Bachelor Thesis

Computer Science and Engineering 300 credits

Edge Computing on Embedded Systems using WebAssembly

Computer Science and Engineering 15 credits

Halmstad June 23, 2025

Mattias Neidenström

Ernst Peterson

Abstract

This thesis explores the viability and performance of WebAssembly (Wasm) for edge computing on embedded systems. Specifically, it evaluates whether WebAssembly, when executed on a lightweight embedded operating system such as Zephyr, can deliver adequate computational efficiency for typical edge computing scenarios. To assess this, a project was built using WebAssembly Micro Runtime (WAMR) integrated in Zephyr. A benchmarking program was developed to test common edge computing performance indicators and tasks such as Fast Fourier Transform (FFT) and Matrix Multiplication under varying code optimization levels and runtime configurations. Results show that WebAssembly introduces significant overhead compared to native code execution when running in interpreter modes. However, it provides benefits with respect to portability, security, and sandboxing. The findings suggest that WebAssembly running in interpreter modes is not suitable for compute-intensive edge workloads on the i.MX RT1180 microcontroller because of the execution time overhead introduced and the lack of support for the WebAssembly System Interface (WASI) standard.

Keywords WASM, Edge, Embedded, IoT, WAMR

Sammanfattning

Detta arbete undersöker brukbarheten och prestandan hos WebAssembly (Wasm) för edge computing på inbyggda system. Specifikt utvärderas huruvida WebAssembly, när det körs på ett lättvikts operativsystem för inbyggda system såsom Zephyr, kan leverera tillräcklig beräkningsprestanda för typiska edge computing-scenarier. För att undersöka detta byggdes ett projekt som använde WebAssembly Micro Runtime (WAMR) integrerat i Zephyr. Ett benchmark-program utvecklades för att testa vanliga prestandaindikatorer och arbetsuppgifter inom edge computing, såsom Fast Fourier Transform (FFT) och matrismultiplikation, med olika nivåer av kodoptimering och runtime konfigurationer. Resultaten visar att WebAssembly medför betydande overhead jämfört med körning av native-kod, när det körs i interpreter-lägen. Däremot erbjuder det fördelar i form av portabilitet, säkerhet och sandboxing. Testresultaten ger stöd för att WebAssembly, när det körs i interpreter-lägen, inte är lämpligt för beräkningsintensiva edge-arbetslaster på i.MX RT1180 mikrokontrollern, på grund av den extra körningstiden som introduceras och avsaknaden av WebAssembly System Interface (WASI)-stöd.

Nyckelord WASM, Edge, Embedded, IoT, WAMR

Preface

This work was done in collaboration with HMS Networks, which specified the requirements for the prototype.

Both authors contributed across most aspects of the project, but each assumed responsibility for specific areas of the work: Mattias Neidenström concentrated on software development and hardware integration, while Ernst Peterson focused more on research and report writing.

We would first like to thank Felix Nilsson from HMS for his practical guidance and support throughout the project.

We would also like to thank our supervisor, Wojciech Mostowski from Halmstad University, for his academic guidance and suggestions.

Finally, we want to thank our language coach, Olivia Semler, for all the text improvement suggestions used when writing the thesis. ChatGPT, Writefull, and Grammarly have also been used as tools for correcting grammatical errors in this report.

Contents

Abstract . . . iii
Sammanfattning . . . v
Preface . . . vii
Contents . . . ix
List of Figures . . . xi
List of Tables . . . xiii
Listings . . . xv
Acronyms . . . xvii
1 Introduction . . . 1
1.1 Background . . . 1
1.2 Problem Statement . . . 2
1.3 Purpose and goals . . . 2
1.4 Requirements . . . 3
1.5 Related works . . . 3
2 Technical Background . . . 7
2.1 What is WebAssembly . . . 7
2.2 WebAssembly for Edge Computing . . . 8
2.3 Zephyr OS . . . 8
2.4 WebAssembly Micro Runtime . . . 8
3 Method . . . 11
3.1 Development Process . . . 11
3.1.1 Building a WebAssembly project in Zephyr . . . 11
3.1.2 Performance testing . . . 11
3.1.3 Benchmarks . . . 12
3.2 Technical Design . . . 14
4 Implementation . . . 17
4.1 Constructing the Benchmark . . . 17
4.2 Using the Benchmark Program . . . 20
4.2.1 Bare-metal . . . 20
4.2.2 WAMR . . . 20
4.2.3 Compiler Optimization Levels . . . 21
4.3 I/O Test . . . 21
5 Results . . . 23
5.1 Benchmark Characteristics . . . 23

5.2 Matrix Multiplication 24
5.3 Insertion Sort 25
5.4 Linked list 26
5.5 Fibonacci Sequence 28
5.6 Fast Fourier Transform (FFT) 28
5.7 Arithmetic Operations 29
5.8 Summary of the CPU Intensive Benchmarks 30
5.9 I/O test 31
6 Discussion 35
6.1 Societal Aspects 37
7 Conclusion 39
7.1 Future Work 40
Bibliography 41
A Source Code 45
B Extra Material 59
B.1 Setting up dependencies 59
B.2 Low Level settings 60

List of Figures

3.1 Development board 15
3.2 Hierarchy Chart 16
3.3 Solution structure 16

List of Tables

3.1 Table of Contents . . . 15

5.1 A table of Benchmark Characteristics . . . 24
5.2 Results of the Matrix Multiplication Benchmarks . . . 24
5.3 Results of the Insertion Sort Benchmark . . . 25
5.4 Results of the Linked List Head Insertion Benchmark . . . 26
5.5 Results of the Linked List Tail Insertion Benchmark . . . 27
5.6 Results of the Linked List Random Access Benchmark . . . 27
5.7 The Results of the Fibonacci Sequence Benchmark . . . 28
5.8 The Results of the Fast Fourier Transform (FFT) Benchmark . . . 28
5.9 Results of the 32-bit Arithmetic Benchmark . . . 29
5.10 Results of the 64-bit Arithmetic Benchmark . . . 29
5.11 Cold Start Delay . . . 31

Listings

4.1 Excerpt from benchmark.c 18
4.2 Math functions needed for FFT 19
4.3 Wasm function export example 21
4.4 Function interface which is used by the I/O test program 21

A.1 Benchmark Suite 45
A.2 Arithmetic Benchmark Implementation 49
A.3 FFT Benchmark Implementation 50
A.4 Fibonacci Benchmark Implementation 53
A.5 Insertion Sort Benchmark Implementation 53
A.6 Linked List Benchmark Implementation 53
A.7 Matrix Multiplication Benchmark Implementation 55
A.8 Stopwatch Implementation 56
A.9 I/O Test Implementation 56

Acronyms

AoT Ahead-of-Time. 4, 9, 12, 36, 37, 60

API Application Programming Interface. 7, 8, 12, 14, 15, 21

BLAS Basic Linear Algebra Subprograms. 13

CPU Central Processing Unit. 3, 20, 60

FFT Fast Fourier Transform. iii, v, x, xiii, xv, 14, 18, 19, 24, 28, 29, 35

IoT Internet of Things. 1, 5, 7, 8, 12, 40

JiT Just-in-Time. 12, 37

JVM Java Virtual Machine. 1–4, 7, 36, 40

MCU Microcontroller Unit. 11, 14, 15, 35, 36

RAM Random Access Memory. 3, 14

RTOS Real-Time Operating System. 5, 8, 11, 14, 15, 36, 37, 39, 59

SDK Software Development Kit. 20, 59

WALI WebAssembly Linux Interface. 4, 5, 37

WAMR WebAssembly Micro Runtime. iii, v, ix, 4, 8, 9, 11, 12, 14, 15, 17, 19–21, 35–37, 39, 59, 60

WASI WebAssembly System Interface. iii, v, 4, 5, 7, 11, 15, 37, 39, 40, 60

Wasm WebAssembly. iii, v, ix, xv, 1–5, 7–9, 11, 12, 14, 15, 17, 19–21, 23, 24, 30, 35–37, 39, 40, 59, 60

WAZI WebAssembly Zephyr Interface. 4, 37

Chapter 1

Introduction

Containerization and virtualization have enabled smaller devices to function as edge computing solutions within larger networks. However, these technologies have limitations in reducing storage and computational requirements while maintaining flexibility for programmers to use various programming languages. On small embedded devices, traditional solutions often require virtualization layers, such as Java Virtual Machine (JVM) [1], which introduce additional overhead. This project explores whether WebAssembly (Wasm) can serve as a viable alternative for edge computing on embedded systems, providing a lightweight and efficient execution environment.

1.1 Background

The rapid spread of Internet of Things (IoT), a common term for devices capable of collecting and exchanging data over the internet, has introduced a growing demand for data processing and storage. Traditionally, this problem has been solved by cloud computing, where data from edge devices is sent to a centralized computer system that manages computational tasks and long-term storage. As the number of edge devices grows, the amount of transferred information within the network increases, negatively impacting network bandwidth and latency [2].

Edge computing is another computing paradigm capable of solving some of the shortcomings of cloud computing by transferring the data processing closer to the source where the data is generated. While setting up a conventional computer system as an edge computing platform is fairly trivial, the same cannot be said for small embedded microcontrollers. These systems are comparatively weak and incompatible with the software used by more powerful systems, which presents several challenges, three of which are low-level programming, portability, and performance [3]. Firstly, embedded platforms are often programmed using low-level languages, and while they are performant, they have downsides like slow development, error-prone code, and an increasing difficulty in finding skilled developers. Broader language support allows for faster development and easier recruitment of developers. Secondly, microcontrollers come in a wide variety of architectures,

feature sets, and methods for hardware interaction, which can make porting an application to a new system complicated and time-consuming. Thirdly, these kinds of systems have limited hardware resources. Conversely, other aspects of microcontrollers, such as high power efficiency and low cost, can make them attractive for use as edge computing devices.

Two common approaches to edge computing are *containers* and *virtual machines*, which set up execution environments on a host machine to run code [4]. Examples of these technologies include JVM and Docker [5]. Although possible to implement on more powerful microcontrollers, these methods introduce significant performance overhead, high memory requirements, and slow startup/shutdown times, making these technologies unfit for devices such as the one targeted in this project. An alternative solution is *WebAssembly*. This is a binary instruction format that takes advantage of capabilities found in many computer platforms, enabling code to be compiled from a high-level language and run efficiently on a wide variety of machines in a safe environment [6]. Using WebAssembly, the capabilities of low-power embedded systems could be leveraged to run edge computing tasks.

1.2 Problem Statement

Edge computing on resource-constrained systems presents several challenges. Traditional approaches often introduce enough overhead to make smaller embedded systems unviable as a platform due to high processing power and memory requirements. On a factory floor, this results in inefficient use of local computing capabilities, as the data-collecting embedded systems at the edge need another computer system to process that data.

From this problem, the following question arises: is WebAssembly viable as an edge computing solution for embedded systems, and how does it compare to alternative solutions such as Docker and JVM?

1.3 Purpose and goals

This project aims to evaluate WebAssembly's viability as an edge computing technology on embedded systems in an industrial environment. A prototype edge computing device is developed to investigate the possibility of using WebAssembly as an edge computing technology. Furthermore, a technical investigation is conducted to compare the WebAssembly solution to technologies like Docker and JVM in terms of performance, cold start delay, and memory footprint.

To provide practical relevance, the system is evaluated in the context of a potential industrial application: real-time sensor signal preprocessing on a factory floor. In this use case, embedded devices are expected to collect raw data from sensors (e.g., vibration, temperature), apply basic transformations such as filtering or FFT analysis, and transmit summary statistics to a central controller. These

tasks require moderate compute performance with low latency, and they benefit from code portability and sandboxing for security and update management. This scenario guides the selection of benchmarks and helps assess whether WebAssembly performance is sufficient for such edge tasks.

1.4 Requirements

The company wants the solution to be built around the i.MX RT1180 microcontroller. They have also specified features and metrics to be evaluated, which are divided into two categories:

Functional Requirements

- **Core functionality:** The system must *support execution of WebAssembly applications*.
- **Portability:** *Existing edge applications* should be adaptable to the system.
- **Networking (low priority):** A *network interface* should be implemented to allow deployment of applications using *TCP* or *UDP*.
- **Sandboxing (low priority):** The system should be able to *instantiate multiple isolated program execution environments* and have the ability to *restrict their RAM and CPU usage* individually.
- **Flash protection (low priority):** The system should have a *protective mechanism* that *hinders excessive writing* to its flash memory.

Non-Functional Requirements

- **Performance:** The performance must be adequate enough to *run real-world edge applications in real-time*.
- **Cold start delay:** *Initialization time* of newly deployed applications *must be shorter* compared to Docker and JVM. According to the company, a Linux system using a Cortex-A CPU can start a Docker container in 10 to 30 seconds.

Note: All functional requirements marked as "low priority" are nice-to-have features only and are not required for the working prototype. These requirements will only be explored if the system already satisfies the main functional requirements and there is project time to spare.

1.5 Related works

A similar project called WARDuino was conducted by T. Lauwaerts, R. G. Singh, and C. Scholliers [3], where a virtual machine was developed to interpret WebAssembly bytecode instructions line by line on an Arduino microcontroller. This project targets a different hardware platform than ours, as the i.MX-RT1180 is not supported in their implementation.

E. Wen and G. Weber developed a proprietary operating system (OS) called WASMachine [7], which is tailor-made to run WebAssembly applications. This so-

lution directly integrates a Wasm runtime into the OS, which, according to their findings, improves efficiency. Since this solution supports ARMv8, it could be implemented as part of our solution on the Cortex M33 chip, however, it lacks support for the Cortex M7 chip. Another downside is the freedom of choice regarding the OS. Furthermore, it does not align with our project objectives as it requires a manual porting effort of WASMachine for each microcontroller.

A similar solution called KESO [1] was implemented by M. Stilkerich, et al., where a specialized JVM was developed from scratch to run efficiently on embedded devices. Testing found execution time overhead in calculating datasets compared to C ranging from 4% when using trigonometric functions from the C library to 23% using the default variant from the Java Math library. The key difference between this solution and one using WebAssembly is that the JVM solution requires the programmer to write code in Java.

The conference article by L. Deshpande and K. Liu [8] shows how Docker can be used on edge devices. This solution requires a central virtual machine running Linux, specifically Ubuntu. The virtual machine then communicates with several deployments of the Docker engine on edge devices using HTTP. These edge devices need to be more powerful than the embedded device targeted in this project, e.g. Raspberry Pi.

A previous study made in the conference article [9] compares native performance and WAMR in AoT and interpreter mode on an embedded device using CoreMark, a single-score benchmark utility (higher score is better). Performance testing resulted in scores of 1157, 611, and 32 for native mode, AoT mode, and interpreter mode, respectively. Devices with a similar score include the Broadcom BCM2835 processor with a score of 1303.78 [10], which is featured on several Raspberry Pi models such as the Raspberry Pi 1 family and Raspberry Pi Zero. In comparison, the AMD Ryzen 9 5900X desktop processor scores 622,272 points.

An interface called WebAssembly Linux Interface (WALI), and a derivative of it called WebAssembly Zephyr Interface (WAZI), was developed by A. Ramesh et al. with the purpose of simplifying development using the WebAssembly System Interface (WASI). The interface adds a level of abstraction between the WebAssembly module and WASI by having a standard list of system calls, which is shared by the entire OS. This can simplify the workload for WASI developers since only platform-specific calls need to be added [11].

An open source project called Boxer made by Dan Phillips is currently in development, and seeks to write containers as Wasm binaries called "Boxes" or "WASM-Boxes". These take Docker files as input and convert them into Wasm binaries. These binaries will not need Docker engine and can instead be run on machines with a Wasm runtime [12].

In summary, many different approaches have been explored, which aim to improve code portability and/or execution speed on embedded systems and edge devices. The most comparable solution to our project is the JVM solution, which targets similar hardware and runs bytecode in a virtual environment. Similar metrics (execution time and binary size) were also chosen to evaluate the performance of

the system. The key differences between the solutions are the multi-language and networking support. The Docker solution targets more powerful hardware and focuses more on sandboxing and container orchestration than execution performance. Thus, the Docker solution is evaluated primarily on isolation capabilities rather than computational and memory efficiency. Boxer tries to create a similar container solution to Docker, but using WebAssembly instead. It offers a set of tools that convert Docker images to Wasm binaries, which can be executed by a Wasm runtime. It is also evaluated by its containerization compatibility and not performance.

The WASMachine OS, while supporting hardware akin to ours (i.MX 8M), imposes constraints by tightly coupling the runtime with a custom OS, thereby limiting flexibility. Our approach, in contrast, integrates a WebAssembly runtime with the Zephyr RTOS to preserve freedom of operating system choice and utilize its built-in security features. Another key difference is the evaluation criteria, where the WASMachine is evaluated by its ability to run WebAssembly itself instead of evaluating the performance of running Wasm files. Likewise, the Arduino-based WebAssembly VM project shares hardware similarities. The main difference is that the VM focuses on testing and development while we try to evaluate the performance of WebAssembly on industrial IoT. The evaluation criteria also differ between the projects, since the VM checks for conformity with the WebAssembly standard and compatibility with Wasm functions instead of performance of running Wasm files. Lastly, efforts like WALI and WAZI address developer usability by abstracting system calls for WASI environments. These interfaces are promising for future work, but are not yet mature or fully accessible during the course of this project.

Ultimately, our work differentiates itself through its focus on performance benchmarking of WebAssembly on industrial-grade microcontrollers using a lightweight RTOS, offering insights into the feasibility of deploying high-level code in resource-constrained edge environments.

Chapter 2

Technical Background

The technical background area of the project is Embedded Systems for Industrial IoT with WebAssembly. Knowledge from the courses: Computer Systems Engineering, Data Communication, and Programming forms the foundation for the project. The research focuses specifically on WebAssembly.

2.1 What is WebAssembly

WebAssembly is a standard that defines a stack-based virtual machine with a compact assembly-like, binary instruction format for running programs [6]. This format is a portable compilation target for other programming languages that enables application deployment on the web for clients and servers. By taking advantage of common hardware capabilities, this standard aims to offer fast load times, small binary sizes, and efficient code execution on a wide variety of platforms. Like Java and JVM, WebAssembly code can be executed on any platform with a WebAssembly runtime. The standard also describes a sandboxed execution environment which ensures memory integrity by constraining programs to their own resizeable, linear memory space. While primarily designed for highly optimized and secure web pages, WebAssembly's focus on efficient code execution on many different architectures makes it a viable option for running portable code on weaker computer systems.

To extend the capabilities of Wasm, the runtime needs to support the WebAssembly System Interface. This is a standardised API that implements clocks, command line interface, HTTP, file systems, and more.

However, WebAssembly also comes with certain drawbacks. Performance overhead in interpreter mode can be significant. WARDuino is on average WebAssembly 425.93 times slower [3] compared to native execution, while an ARM Cortex A7 MCU is 36.16 times slower in Coremark [9]. Additionally, not all combinations of hardware and runtimes have WASI implementations yet, which limits the usability of WebAssembly.

2.2 WebAssembly for Edge Computing

Offloading computation from central servers is a core issue for many industries worldwide. A path for abstraction has been developed with significant milestones in containerization using Docker and Kubernetes, which has simplified development of applications at the edge [13]. However, these solutions inherently suffer from high memory usage and slow cold boot times, which can be problematic for certain use cases. WebAssembly could enable edge computing on smaller embedded devices than containers allow. That said, WebAssembly is not a universal replacement for all embedded edge solutions. It introduces additional computational overhead, particularly in interpreter mode, which may not be acceptable for ultra-low-power applications.

2.3 Zephyr OS

Zephyr is a Real-Time Operating System (RTOS) developed with security, connectivity, and future-proofing in mind. It uses a small-footprint kernel designed for resource-constrained systems. The main benefit of using Zephyr in this context is that it provides standardized APIs for hardware interaction for many different microcontrollers. The RTOS is trusted and used by many commercial products such as "Gardena Smart Gateway" and "Framework Laptop 13 DIY Edition (AMD Ryzen™ 7040 Series)" but is also used in smartwatches and IoT applications [14]. The Zephyr project employs mainly the following security measures:

- Cryptographic algorithms and protocols through PSA Crypto
- Quality Assurance of the development of Zephyr through code reviews
- Execution Protection including thread separation, stack and memory protection

Although security is not a primary focus in the context of this project, Zephyr and WebAssembly together result in a good foundation for building a secure system. The combination already defines a safe execution environment, but it can be improved even more by utilizing the security features provided by Zephyr.

To combat the wear distribution problem of flash memory, Zephyr has a built-in file and partition service called *Non-Volatile Storage (NVS)* [15]. This system stores data as key-value pairs in a circular buffer where new data is appended rather than overwritten in the same location. Older data is marked as obsolete and erased when no longer in use. This system provides a more uniform distribution of wear on flash memory, which can prolong its lifetime.

2.4 WebAssembly Micro Runtime

WebAssembly Micro Runtime (WAMR) is a lightweight WebAssembly runtime with a focus on a small memory footprint and high performance for embedded devices. WAMR is developed to have highly configurable features tailored for IoT and

edge devices, including support for *Trusted Execution Environment (TEE)*, *smart contracts*, and *cloud native applications* [16]. The runtime has a compact *binary size* which varies depending on which execution mode is used. When the runtime is compiled with *Ahead-of-Time (AoT)* support, the size can be as small as 29.4 KB while the *interpreter* modes range between 56.3 KB and 58.9 KB. This allows the runtime to fit on resource-constrained systems using lightweight processors such as the Cortex-A15 and Cortex-M7 series. WAMR's memory model is divided into four categories [17]:

- Runtime memory: Used by the runtime globally.
- Wasm module memory, which reserves memory for module structure, file buffers, and pre-compiled code.
- Wasm module instance memory, which reserves memory for needed functions, exports, and data.
- Execution environment memory, which reserves memory needed to execute the Wasm functions.

The AoT running mode promises to run code with near-native speed. WAMR includes ports for several architectures, including x86 and ARM, as well as operating systems, including Linux and Zephyr.

Chapter 3

Method

This chapter describes the approach taken to develop a WebAssembly-based edge computing system using an Real-Time Operating System (RTOS) on a microcontroller.

3.1 Development Process

The solution is built around the ARM-based Microcontroller Unit (MCU) i.MX-RT1180 from NXP [18] and is programmed in C [19] using VSCode with the MCUXpresso extension. Installation of an RTOS allows the integration of a WebAssembly runtime, which enables the execution of WebAssembly applications on the system. The project is divided into four phases.

3.1.1 Building a WebAssembly project in Zephyr

The project begins with hardware and software setup. The goal is to install Zephyr on the platform and integrate WAMR into it using the WAMR-Zephyr port, which allows execution of compiled WebAssembly programs.

3.1.2 Performance testing

During the second phase, the system's performance is tested by executing computationally intensive applications on the platform to measure the overhead of Zephyr and WAMR. This was planned to be done by using the CoreMark MCU benchmarking suite for its widespread use in embedded systems performance testing. Unfortunately, Zephyr's lack of support for the WebAssembly System Interface in WAMR means that the WAMR built-in Standard C Library libc had to be used instead. This bare-bones library lacks support for many features, including file systems, clocks, random numbers, and math functions such as square roots and trigonometry. As a consequence, using CoreMark or any other benchmarking suite is impractical for this project, which also makes comparing performance statistics to other systems difficult.

A custom benchmarking suite has been created around the previously mentioned limitations. To make the performance testing more representative of real-world workloads, a number of commonly used test methods have been devised, which are described in more detail in a later chapter 3.1.3. These tests are implemented in a single C file that can be easily ported to WebAssembly. The benchmark will be run in all Wasm execution modes available to WAMR, and different compiler optimization settings will also be tested. Only WAMR's classic interpreter and fast interpreter could be tested since WAMR lacks support for AoT and JiT modes for Zephyr.

Cold start delay is measured with the help of Zephyr's clocks API. This API provides a function that returns the system uptime measured in clock cycles. By immediately retrieving the current uptime and printing it to the console when a Wasm program starts, the delay between the early stages of the Zephyr boot process and Wasm program execution can be measured. Knowing the M33 core operates at a constant frequency of 240 MHz, the number of clock cycles can be converted to seconds.

Energy consumption is compared between the bare-metal and Wasm programs using a USB tester; the FNIRSI FNB48S. This tester monitors power using a sampling rate between 2 and 100 samples per second with a resolution of 0.00001 W and an accuracy of $\pm 0.5‰$. It measures the average total power consumed by the development board while the benchmark is running. Once the benchmark has finished, the average power draw is multiplied by the program execution time to calculate the amount of energy consumed by the board to finish executing the program.

Since the target use case for this system is within industrial IoT, it makes sense to include a test that examines the I/O capabilities of the system. This will be done by using Zephyr's API to construct a simple program that measures sequential read and write performance of a Kingston SDC4 micro SDHC card. The same program will be executed in Zephyr, and in WebAssembly to determine if there is a discernible performance difference between the two.

3.1.3 Benchmarks

To evaluate whether WebAssembly is suitable for typical edge workloads, the benchmarks were designed to reflect tasks that could realistically be performed in a real-time industrial sensor preprocessing scenario. This includes operations such as signal transformation, lightweight data structuring, and computation on small datasets, all of which are common in edge nodes processing raw sensor data before transmission. The selected benchmarks therefore aim to simulate compute patterns found in such applications.

Matrix Multiplication

Linear algebra, including operations targeting vectors and matrices, is commonly used in embedded computing solutions. This motivates focused testing of these

operations since the performance of the operations themselves cascades to affect the entire performance of programs. A selection of dense linear algebra operations has been generalized as the Basic Linear Algebra Subprograms (BLAS) which motivates the choice of tests [20]. BLAS is divided into three levels:

- Vector-Vector operations, such as Vector addition.
- Vector-Matrix operations, such as Matrix-Vector multiplication.
- Matrix-Matrix operations, such as Matrix multiplication.

The choice of testing matrix multiplication specifically is because it is one of the heavier operations in BLAS and will indicate the performance of the other BLAS programs.

Insertion Sort

Reading and writing to statically allocated memory is part of all software programs. A common example of manipulating such data is to sort an array of values. Therefore, this benchmark includes insertion sort, one of the most popular and simple sorting algorithms, which indicates how fast the system handles random reads and writes to static memory. The choice of this particular algorithm is founded in its high performance on small datasets and iterative implementation, which are important factors for embedded systems [21].

Linked List

The linked list is a fundamental data structure in programming and is used in the program to test dynamic memory allocation performance. A singly linked list is created, in which data values are inserted, retrieved, and removed. These operations test heap memory allocation, deallocation, and data retrieval.

Fibonacci Sequence

The Fibonacci sequence is a classical mathematical model that is particularly useful for testing recursion on computer systems. Each number in the sequence is defined as the sum of the two previous ones where the starting numbers are defined as $fib(0) = 0$ and $fib(1) = 1$, resulting in the sequence $\{0, 1, 1, 2, 3, 5, ...\}$. The sequence is programmed using recursion, where each function calls upon itself to create the following number in the sequence. This means that the sequence is a binary tree where each child tree differs in height by one. Testing the Fibonacci sequence gives insight into the system's capability of handling recursive function calls. While this approach is generally to be avoided over an iterative approach for embedded systems, according to the MISRA C 2012 guidelines [22], the recursive algorithm was explicitly chosen to test function call performance.

FFT

Fast Fourier Transform is an algorithm for computing a discrete Fourier transform (DFT). The most common algorithm for computing FFT is the one proposed by J.W. Cooley and T.W. Tukey in 1965, which has a computational complexity of $O(N \log N)$ compared to $O(N^2)$ Gauss algorithm for DFT [23].

J.W. Cooley and T.W. Tukey's FFT algorithm works by recursively breaking down the DFT into smaller chunks and leveraging the properties of complex exponentials. An iterative version of the algorithm also exists, which is more suitable for resource-constrained environments [24]. Since the target is an embedded system, the iterative approach is used to conserve memory and function call overhead.

FFT algorithms as benchmarks closely emulate real-world signal processing, which is a typical practical workload for embedded systems [25].

Arithmetic

The arithmetic test is a simple program that tests the four basic mathematical operations: addition, subtraction, multiplication, and division of 32 and 64-bit integers and floating-point numbers.

3.2 Technical Design

The embedded system used in this project is the NXP MIMXRT1180-EVK development board [26]. It features an NXP i.MX-RT1180 [18], a dual-core ARM-based MCU suitable for industrial applications thanks to its energy efficiency and wide selection of connectivity interfaces, security features, and networking capabilities. Its target applications include industrial control, AC/servo drives, and in-vehicle networking. In terms of memory, this particular platform comes with 1.5 MB of RAM, and while it is considered to be a lot for low-cost applications, it still severely limits OS and WebAssembly runtime choices. It is also too underpowered to run other edge computing solutions like Docker. The system is programmed using a PC connected with a USB cable to the debug USB port on the development board, as seen in the figure 3.1.

WAMR was chosen because of its small memory footprint and Zephyr OS support [16]. Zephyr was chosen for its WAMR compatibility, low storage requirements, and because it is supported by NXP for the development board. The inclusion of an OS makes the integration of the runtime easier since its API calls are handled by the OS instead of having to be implemented manually on this specific platform. A hierarchy of the system can be seen in figure 3.2, which shows a simplified view of how the different system components build on each other.

Figure 3.3 shows an overview of the project structure. A PC is used together with the MCUXpresso extension in VSCode to upload and debug the RTOS and WebAssembly runtime code to the MCU. Applications are compiled to WebAssembly

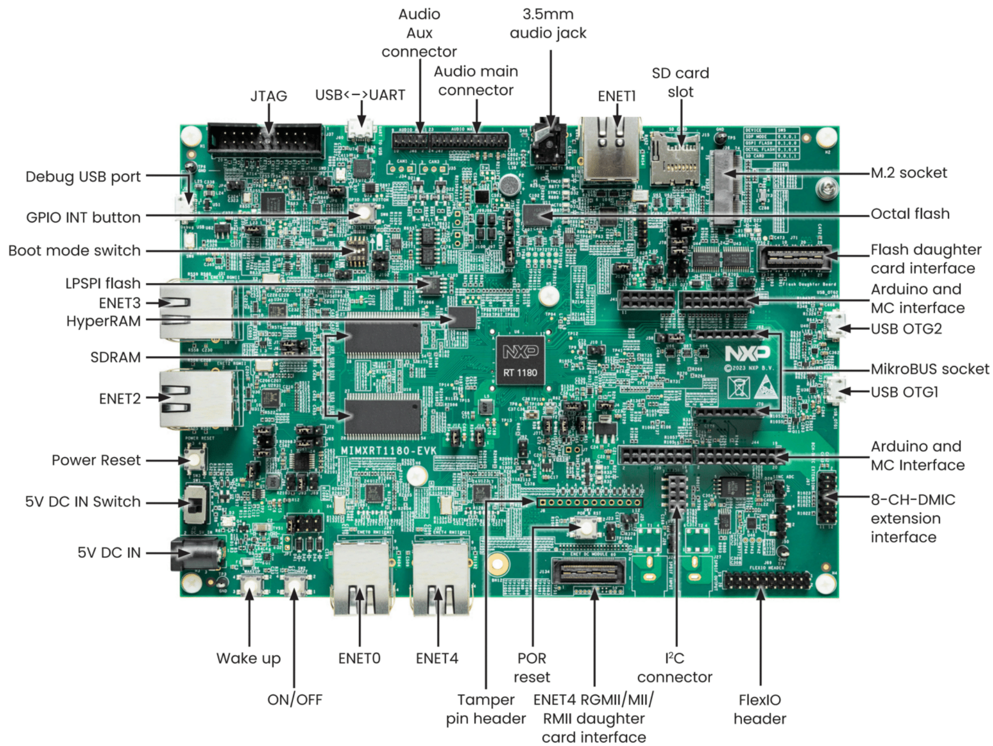

Figure 3.1: Development board front callouts

bytecode on the PC and sent to the development board over USB to be executed by the runtime in isolation from the hardware. The MCU can also communicate with external data-generating devices, which should be controllable by the application. The WebAssembly runtime supports the WASI interface, which should make Zephyr's communication interfaces accessible from within Wasm modules.

The runtime is integrated as a module in Zephyr, ensuring that all the RTOS function calls, including IO interfaces, can be accessed by the WebAssembly runtime through the API. This also keeps all of the RTOS functionality intact, allowing e.g. scheduling, memory management, and thread execution when code is compiled to WebAssembly and not only native C code. Since WAMR only interacts with the Zephyr API, the security in the RTOS remains intact. A list of specific implementations along with our contributions can be found in table 3.1.

Table 3.1: Table of Contributions

Component	Source	Contribution
Zephyr	Open Source (Zephyr Project)	Configuring NXPs port for the board
WAMR	Open Source (ByteCode Alliance)	Integration with NXPs port using the WAMR-Zephyr port
Benchmarking Suite	Custom	Implemented in C and exported to Wasm
Communication interface	Custom	Implemented in C and used in WAMR-Zephyr

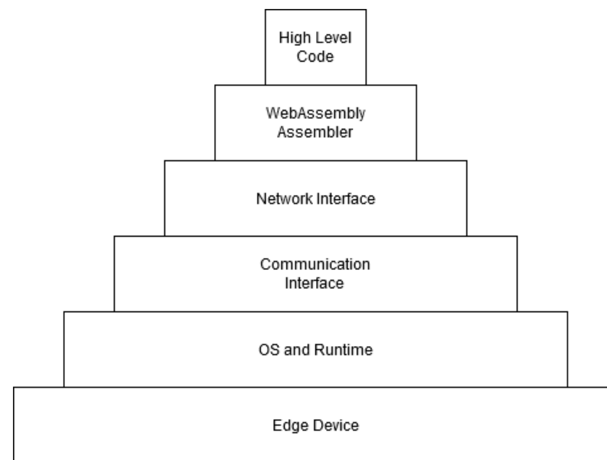

Figure 3.2: A chart of the system hierarchy

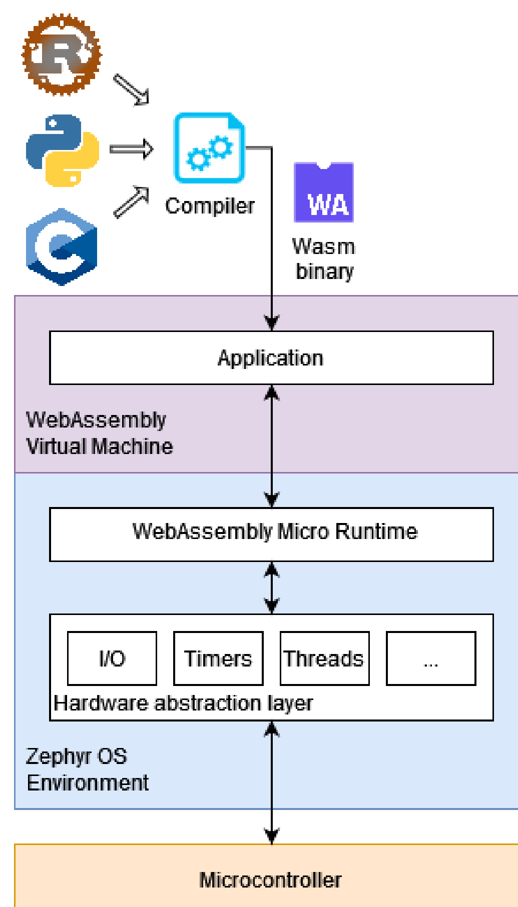

Figure 3.3: A simple overview of the project structure

Chapter 4

Implementation

4.1 Constructing the Benchmark

A single folder C project is created with the following structure:

```
/Benchmark
 ├── benchmark.c ... Main benchmark program
 ├── platform_common.h ... Interface for init, printf, and clock functions
 ├── Stopwatch.h ... Uses the SysTick to measure elapsed time
 ├── xyz.h ... Data structures and algorithms needed for testing
 ├── script.py ... Compiles benchmark.c to a Wasm header file
 └── test_wasm.h ... Plaintext WebAssembly generated by script.py
```

The main benchmark program imports the platform_common.h file, which contains declarations of functions for printing to the console and reading the system clock. Since these functions need separate implementations for the bare-metal and WAMR projects, this header file searches for a specific package only present in the bare-metal project. Once the platform is identified, the header imports the appropriate implementations of these functions, which are then passed on to the main program.

The program first runs the init method, which contains platform-specific setup code. The bare-metal program uses this method to configure the SysTick while the WAMR program uses it to print boot time information to the terminal. The stopwatch gets initialized by calling the stopwatch_init function. This function measures the average delay between consecutive calls of the stopwatch_start and stopwatch_stop functions, which is taken into account when measuring elapsed time.

After setting up the system, the benchmarks start. The test settings can be easily configured for each individual test by changing the values of the macros defined at the top of the program. Code listing 4.1 shows an excerpt of the benchmark program, which shows how the program is structured. Using the previously mentioned algorithms, the following tests have been implemented into the bench-

mark:

- **Recursive Fibonacci:** Calculates the n'th Fibonacci number recursively.
- **Linked List - Insert head:** Inserts a set number of nodes at the head of a linked list, deletes the list, and repeats the process a set number of times.
- **Linked List - Insert tail:** Same as the test above, but inserts the nodes at the tail of the linked list instead of the head.
- **Linked List - Random access:** Creates a linked list with a set number of nodes and retrieves a set number of values from random indices.
- **Matrix multiplication - Integer:** Repeatedly multiplies two 32-bit integer, square matrices together.
- **Matrix multiplication - Float:** Same as the test above, but using 32-bit floating point matrices instead.
- **Insertion sort:** Inserts random values into an array of a set length, sorts it using insertion sort, and repeats the process a set number of times.
- **Fast Fourier Transform:** Applies the FFT algorithm on an array of a set length. This is repeated a set number of times.
- **Arithmetic - int32:** Repeatedly adds, subtracts, multiplies, and divides 32 bit integers.
- **Arithmetic - int64:** Same as the test above, but using 64-bit integer numbers.
- **Arithmetic - float32:** Same as the test above, but using 32-bit floating-point numbers.
- **Arithmetic - float64:** Same as the test above, but using 64-bit floating-point numbers.

Listing 4.1: Excerpt from benchmark.c

```
/* TEST SETTINGS */

// Fibonacci
#define FIB_N 40

// Linked list
#define LLIST_INSERT_HEAD_COUNT 7000
#define LLIST_INSERT_HEAD_ITER  700
...
/* TEST CONFIGURATION */

#define USE_FIBONACCI 1

#define USE_LLIST_INSERT_HEAD 1
...

#include "platform_common.h"
#include "Stopwatch.h"

#include "Fibonacci.h"
#include "LinkedList.h"
...
int main(void)
{
```

```
    /* Setup */
    init();
    stopwatch_init();

    #if USE_FIBONACCI
        // Do Fibonacci test
    #endif
    #if USE_LLIST_INSERT_HEAD
        // Do Linked List insert head test
    #endif
    ...
}
```

Worth noting is that all algorithms and benchmarks are implemented without any C library functions to ensure compatibility with WAMR's very limited Standard C Library libc WebAssembly library, which only provides basic functions such as malloc, free, and printf. This posed a problem for the FFT algorithm, which relies on multiple math functions for real and complex numbers, which can be seen in listing 4.2, all of which had to be reimplemented in C without the help of library functions. While they could have been exported from the Zephyr environment, that would not have resulted in an accurate representation of Wasm's computing capabilities since, in that case, the calculations would not have been done in Wasm code.

Listing 4.2: Math functions needed for FFT

```
float fmodf(float x, float y) {
 // Calculates the remainder of the division between
 // the two floating point numbers x and y
}

float sinf(float x) {
 // Calculates sine of x using the Taylor series approximation
}

float cosf(float x) {
 // Returns sinf(x + PI/2)
}

float expf(float x) {
 // Calculates e^x using the Taylor series approximation
}

complex float __mulsc3(float a, float b, float c, float d) {
 // Calculates the complex number (a+bi)(c+di)
}

complex float cexpf(complex float z) {
 // Calculates e^z where z is a complex number
}
```

4.2 Using the Benchmark Program

The following section describes the process of integrating and using the benchmark program in a bare-metal and WAMR project.

4.2.1 Bare-metal

This project is based on the rled_blinky_cm33 demo program from the MCUXpresso Software Development Kit (SDK) [27]. The following changes are made to the project:

- **CMakeLists.txt** The included source file is replaced with the benchmark.c file. To allow the inclusion of additional header files, the project folder is added to the include path. Compiler optimization flags are also changed here.
- **platform_baremetal.h** A header file is added to the project folder that implements the platform_common.h interface. It initializes the SysTick interrupt and SysTick handler to count the number of CPU cycles passed.
- **Linker Script** By default, the rled_blinky_cm33 project uses a 1 KB heap and 1 KB stack. This is enough to blink an LED, but the benchmark program needs more heap memory for dynamic memory allocation. By editing the linker script located at
  `mcux_sdk_v25_03_00/mcuxsdk/devices/RT/RT1180/MIMXRT1189/gcc/MIMXRT1189xxxxx_cm33_ram.ld`
  the heap size is increased to 128 KB by changing the m_heap_size value.

4.2.2 WAMR

The WAMR repository [28] contains a sample Zephyr project called the mini-product, which is used as a base for the benchmark program to which the following modifications had to be made:

- **CMakeLists.txt** The main program of this project imports the WebAssembly code as a C header file. The benchmark folder is added to the include path to use its Wasm header file instead of the included one. Additionally, a modification was made that causes script.py to be executed every time the project is built. This custom script compiles benchmark.c to a Wasm binary, and converts that binary to a C header file, which is then used by the main program.
- **platform_wasm.h** Similar to platform_baremetal.h, but implements the interface using native functions. It also declares Wasm function exports for these custom functions and the C rand function. Listing 4.3 shows how one of the functions is exported:

Listing 4.3: Wasm function export example

```
// This method executes outside the wasm module
uint64_t get_system_ticks(wasm_exec_env_t exec_env) {
    return k_cycle_get_64(); // Zephyr function to read the system clock
}

// Make a method with name "get_system_ticks" available to the wasm module
static NativeSymbol native_symbols[] =
{
    // "()": The method takes void as argument
    //  "I": The method returns a 64-bit integer
    EXPORT_WASM_API_WITH_SIG(get_system_ticks, "()I"),
};
```

- Additional low-level Zephyr and WAMR configuration settings were changed which is outlined in appendix B.2.
- **main.c** platform_wasm.h is imported and the functions registered to the Wasm module. The module heap size is also adjusted to match that of the bare-metal project, but the stack size had to be increased to 2 KB for the program to run. This decision was based on an observed stack overflow during the Fibonacci test.

4.2.3 Compiler Optimization Levels

The choice of compiler optimization flags can significantly impact both execution time and energy consumption, as demonstrated by D. Branco and P. R. Henriques [29]. Their report concluded that energy usage is closely correlated to program execution time, which can be reduced by selecting appropriate compiler optimization flags. For this project, we chose to use O2 as a starting point, as it is the recommended optimization level. Additionally, the O3 and Os flags have been tested. O3 uses even more aggressive optimizations compared to O2, and Os reduces code size by removing O2 flags that increase it. The optimization level is set for the bare-metal project and Wasm compilers. Zephyr has its own setting, which is set to O2 for all tests.

4.3 I/O Test

The I/O test performs sequential data reads and writes to an SD card. To implement this, an interface was created for SD card data transfer functions, shown in listing 4.4, which directly uses Zephyr's SD card API. These functions are also exported by WAMR so Wasm modules can access them. The compile script and stopwatch header file were reused for this program.

Listing 4.4: Function interface which is used by the I/O test program

```
void sd_read(uint8_t *data_buf, uint32_t start_sector, uint32_t num_sectors);
void sd_write(uint8_t *data_buf, uint32_t start_sector, uint32_t num_sectors);
uint64_t get_system_ticks();
```

This program works by repeatedly performing linear read and write operations on a set number of sectors (each SD card sector is 512 bytes). After a short warm-up period that aims to improve the consistency of the measurements, the program measures the accumulated time and calculates an average latency value, which represents the time it takes to perform the operation. The program also converts this number into a data throughput rate.

Chapter 5

Results

Firstly, appropriate settings had to be found for all tests through trial and error. The goal was to ensure that all tests would run for at least ten seconds in the fastest configuration, while avoiding any kind of memory overflow. This duration was chosen arbitrarily to let the power measurement stabilize, while being short enough not to cause the Wasm program to run for multiple hours.

All tests were enabled, the benchmark was started, and the USB tester's average power measurement was reset. Once the program had finished execution, the duration of each individual test was noted down, together with the memory usage information from the MCUXpresso extension and average power usage. Worth noting is that the memory footprint of the Wasm program also includes the runtime and OS. This process was repeated for the bare-metal, fast interpreter, and classic interpreter configurations using the O2, O3, and Os compiler optimizations.

The cold start time was measured for the fast and classic interpreters by repeatedly resetting the MCU ten times and noting down the delay. The result was then presented as a confidence interval with a confidence level of 99%.

5.1 Benchmark Characteristics

To evaluate WebAssembly's suitability for edge computing workloads, it is essential to understand the characteristics of the benchmarks used. Edge applications vary significantly in their resource profiles — some are CPU-intensive (e.g., signal processing), others are memory- or I/O-bound (e.g., sensor polling or protocol handling). By classifying each benchmark, we provide a clearer context for interpreting the performance results and identifying coverage gaps in our testing.

Table 5.1 categorizes each benchmark in terms of its type, memory usage, and CPU intensity. This classification is based on the underlying implementation patterns and resource usage during execution.

Table 5.1: A table of Benchmark Characteristics

Benchmark	Type	Memory Usage	CPU Intensity	I/O Dependency
Matrix Multiplication	Numerical Computation	Medium (heap)	High	None
Insertion Sort	Memory-bound Algorithm	Medium (heap)	Moderate	None
Linked list	Dynamic Memory Access	Medium (heap)	Moderate	None
Fibonacci Sequence	Recursive Computation	Low (stack)	Very High	None
FFT	Transform Algorithm	High (heap)	Very high	None
Arithmetic Operations	Primitive Operation Test	Very Low	Low to Moderate	None
I/O Test	Read/Write	Very High	Low	SD Card Access

As shown in table 5.1, most benchmarks are synthetic and predominantly CPU-bound, with no real I/O operations, except for the I/O test. Memory usage varies between the tests, FFT uses a high amount of memory through the use of arrays while the arithmetic operations test barely uses any memory but instead mainly relies on the CPU. Since the tests are skewed towards CPU-heavy operations, this means that the results primarily reflect the raw computational performance of Wasm and do not account for asynchronous or event-driven edge workloads, which are common in real deployments.

This limitation should be considered when interpreting the results. Many real-world edge applications rely heavily on I/O, such as periodic data acquisition from sensors, GPIO-triggered control logic, or low-latency communication with other nodes. WebAssembly's performance under these conditions remains largely unexplored in this study with the exception of the flash disk access test. A more in-depth study including a network access test presents, would help round out the evaluation.

5.2 Matrix Multiplication

The size of the matrices was set to 62x62, and the test was set to perform 2100 multiplications for each matrix type.

Table 5.2: Results of the Matrix Multiplication Benchmarks

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time int32 O2 (s)	16.887	375.70	819.45
Execution Time int32 O3 (s)	16.958	376.34	819.64
Execution Time int32 Os (s)	16.990	422.26	1055.9
Execution Time float32 O2 (s)	16.956	384.75	819.65
Execution Time float32 O3 (s)	17.023	384.71	820.31
Execution Time float32 Os (s)	16.991	433.00	1055.1

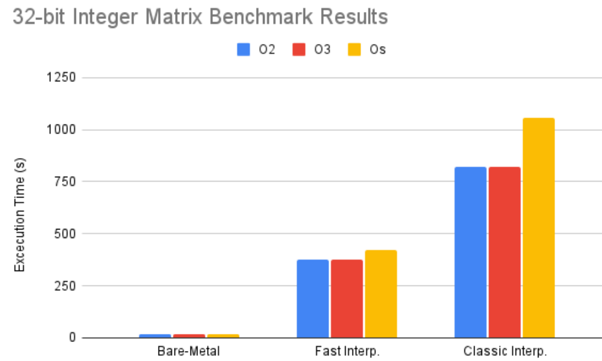

Figure 5.1: Graph of Integer Matrix Multiplication Benchmark

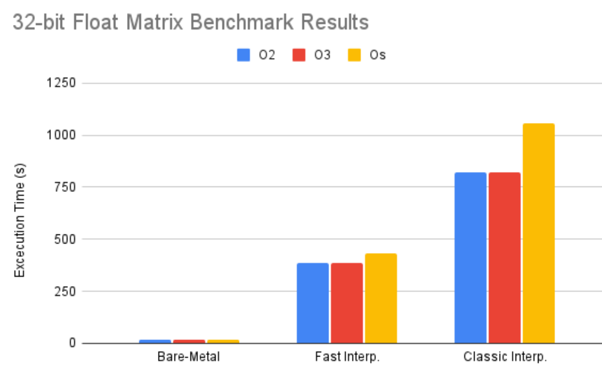

Figure 5.2: Graph of Float Matrix Multiplication Benchmark

5.3 Insertion Sort

The insertion sort test was configured to sort an array of 4000 elements 120 times.

Table 5.3: Results of the Insertion Sort Benchmark

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time O2 (s)	20.009	421.19	918.14
Execution Time O3 (s)	20.019	421.33	918.19
Execution Time Os (s)	14.016	459.23	1141.7

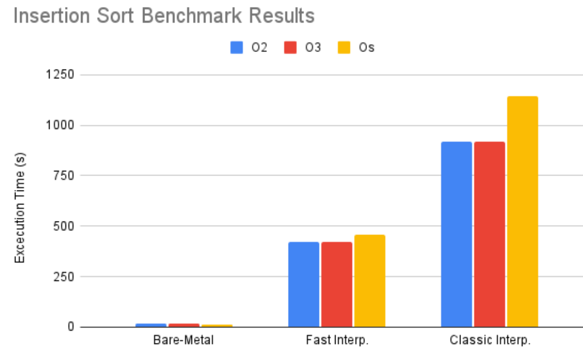

Figure 5.3: Graph of the Insertion Sort Benchmark

5.4 Linked list

All linked list tests were configured to use a list size of 7000 elements. The head insertion, tail insertion, and random access tests were run for 2000, 6, and 35000 iterations respectively.

Table 5.4: Results of the Linked List Head Insertion Benchmark

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time O2 (s)	11.559	253.57	414.45
Execution Time O3 (s)	11.434	158.84	312.28
Execution Time Os (s)	13.834	301.52	1107.7

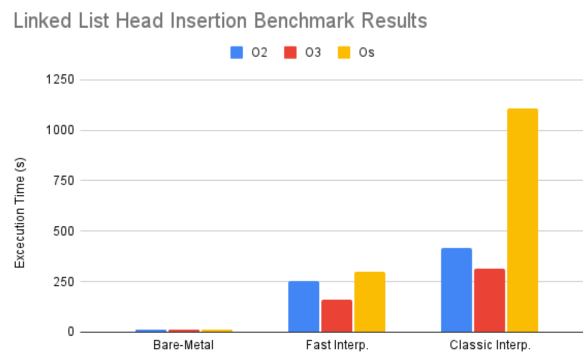

Figure 5.4: Graph of the Linked List Head Insertion Benchmark

Table 5.5: Results of the Linked List Tail Insertion Benchmark

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time Tail O2 (s)	12.303	30.443	42.246
Execution Time Tail O3 (s)	12.297	29.606	42.271
Execution Time Tail Os (s)	12.415	58.972	121.13

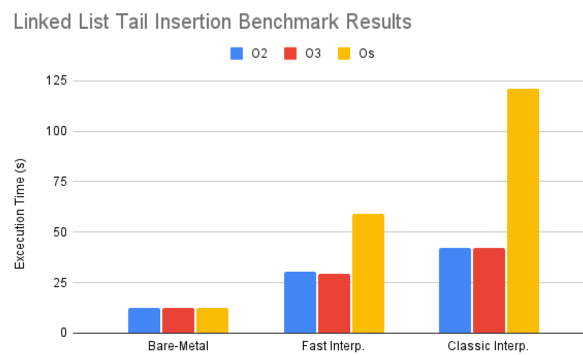

Figure 5.5: Graph of the Linked List Tail Insertion Benchmark

Table 5.6: Results of the Linked List Random Access Benchmark

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time Random Access O2 (s)	10.882	91.417	140.52
Execution Time Random Access O3 (s)	10.883	86.637	136.13
Execution Time Random Access Os (s)	10.934	92.174	141.13

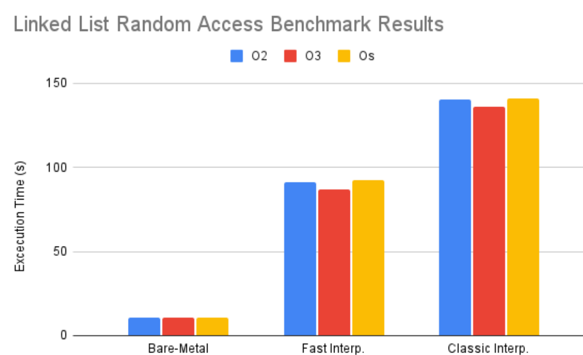

Figure 5.6: Graph of the Linked List Random Access Benchmark

5.5 Fibonacci Sequence

The test was configured to calculate the 41st Fibonacci number.

Table 5.7: The Results of the Fibonacci Sequence Benchmark

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time O2 (s)	14.227	531.47	946.15
Execution Time O3 (s)	19.211	532.59	946.22
Execution Time Os (s)	33.489	530.36	946.23

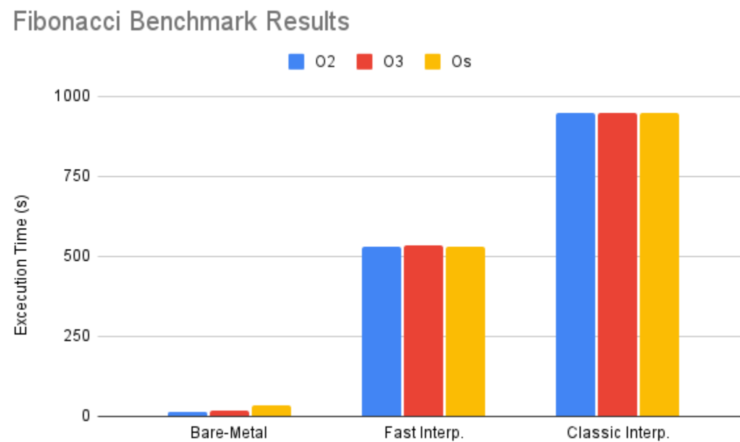

Figure 5.7: Graph of the Fibonacci Sequence Benchmark

5.6 Fast Fourier Transform (FFT)

The test was configured to use a data set of 2048 data points on which the FFT algorithm would be applied 3400 times.

Table 5.8: The Results of the FFT Benchmark

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time O2 (s)	15.082	245.83	531.79
Execution Time O3 (s)	12.376	243.74	527.06
Execution Time Os (s)	15.299	263.46	593.40

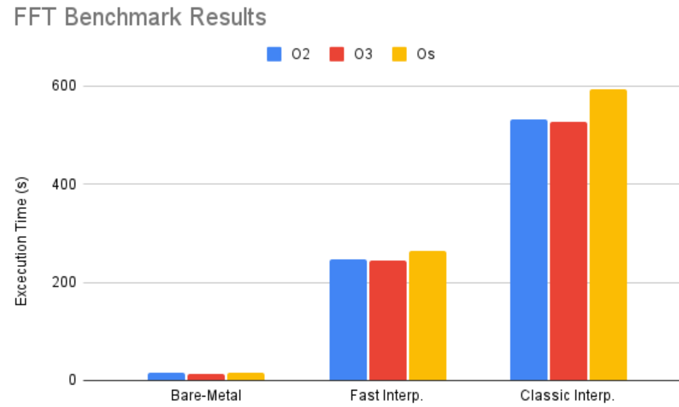

Figure 5.8: Graph of the FFT Benchmark

5.7 Arithmetic Operations

The arithmetic test was run for 30,000,000 iterations for each data type.

Table 5.9: Results of the 32-bit Arithmetic Benchmark

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time int32 O2 (s)	11.625	238.88	807.89
Execution Time int32 O3 (s)	11.625	238.88	807.89
Execution Time int32 Os (s)	12.125	238.64	807.93
Execution Time float32 O2 (s)	18.125	265.88	862.77
Execution Time float32 O3 (s)	18.125	265.88	862.77
Execution Time float32 Os (s)	21.000	265.90	862.83

Table 5.10: Results of the 64-bit Arithmetic Benchmark

Parameter	Bare-Metal	Fast Interpreter	Classic Interpreter
Execution Time int64 O2 (s)	66.000	330.26	912.90
Execution Time int64 O3 (s)	66.000	330.26	912.90
Execution Time int64 Os (s)	67.625	330.26	912.90
Execution Time float64 O2 (s)	174.63	488.74	1131.3
Execution Time float64 O3 (s)	174.63	488.92	1131.3
Execution Time float64 Os (s)	174.50	513.62	1131.3

5.8 Summary of the CPU Intensive Benchmarks

In summary, Wasm introduces a significant performance overhead compared to native C, where the average power consumption is 9.19 times that of native for the fast interpreter and 21.67 times for the classic interpreter.

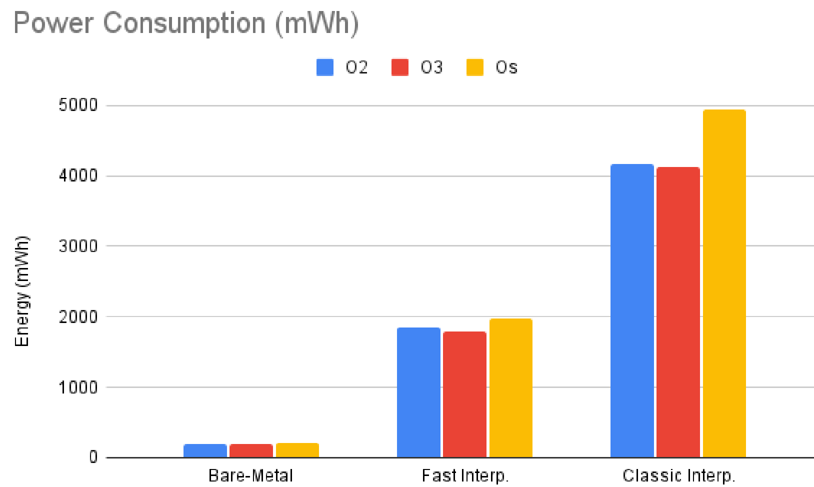

Figure 5.9: Graph of the Total Power Consumed by the Program

The average execution time is 9.36 times native performance for the fast interpreter and 22.24 times more for the classic interpreter.

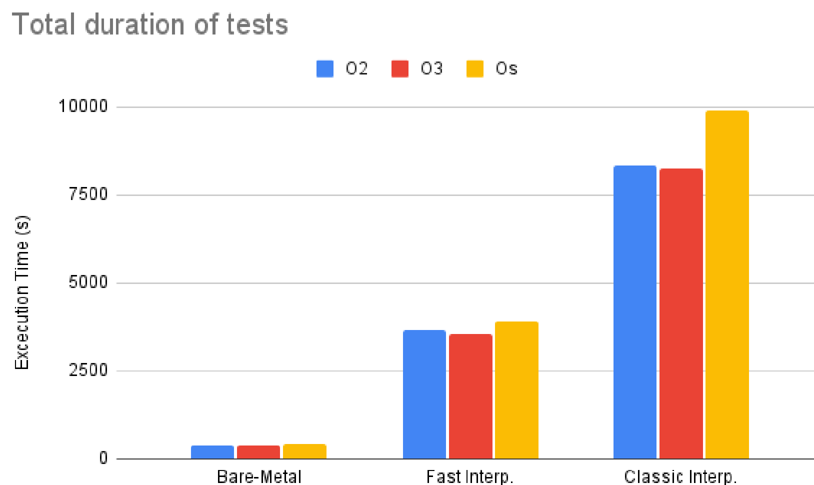

Figure 5.10: Graph of the Total Execution Time

The average binary sizes are 1.72 times larger than native for the fast interpreter and 1.62 times larger for the classic interpreter.

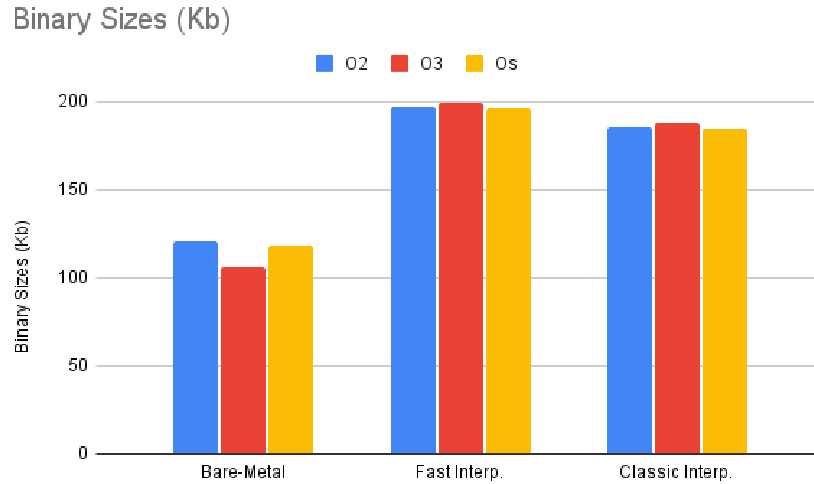

Figure 5.11: Graph of the Binary Sizes

The cold start delay was measured 10 times for each interpreter and is presented with a confidence level of 99%.

Table 5.11: Cold Start Delay

Parameter	Fast Interpreter	Classic Interpreter
Cold Start Delay (ms)	52.014±1.8352	28.819±0.420632

5.9 I/O test

The SD card I/O test ran for 1000 iterations for each data size. To capture trends in both small and large data sets, data sizes from 1 sector (512 bytes) to 4096 sectors (~2 megabytes) were used. The sizes scale logarithmically with powers of two.

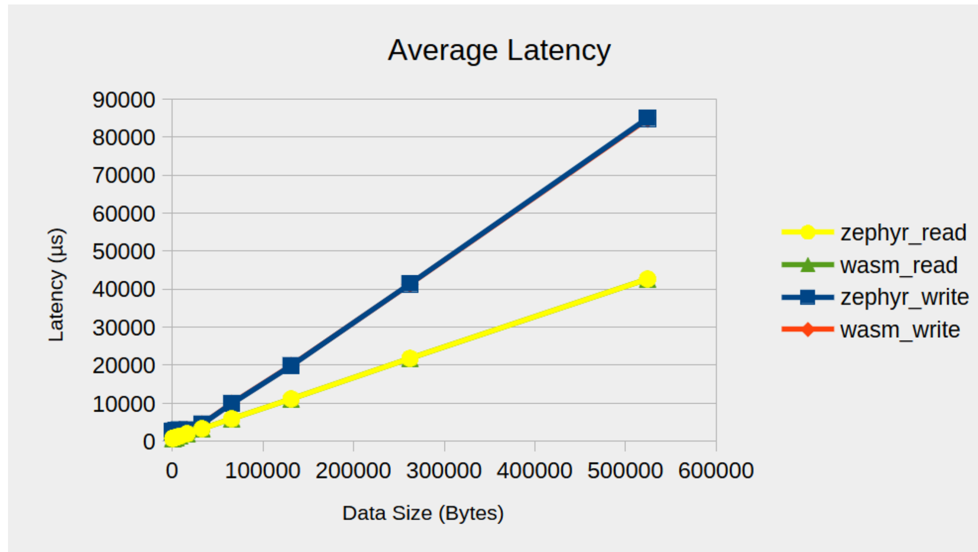

Figure 5.12: Graph of the Average Read and Write Latencies of the I/O test

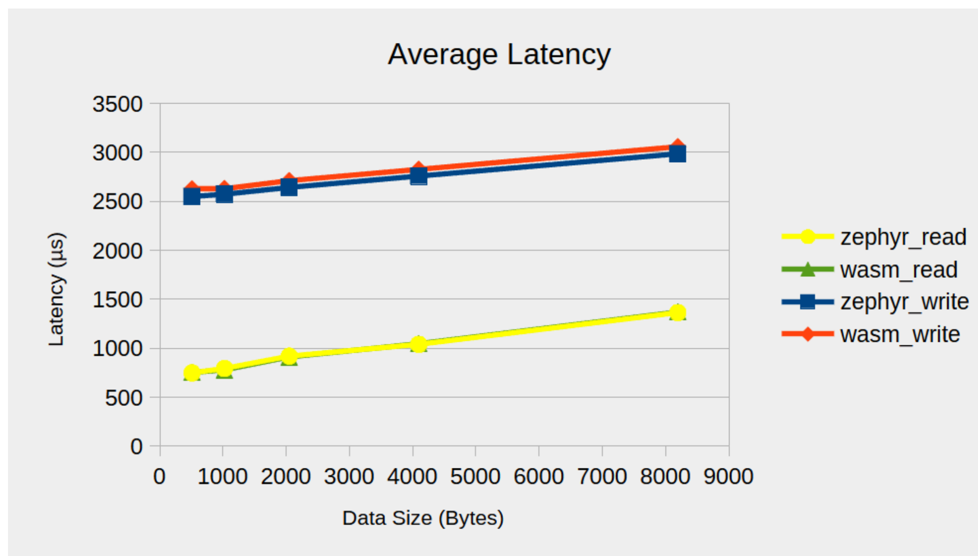

Figure 5.13: Figure 5.12 Zoomed in

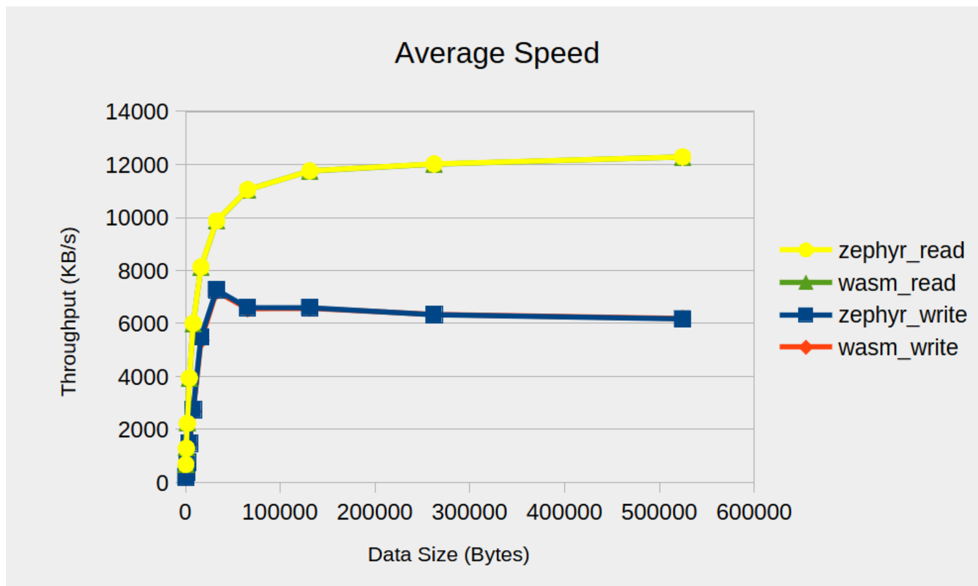

Figure 5.14: Graph of the Average Read and Write Speeds of the I/O test

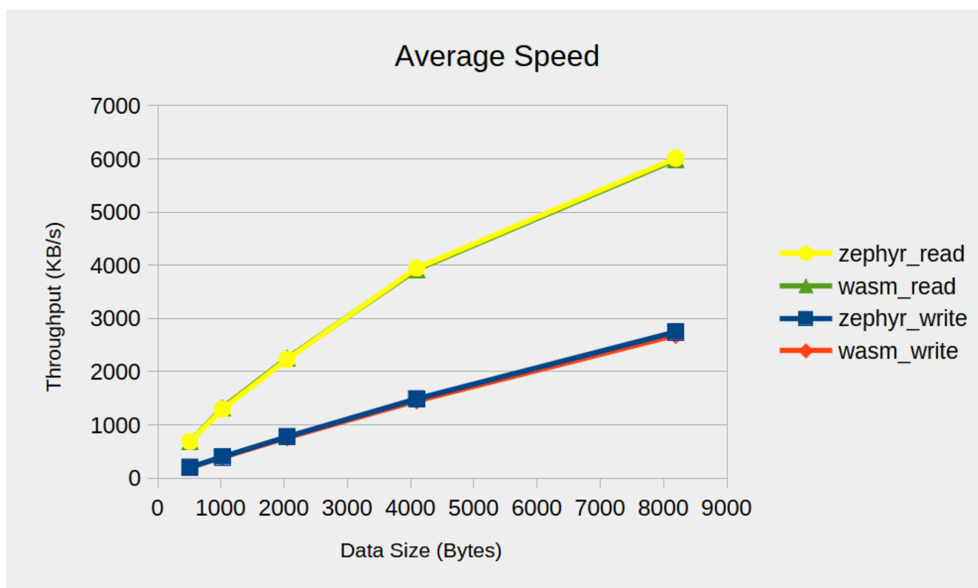

Figure 5.15: Figure 5.14 Zoomed in

Chapter 6

Discussion

The results from benchmarking show a significant performance overhead compared to native C in all tests conducted. On average, the execution times were 9.36 and 22.24 times higher for the fast and classic interpreters respectively. There were a few outliers; linked list tail insertion was only 3.21 (fast interpreter) and 5.54 (classic interpreter) slower than native. The recursive Fibonacci sequence had the most overhead, taking 37.36 (fast interpreter) and 66.50 (classic interpreter) times longer than native. While not directly comparable, the average execution overhead of the fast interpreter lands in the same ballpark as the ESP32 C3 MCU, which scored a 36 times higher CoreMark score in native code compared to the WAMR interpreter [9].

The integer matrix, insertion sort, and 32-bit integer arithmetic tests all had a very similar overhead of roughly 21 times native performance. A common factor between these tests is the usage of tight loops. Tests like the FFT algorithm and 64-bit float arithmetic also make heavy use of loops, but these two tests only introduce an overhead of 16.3 and 2.80 times, respectively, compared to native. These two tests perform more computationally expensive operations within the loops compared to the first three tests mentioned, which only use a few lines of 32-bit integer math in every loop iteration. Together with the results of the Fibonacci test, we can conclude that Wasm incurs a greater performance penalty for programs that make use of many fast loops or function calls.

Regarding average power consumption, the numbers closely follow the execution time in a linear fashion, meaning that reducing program execution time is key to reducing power consumption, which aligns with D. Branco's and P. R. Henriques's findings [29].

A general recommendation for writing programs for this system is to avoid 64-bit numbers. The arithmetic test shows that 64-bit data types are several times slower, both in bare-metal C and in Wasm, which should not be very surprising considering the fact that the Cortex M33 is a 32-bit processor.

It can also be concluded that the choice between O2 and O3 optimizations had a negligible performance impact. Using Os optimizations, however, significantly increased the execution time for many of the tests, leading to a 10% longer total

execution time. The difference in binary sizes between the optimization levels was minimal, but that could be a result of the small size of the benchmark program. For small programs such as this, O2 or even O3 optimizations can be enabled for a speed boost without any noteworthy disadvantages. In the case of the linked list head insertion test, O3 caused a substantial execution time reduction of 32% compared to O2, which shows that performance profiling is important, as it can aid in finding optimal settings for a specific application.

From the results observed in the I/O test in section 5.9, we can conclude that:

- As seen in figure 5.12 and 5.13, the read and write latency is unaffected by Wasm apart from a delay of a few microseconds which is most likely the result of function call overhead. This is expected considering the Wasm program lets the operating system manage the entire data transfer process. It can also be observed that the latency scales linearly; when the amount of data transfer is doubled, the data transfer will take twice as much time.
- The read and write speeds were similarly unaffected by Wasm, as seen in figure 5.14 and 5.13. Judging from the graphs, it is evident that reading and writing larger chunks of data at a time is generally faster, although the throughput hits a plateau at a read speed of 12 MB/s and a write speed of 6 MB/s. These bottlenecks stem from the capabilities of the SD card, hardware, or the operating system.

These two points show that Wasm can be used for simple I/O tasks without a performance penalty. Since these operations rely on exported function calls from the RTOS we believe that most, if not all, similar I/O tasks will have similar results. This could prove useful in some simple edge computing tasks, such as collecting and sending data to a server.

We have found that a Wasm solution on a microcontroller is not a substitute for a Docker container based solution. This is because of the lack of support for additional functions such as the C standard library using WAMR. However, the work currently being done on Boxer, mentioned in section 1.5, suggests that it might be possible in the near future. A direct Wasm-Docker performance comparison was also not possible since Docker cannot run on this MCU.

When comparing Wasm with JVM in terms of execution times, JVM wins by a large margin. This may change if AoT can be used, but compared with the Wasm interpreters, the KESO JVM executes programs in a fraction of the time of Wasm, as seen in section 1.5, only 4% to 23% compared to native C.

Issues came up with the software development on the development board, namely with the lack of Zephyr support in the official IDE developed by NXP. This led us to work with the VSCode extension made by NXP instead, with the downside of having fewer debugging and troubleshooting methods.

Since WAMR is not yet implemented as a complete module in Zephyr, this caused issues with creating projects using WebAssembly. The current solution of building a sample program running WAMR is insufficient for more complex Wasm solutions where one wants access to all the functionality included in Zephyr. Com-

plete module support would also integrate with the work done by NXP for board support on the RTOS.

Issues with compilation and code portability arose as soon as WAMR was integrated with Zephyr. The biggest issue was the lack of a finalized WASI implementation for Zephyr, which makes it very difficult to write programs for the platform since almost all standard library functions are missing. Zephyr also does not seem to support JiT or AoT compilation modes in WAMR which restricted us to only using the significantly slower interpreters. We also looked into using the WAZI system mentioned in section 1.5 developed by A. Ramesh et al., however, only the WALI system was available on GitHub [11].

We have found several issues with using Wasm for edge computing on embedded systems. Many of them are related to the lack of WASI implementation, which severely impacts the ease of implementing the networking features needed in edge devices. However, the results do show promise in the raw performance needed to do simple edge operations. The execution times are sufficient for many edge computing programs, especially ones pertaining to arithmetic operations. We also found that the Wasm binaries fit these types of programs on the microcontroller. In summary, WAMR based solutions can work for edge computing on microcontrollers if the increased execution time is acceptable for the specific application.

6.1 Societal Aspects

The increased overhead introduced by the WebAssembly interpreters decreases energy efficiency compared to running the software natively, which would be a problem if Wasm is used as a full replacement of native code. On the other hand, if one can replace a Docker solution with one running WAMR on a microcontroller, this would significantly improve energy efficiency.

Additionally, if WebAssembly can serve as an edge computing solution despite its overhead, it could reduce reliance on large data centers. By bringing computing closer to the data sources, the need for centralized computing infrastructure diminishes. Centralized data centers often strain local economies by consuming significant amounts of energy, a concern explored in greater detail in the article by Frans Libertson et al. [30].

Chapter 7

Conclusion

To address the exact research question posed in section 1.2 for this thesis: "is WebAssembly viable as an edge computing solution for embedded systems, and how does it compare to alternative solutions such as Docker and JVM?". We compare our results with our posed requirements in section 1.4:

- The core functionality of Wasm for the RT1180 was investigated in depth, and we found that using the Zephyr RTOS and WAMR runtime, the microcontroller is capable of running WebAssembly programs, although an implementation of WASI is needed for writing more advanced programs.
- Code portability was investigated, and we can conclude that existing edge applications will, in most cases, not be adaptable to the system, mainly caused by the lack of a WASI implementation for Zephyr. This means programmers would only have access to WAMR's very limited libc standard library. If any other C library functions are needed which are not included in libc, they would either have to be written from scratch or exported from the Zephyr environment.
- Program sandboxing is possible using our solution as it is a core feature of the Wasm standard and WAMR allows for instantiating multiple Wasm modules. The module sizes can be easily controlled, but per-module CPU usage restriction was not investigated. Module deployment over a network connection was not tested because of project time constraints, although its implementation should be possible using Zephyr's interfaces.
- Flash protection was investigated; however, more in-depth work would have to be done in order to make it work, which was not possible due to time constraints.
- Performance of Wasm was examined using different algorithms, and we found that the overhead introduced by the Wasm interpreters is several times that of native performance, but it is fast enough for small-scale data processing. The results from the limited I/O test do, however, show some promise, since the overhead observed is almost non-existent.
- The cold start delay requirement for Wasm programs is met since it is significantly shorter compared to Docker and negligible compared to the actual

running time of the benchmark program.

This leads to the conclusion that Wasm is not suitable as an edge computing solution for embedded systems in terms of performance in industrial applications, unless code portability and sandboxing are essential features needed in a system. It does however show promise in I/O intensive control tasks. We found that it is not a substitute for Docker, and it produces more performance overhead than the KESO [1] JVM solution. We could, however, not do a direct comparison between our solution and JVM, since the same tests could not be repeated on both platforms.

Especially for the RT1180 microcontroller, the usability of Wasm is limited. Until support for more operating systems and Wasm runtimes is added, it would make more sense to use other microcontrollers such as Arduino or ESP32 that have a wider range of operating systems and runtime choices.

7.1 Future Work

Future testing of WebAssembly on edge devices should include testing networking performance and application deployment. To create a solid industrial IoT solution, a WASI implementation should also be available for the specific device.

An important extension would be to implement and evaluate event-driven WebAssembly workloads and network transfer, to better reflect the needs of real-world embedded edge deployments.

Bibliography

[1] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat, “Tailor-made JVMs for statically configured embedded systems,” eng, *Concurrency and computation*, vol. 24, no. 8, pp. 789–812, 2012, ISSN: 1532-0626.

[2] K. Cao, Y. Liu, G. Meng, and Q. Sun, “An Overview on Edge Computing Research,” *IEEE Access*, vol. 8, pp. 85 714–85 728, 2020, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2991734. [Online]. Available: https://ieeexplore.ieee.org/document/9083958 (visited on 01/27/2025).

[3] T. Lauwaerts, R. G. Singh, and C. Scholliers, “WARDuino: An embedded WebAssembly virtual machine,” eng, *Journal of computer languages (Online)*, vol. 79, pp. 101 268–, 2024, ISSN: 2590-1184.

[4] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, “Challenges and Opportunities for Efficient Serverless Computing at the Edge,” in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, ISSN: 2575-8462, Oct. 2019, pp. 261–2615. DOI: 10.1109/SRDS47363.2019.00036. [Online]. Available: https://ieeexplore.ieee.org/document/9049531 (visited on 01/27/2025).

[5] *What is a Container? | Docker*. [Online]. Available: https://www.docker.com/resources/what-container/ (visited on 01/27/2025).

[6] *WebAssembly*, en. [Online]. Available: https://webassembly.org/ (visited on 01/27/2025).

[7] E. Wen and G. Weber, “Wasmachine: Bring the Edge up to Speed with A WebAssembly OS,” eng, in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, IEEE, 2020, pp. 353–360, ISBN: 9781728187808.

[8] L. Deshpande and K. Liu, “Edge computing embedded platform with container migration,” eng, in *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, IEEE, 2017, pp. 1–6, ISBN: 1538604353.

[9] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, “Potential of webassembly for embedded systems,” in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–4. DOI: 10.1109/MECO55406.2022.9797106.

[10] *EEMBC*, en. [Online]. Available: https://www.eembc.org/coremark/scores.php (visited on 04/29/2025).

[11] A. Ramesh, T. Huang, B. L. Titzer, and A. Rowe, “Empowering webassembly with thin kernel interfaces,” in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys ’25, Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 1–20, ISBN: 9798400711961. DOI: 10.1145/3689031.3717470. [Online]. Available: https://doi.org/10.1145/3689031.3717470.

[12] D. Phillips, *Dphilla/boxer*, original-date: 2022-03-08T23:17:01Z, Apr. 24, 2025. [Online]. Available: https://github.com/dphilla/boxer (visited on 04/24/2025).

[13] *How WebAssembly will transform edge computing*, en. [Online]. Available: https://www.infoworld.com/article/2338802/how-webassembly-will-transform-edge-computing.html (visited on 01/27/2025).

[14] *Zephyr Project*, Zephyr Project. [Online]. Available: https://www.zephyrproject.org/ (visited on 01/29/2025).

[15] *Non-Volatile Storage (NVS) — Zephyr Project Documentation*. [Online]. Available: https://docs.zephyrproject.org/latest/services/storage/nvs/nvs.html (visited on 05/08/2025).

[16] *WebAssembly Micro Runtime*, WAMR. [Online]. Available: https://bytecodealliance.github.io/wamr.dev/ (visited on 01/28/2025).

[17] X. Wang and W. Huang. “The WAMR memory model,” WAMR. (Mar. 21, 2023), [Online]. Available: https://bytecodealliance.github.io/wamr.dev/blog/the-wamr-memory-model/ (visited on 03/05/2025).

[18] *i.MX RT1180: Crossover MCU with TSN Switch and EdgeLock®*. [Online]. Available: https://www.nxp.com/products/i.MX-RT1180 (visited on 01/27/2025).

[19] *The GNU C Reference Manual*. [Online]. Available: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html (visited on 01/29/2025).

[20] BLAS Technical Forum, “Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard,” University of Tennessee, Tech. Rep., Aug. 2001. [Online]. Available: https://netlib.org/blas/blast-forum/blas-report.pdf.

[21] T. Bingmann, J. Marianczuk, and P. Sanders, "Engineering faster sorters for small sets of items," *Software: Practice and Experience*, vol. 51, no. 5, pp. 965–1004, 2021. DOI: `https://doi.org/10.1002/spe.2922`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2922`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2922`.

[22] M. I. S. R. Association, *MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems*. MIRA, 2013, ISBN: 978-1-906400-10-1. [Online]. Available: `https://books.google.se/books?id=3yZKmwEACAAJ`.

[23] M. T. Heideman, D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast fourier transform," *Archive for History of Exact Sciences*, vol. 34, no. 3, pp. 265–277, Sep. 1, 1985, ISSN: 1432-0657. DOI: `10.1007/BF00348431`. [Online]. Available: `https://doi.org/10.1007/BF00348431`.

[24] "An algorithm for the machine calculation of complex fourier series," in *Papers on Digital Signal Processing*, A. V. Oppenheim, Ed., The MIT Press, Nov. 15, 1969, p. 0, ISBN: 978-0-262-31084-0. DOI: `10.7551/mitpress/5222.003.0014`. [Online]. Available: `https://doi.org/10.7551/mitpress/5222.003.0014` (visited on 08/04/2025).

[25] F. Stefani, A. Moschitta, D. Macii, and D. Petri, "Fft benchmarking for digital signal processing technologies," Jan. 2004.

[26] *i.MX RT1180 Evaluation Kit*. [Online]. Available: `https://www.nxp.com/design/design-center/development-boards-and-designs/MIMXRT1180-EVK` (visited on 01/29/2025).

[27] *Nxp-mcuxpresso/mcux-sdk*, Apr. 2025. [Online]. Available: `https://github.com/nxp-mcuxpresso/mcux-sdk` (visited on 04/25/2025).

[28] *Bytecodealliance/wasm-micro-runtime*, Apr. 2025. [Online]. Available: `https://github.com/bytecodealliance/wasm-micro-runtime` (visited on 04/25/2025).

[29] D. Branco and P. R. Henriques, "Impact of GCC Optimization Levels in Energy Consumption During Program Execution," en, *Acta Electrotechnica et Informatica*, vol. 16, no. 1, pp. 20–26, Mar. 2016, ISSN: 13358243, 13383957. DOI: `10.15546/aeei-2016-0004`. [Online]. Available: `http://www.aei.tuke.sk/papers/2016/1/04_Branco.pdf` (visited on 04/27/2025).

[30] F. Libertson, J. Velkova, and J. P. and, "Data-center infrastructure and energy gentrification: Perspectives from sweden," *Sustainability: Science, Practice and Policy*, vol. 17, no. 1, pp. 152–161, 2021. DOI: `10.1080/15487733.2021.1901428`. eprint: `https://doi.org/10.1080/15487733.2021.1901428`. [Online]. Available: `https://doi.org/10.1080/15487733.2021.1901428`.

Appendix A

Source Code

Listing A.1: Benchmark Suite

```
/* TEST SETTINGS */

// Fibonacci
#define FIB_N 41

// Linked list
#define LLIST_INSERT_HEAD_COUNT 7000
#define LLIST_INSERT_HEAD_ITER  2000

#define LLIST_INSERT_TAIL_COUNT 7000
#define LLIST_INSERT_TAIL_ITER  6

#define LLIST_RAND_ACCESS_LIST_SIZE 7000
#define LLIST_RAND_ACCESS_ITER      35000

// Matrix
#define MATRIX_SIZE 62
#define MATRIX_ITER 2100

// Insertion sort
#define INSERTION_SORT_SIZE 4000
#define INSERTION_SORT_ITER 120

// Fast Fourier Transform
#define FFT_SIZE 2048 // Must be a power of 2
#define FFT_ITER 3400

// Integer and double arithmetic
#define ARITHMETIC_ITER 30000000

/* TEST CONFIGURATION */

#define USE_FIBONACCI 1

#define USE_LLIST_INSERT_HEAD 1
#define USE_LLIST_INSERT_TAIL 1
#define USE_LLIST_RANDOM_ACCESS 1

#define USE_MATRIX_MUL_INT 1
#define USE_MATRIX_MUL_FLOAT 1
```

```
#define USE_INSERTION_SORT 1

#define USE_FFT 1

#define USE_ARITHMETIC_INT32 1
#define USE_ARITHMETIC_INT64 1
#define USE_ARITHMETIC_FLOAT32 1
#define USE_ARITHMETIC_FLOAT64 1

/* ------------------- */

#include "platform_common.h"
#include "Stopwatch.h"

#include "Matrix.h"
#include "FFT.h"
#include "Fibonacci.h"
#include "LinkedList.h"
#include "InsertionSort.h"
#include "Arithmetic.h"

#define print_test(test_name) {\
    printf("\nStarting␣test:␣%s\n----------------------------\n", test_name)\
    }

int main(void)
{
    /* Setup */
    init();
    stopwatch_init();
    uint64_t start_time = get_system_ticks();

    /* Fibonacci */
    print_test("Recursive␣Fibonacci");
    #if USE_FIBONACCI
        printf("Calculating␣fib(%d)...", FIB_N);
        stopwatch_start();
        uint32_t f = fib(FIB_N);
        stopwatch_stop();
        printf("=%d␣(%dms)\n", f, stopwatch_millis_elapsed());
    #endif

    /* Linked list */
    print_test("Linked␣List");
    // Head insertion
    #if USE_LLIST_INSERT_HEAD
        printf("Insertion␣at␣head␣(%d␣nodes,␣%d␣iterations)...",
                LLIST_INSERT_HEAD_COUNT, LLIST_INSERT_HEAD_ITER);
        stopwatch_start();
        for (int i = 0; i < LLIST_INSERT_HEAD_ITER; i++) {
            LinkedList *l = LinkedList_();

            for (int i = 0; i < LLIST_INSERT_HEAD_COUNT; i++)
                llist_insert(l, 0, 0);

            llist_delete(l);
        }
        stopwatch_stop();
        printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
```

```
#endif
// Tail insertion
#if USE_LLIST_INSERT_TAIL
    printf("Insertion␣at␣tail␣(%d␣nodes,␣%d␣iterations)...",
            LLIST_INSERT_TAIL_COUNT, LLIST_INSERT_TAIL_ITER);
    stopwatch_start();
    for (int i = 0; i < LLIST_INSERT_TAIL_ITER; i++) {
        LinkedList *l = LinkedList_();

        for (int i = 0; i < LLIST_INSERT_TAIL_COUNT; i++)
            llist_insert(l, 0, llist_size(l));

        llist_delete(l);
    }
    stopwatch_stop();
    printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
#endif
// Random data access
#if USE_LLIST_RANDOM_ACCESS
    printf("Random␣data␣access␣(%d␣nodes,␣%d␣iterations)...",
            LLIST_RAND_ACCESS_LIST_SIZE, LLIST_RAND_ACCESS_ITER);
    LinkedList *l = LinkedList_();
    for (int i = 0; i < LLIST_RAND_ACCESS_LIST_SIZE; i++)
        llist_insert(l, 0, 0);

    volatile data_t getVal;
    stopwatch_start();
    for (int i = 0; i < LLIST_RAND_ACCESS_ITER; i++)
        getVal = llist_get(l, rand() % llist_size(l));
    stopwatch_stop();

    llist_delete(l);
    printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
#endif

/* Matrix */
print_test("Matrix");
// Integer
#if USE_MATRIX_MUL_INT
    printf("[%dX%d]␣integer␣matrix␣multiplication␣(%d␣iterations)...",
            MATRIX_SIZE, MATRIX_SIZE, MATRIX_ITER);
    static iMatrix ia, ib;
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            ia[i][j] = rand();
            ib[i][j] = rand();
        }
    }
    stopwatch_start();
    for (int i = 0; i < MATRIX_ITER; i++)
        mat_i_mul(&ia, &ib);
    stopwatch_stop();
    printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
#endif
// Float
#if USE_MATRIX_MUL_FLOAT
    printf("[%dX%d]␣float␣matrix␣multiplication␣(%d␣iterations)...",
            MATRIX_SIZE, MATRIX_SIZE, MATRIX_ITER);
    static fMatrix fa, fb;
    for (int i = 0; i < MATRIX_SIZE; i++) {
```

```
        for (int j = 0; j < MATRIX_SIZE; j++) {
            fa[i][j] = (float)rand();
            fb[i][j] = (float)rand();
        }
    }
    stopwatch_start();
    for (int i = 0; i < MATRIX_ITER; i++)
        mat_f_mul(&fa, &fb);
    stopwatch_stop();
    printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
#endif

/* Insertion sort */
print_test("Insertion␣sort");
#if USE_INSERTION_SORT
    printf("Sorting␣(%d␣elements,␣%d␣iterations)...",
            INSERTION_SORT_SIZE, INSERTION_SORT_ITER);
    static int arr[INSERTION_SORT_SIZE];
    uint64_t time_sum = 0;
    for (int i = 0; i < INSERTION_SORT_ITER; i++) {
        // Fill array with random numbers
        for (int j = 0; j < INSERTION_SORT_SIZE; j++)
            arr[j] = rand();
        stopwatch_start();
        insertion_sort(arr, INSERTION_SORT_SIZE);
        stopwatch_stop();
        time_sum += stopwatch_ticks_elapsed();
    }
    printf("␣Done␣(%dms)\n", (uint32_t)((1000 * time_sum) / CPU_FREQ));
#endif

/* Fast Fourier Transform */
print_test("Fast␣Fourier␣Transform");
#if USE_FFT
    printf("FFT␣(%d␣data␣points,␣%d␣iterations)...", FFT_SIZE, FFT_ITER);
    static complex float arr2[FFT_SIZE];
    for (int i = 0; i < FFT_SIZE; i++)
        arr2[i] = (float)rand() + (float)rand() * I;

    stopwatch_start();
    for (int i = 0; i < FFT_ITER; i++)
        fft(arr2, FFT_SIZE);
    stopwatch_stop();
    printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
#endif

/* Arithmetic */
print_test("Arithmetic␣operations");
#if USE_ARITHMETIC_INT32
    printf("int32␣(%d␣iterations)...", ARITHMETIC_ITER);
    stopwatch_start();
    arithmetic_int32(ARITHMETIC_ITER);
    stopwatch_stop();
    printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
#endif
#if USE_ARITHMETIC_INT64
    printf("int64␣(%d␣iterations)...", ARITHMETIC_ITER);
    stopwatch_start();
    arithmetic_int64(ARITHMETIC_ITER);
    stopwatch_stop();
```

```
        printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
    #endif
    #if USE_ARITHMETIC_FLOAT32
        printf("float32␣(%d␣iterations)...", ARITHMETIC_ITER);
        stopwatch_start();
        arithmetic_float32(ARITHMETIC_ITER);
        stopwatch_stop();
        printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
    #endif
    #if USE_ARITHMETIC_FLOAT64
        printf("float64␣(%d␣iterations)...", ARITHMETIC_ITER);
        stopwatch_start();
        arithmetic_float64(ARITHMETIC_ITER);
        stopwatch_stop();
        printf("␣Done␣(%dms)\n", stopwatch_millis_elapsed());
    #endif

    uint64_t end_time = get_system_ticks();
    uint32_t total_time = (uint32_t)((1000 * (end_time - start_time)) / CPU_FREQ);
    printf("\nTotal␣benchmark␣duration:␣%dms", total_time);
}
```

Listing A.2: Arithmetic Benchmark Implementation

```
#ifndef ARITHMETIC_H
#define ARITHMETIC_H

volatile int32_t i_a, i_b, i_c, i_d, i_e, i_f, i_g, i_h, i_i, i_j;
volatile int64_t I_a, I_b, I_c, I_d, I_e, I_f, I_g, I_h, I_i, I_j;
volatile float   f_a, f_b, f_c, f_d, f_e, f_f, f_g, f_h, f_i, f_j;
volatile double  F_a, F_b, F_c, F_d, F_e, F_f, F_g, F_h, F_i, F_j;

void arithmetic_int32(size_t iterations) {
    for (size_t i = 0; i < iterations; i++) {
        i_a += i_b * i_c;
        i_b -= i_c / (i_d + (i_d == 0));
        i_c *= i_d + i_e;
        i_d /= i_e + i_f + ((i_e + i_f) == 0);
        i_e += i_f - i_g;
        i_f -= i_g * i_h;
        i_g *= i_h + i_i;
        i_h /= i_i + i_j + ((i_i + i_j) == 0);
        i_i += i_j - i_a;
        i_j -= i_a * i_b;
    }
}

void arithmetic_int64(size_t iterations) {
    for (size_t i = 0; i < iterations; i++) {
        I_a += I_b * I_c;
        I_b -= I_c / (I_d + (I_d == 0));
        I_c *= I_d + I_e;
        I_d /= I_e + I_f + ((I_e + I_f) == 0);
        I_e += I_f - I_g;
        I_f -= I_g * I_h;
        I_g *= I_h + I_i;
        I_h /= I_i + I_j + ((I_i + I_j) == 0);
        I_i += I_j - I_a;
        I_j -= I_a * I_b;
    }
```

```
}

void arithmetic_float32(size_t iterations) {
    for (size_t i = 0; i < iterations; i++) {
        f_a += f_b * f_c;
        f_b -= f_c / (f_d + (f_d == 0));
        f_c *= f_d + f_e;
        f_d /= f_e + f_f + ((f_e + f_f) == 0);
        f_e += f_f - f_g;
        f_f -= f_g * f_h;
        f_g *= f_h + f_i;
        f_h /= f_i + f_j + ((f_i + f_j) == 0);
        f_i += f_j - f_a;
        f_j -= f_a * f_b;
    }
}

void arithmetic_float64(size_t iterations) {
    for (size_t i = 0; i < iterations; i++) {
        F_a += F_b * F_c;
        F_b -= F_c / (F_d + (F_d == 0));
        F_c *= F_d + F_e;
        F_d /= F_e + F_f + ((F_e + F_f) == 0);
        F_e += F_f - F_g;
        F_f -= F_g * F_h;
        F_g *= F_h + F_i;
        F_h /= F_i + F_j + ((F_i + F_j) == 0);
        F_i += F_j - F_a;
        F_j -= F_a * F_b;
    }
}

#endif
```

Listing A.3: FFT Benchmark Implementation

```
#ifndef FFT_H
#define FFT_H

#include <complex.h>
#undef cexpf
#undef __mulsc3

#define M_PI   3.141592653589793238462643383279502884197  // pi
#define M_PI_2 1.570796326794896619231321691639751442099  // pi/2
#define M_PI_4 0.7853981633974483096156608458198757210493 // pi/4
#define M_2_PI 6.283185307179586476925286766559005768394  // 2*pi

float fmodf(float x, float y) {
    int quotient = (int)(x / y);
    float remainder = x - y * quotient;
    if (remainder < 0.0f)
        remainder += y;
    return remainder;
}

float sinf(float x) {
    // -pi < x < pi
    x = fmodf(x, M_2_PI);
    if (x > M_PI)
```

```
        x -= M_2_PI;

    const float coefficients[] = {
        -1.0f/6.0f,            // -x^3/3!
         1.0f/120.0f,          //  x^5/5!
        -1.0f/5040.0f,         // -x^7/7!
         1.0f/362880.0f,       //  x^9/9!
        -1.0f/39916800.0f,     // -x^11/11!
         1.0f/6227020800.0f,   //  x^13/13!
        -1.0f/1307674368000.0f //  x^15/15!
    };

    float result = x;
    float term = x;
    float xx = x * x;

    for (int i = 0; i < sizeof(coefficients)/sizeof(float); i++) {
        term *= xx;
        result += term * coefficients[i];
    }

    return result;
}

float cosf(float x) {
    return sinf(x + M_PI_2);
}

float expf(float x) {
    const float coefficients[] = {
        1.0f,                 // x^1/1!
        1.0f/2.0f,            // x^2/2!
        1.0f/6.0f,            // x^3/3!
        1.0f/24.0f,           // x^4/4!
        1.0f/120.0f,          // x^5/5!
        1.0f/720.0f,          // x^6/6!
        1.0f/5040.0f,         // x^7/7!
        1.0f/40320.0f,        // x^8/8!
        1.0f/362880.0f,       // x^9/9!
        1.0f/3628800.0f,      // x^10/10!
        1.0f/39916800.0f,     // x^11/11!
        1.0f/479001600.0f,    // x^12/12!
        1.0f/6227020800.0f,   // x^13/13!
        1.0f/87178291200.0f,  // x^14/14!
        1.0f/1307674368000.0f // x^15/15!
    };

    float result = 1.0f;
    float term = 1.0f;

    for (int i = 0; i < sizeof(coefficients)/sizeof(float); i++) {
        term *= x;
        result += term * coefficients[i];
    }

    return result;
}

complex float __mulsc3(float a, float b, float c, float d) {
    return (a*c - b*d) + (a*d + b*c)*I;
```

```
}

complex float cexpf(complex float z) {
    float x = crealf(z);  // Real part
    float y = cimagf(z);  // Imaginary part

    float expx = expf(x);
    return expx * cosf(y) + (expx * sinf(y)) * I;
}

uint16_t reverse_bits(uint16_t n, uint16_t nBits) {
    uint16_t reversed = 0;
    for (uint16_t i = 0; i < nBits; i++) {
        reversed <<= 1;
        reversed |= n & 1;
        n >>= 1;
    }
    return reversed;
}

void reorder_data(float complex *data, size_t n) {
    // Calculate number of bits needed for bit reversing indexes
    uint16_t nBits = 0;
    for (size_t i = n; i > 1; i >>= 1) // log2(n)
        nBits++;

    // Reorder data
    for (size_t i = 0; i < n; i++) {
        uint16_t newIndex = reverse_bits(i, nBits);
        // Only swap with indexes >i to avoid double swap
        if (newIndex > i) {
            // Swap values at indexes newIndex and i
            complex float tmp = data[i];
            data[i] = data[newIndex];
            data[newIndex] = tmp;
        }
    }
}

void fft(complex float *data, size_t n) {
    reorder_data(data, n);

    for (size_t s = 2; s <= n; s <<= 1) {
        complex float wm = cexpf(-2.0f*M_PI * I / s);
        for (size_t k = 0; k < n; k+=s) {
            complex float w = 1.0f;
            for (size_t j = 0; j < s/2; j++) {
                complex float t = w * data[k + j + s/2];
                complex float u = data[k + j];
                data[k + j] = u + t;
                data[k + j + s/2] = u - t;
                w *= wm;
            }
        }
    }
}

#endif
```

Listing A.4: Fibonacci Benchmark Implementation

```
#ifndef FIBONACCI_H
#define FIBONACCI_H

uint32_t fib(uint32_t n) {
    return n < 2 ? n : fib(n - 2) + fib(n - 1);
}

#endif
```

Listing A.5: Insertion Sort Benchmark Implementation

```
#ifndef INSERTION_SORT_H
#define INSERTION_SORT_H

void insertion_sort(int data[], size_t n) {
    for (size_t i = 1; i < n; i++) {
        int x = data[i];
        int j = i - 1;
        while (j >= 0 && data[j] > x) {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = x;
    }
}

#endif
```

Listing A.6: Linked List Benchmark Implementation

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

#include "platform_common.h"
#include <stdlib.h>

typedef int32_t data_t;

typedef struct Node {
    data_t data;
    struct Node *pNext;
} Node;

typedef struct LinkedList {
    size_t size;
    Node *pHead;
} LinkedList;

// Exception message macro
#define ll_except(condition, msg) {\
        if (condition) {\
            printf("\nLinkedList␣Error:␣%s\n", msg);\
            exit(1);\
        }\
    }

LinkedList *LinkedList_() {
    LinkedList *ll = (LinkedList *)malloc(sizeof(LinkedList));
```

```
    ll_except(!ll, "List␣memory␣allocation␣failed");
    ll->pHead = NULL;
    ll->size = 0;
    return ll;
}

Node *Node_(data_t value) {
    Node *n = (Node *)malloc(sizeof(Node));
    ll_except(!n, "Node␣memory␣allocation␣failed");
    n->data = value;
    return n;
}

void llist_insert(LinkedList *list, data_t value, size_t index) {
    ll_except(!list, "Attempted␣insertion␣into␣NULL␣list");
    ll_except(index > list->size, "Index␣out␣of␣bounds");

    Node *n = Node_(value);

    if (index == 0) {
        n->pNext = list->pHead;
        list->pHead = n;
    } else {
        Node *tmp = list->pHead;
        size_t position = 0;
        while (++position != index)
            tmp = tmp->pNext;

        n->pNext = tmp->pNext;
        tmp->pNext = n;
    }

    list->size++;
}

data_t llist_get(LinkedList *list, size_t index) {
    ll_except(!list, "Attempted␣fetching␣from␣NULL␣list");
    ll_except(index >= list->size, "Index␣out␣of␣bounds");

    Node *tmp = list->pHead;

    size_t position = 0;
    while (position++ != index)
        tmp = tmp->pNext;

    return tmp->data;
}

void llist_remove(LinkedList *list, size_t index) {
    ll_except(!list, "Attempted␣deletion␣from␣NULL␣list");
    ll_except(index >= list->size, "Index␣out␣of␣bounds");

    Node *tmp = list->pHead;

    if (index == 0) {
        list->pHead = tmp->pNext;
        free(tmp);
    } else {
        size_t position = 0;
        while (++position != index)
```

```
            tmp = tmp->pNext;

        Node *delNode = tmp->pNext;
        tmp->pNext = delNode->pNext;
        free(delNode);
    }

    list->size--;
}

void llist_delete(LinkedList *list) {
    Node *tmp = list->pHead;
    for (size_t i = 0; i < list->size; i++) {
        Node *delNode = tmp;
        tmp = delNode->pNext;
        free(delNode);
    }
    free(list);
    list = NULL;
}

size_t llist_size(LinkedList *list) {
    return list->size;
}

void llist_print(LinkedList *list) {
    ll_except(!list, "Attempted printing NULL list");
    printf("{");
    for (size_t i = 0; i < list->size; i++) {
        printf("%d", llist_get(list, i));
        if (i < list->size-1)
            printf(", ");
    }
    printf("}");
}

#endif
```

Listing A.7: Matrix Multiplication Benchmark Implementation

```
#ifndef MATRIX_H
#define MATRIX_H

#ifndef MATRIX_SIZE
#define MATRIX_SIZE 16
#endif

typedef float fMatrix[MATRIX_SIZE][MATRIX_SIZE];
typedef int   iMatrix[MATRIX_SIZE][MATRIX_SIZE];

void mat_i_mul(iMatrix *mat1, iMatrix *mat2) {
    for (size_t m = 0; m < MATRIX_SIZE; m++) {
        for (size_t n = 0; n < MATRIX_SIZE; n++) {
            int sum = 0;
            for (size_t i = 0; i < MATRIX_SIZE; i++)
                sum += (*mat1)[m][i] * (*mat2)[i][n];
            (*mat1)[m][n] = sum;
        }
    }
}
```

```
void mat_f_mul(fMatrix *mat1, fMatrix *mat2) {
    for (size_t m = 0; m < MATRIX_SIZE; m++) {
        for (size_t n = 0; n < MATRIX_SIZE; n++) {
            float sum = 0;
            for (size_t i = 0; i < MATRIX_SIZE; i++)
                sum += (*mat1)[m][i] * (*mat2)[i][n];
            (*mat1)[m][n] = sum;
        }
    }
}

#endif
```

Listing A.8: Stopwatch Implementation

```
#ifndef STOPWATCH_H
#define STOPWATCH_H

#define CPU_FREQ 240000000
#define INIT_ITERATIONS 10000

uint64_t sw_overhead = 0;
uint64_t sw_value = 0;

void stopwatch_start() {
    sw_value = get_system_ticks();
}

void stopwatch_stop() {
    sw_value = get_system_ticks() - sw_value;
}

uint64_t stopwatch_ticks_elapsed() {
    return sw_value - sw_overhead;
}

uint32_t stopwatch_millis_elapsed() {
    return (uint32_t)((1000 * stopwatch_ticks_elapsed()) / CPU_FREQ);
}

void stopwatch_init() {
    uint64_t sum = 0;
    for (int i = 0; i < INIT_ITERATIONS; i++) {
        stopwatch_start();
        stopwatch_stop();
        sum += stopwatch_ticks_elapsed();
    }
    sw_overhead = sum / INIT_ITERATIONS;
}

#endif
```

Listing A.9: I/O Test Implementation

```
#ifndef BENCHMARK_H
#define BENCHMARK_H

#include "Interface.h"
```

```
#include "Stopwatch.h"
#include <stdio.h>

#define SECTOR_SIZE 512

#define TICKS_PER_US (CPU_FREQ / 1000000ULL)

#define TEST_READ 1
#define TEST_WRITE 1

static uint8_t buffer[SECTOR_SIZE * 4096];

void do_test(int sector_count, int iterations)
{
    uint64_t data_size = sector_count * SECTOR_SIZE;
    printf("Testing␣read␣and␣write␣delay␣of␣%d␣bytes␣for␣%d␣iterations...\n",
        (uint32_t)data_size, iterations);

    // These variables store the time (in ticks) it takes
    // to complete the r/w operation
    uint64_t min, max, total;

    #if TEST_READ
        min = -1; max = 0; total = 0;
        // Warm-up
        for (int i = 0; i < 200; i++)
            sd_read(buffer, 0, sector_count);
        // Read
        for (int i = 0; i < iterations; i++) {
            stopwatch_start();
            sd_read(buffer, 0, sector_count);
            stopwatch_stop();
            uint64_t elapsed = stopwatch_ticks_elapsed();

            total += elapsed;
            if (elapsed < min) min = elapsed;
            if (elapsed > max) max = elapsed;
        }
        printf("Read:\n");
        printf("␣␣␣␣min:%dus␣(%dKB/s)\n",
            (uint32_t)(min / TICKS_PER_US),
            (uint32_t)((data_size * CPU_FREQ) / (min * 1000)));
        printf("␣␣␣␣max:%dus␣(%dKB/s)\n",
            (uint32_t)(max / TICKS_PER_US),
            (uint32_t)((data_size * CPU_FREQ) / (max * 1000)));
        printf("␣␣␣␣avg:%dus␣(%dKB/s)\n",
            (uint32_t)(total / (iterations * TICKS_PER_US)),
            (uint32_t)((data_size * CPU_FREQ * iterations) / (total * 1000)));
    #endif

    #if TEST_WRITE
        min = -1; max = 0; total = 0;
        // Warm-up
        for (int i = 0; i < 200; i++)
            sd_write(buffer, 0, sector_count);
        // Write
        for (int i = 0; i < iterations; i++) {
            stopwatch_start();
            sd_write(buffer, 0, sector_count);
            stopwatch_stop();
```

```
            uint64_t elapsed = stopwatch_ticks_elapsed();

            total += elapsed;
            if (elapsed < min) min = elapsed;
            if (elapsed > max) max = elapsed;
        }
        printf("Write:\n");
        printf("␣␣␣␣min:%dus␣(%dKB/s)\n",
            (uint32_t)(min / TICKS_PER_US),
            (uint32_t)((data_size * CPU_FREQ) / (min * 1000)));
        printf("␣␣␣␣max:%dus␣(%dKB/s)\n",
            (uint32_t)(max / TICKS_PER_US),
            (uint32_t)((data_size * CPU_FREQ) / (max * 1000)));
        printf("␣␣␣␣avg:%dus␣(%dKB/s)\n",
            (uint32_t)(total / (iterations * TICKS_PER_US)),
            (uint32_t)((data_size * CPU_FREQ * iterations) / (total * 1000)));
    #endif

    printf("\n");
}

void run_benchmark()
{
    stopwatch_init();

    for (int i = 0; i <= 12; i++)
        do_test(1 << i, 1000);
}

#endif
```

Appendix B

Extra Material

B.1 Setting up dependencies

The project begins with a hardware and software setup. The RTOS Zephyr [14] is installed on the platform with the WebAssembly Micro Runtime (WAMR) on top of it [16] which allows for execution of compiled Wasm programs. This is done by:

1. Setting up the environment
    a. Ubuntu version 24.04.2, a Linux distribution, is installed on a workspace computer. This is recommended to simplify declaration of environment variables and use of communication ports. The same steps can still be followed on a Windows machine.
    b. Zephyr dependencies and SDK are installed in the workspace.
    c. West, a command-line tool for Zephyr is installed via pip (included in Python).
    d. The Zephyr repository is then downloaded and initialized using West.
2. Configuring Zephyr
    a. An environment variable is added to the base of the Zephyr installation. This is a necessary step for allowing programs built outside of the repository access to the RTOS.
    b. VSCode is installed along with the MCUXpresso extension which provides the necessary programs for flashing the development board.
    c. We chose to use a program called LinkServer, provided by NXP to flash the board. LinkServer is added to the system path in order to allow it to be used outside of the VSCode environment.
3. Integrating WebAssembly Micro Runtime (WAMR)
    a. The WAMR repository is downloaded using Git.
    b. A Zephyr instanced program is included in the repository, which is then compiled. This program is device dependent, and requires definitions of certain device specific functions to be specified in a configuration

file.

c. WAMR is tested by running a simple "hello world" program compiled to Wasm bytecode as input.

B.2 Low Level settings

1. **prj.conf** The following Zephyr configuration arguments are added:
   CONFIG_SPEED_OPTIMIZATIONS=y (Enable O2 optimizations)
   CONFIG_FPU=y (Enable the CPU's floating point unit)
2. **settings.json** The following WAMR configuration arguments are added:
   "cmake.configureArgs":[
   "-DWAMR_BUILD_TARGET=THUMBV8", // CPU architecture
   "-DWAMR_BUILD_AOT=0", // Disable AoT support
   "-DWAMR_BUILD_FAST_INTERP=1", // Enable fast interpreter
   "-DWAMR_BUILD_LIBC_BUILTIN=1", // Use WAMR libc
   "-DWAMR_BUILD_LIBC_WASI=0", // Disable WASI
   "-DWAMR_BUILD_GLOBAL_HEAP_POOL=0" // Use Zephyr malloc
   ]