measurements (see Appendix B), a statistical analysis was employed to extract meaningful insights from the collected data and minimize the influence of potential outliers or anomalies. To minimize the impact of potential noise, the top 10% of results were trimmed from dataset. Subsequently, the trimmed average and the 90th percentile were calculated. The 90th percentile provides insight into the reliability of the trimmed average.

For modules containing loops, both the trimmed average and 90th percentile had the corresponding average of the empty loop (specific to the language and runtime) subtracted. This adjustment isolates the true execution time of the tested component.

Appendix C presents additional statistical analyses that were performed. However, these have been omitted from the main text to maintain conciseness.

4. Results

This section presents the results of the performance evaluation, organized by test module. Each module is accompanied by relevant information and analysis. All plots throughout the study adhere to the legends defined in Figure 2.

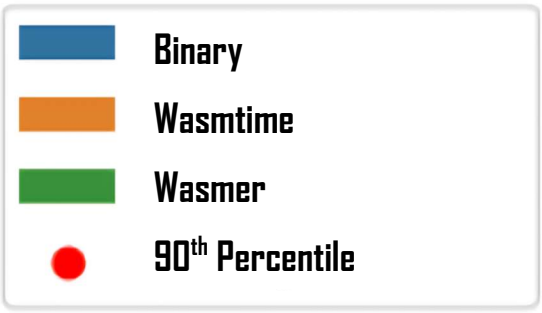

Figure 2: Legends for the result plots used throughout the study.

4.1 Loop

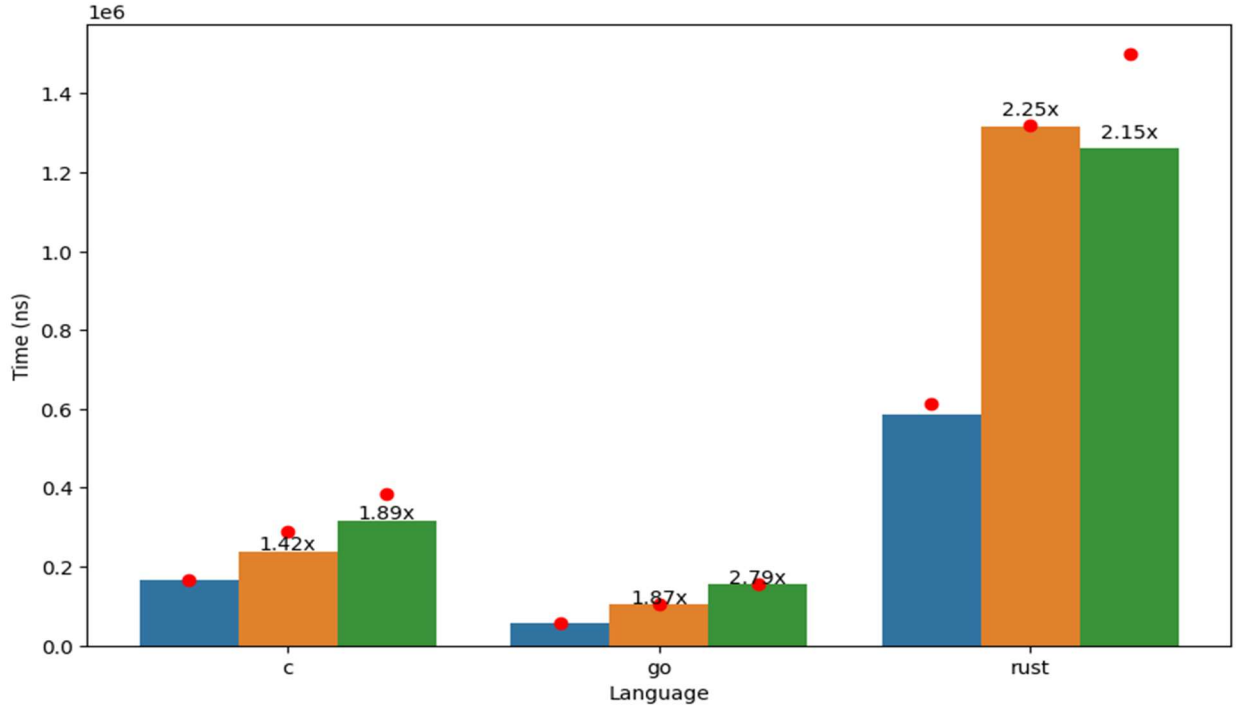

Figure 3: Average (Noise-Trimmed) Time Taken, Loop Module

The Loop module serves as the baseline for empty loop performance. It consists of a loop that increments a variable. As previously stated, this loop iterates 50,000 times, and in each iteration, the variable of the loop is increased by 1. We are going to use this data to subtract it from other test cases to obtain a more accurate measurement for the actual processing time.

The plot reveals that Wasmtime-C exhibits the lowest performance slowdown, requiring 42% more execution time than a native C binary. Conversely, Wasmer-Go demonstrates the highest slowdown, taking 2.79 times longer than its native counterpart.

	Binary	Wasmtime	Wasmer
C	1	1.22	1.22
Go	1	1.01	1
Rust	1.05	1	1.19

Table 1: Ratios of 90th Percentile to Trimmed Mean, Addition Module

Generally, ratios close to 1 indicate consistent data, while higher ratios suggest greater variability. As observed in Table 1, most modules exhibit ratios near 1, signifying reliable measurements. However, Wasmtime-C, Wasmer-C and Wasmer-Rust present ratios around 1.2, which implies more variable data in these specific datasets.

Side Note: It's interesting to observe that despite turning off optimizations as much as possible, Native Go is performing around 3 times faster than Native C, and Rust is twice as slow than C. This might seem unconventional. This phenomenon is potentially due to unavoidable optimizations within the compiler itself, even with disabled optimizations. The way different compilers translate the code can also play a role in these performance variations.

4.2 Fibonacci

The Fibonacci module provides a benchmark for assessing how WASI-enabled WASM code generally performs within the chosen runtime environments.

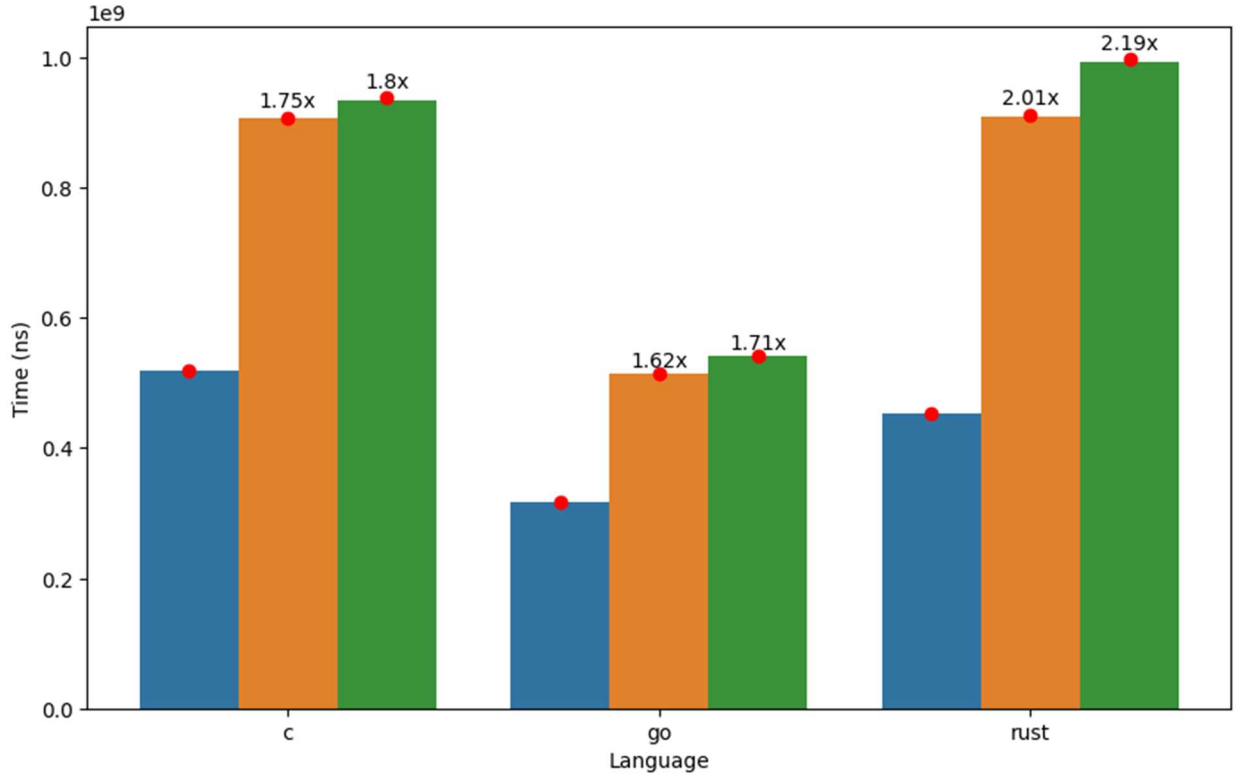

Figure 4: Average (Noise-Trimmed) Time Taken, Fibonacci Module

This is the only module that is not built on top of the loop module, and the results are not affected by the loop. This comparison establishes a reference point for understanding the runtime impact on binary versus WASM execution. Specifically, it calculates the 37th element of the Fibonacci series through a recursive call without any intermediate caching. This approach intentionally avoids optimizations to purely measure the computational efficiency and the overhead of recursion in WASM environments, offering insight into how these environments handle intensive computational tasks without assistance from optimization techniques such as caching.

The average execution times reveal performance ratios ranging from 1.75 for Wasmtime-C to 2.19 for Wasmer-Rust. It's worth noting that this module exhibits lower variability compared to the Addition module.

Interestingly, the close alignment of the 90th percentile and the trimmed average indicates consistent timing results. This consistency could be attributed to the computationally demanding nature of the Fibonacci calculation, where execution times fall within a larger time scale (1e9 scale on Y-axis), potentially improving measurement precision.

4.3 Dynamic Memory Allocation

Dynamic Memory Allocation module incorporates a loop where a linked list is dynamically expanded by adding a new element with each iteration, its performance metrics— 90th percentile and trimmed average—were adjusted by subtracting the corresponding values of the Loop module. This adjustment ensures a more accurate representation of the true overhead associated with dynamic memory operations within the WASM runtimes. It's important to note that the time measurements reported here are exclusively for the allocation of memory. The deallocation of memory, which could also influence performance, is beyond the scope of this thesis and has not been accounted for in the analysis.

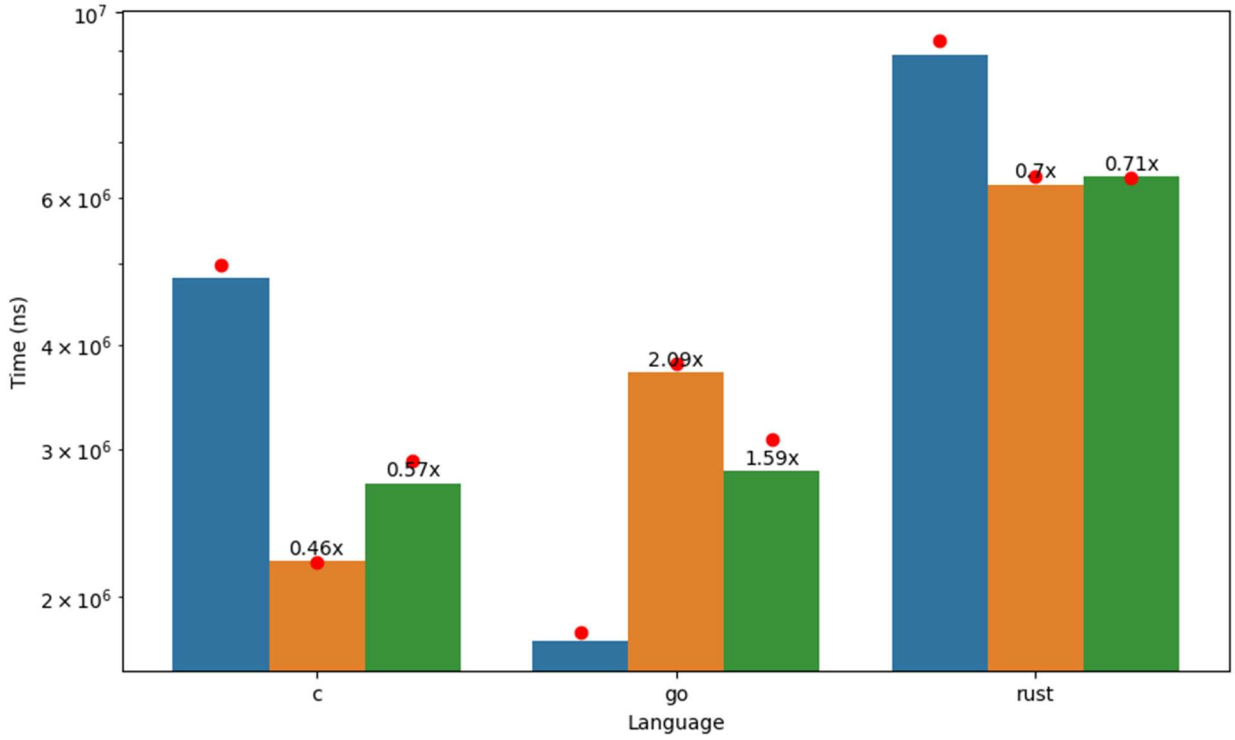

Figure 5: Adjusted Average (Noise-Trimmed) Time Taken, DMA Module

Surprisingly, the results demonstrate that WASM execution with WASI outperforms native C and Rust. While a definitive explanation lies beyond the scope of this study, a plausible hypothesis is that WASM runtimes possibly preallocate dynamic memory in fixed segments. This approach would reduce the frequency of memory reallocation and data transfers. Alternatively, or in addition, this performance advantage might also be the effect of compilation strategies employed by the WASM runtimes.

	Binary	Wasmtime	Wasmer
C	1.04	1	1.06
Go	1.02	1.03	1.09
Rust	1.04	1.02	1

Table 2: Ratios of Adjusted 90th Percentile to Adjusted Trimmed Mean, Dynamic Memory Module Despite

Despite the appearance in the plot due to the logarithmic scale on the Y-axis, the ratios between the 90th percentile and the trimmed mean remain consistent, indicating reliability in the average.

4.4 I/O Operation (Standard Output)

I/O operation in the context of this test happens by printing the loop iteration number into standard output. For this test, we redirected the standard output to a temporary file—a process handled by the operating system and not the program itself. Moreover, to ensure fairness and consistency across tests, a new temporary file is created for each test, and a flush (sync command)[39] is executed before each test to clear any potential buffers. As with the Dynamic Memory Allocation module, performance metrics for the I/O Operation module were adjusted by deducting the corresponding values from the Loop module.

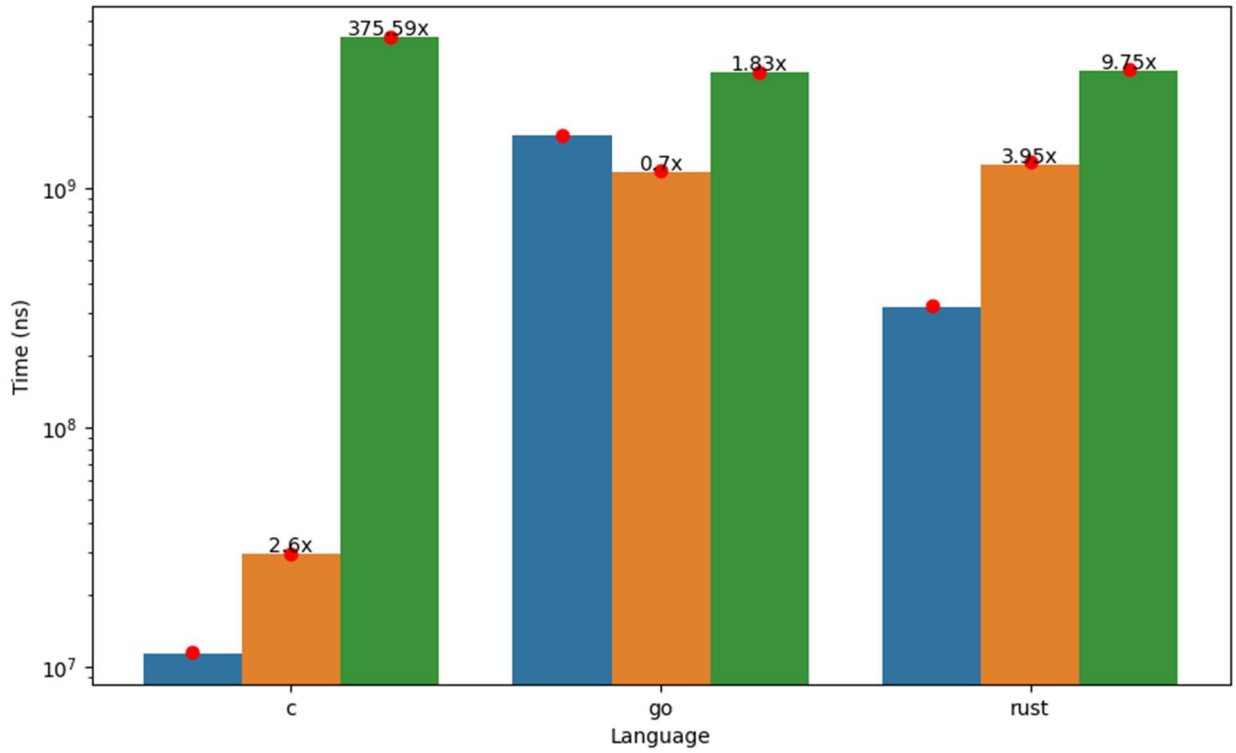

Figure 6: Adjusted Average (Noise-Trimmed) Time Taken, Standard Output Module

The data reveals significant variations in execution time between different language-runtime combinations, even outperforming Go in Wasmtime runtime. Wasmer exhibits considerably slower performance for all three languages compared to Wasmtime. Notably, Wasmer-C demonstrates the most significant slowdown, taking approximately

375 times longer than native C code to perform the same I/O operation.

The large slowdowns observed with Wasmer suggest potential shortcomings in its handling of write operations to standard output. While the extent of the slowdown varies across languages, the overall WASM execution times remain comparable for Go and Rust. This consistency might suggest a fixed overhead for these languages, irrespective of the source language used for WASM compilation.

The consistent behavior of the 90th percentile in most instances suggests reliable measurements, eliminating the need for further analysis of this metric in this module.

4.5 Time (System Call)

Time is one of the test cases designed to measure the efficiency of system calls. In this scenario, the system calls to obtain the current time are executed in each iteration of the loop. Like previous modules, the presented values have been adjusted to account for the loop's influence.

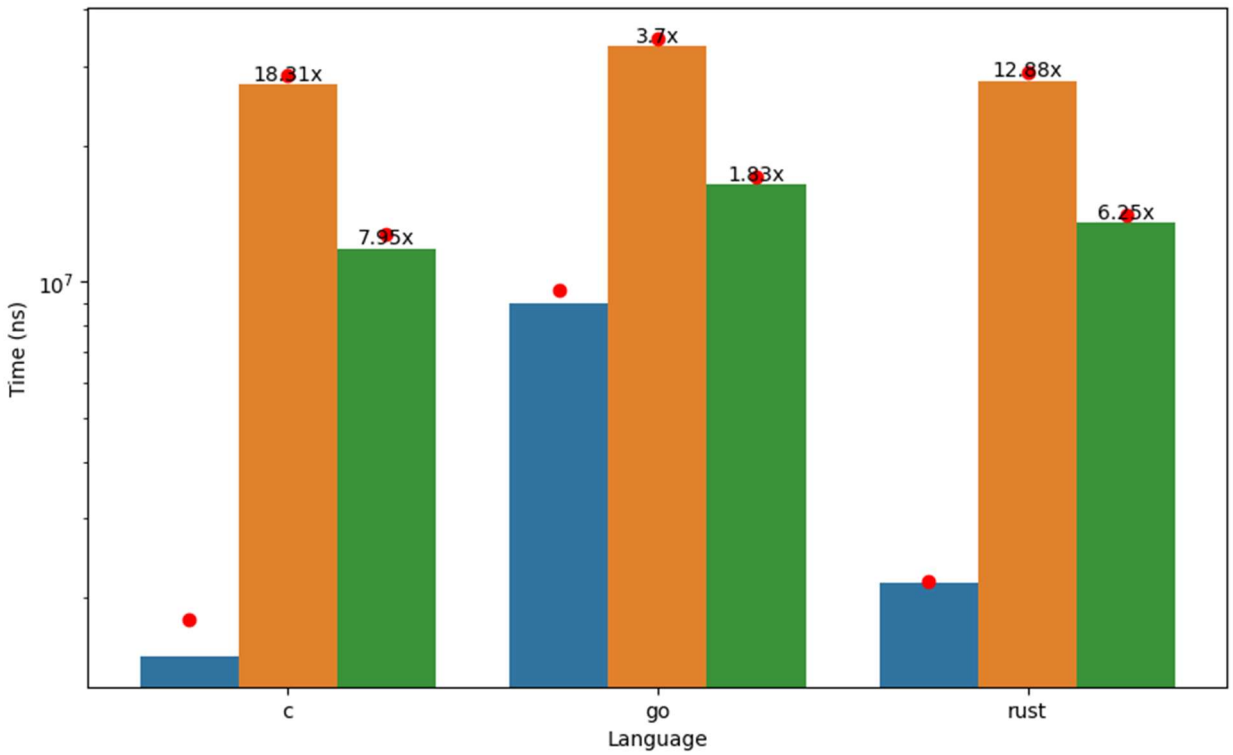

Figure 7: Adjusted Average (Noise-Trimmed) Time Taken, Time Module

The graph reveals the most significant slowdown for C code compiled to WASM, with both Wasmtime and Wasmer exhibiting slower performance compared to native execution. This might be attributed to the inherent use of the same system component within the C language and the Linux runtime, and potentially larger difference when it compiles to WASM.

It's also noteworthy that Go demonstrates the smallest slowdown percentage compared to its native binary counterpart, even though its overall execution time is the highest among the three languages.

We can see that execution times in WASM runtimes are comparable across different languages, further indicating that WASI manages this aspect without language dependency.

	Binary	Wasmtime	Wasmer
C	1.2	1.05	1.08
Go	1.07	1.04	1.04
Rust	1.01	1.05	1.04

Table 3: Ratios of Adjusted 90th Percentile to Adjust Trimmed Mean, Time Module

4.6 Get Working Directory (System Call)

This module evaluates the performance of executing the system call to get the current working directory[40], which serves as another targeted case study for assessing system interaction within our testing framework. For each iteration of the loop, the system calls to retrieve the current working directory is made. Like the approach taken with previous modules, the results have been adjusted to negate the influence of the loop structure itself.

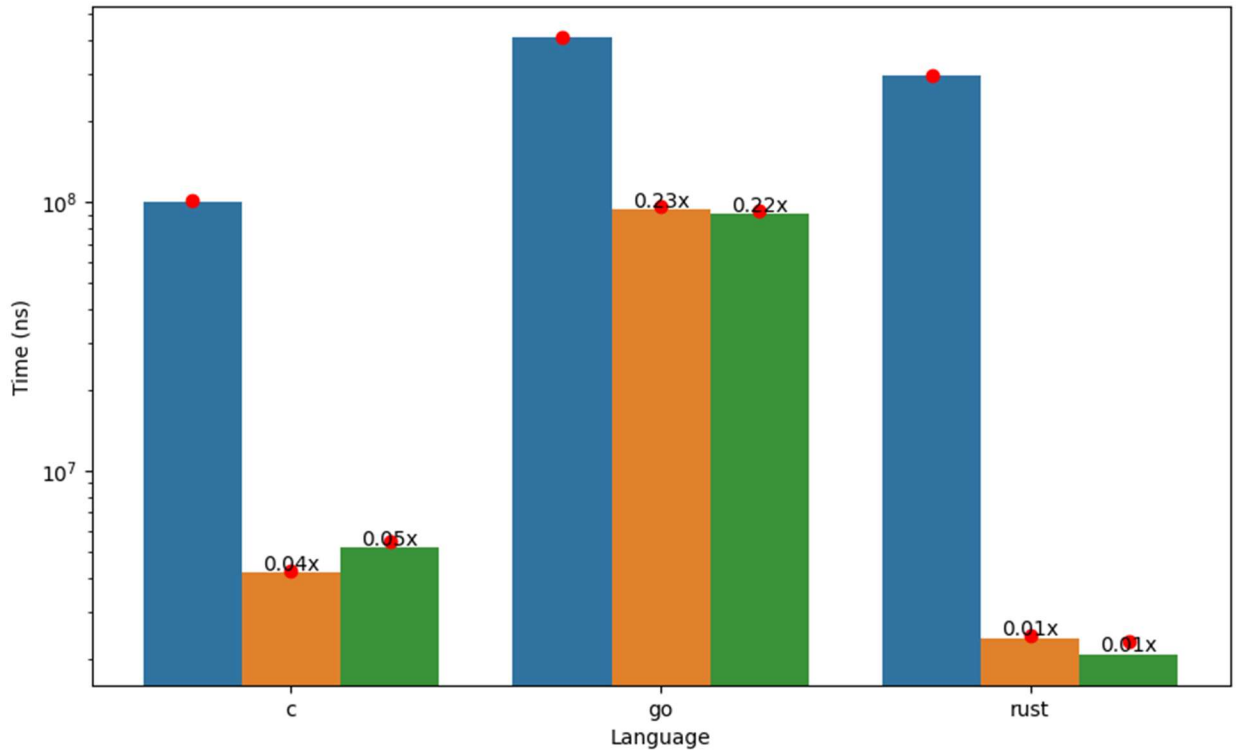

Figure 8: Adjusted Average (Noise-Trimmed) Time Taken, Get_WD Module

This is the most unexpected result we have observed so far: both WASM runtimes outperformed their native counterparts significantly. Although a thorough investigation into the precise reasons for this discrepancy lies beyond the scope of this thesis, one possible explanation could be that WASM runtimes optimize performance by caching the result of the first get_wd system call. They might not invoke a second system call unless there has been a change in the directory, effectively short-circuiting the command.

5. Future Work

Building on the insights gleaned from this study, future research could delve deeper into the mechanisms underlying the performance characteristics observed across different WASM runtimes and programming languages. By unraveling these, there lies a potential treasure trove of optimization opportunities within WASI that could significantly enhance execution efficiency and broaden the applicability of WASM in diverse computing environments.

Moreover, incorporating more proposals of WASI and possibly utilizing other metrics to gauge performance, alongside testing on larger scales with real-world compound examples, could shed light on how WASM behaves under different operational stresses and in varied application contexts. Such comprehensive exploration is pivotal for charting the path forward in the optimization of WASI-enabled WASM code, ultimately facilitating the development of high-performance, cross-IoT applications.

Additionally, an exploration into the compilation strategies employed by WASM runtimes—whether Just-In-Time or Ahead-Of-Time —and their impact on performance and resource utilization offers a promising avenue for research. Understanding the trade-offs and benefits of these compilation strategies could provide insights into their contribution to the results observed in this study and how they might be leveraged for further optimization.

Furthermore, given the non-OS independence of system calls, investigating the behavior of WASM across a broader array of system call implementations and scenarios becomes crucial. Testing a wide range of system calls across multiple operating systems and environments can unveil performance discrepancies and opportunities for improvement.

6. Conclusion

In this study, we explored the performance characteristics of WebAssembly across different runtimes (Wasmtime and Wasmer) and programming languages (C, Go, and Rust), focusing on a variety of computational and system interaction modules when WASI is enabled.

The Fibonacci and Loop modules demonstrated more consistent results, suggesting that parts of the code that do not directly interact with the WASI SDK can expect a predictable percentage-wise slowdown. This consistency provides valuable insights into the expected performance overhead for computational tasks within a WASM environment.

Furthermore, the study shed light on the impact of memory management and the compilation process on performance. The observed faster execution times in certain tests

highlight the significant role that the choice of compiler and the mapping of code to WASM play in overall application performance. These findings suggest that even within the

constraints of a WASM execution environment, there are opportunities to optimize performance through careful selection of programming languages and compilation strategies.

When examining write operations and system calls, which require more direct interaction with the underlying system, the effect of the chosen language and runtime becomes more pronounced, with results varying dramatically. This variability underscores the complexity of optimizing WASM performance for system-level operations. Interestingly, the overhead introduced by the WASM runtimes, particularly in terms of execution time, appears to be somewhat fixed, hinting at the possibility of achieving better relative performance in larger-scale applications.

It is important to note that the performance metrics obtained in this study differ significantly from those discussed in the original article on WASM and from the findings analyzed in the related work section. This divergence is not unexpected, given the myriad changing parameters in our experimental setup, including the introduction of the WASI target, the use of runtimes as standalone environments, and the disabling of optimizations. Our approach was designed to establish a baseline for what could be expected from a direct translation of code to WASM without the benefit of optimization, providing a foundation for future research and development in this area.

In conclusion, this thesis contributes to a deeper understanding of WASM's performance landscape, highlighting the influence of language choice, runtime environment, and the nature of the task on execution efficiency. Our findings point to both the potential and the challenges of leveraging WASM for cross-platform application development, offering a steppingstone for further exploration and optimization in the field.

7. Appendices

Appendix A: GitHub Repository

URL:

https://github.com/LiquidAI-project/wasm-comparison

Appendix B: Results File

URL:

https://github.com/LiquidAI-project/wasm-comparison/blob/master/results/2024-03-08-19-00.txt

Appendix C: Analysis Notebook

URL:

https://github.com/LiquidAI-project/wasm-comparison/blob/master/results/analysis_500.ipynb

8. References

[1] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 185–200, Sep. 2017, doi: 10.1145/3140587.3062363.

[2] A. Hilbig, D. Lehmann, and M. Pradel, "An Empirical Study of Real-World WebAssembly Binaries," in *Proceedings of the Web Conference 2021*, New York, NY, USA: ACM, Apr. 2021, pp. 2696–2708. doi: 10.1145/3442381.3450138.

[3] P. P. Ray, "An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions," *Future Internet*, vol. 15, no. 8, p. 275, Aug. 2023, doi: 10.3390/fi15080275.

[4] Kenton Varda, "WebAssembly on Cloudflare Workers." Accessed: Mar. 08, 2024. [Online]. Available: https://blog.cloudflare.com/webassembly-on-cloudflare-workers

[5] Arturo Pizano, "Siemens joins WebAssembly Research Center at Carnegie Mellon University." Accessed: Mar. 08, 2024. [Online]. Available: https://ecosystem.siemens.com/researchandinnovation/siemens-joins-webassembly-research-center-at-carnegie-mellon-universityhttps://ecosystem.siemens.com/researchandinnovation/siemens-joins-webassembly-research-center-at-carnegie-mellon-university

[6] "MLOps for the Edge", Accessed: Mar. 08, 2024. [Online]. Available: https://hotg.dev/

[7] "Emscripten ", Accessed: Jan. 04, 2024. [Online]. Available: https://emscripten.org/

[8] "AssemblyScript official website ." Accessed: Jan. 07, 2024. [Online]. Available: https://www.assemblyscript.org/

[9] "Wasmtime." Accessed: Jan. 05, 2024. [Online]. Available: https://wasmtime.dev/

[10] "Wasmer." Accessed: Jan. 06, 2024. [Online]. Available: https://wasmer.io/

[11] Lin Clark, "Standardizing WASI: A system interface to run WebAssembly outside the web." Accessed: Mar. 07, 2024. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[12] "Use Cases." Accessed: Jan. 04, 2024. [Online]. Available: https://webassembly.org/docs/use-cases/

[13] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the performance of webassembly applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, New York, NY, USA: ACM, Nov. 2021, pp. 533–549. doi: 10.1145/3487552.3487827.

[14] A. Romano and W. Wang, "WASim," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA: ACM, Dec. 2020, pp. 1321–1325. doi: 10.1145/3324884.3415293.

[15] M. Kim, H. Jang, and Y. Shin, “Avengers, Assemble! Survey of WebAssembly Security Solutions,” in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, IEEE, Jul. 2022, pp. 543–553. doi: 10.1109/CLOUD55607.2022.00077.

[16] J. De Macedo, R. Abreu, R. Pereira, and J. Saraiva, “On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet?,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, IEEE, Nov. 2021, pp. 255–262. doi: 10.1109/ASEW52652.2021.00056.

[17] “WebAssembly Working Group.” Accessed: Jan. 06, 2024. [Online]. Available: https://www.w3.org/groups/wg/wasm/

[18] Leaning Technologies, “Cheerp: The C++ compiler for the Web.” Accessed: Feb. 24, 2024. [Online]. Available: https://leaningtech.com/cheerp/

[19] B. Spies and M. Mock, “An Evaluation of WebAssembly in Non-Web Environments,” in *2021 XLVII Latin American Computing Conference (CLEI)*, IEEE, Oct. 2021, pp. 1–10. doi: 10.1109/CLEI53233.2021.9640153.

[20] “Raspberry Pi 4 Tech Specs”, Accessed: Feb. 24, 2024. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[21] “Industrial microSD Memory Card”, Accessed: Feb. 24, 2024. [Online]. Available: https://shop.kingston.com/products/industrial-microsd-card-memory-card?variant=40558543372480

[22] “Raspberry Pi OS.” Accessed: Mar. 09, 2024. [Online]. Available: https://www.raspberrypi.com/software/

[23] A. D. Raju, I. Y. Abualhaol, R. S. Giagone, Y. Zhou, and S. Huang, “A Survey on Cross-Architectural IoT Malware Threat Hunting,” *IEEE Access*, vol. 9, pp. 91686–91709, 2021, doi: 10.1109/ACCESS.2021.3091427.

[24] “CPU frequency and voltage scaling code in the Linux(TM) kernel.” Accessed: Feb. 26, 2024. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[25] Stephen Akinyemi, “Awesome WebAssembly Languages.” Accessed: Feb. 26, 2024. [Online]. Available: https://github.com/appcypher/awesome-wasm-langs

[26] STEPHEN CASS, “The Top Programming Languages 2023.” Accessed: Feb. 26, 2024. [Online]. Available: https://spectrum.ieee.org/the-top-programming-languages-2023

[27] “Clang: a C language family frontend for LLVM.” Accessed: Mar. 08, 2024. [Online]. Available: https://clang.llvm.org/

[28] “WASI SDK.” Accessed: Feb. 26, 2024. [Online]. Available: https://github.com/WebAssembly/wasi-sdk

[29] “Cortex-A72.” Accessed: Mar. 08, 2024. [Online]. Available: https://developer.arm.com/Processors/Cortex-A72

[30] “What is rustc?” Accessed: Mar. 02, 2024. [Online]. Available: https://doc.rust-lang.org/rustc/what-is-rustc.html

[31] TinyGo Authors, “TinyGo - A Go Compiler For Small Places.” Accessed: Mar. 02, 2024. [Online]. Available: https://tinygo.org/

[32] TinyGo Authors, “Differences from Go.” Accessed: Mar. 02, 2024. [Online]. Available: https://tinygo.org/docs/concepts/compiler-internals/differences-from-go/

[33] Rust Documentation, “Module std::time”, Accessed: Mar. 03, 2024. [Online]. Available: https://doc.rust-lang.org/std/time/index.html

[34] Go Documentation, “Time package.” Accessed: Mar. 03, 2024. [Online]. Available: https://pkg.go.dev/time

[35] GNU Library, “Getting the Time.” Accessed: Mar. 03, 2024. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Getting-the-Time.html

[36] Linux manual, “nice.” Accessed: Mar. 05, 2024. [Online]. Available: https://man7.org/linux/man-pages/man1/nice.1.html

[37] “WebAssembly W3C Process.” Accessed: Mar. 06, 2024. [Online]. Available: https://github.com/WebAssembly/meetings/blob/main/process/phases.md

[38] “wasi-threads.” Accessed: Mar. 06, 2024. [Online]. Available: https://github.com/WebAssembly/wasi-threads

[39] “sync(2) - Linux man page.” Accessed: Mar. 09, 2024. [Online]. Available: https://linux.die.net/man/2/sync

[40] “getcwd(3) — Linux manual page.” Accessed: Mar. 09, 2024. [Online]. Available: https://man7.org/linux/man-pages/man3/getcwd.3.html