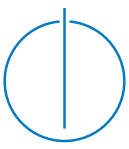
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Evaluation of WebAssembly IoT Runtimes on a ESP32 Microcontroller

Lukas Heddendorp



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Evaluation of WebAssembly IoT Runtimes on a ESP32 Microcontroller

# Evaluation von WebAssembly IoT Runtimes auf einem ESP32 Microcontroller

| | |
|---|---|
| Author: | Lukas Heddendorp |
| Supervisor: | Prof. Dr.-Ing. Jörg Ott |
| Advisor: | M.Sc. Teemu Kärkkäinen |
| Submission Date: | 16.03.2020 |

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.03.2020 Lukas Heddendorp

# Abstract

Microcontrollers are all around us and used in many different devices. They fulfill particular tasks and are subject to many constraints, such as small memory and less processing power. In this thesis, we will look at the feasibility of running WebAssembly on an ESP32 Microcontroller. WebAssembly is a newly developed bytecode meant to serve as a compilation target that can be used in any browser to execute optimized code at near-native speeds. Recently the interest around running WebAssmbly on embedded devices has picked up, and we want to evaluate how WebAssembly can be run on the ESP32 microcontroller. For this, we found the WASM3 runtime that can interpret and execute WebAssembly on the ESP32 and performs better than all other currently known WebAssembly Interpreters. To test the execution, we designed a collection of test workloads inspired by requirements that programs on a Microcontroller might have. We ran them while measuring the execution time of native code compared to WebAssembly code. Our tests show that the execution times increase by up to 90x when interpreting the code as WebAssmbly. While the lower performance and limited support for system interaction pose severe drawbacks to using WebAssembly, there are also significant advantages. New languages that support WebAssembly as a compilation target can be used without being explicitly supported by the platform and modules can be dynamically loaded over the air and executed on the microcontroller without the need for flashing the system.

# Contents

# 1 Introduction

WebAssembly is enabling new experiences on the web and could become a widely used universal bytecode outside of browsers as well. Since its introduction, WebAssembly has enabled many new experiences on the web. In 2019 the first runtimes meant to be used on embedded devices where published. In this thesis, we want to evaluate the concept of running WebAssembly on the ESP32 microcontroller and see what possible drawbacks it has.

**Microcontrollers** Meant for executing specific tasks, microcontrollers are small computers with minimal resources. They are designed with the aim to have just enough resources while keeping costs low. A popular system on a chip in this class is the ESP32 family. They are very affordable and can be used from experimentation and prototyping to production products. Their connectivity options and CPU performance makes them a great fit for the internet of things devices. We will focus our testing on running WebAssembly on the ESP32 in this thesis.

**WebAssembly** Since its beginning with static pages, the web has evolved to become a universal platform for applications, available on many different devices. However, even though browser engines have made significant progress at optimizing JavaScript, the only natively supported language on the web, there are still performance inconsistencies. To solve this problem, WebAssembly was created. It is a new, low-level bytecode format that allows running optimized code on browsers at near-native speeds. Being adopted by all major browser vendors, it is now almost universally available.

But since WebAssembly has not explicit dependencies on the web platform, its attributes such as portability, safety, and speed, making it very useful outside of the browser too. Runtimes meant to be used on embedded devices have been becoming available recently and might open exciting new angles of programming a microcontroller.

## Assessing WebAssembly

In order to assess the current state of WebAssembly on the ESP32, we found a runtime, WASM3, which has support for the ESP32 running FreeRTOS. While other runtimes are available already, most of them only target desktop PCs. The only other runtime for embedded use, the WebAssembly micro runtime, does not support the ESP32 operating system. WASM3 also achieves the best execution speeds amongst WebAssembly runtimes in benchmarks.

The comparison we are interested in is between the execution of code compiled to WebAssembly and compiled to native code. For this, we designed a collection of Workloads that are inspired by real-world applications. We ran those tests as WebAssembly and native code and measured the different behavior to gain more insight into the drawbacks and advantages of running WebAssembly.

Being a new development, there is not much previous work on running WebAssembly on embedded devices. Most of the research is currently focused on the applications inside the browser. We are interested in this new format that can provide value to the internet of things devices using the ESP32 and what the currently available ecosystem looks like.

# 2 Background

Since WebAssebly on embedded devices is a new topic that just recently surfaced, we will introduce some concepts that are important to follow the thesis. While both microcontrollers and WebAssebly are widely available, there are specific details about them that are essential to be aware of in order to assess the use of WebAssebly on the ESP32 microcontroller.

## 2.1 Microcontrollers

Since this thesis is focused on the use of WebAssembly on microcontrollers, we will shortly explain the concept and limitations that come with it. A microcontroller (**MCU** for microcontroller unit) is a small computer, meant to fulfill a particular requirement without a complex operating system. They are designed for embedded applications from implantable medical devices to toys and very prominently in IoT devices. Bigger devices will often have multiple microcontrollers, each responsible for a particular function. A car, for example, could include, amongst others, an MCU to control the mirror adjustments, one to handle fuel injection and another one for traction control.

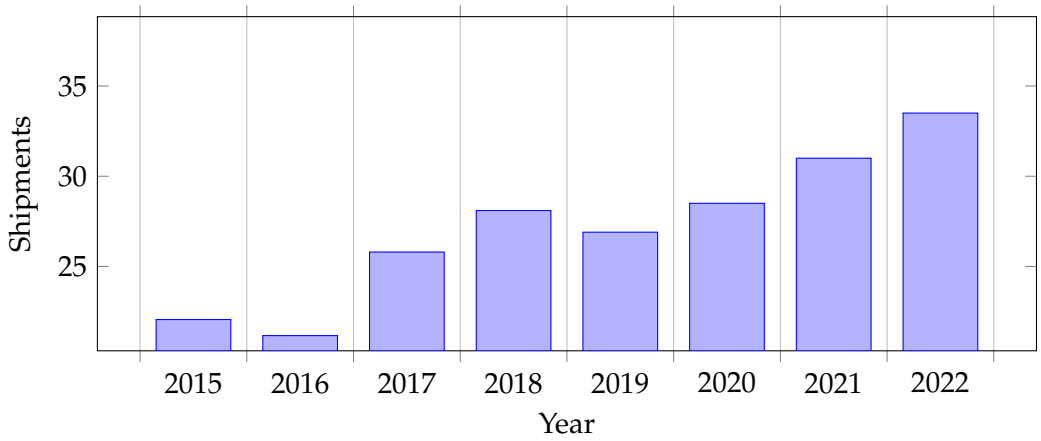



Figure 2.1: MCU shipments worldwide from 2015 to 2023 (in billions)

Core elements of an MCU are the processor, memory, and I/O peripherals. The Processor (CPU) can be thought of as the brain of the MCU. It performs basic arithmetic, logic, and I/O operations. Memory is where any data is stored; the processor needs to fulfill its tasks. Lastly, the I/O peripherals are the controller's connection to the outside world; they allow the receiving and sending of information, such as receiving a signal from a switch and turning on a light in response.

### 2.1.1 ESP32

For this thesis, we are not looking at big devices using multiple MCUs but instead at the ESP32 system on a chip (SoC) specifically. A very popular low-cost, low power series of microcontrollers with integrated WiFi and Bluetooth. Developed by the Shanghai-based company Espressif Systems this successor to the ESP8266 offers a great platform for IoT and embedded projects[9]. The ESP32 has the additional processing power and I/O options that make it a great platform for developing secure IoT devices. It gained popularity fast after being released in September of 2016.

The ESP32 systems family provides an excellent base for many IoT applications. There are multiple versions available from ones very well suited for hobbyists to ones used for industrial manufactures. With a low price point and area footprint, they still provide significant performance and many operational features[16].

To reach price and power consumption targets, the ESP32 has significant hardware limitations. This introduces some constraints when working with the platform, such as 4MB flash memory and 520KB RAM available. The Operating system, for example, can hardly be called that. Compared to popular operating systems like Windows and Linux, the used FreeRTOS is much more specialized. This will be further explained in the following passage.

### 2.1.2 FreeRTOS

Many MCUs are used in applications where throughput is less important than a guaranteed performance. This is why the ESP32 uses FreeRTOS, a real-time operating system (RTOS) is specifically intended to be used in time-critical situations. A key characteristic of such an operating system is the predictable behavior of the scheduler, the part of the operating system that decides which task should be run by the CPU at any given time. Most schedulers allow the user to set priorities for tasks in order to decide which task should be run next.

FreeRTOS specifically is the leading RTOS amongst MCUs and is designed to be small enough to run on a microcontroller[10]. Since most applications in which MCUs are used do not warrant the use of a full RTOS, FreeRTOS only provides the core scheduling

functionality, timing, and synchronization primitives. It can, however, be extended by using addon components, for example, to make use of a specific networking stack. FreeRTOS also built a significant community and support for many platforms in its 15-year development.

In more recent history, Amazon has taken over stewardship of FreeRTOs and also offers their own extension a:FreeRTOS[15]. This version additionally comes with some direct integration into Amazons AWS service[11]. It is supposed to make the development of new IoT devices easier, especially when using Amazon's platform for the server-side processing; the core of FreeRTOS remains open source.

## 2.2 WebAssembly

Beginning with static HTML pages, the web has since developed into a universal application platform, accessible from many different devices running different operating systems. JavaScript is the only natively supported language on the web. However, even though it is universally used and made much progress in modern implementations, it still has some problems as a compilation target. WebAssembly addresses these issues and provides a compilation target for the web [7].

WebAssembly (WASM) was first announced in June 2015 [23] and reached a cross-browser consensus in March 2017 [24]. Its goal was to provide near-native performance for browser-based applications, which could only be written in JavaScript for a long time. Since being published in March 2017, it is currently usable for about 90% of global internet users. More recently, the interest picked up around usage outside of the browser, which is also the primary concern of this thesis.

In a study conducted on the use of WebAssebly in June of 2019, researchers looked at the top 1 million Alexa websites and found that WebAssebly was used in 1 of 600 websites [17]. The most common use-cases identified where malicious. Websites use it to obfuscate additional code and mine cryptocurrencies on the machines of visitors.

Being meant for the web, WASM has to achieve specific goals that give the platform new properties. It has to be safe since, on the web, code is loaded mainly from untrusted sources. It has to be fast as the primary motivation to introduce WebAssembly was to provide a compile target on the web with reliable performance. Other than the usual low-level code such as regular assembly, WebAssembly has to be portable and work in all the different circumstances the web is currently used. Lastly, because the code is transmitted over the network, it has to be as small as possible to reduce bandwidth and improve latency [19].

WASM is a bytecode format, designed to be a portable target for high-level languages like C

C++ or Rust. It is executed on a stack-based virtual machine on which it executes in near-native speed due to its low-level design. Still, it runs in a memory-safe environment inside the browser and is subject to the same security policies as JavaScript code would be. WebAssembly modules are loaded with the application and provide bindings to JavaScript that make them usable in the browser.

Together with the binary format of WebAssembly, there is a text format that defines a programming language with syntax and structure. Every WASM binary is a self-contained module with functions, globals, tables, imports, and exports. This concept provides both encapsulation and sandboxing since modules can only interact with their environment using imports, and the client can only access the specified exports. Inside the module, the code ins organized in functions that can call each other even recursively.

Other than most stack machines, WebAssebly provides structured control flow instead of arbitrary jumps [13]. Figure 2.2 shows the control flow syntax in WebAssebly, which shows that all block, loop, and if constructs must be terminated with the end instruction and have to be correctly nested. They bracket nested sequences of instructions referred to as blocks in WASM; each block has an implicit label, which is the target for branch instructions.

$$
\begin{array}{lll}
\mathit{instr} & ::= & \ldots \\
& | & \mathsf{nop} \\
& | & \mathsf{unreachable} \\
& | & \mathsf{block}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{end} \\
& | & \mathsf{loop}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{end} \\
& | & \mathsf{if}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{else}\ \mathit{instr}^*\ \mathsf{end} \\
& | & \mathsf{br}\ \mathit{labelidx} \\
& | & \mathsf{br\_if}\ \mathit{labelidx} \\
& | & \mathsf{br\_table}\ \mathit{vec}(\mathit{labelidx})\ \mathit{labelidx} \\
& | & \mathsf{return} \\
& | & \mathsf{call}\ \mathit{funcidx} \\
& | & \mathsf{call\_indirect}\ \mathit{typeidx}
\end{array}
\tag{2.1}
$$

Figure 2.2: WebAssembly control flow syntax

Since they are used later on, we want to explain the `br_if`, `return` and `call` instructions. Both of them are branch instructions that will reference a specific block label. `br_if` performs a conditional branch and is used, for example, to execute if statements. `return` is a special unconditional branch that always branches to the outermost block.

The `call` instruction invokes another function by taking the necessary arguments from the stack and returning the result of the execution of the function.

The WebAssebly runtime keeps all the global state that can be manipulated by the module in the store. Besides the store, almost all instructions for an implicit stack. This stack contains values, labels, and call frames of active function calls. Thus, if a function returns a result, it will be the first entry on the stack after the execution has finished [20].

### 2.2.1 WebAssembly for IoT

While WASM is developed for the web, it carefully avoids any dependencies on the web. It is meant to be an open standard that can be embedded in a variety of ways [18]. The goals mentioned above, which WebAssembly achieves, make it an exciting format to explore on embedded devices. Due to its aim to be universal, it would allow the use of languages on MCUs that were not previously supported, and since it is already meant to be transmitted over the network, also over the air updates of code running on the controller are possible. To achieve portability, the source level interface libraries would have to map the host environments' capabilities either at build time or runtime.

While WebAssebly in an IoT context is an up-and-coming concept, it is also a very recent development. The best support for WASM right now is in the browser, of course, but out of browser runtimes keep surfacing, implemented in various languages, and providing an exciting execution environment [1]. Runtimes meant to be used on MCUs are much rarer and not as mature yet. Given the significant interest in the idea, though, and the working groups' avoidance of web dependencies, it can be assumed that this situation will change in the future.

### 2.2.2 Interpreters

With our specific use-case in mind, the WASM3 [27] engine was chosen since its specific goal is to run WebAssembly on MCUs. Other than many other engines, it does not follow a just in time compilation pattern, though, but instead acts as an interpreter. Thus we will shortly introduce the concept and advantages.

Interpreters are computer programs that execute a program. They pose a different concept to compiled execution, where a program would be translated to machine code before being run directly on the CPU. While offering multiple advantages, the main drawback is the execution speed compared to native code execution, which is often slower by order of magnitude and sometimes more. The overhead is generated by the interpreter having to analyze the program code before it can be executed.

Interpreters thus offer benefits in development speed since the code does not have

to be recompiled to run and in portability because the same code could be run on multiple platform-specific interpreters without the need to compile it into the native machine code of multiple platforms. For our use-case, the interpreter executes WASM instructions, allowing the dynamic loading of modules and running them in the chosen environment.

## 2.3 Microbenchmarking

Benchmarking is any form of measurement to qualify the behavior of a system. The most obvious examples would be measuring performance, energy, or memory consumption, but also reliability and temperature stability could contribute to a benchmark. Building useful benchmarks is hard because a program has to be created that yields repeatable and consistent results. A notable effort in the world of microcontroller benchmarking is the EEMBC [8].

The embedded microprocessor benchmark consortium is an industry association that has been designing benchmarks for over 20 years. The consortium offers multiple benchmarks, all meant to cover specific use-cases of embedded controllers from ultra-low-power IoT applications, over processor performance measurements to a recent benchmark designed to assess machine learning performance [3].

## 2.4 Summary

Microcontrollers are tiny, and restricted computers meant to perform a specific task. They are part of our everyday life and are used in a great variety of applications. The ESP32 system on a chip is a prevalent family of Microcontrollers that are very well suited for use in the Internet of Things devices. They are affordable but still offer WiFi and Bluetooth connectivity.

WebAssebly is a new bytecode that was designed for browsers, with the aim of allowing developers to run optimized low-level code at near-native speeds. It is supported by all major browsers and in active development. However, people are not only interested in WebAssebly on the web itself but also recognize it is potential as a new universal and portable bytecode outside of the browser.

# 3 Methodology

The goal of this thesis is to evaluate the use of WebAssembly for programming an ESP 32 Microcontroller. After we have introduced the concepts around MCUs and WebAssembly, we would like to explain our approach of measuring the impact of running WebAssembly on MCUs.

## 3.1 Running WebAssembly

First, as explained earlier, WebAssembly always needs a runtime, which is usually provided by the browser. In this case, we do not need all the features a browser would provide, just a way to execute WASM. Since the momentum around running WASM outside of a browser environment is picking up recently, more and more runtimes become available. A big push for WebAssembly on new platforms came in November of 2019 in the form of the Bytecode Alliance [4]. An open-source community dedicated to creating the foundations needed to run WASM on multiple platforms in a secure way.

The Bytecode Alliance maintains a couple of different compilers and runtimes for WebAssembly. This project includes the WebAssembly Micro Runtime (WAMR), which is an interpreter based runtime, specifically meant to run on embedded devices such as the ESP32 [5]. While looking at this project, we noticed that the OS used by the ESP32 (FreeRTOS) was not yet supported, with no current plans to change that.

While a couple of other WebAssembly runtimes are available, [1] WAMR used to be the only one capable of running on an embedded device. In late 2019 the second runtime for embedded devices was released in WASM3 [27]. This runtime is the first one we know of to support the ESP32 and FreeRTOS. It also performs significantly better than WAMR in benchmarks [21]. Thus we decided that using WASM3 was the way to go about running WASM on the ESP32.

According to the developers' measurements [21], WASM3 is currently the fastest available WebAssembly interpreter, being about 4x slower than current just in time compiling runtimes and about 12x slower than native execution. Because of the strict constraints that embedded devices have, WASM3 uses an interpreter model, which is more memory efficient and provides better startup times than JIT compilation. It also makes portability and security much more comfortable to achieve and maintain.

The speed of WASM3 is impressive, considering that even in browsers, a performance loss of up to 3x can be experienced when comparing WebAssembly to native code execution [14]. For our tests, it is crucial to keep in mind that the ESP32 is not a reference platform due to its limited capacities and features.

While the most basic interpreter can be thought of as a loop around a big switch statement that matches all possible instructions in the interpreted code, WASM3 follows a model dubbed M3 [22]. In WASM3, the bytecode first gets compiled into operations for the meta machine, which is traversed by one operation calling the next, which relies on tail-call optimization by the compiler. This leads to an efficient and elegant execution model for their virtual machine.

## 3.2 Comparing the platform

After finding a way to run WebAssembly on the ESP32 and verifying it with basic tests, we had to find a way to measure how well that works. In order to compare different platforms, a popular tool is benchmarking, in which the same workload is run on multiple platforms to generate values to compare those platforms.

While most benchmarks are meant to provide a comparison of two hardware platforms, in this specific case, we are not interested in the performance of the platform. Instead, we are interested in the performance of different execution models on the same platform, being the default native execution of code explicitly compiled for the ESP32 and the interpreted execution of the WASM code that could run anywhere.

Our desired comparison makes the test setup quite simple, the basic idea is to run the same code on the ESP32 twice, but once compiled to WASM. This approach has worked for the most part, with small detours being made when testing the import of outside functions into WebAssembly. All tests consist of a `run` method, which is once called from the main file and once loaded into the engine and run as a WASM function.

In line with how benchmarks work, we set out to design a couple of workloads we could run in both the native and the WebAssembly environments and compare the way they execute. In order to generate meaningful results, we tried to find simple tests that are not too far from what an MCU would execute. Keeping the tests limited and simple also allows us to look at the WASM output and understand the exact instructions in some instances.

### 3.2.1 Specialized Workloads

The first and most basic test is recursively calculating a Fibonacci number, while extensive recursive calls are not a part of most applications, functions calls in general

are. This also shows some WebAssembly specific features since it doe not only have jumps available but allows functions and function calls in the assembler code.

Secondly, switch and if statements are an integral part of any application. So comparing the performance of a switch statement is another indicator of how good applications would perform.

Of course, every application needs memory access, so we decided to implement two memory tests—one using direct access and one using random access to see if there is any impact on performance. Combining memory access and calculation, we also ran a matrix multiplication. This algorithm is the basis for many more complex algorithms, and the tight inner loop offers itself for optimization on the hardware and during compilation.

Of course, it is essential for embedded devices also to have hardware access. Currently, WebAssembly does not have a model of specific hardware features, network stack, or even CPU cores. All this functionality is assumed to be in the browser environment. The runtime we selected offers a mechanic to link external functions that can then be called from the WASM code, so we designed tests to see if outside calls came with a significant overhead that would impact applications using them.

Lastly, we implemented the Fibonacci test again but in AssemblyScript instead of C++. AssemblyScript [2] compiles a subset of TypeScript to WebAssembly. This is exciting for developers with a background in web development, as TypeScript will probably be already familiar to them. We implemented this test to show the new options using WebAssembly opens on the ESP32, which does not natively support a way to run TypeScript. TypeScript itself is an extension of JavaScript that supports strict types.

## 3.3 Running tests

All tests were run on the ESP-WROOM-32 using the ESP-devkit provided by espressif. The same C++ code was compiled to WebAssembly and also imported into the test program to allow for native execution. Then the test code was run multiple times to generate statistically significant results. The results from these tests will be explained in more detail in a specific section for each test.

### 3.3.1 Testing setup

All tests share a very similar main program to execute and time the tests, which we would like to explain now.

Listing 3.1: Main testing method

```
1  extern "C" void app_main(void) {
2      // Variable initialization
3
4      setup_wasm();
5
6      for (long long &wasm_time : wasm_times) {
7          int64_t start_time = esp_timer_get_time();
8          for (int j = 0; j < 10; ++j) {
9              run_wasm("20");
10         }
11         int64_t end_time = esp_timer_get_time();
12         wasm_time = (end_time - start_time) / 10;
13     }
14
15     for (long long &native_time : native_times) {
16         int64_t start_time = esp_timer_get_time();
17         for (int j = 0; j < 10; ++j) {
18             long value = run(20);
19         }
20         int64_t end_time = esp_timer_get_time();
21         native_time = (end_time - start_time) / 10;
22     }
23
24     printf("|Run|WASM|NATIVE|\n|---|---|---|\n");
25     for (int i = 0; i < sizeof(wasm_times) / sizeof(wasm_times[0]); ++i) {
26         printf("|%d|%lld|%lld|\n", i + 1, wasm_times[i], native_times[i]);
27     }
28     sleep(100);
29     printf("Restarting...\n\n\n");
30     esp_restart();
31 }
```

The primary testing method in listing 3.1 starts with setting up the WebAssembly runtime, which will be further explained with listing 3.2. This process is timed to see how much overhead the runtime initialization introduces. Next, there are two arrays set up to hold the test results. Then the test is run for the WASM with the function described in listing 3.3, by taking the average time over ten runs and saving that into the previously declared array. Following, the same is done for the native version of the test code. Lastly, the results are printed to the console, and the controller is eventually

restarted to rerun the tests.

It is important to note that with this setup, the runtime will be loaded in memory during both tests. Also, the native code is shipped together with the WASM test code. We do not expect interference due to the minimal footprint of both test cases.

Listing 3.2: Runtime setup

```
1  IM3Environment env;
2  IM3Runtime runtime;
3  IM3Module module;
4  IM3Function f;
5
6  static void setup_wasm() {
7          M3Result result = m3Err_none;
8
9          auto *wasm = (uint8_t *) wasm_test_cpp_wasm;
10         uint32_t fsize = wasm_test_cpp_wasm_len - 1;
11
12         env = m3_NewEnvironment(); // Error output omitted
13         runtime = m3_NewRuntime(env, 2048, NULL);
14         result = m3_ParseModule(env, &module, wasm, fsize);
15         result = m3_LoadModule(runtime, module);
16         result = LinkThesis(runtime);
17         result = m3_FindFunction(&f, runtime, "run");
18     }
19 }
```

Setting up the runtime is pretty straight forward, initially, the WASM module is imported from a header file, together with its length. Then the environment and runtime are created, followed by parsing the module. If the runtime has to provide functions that the WASM module relies upon, they are linked after loading the module. An example can be found in section 3.3.6 in which lines 24 and 25 of listing 3.2 are not commented out. Once the runtime is fully set up, the function itself is searched for. In our case, the function's name is always "run".

Listing 3.3: WASM execution

```
1  static void run_wasm(const char *input1) {
2      M3Result result = m3Err_none;
3
4      const char *i_argv[3] = {input1, NULL};
5      result = m3_CallWithArgs(f, 1, i_argv);
```

```
6
7       if (result) FATAL("m3_CallWithArgs: %s", result);
8
9       long value = *(uint64_t *) (runtime->stack);
10  }
```

The execution of the WASM function is a matter of calling the previously found function with the runtimes `m3_CallWithArgs()` method and supplying it with the input arguments. The return value of the operation can be found on the virtual machines stack afterward.

### 3.3.2 Testing matrix multiplication

To test performance during matrix multiplication, we use the code of listing 3.4. The test function takes one argument, the matrix size. It then creates two n×n matrices and multiplies them. In the end, it returns one value of the resulting matrix. This is to prevent the compiler from optimizing the actual calculation during compilation.

Listing 3.4: Matrix multiply test

```
1   uint32_t run(uint32_t n) {
2       uint32_t a[n][n], b[n][n], mul[n][n];
3
4       for (uint32_t i = 0; i < sizeof(a) / sizeof(a[0]); ++i) {
5           for (uint32_t j = 0; j < sizeof(a[0]) / sizeof(a[0][0]); ++j) {
6               a[i][j]=i+1;
7               b[i][j]=i+2;
8           }
9       }
10
11      for (uint32_t i = 0; i < sizeof(a) / sizeof(a[0]); ++i) {
12          for (uint32_t j = 0; j < sizeof(a[0]) / sizeof(a[0][0]); ++j) {
13              mul[i][j] = 0;
14              for (uint32_t k = 0; k < sizeof(a[0]) / sizeof(a[0][0]); ++k) {
15                  mul[i][j] += a[i][k] * b[k][j];
16              }
17          }
18      }
19      return mul[n-1][n-1];
20  }
```

Running this function in the main program described in section 3.3.1 results in the Following measurements for the average execution times of 100 runs.

### 3.3.3 Testing memory performance

Of course, memory performance is an important aspect of any computing platform, so we designed a test to compare it between native and WASM execution.

Listing 3.5: Linear memory test

```
uint32_t run(uint32_t n) {
    uint32_t array1[n];

    for (int i = 0; i < n; ++i) {
        array1[i] = i+1;
    }
    array1[n-1] = 0;
    uint32_t nextStep = 1;
    while(nextStep){
        nextStep = array1[nextStep];
    }
    return nextStep;
}
```

The setup is fairly simple; an array is created and filled with the indices of the respective following elements, imitating a linked list. Then it is read from, starting at index one and saving whatever index was found there into a variable that defines the next index to be read until finally, the next index ends up being 0. Lastly, the next index is returned to prevent compiler optimization.

### 3.3.4 Testing recursive calls

The test of recursive calls is run by calculating Fibonacci numbers with the following code.

Listing 3.6: Recursive calling test

```
uint32_t run(uint32_t n) {
    if (n < 2) {
        return n;
    }
    return run(n - 1) + run(n - 2);
}
```

Listing 3.7: WASM code excerpt

```
1  (module
2    (type $t1 (func (param i32) (result i32)))
3    (func $run (export "run") (type $t1) (param $p0 i32) (result i32)
4      (block $B0
5        (br_if $B0
6          (i32.lt_u
7            (local.get $p0)
8            (i32.const 2)))
9        (return
10         (i32.add
11           (call $run
12             (i32.add
13               (local.get $p0)
14               (i32.const -1)))
15           (call $run
16             (i32.add
17               (local.get $p0)
18               (i32.const -2))))))
19     (local.get $p0)))
```

Listing 3.7 shows the WebAssembly text format generated for the Fibonacci function in listing 3.6 and we would like to take a closer look at what the WASM module looks like for this specific example. After the module opening, the type of our `uint32_t run(uint32_t n)` function is defined and reused in the function definition in line 3, in this line the functions input and return types are also defined. The input is assigned to the $p0 variable for later use. In line 4 the block `$B0` is started, it contains the main function body. In line 5, we can see a `br_if` instruction; this is a conditional break that breaks the execution of the passed block if the condition is true. The condition, in this case, is the rest of the instructions included in the parentheses. Namely the comparison of the accepted parameter with 2 to see if it is smaller if that is the case the remaining block is skipped and code execution would continue in line 19 where the parameter is pushed on the stack, as the topmost value of the stack after the execution is the return value of a WASM function. Alternatively, the execution could continue in line 9 with the `return` instruction, which executes the instructions inside the parentheses and prevents any further code execution after that, mirroring the common return instruction in C. the Value return is the result of the two recursive calls of line 5 in the listing 3.6. The `run` function is called again by using the `call` instruction in lines 11 and 15.

It is important to note that this text format is not strictly WebAssembly, but one version to make it readable for humans. To make it more similar to the look of common programming languages for this listing, code folding was applied. To make the difference visible, listing 3.8 shows the WASM code without folding.

Listing 3.8: WASM code without folding

```
(func $run (export "run") (type $t1) (param $p0 i32) (result i32)
  block $B0
    local.get $p0
    i32.const 2
    i32.lt_u
    br_if $B0
    local.get $p0
    i32.const -1
    i32.add
    call $run
    local.get $p0
    i32.const -2
    i32.add
    call $run
    i32.add
    return
  end
  local.get $p0)
```

### 3.3.5 Testing switch statements

Listing 3.9: Switch statement test code

```
uint32_t run(uint32_t n) {
    uint32_t array1[20] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
    uint32_t result = 200;
    for (int i = 0; i < n; ++i) {
        uint32_t compare = i % 20;
        switch (compare) {
            case 0:
                result = array1[0];
                break;
            case 1:
```

```
11                 result = array1[1];
12                 break;
13             // Some cases omitted
14             case 18:
15                 result = array1[18];
16                 break;
17             case 19:
18                 result = array1[19];
19                 break;
20             default:
21                 result = 100;
22                 break;
23         }
24     }
25     return result;
26 }
```

As previously mentioned, switch statements are a widespread occurrence in software running on microcontrollers and thus a thing to be tested. The test method for this is a big switch statement that is looped over. To prevent optimization, the return value is updated from an array and returned by the function.

### 3.3.6 Testing native calls

A vital function of the runtime is to expose outside functions to the WASM module and allow the interaction with other libraries from within the WASM code. For this, we designed two reasonably simple tests that call functions not defined in the WASM code.

Listing 3.10: Outside call test code

```
1 #include "test_api.h"
2
3 WASM_EXPORT
4 void run(uint32_t n) {
5     mark();
6 }
```

As is obvious from listing 3.10, the test code just calls the outside `mark()` function. There is also an import of the test API header in which the external function is defined to make the test code compile.

Listing 3.11: Test api defintiion

```
1 #ifndef WASM3_TEST_API_H
```

```
2  #define WASM3_TEST_API_H
3
4  #include <stdint.h>
5
6  #define WASM_EXPORT extern "C" __attribute__((used)) __attribute__((visibility ("default")
7  #define WASM_EXPORT_AS(NAME) WASM_EXPORT __attribute__((export_name(NAME)))
8  #define WASM_IMPORT(MODULE, NAME) __attribute__((import_module(MODULE))) __attribute__((im
9
10 extern "C" {
11
12 WASM_IMPORT("thesis", "sendValue") uint32_t sendValue (void);
13 WASM_IMPORT("thesis", "mark") void mark (void);
14
15 }
16
17 #endif //WASM3_TEST_API_H
```

The resulting WASM code does not include the `mark` method, but instead imports it from the `thesis` module that is expected to be available at runtime.

Listing 3.12: WASM code for the outside call

```
1 (module
2   (type $t0 (func))
3   (type $t1 (func (param i32)))
4   (import "thesis" "mark" (func $thesis.mark (type $t0)))
5   (func $run (export "run") (type $t1) (param $p0 i32)
6     (call $thesis.mark)))
```

As displayed in listing 3.12 line 4 the mark function from the thesis module is imported as defined in listing 3.11. The run function then just calls the imported function in line 6.

In order to provide this imported function at runtime, the setup for our tests has to be changed slightly; namely, it has to be linked during the runtime setup in listing 3.2.

Listing 3.13: Function linking

```
1 int64_t native_timestamp;
2
3 m3ApiRawFunction(m3_thesis_mark) {
4     native_timestamp = esp_timer_get_time();
5     m3ApiSuccess();
6 }
```

```
7
8  M3Result LinkThesis(IM3Runtime runtime) {
9      IM3Module module = runtime->modules;
10     const char *thesis = "thesis";
11
12     m3_LinkRawFunction(module, thesis, "mark", "i()", &m3_thesis_mark);
13     return m3Err_none;
14 }
15
16 void mark() {
17     native_timestamp = esp_timer_get_time();
18 }
```

In listing 3.13 line 1 we introduce a variable to hold a timestamp after the `mark` function was called. In line 3, we define the function, which just saves the current timestamp and ends with success. This is then linked into the runtime in line 12. To compare native execution this time, we can not call the exact same function since it was not compiled to WASM at all, so we implement a similar function in line 16 that is called during the test of native execution.

### 3.3.7 Running TypeScript

In listing 3.14 the same function as for 3.3.4 is implemented but in TypeScript instead of C++. This is to explore the new options opened by WebAssembly on the ESP32. The test is then run the same way the other tests work. We are once again comparing the execution performance with the native C++ code.

Listing 3.14: TypeScript testing code

```
1 export function run(n: i32): i32 {
2   if(n < 2 ) {
3     return n;
4   }
5   return run(n - 1) + run(n - 2);
6 }
```

The generated WebAssembly code is almost identical as can be seen in listing 3.15. In line 4 however `if` block is used instead of a conditional branch. That block is closed with the `then` statement in line 8. Another difference is the use of the `sub` instruction instead of adding a negative number in line 13. In general though, the genrated WASM code is almost identical.

Listing 3.15: WASM code excerpt

```
(module
  (type $t0 (func (param i32) (result i32)))
  (func $run (export "run") (type $t0) (param $p0 i32) (result i32)
    (if $I0
      (i32.lt_s
        (local.get $p0)
        (i32.const 2))
      (then
        (return
          (local.get $p0))))
    (i32.add
      (call $run
        (i32.sub
          (local.get $p0)
          (i32.const 1)))
      (call $run
        (i32.sub
          (local.get $p0)
          (i32.const 2))))))
```

## 3.4 Summary

In order to assess WebAssembly on the ESP32, we used a new Runtime from late 2019, WASM3. It is only the second runtime meant to run on embedded devices and the first one to date that supports FreeRTOS. It is an interpreter for WebAssembly that follows a meta machine pattern and achieves excellent speeds compared to other WebAssembly runtimes currently available.

Secondly, we designed a collection of workloads that can be used to compare the WebAssembly execution to the native code execution. We modeled the tests after requirements, which applications on an MCU will have. These include function calls, memory access, computation, and the use of outside functions to interact with peripherals, for example. Additionally, we tested the ability of WebAssembly to be a target for new languages, not natively available on the controller, and wrote one test in TypeScript.

# 4 Evaluation

## 4.1 Running Benchmarks

When implementing the workloads for our benchmarks, we noticed that using WebAssembly as of now imposes many constraints. Since we are not targeting a browser platform, which offers a wide variety of APIs meant to handle system calls, this means that code targeting WASM on the ESP32 can not make use of APIs available in the browser environment such as memory allocation. This is where WASI comes into play, as mentioned in 2.2, WASI aims to solve the problem of interfacing with the system from WASM code by providing a standardized interface. Once WASI is supported by the runtime, writing code for it will be much more straightforward.

In order to run the test, we had to compile the C++ test code into WASM bytecode. An established tool for this is the emscripten compiler. After using this for the initial tests, though, we noticed problems, since it is meant for compiling WASM that will run in the browser. This leads to modules that expect all the available functions of the browser environment since we are running the tests on an embedded device; those functions are not available to us.

Because emscripten is meant for use in browsers after compilation, we switched to the `wasmc++` compiler, which is part of wasienv, a toolchain for compiling C into WebAssembly [26]. This project provides a couple of utilities to compile C code into WASM modules. `wasmc++` wraps clang++ with the correct configuration for WebAssembly already applied and makes compiling very easy. For our tests, the code was compiled by running `wasmc++ -Os -Wl,--strip-all -nostdlib wasm/test.cpp -o wasm/test.cpp.wasm`.

Lastly, the WASM3 runtime expects the bytecode to be loaded into an array. In Linux, the `xxd` utility is exactly what is needed to achieve that. After compilation, the WASM code can be converted into a C++ header file by running `xxd -i wasm/test.cpp.wasm > main/test.wasm.h`. After these steps, we are able to run our tests just as planned.

After designing tests as described in 3.3.1, we ran all of them on the ESP32 to report times. In general, the tests showed a significant slowdown in execution speed when running the workloads in a WebAssembly context compared to running them compiled natively. For all tests, we could observe the variance of the measured time is very low;

this is to be expected since the tests ran without any other load on the MCU and the deterministic nature of FreeRTOS, as mentioned in section 2.1.2.

It is important to mention when looking at the result of these tests that WASM3 heavily relies on tail call elimination, which currently is not performed by the ESP32 compiler. This leads to excessive use of the native stack and lower performance. Collaborators and authors of WASM3 are currently exploring solutions that would make the runtime faster and more efficient on the platform soon [12].

### 4.1.1 Recursive calls

Calling functions is an integral capability of any application, so this is the first test to compare WASM with the native execution of our code. We are using the test described in 3.3.4. This test has almost no instructions but produces many function calls that could be troubling to handle for the runtime.

Even though most applications on an MCU might not run recursive calculations, this test does show the cost of calling many functions. Table 4.1 shows some of the measured times. It is apparent from the numbers that running the code in the interpreter takes about 41x longer than the native execution.

As seen in the WASM code of listing 3.7, it is straightforward but requires the runtime to manage the execution of the same functions many times over. Compared to some of the following tests, 41x is not a very high slowdown.

To see if both the runtime and native execution behave similarly for different inputs, we ran the test multiple times. Figure 4.1 shows the change in execution time for rising input. As expected, the execution time grows exponentially, but the runtime and native execution maintain their 41x difference in speed.

| Run | WASM Execution | Native Exectuion |
|---|---|---|
| 10 | 41766 | 1000 |
| 11 | 41767 | 1000 |
| 12 | 41766 | 1000 |
| 13 | 41767 | 1000 |
| 14 | 41766 | 1000 |
| 15 | 41767 | 1000 |
| 16 | 41766 | 1000 |
| 17 | 41766 | 1000 |

Table 4.1: Exerpt of the measured times for recursive calls

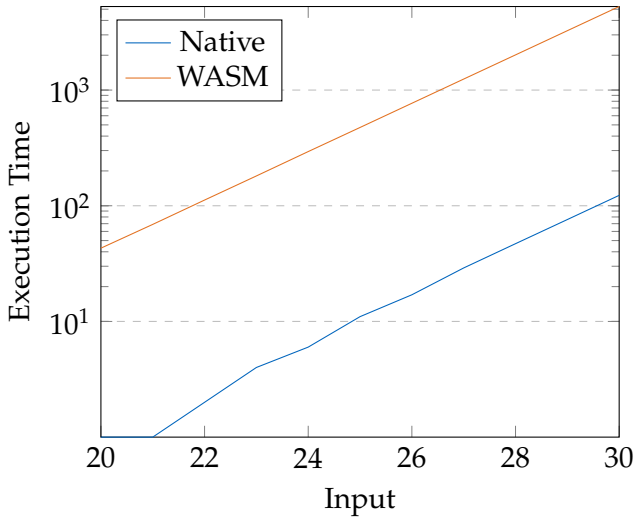



Figure 4.1: Recursive call times for different inputs

### 4.1.2 Swicth statements

Next, we tested the performance of switch statements in the runtime. The times listed in Table 4.2 lead to the conclusion, that the native compiler optimized our switch statement very well, effectively skipping over what we tried to test. Otherwise, the native instruction should take a while longer. Since the WASM code never executed more than 100x slower than native code, it would be reasonable to expect a result in the same range for switch statements.

Since there are no values to compare this statement by, we cannot pass judgment on how much of a performance hit switch statements take from being executed in the runtime. It is fair to assume a similar slowdown to the other tests occurs here.

| Run | WASM Execution | Native Exectuion |
|---|---|---|
| 24 | 1021 | 0 |
| 25 | 1021 | 0 |
| 26 | 1021 | 0 |
| 27 | 1021 | 0 |
| 28 | 1021 | 0 |
| 29 | 1021 | 0 |
| 30 | 1021 | 0 |
| 31 | 1021 | 0 |

Table 4.2: Exerpt of the measured times for switch calls

### 4.1.3 Memory access

Of course, every application requires memory access, so we ran tests that perform linear reads on the memory. The runtime caused a longer execution time of around 33x, as can be seen in the measured times in table 4.3. Since WASM is always run in a virtual machine, memory access optimization is not achieved in the source code but instead taken care of by the runtime. In a browser, for example, the WASM memory is a JavaScript ArrayBuffer.

We were not able to directly compare linear memory access with random memory access. However, when running the test meant to assess random access, we saw the performance loss increase to about 73x. This could be caused by several things, such as the generation of pseudo-random numbers in the test code. We can not make a precise determination if random access has any impact on the performance of the WASM code.

| Run | WASM Execution | Native Exectuion |
|---|---|---|
| 74 | 1802 | 56 |
| 75 | 1801 | 55 |
| 76 | 1802 | 55 |
| 77 | 1802 | 55 |
| 78 | 1802 | 55 |
| 79 | 1802 | 55 |
| 80 | 1801 | 55 |
| 81 | 1802 | 55 |

Table 4.3: Exerpt of the measured times for memory access

### 4.1.4 Matrix multiplication

The matrix multiplication test combines both memory access and calculations and experiences the most significant increase of execution time in any of our tests. As the numbers in table 4.4 show, the interpreted code runs more than 90x slower than the natively executed code.

This test does show that more extended calculations and more sophisticated algorithms will take a significant performance hit when being run as WebAssembly. Even in contact with the contributors of the runtime, we were not able to identify specific tasks that are very expensive in the interpreted environment. Nevertheless, as a general rule, it is not advisable to implement large complex workloads in WebAssembly at this point.

| Run | WASM Execution | Native Exectuion |
|---|---|---|
| 55 | 26277 | 281 |
| 56 | 26277 | 282 |
| 57 | 26277 | 281 |
| 58 | 26277 | 281 |
| 59 | 26277 | 282 |
| 60 | 26277 | 281 |
| 61 | 26277 | 281 |
| 62 | 26277 | 281 |

Table 4.4: Exerpt of the measured times for matrix multiplication

### 4.1.5 Calling of native code

Of course, not all functionality can be included in the WASM module; mainly, native platform features will have to make use of functions only available outside of the module. When measuring the performance of calls outside the test-module, we are taking the total time of ten runs. The timings in table 4.5 show that the overhead introduced by the runtime for such a simple test case is relatively small. The interpreted code took less than 10x longer than the native calls.

In a second test, which also returned a value to the WASM module, we did not see an additional increase in execution time, making this the best performing test in our experiments.

| Run | WASM Execution | Native Exectuion |
|---|---|---|
| 15 | 33 | 7 |
| 16 | 33 | 8 |
| 17 | 33 | 7 |
| 18 | 32 | 8 |
| 19 | 33 | 8 |
| 20 | 32 | 8 |
| 21 | 33 | 8 |
| 22 | 32 | 7 |
| 23 | 33 | 8 |
| 24 | 32 | 7 |
| 25 | 33 | 8 |

Table 4.5: Exerpt of the measured times for external calls

### 4.1.6 Typescript execution

Closely related to 4.1.1 is the test case running TypeScript since it is the same test function and almost the same WASM code being tested. The observed results in table 4.6 are very similar to the times measured when executing WebAssembly code generated from C++. Nonetheless, this test shows the exciting potential of running languages on the ESP32 that are not natively supported. Also, it is interesting to note that, at least for this example, there is no performance loss by using TypeScript instead of C++.

Of course, the compilation of TypeScript to WebAssembly requires a different compiler than `wasmc++`. We used the `asc` compiler that is part of the AssemblyScript toolchain and compiled the TypeScript code by running `asc index.ts -b test.wasm --validate -O3z --runtime none --noAssert`. After compilation, we followed the same steps as with the other tests to run the comparison.

| Run | WASM Execution | Native Exectuion |
|---|---|---|
| 37 | 41493 | 1000 |
| 38 | 41493 | 1000 |
| 39 | 41493 | 1000 |
| 40 | 41493 | 1000 |
| 41 | 41493 | 1000 |
| 42 | 41493 | 1000 |
| 43 | 41493 | 1000 |
| 44 | 41493 | 1000 |
| 45 | 41493 | 1000 |
| 46 | 41494 | 1000 |

Table 4.6: Exerpt of the measured times for TypeScript execution

## 4.2 Learnings

### 4.2.1 Drawbacks of WASM execution

The very obvious drawback of executing WebAssembly on a microcontroller is the performance loss that comes with it. We have shown this with all our tests, and even though the slowdown varies from test to test, code compiled to WASM and executed by an interpreter will run an order of magnitude slower than the same code directly compiled into the main program and executed on the MCUs CPU.

This puts WebAssembly into an unexpected position on embedded devices since it is meant initially to provide better performance. In browsers, WebAssembly made it possible to run adobe lightroom on the web, allows Facebook to compress images before uploading them and Wikipedia to play videos the users' browser does not support [25]. However, even though it reaches near-native speeds on browsers, it performs much worse than the current alternative on embedded devices.

It is fair to assume that this performance decrease causes the MCU to consume much more energy in order to make the same computation. For devices running on a battery, this could pose a challenge. Further experiments could be made to quantify the impact of running WASM and also compare it to other energy-consuming tasks such as network IO.

Apart from the impact on the running of program code on the MCU, we also noticed other limitations. The availability of certain expected functionality in WASM is minimal. For example, the use of `malloc()` is not possible if compiling code to WASM and running it in our runtime. In web browsers, this functionality is available for import from the environment and implemented such that it can be used as expected. WASM3, however, does not offer any functions for import, and implementing dynamic memory management would be a significant effort.

Since the problem of system access outside the browser is prevalent, a subgroup of the WASM community group is working on specifying a system interface for WASM[6]. The WebAssembly system interface (WASI) is meant to provide a foundation for developers to build upon when targeting non-browser platforms. Once specified, code compiled for WASI will run in any WASI-compliant runtime, truly enabling WASMs portability.

### 4.2.2 Potential of WebAssembly on embedded devices

As we demonstrated in our tests, the WebAssembly code is interpreted at runtime; this means that it could also be loaded from the network instead of being included in the code. Dynamically loading code and executing it allows the deployment of new behavior to an MCU without having to perform a flash but rather in the form of an over the air update.

Since WebAssembly support in the browser is excellent already, all the tests we developed can alternatively be run in the browser. This allows embedded developers to test their programs locally in development and be able to make sure everything works as expected even before deploying it to an MCU for the first time. For the ESP32, for example, a browser emulator could be built, which provides all the native functions expected on the platform in JavaScript.

Additionally, we showed in the last test that WebAssembly could open the doors for new languages that are not natively supported on the MCU. In our example, we were

able to program the ESP32 by using TypeScript, which usually compiles to JavaScript. However, in our example, we compiled it to WebAssembly and were able to run it on the ESP32. This enables developers without previous to get into embedded programming from their current field of work.

### 4.2.3 Usecase examples on the ESP32

For embedded use-cases that are CPU bound, WebAssembly could pose a big problem since the performance is much worse than current native execution. Making up for this with multiple devices could be a way of mitigating that problem, but since we have observed a performance loss of close to 100x, that seems like a costly way of overcoming this problem.

Should the most time-consuming things not be calculations and similar tasks, though, but waiting for a slow sensor read or a signal from the outside, the performance loss might not be as significant. Execution time also matters less if the task is performed periodically, and the device has much idle time.

A very promising use-case of WASM is the customization of device behavior. This example is often given and also supported by the results of our tests and relies on the interoperation of the native code with the loaded WASM module. An IoT device could, for example, have an extensive API and allow the user to deploy logic in the form of a WebAssembly module making use of all the API functions. Here, other features of WebAssembly also come into play, such as it is security guarantees, which make it safer to execute unknown code on the device.

## 4.3 Summary

To run the Benchmarks, we compiled the testing code to WebAssembly and set up a test function that can be run on the microcontroller. The test function initialized the runtime and loaded the WebAssembly code. Thereafter it runs both the WASM function and the native implementation 1000 times, taking the average runtime of ten runs. Every test aimed to run the same code once compiled to WebAssembly and once being part of the regular native compilation.

Our tests show a significant slowdown of up to 93x when running the code in WebAssembly, which leads to significant disadvantages when running CPU bound applications on the MCU. With applications that are not limited by the CPU performance but by a slow sensor readout or network interaction, for example, it would not pose as much of a problem. Also, we saw that the TypeScript code, compiled to WebAssembly, ran at a very similar performance to the C++ code compiled to WASM, showing that new languages can be used without additional performance decrease.

# 5 Conclusion

This thesis showed the current status of running WebAssembly on the ESP32 and evaluated its viability. We explored the ecosystem around WebAssembly on embedded devices and established the necessary background to show the significance of our findings. We ran a selection of tests using the current state of the art technology available on the ESP32.

## Process

To run WebAssembly on the ESP32, we found the WASM3 runtime. It performs excellent compared to other WASM runtimes currently available and can be used on MCUs. We were not able to find a second runtime that would be usable on the ESP32 at this point. To run the tests, we set up a test environment, including compiling the test code to WASM.

The test workloads we designed are modeled after tasks, which production application on an MCU would perform and show the behavior of the runtime in different circumstances. We tested function calls, memory access, matric multiplication, and the calling of outside functions. Thes might be needed to control the peripherals of the microcontroller. Additionally, we also ran one test that was written in TypeScript to see the experience of languages, not natively supported by the ESP32.

## Findings

Our tests showed that the interpreted execution of WebAssembly takes 40 - 90x longer than executing the same function natively compiled. Except for outside calls, where we noticed a slowdown lower than 4x. Also, programming for a WASM target is very limited right now since there is no unified system interface available that would allow the code to interact with the underlying OS.

This significant performance decrease makes it a bad fit for CPU bound applications. Mitigating the lost performance with more devices would be very expensive and require much coordination. We also suspect the longer execution times to make the MCU use

more energy than it does when running the same computation in native code, which could pose a problem to ultra low powered devices and should be researched further.

However, our tests also showed the potential of WebAssembly. The test that used TypeScript to implement the same testing function that was originally written in C++ ran at a similar performance. This shows the advantage of WebAssembly being universal and supported as a target for many languages, that are not typically seen in embedded programming.

Also, due to it is specific attributes, WebAssembly is well fitted for network transmission and dynamic execution. It also guarantees memory safety, which makes executing unknown code a much smaller risk. These properties allow network-connected devices to receive new instruction over the air and run them without a flash being required.

**Evaluation** WebAssembly on the ESP32 shows excellent potential for new ways of developing for embedded platforms. It opens systems to new languages and deployment strategies. Due to the performance decrease and missing system interface, it is certainly not a good fit for all applications right now. However, with the WebAssembly system interface being actively worked on and more embedded runtimes becoming available, we expect the attractiveness of WebAssembly to increase further. It stands to see if WebAssembly becomes the new universal bytecode, but we see powerful potential, even in the early stages.

# List of Figures

# List of Tables

# Bibliography


[1] S. Akinyemi. *appcypher/awesome-wasm-runtimes*. original-date: 2018-10-08T10:48:10Z. Mar. 5, 2020. URL: https://github.com/appcypher/awesome-wasm-runtimes (visited on 03/06/2020).

[2] *AssemblyScript/assemblyscript*. original-date: 2017-09-28T11:06:50Z. Mar. 13, 2020. URL: https://github.com/AssemblyScript/assemblyscript (visited on 03/13/2020).

[3] *Benchmark Product List - EEMBC - Embedded Microprocessor Benchmark Consortium*. Benchmark Product List - EEMBC - Embedded Microprocessor Benchmark Consortium. URL: https://www.eembc.org/products/ (visited on 03/13/2020).

[4] *Bytecode Alliance*. Bytecode Alliance. Library Catalog: bytecodealliance.org. URL: https://bytecodealliance.org/ (visited on 03/06/2020).

[5] *bytecodealliance/wasm-micro-runtime*. original-date: 2019-05-02T21:32:09Z. Mar. 6, 2020. URL: https://github.com/bytecodealliance/wasm-micro-runtime (visited on 03/06/2020).

[6] L. Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web – Mozilla Hacks - the Web developer blog*. Mozilla Hacks – the Web developer blog. Library Catalog: hacks.mozilla.org. Mar. 27, 2019. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface (visited on 03/04/2020).

[7] M. contributors. *WebAssembly*. MDN Web Docs. Library Catalog: developer.mozilla.org. URL: https://developer.mozilla.org/en-US/docs/WebAssembly (visited on 03/13/2020).

[8] *Embedded Microprocessor Benchmark Consortium*. Embedded Microprocessor Benchmark Consortium. URL: https://www.eembc.org/ (visited on 03/13/2020).

[9] *ESP32 Overview | Espressif Systems*. URL: https://www.espressif.com/en/products/hardware/esp32/overview (visited on 03/02/2020).

[10] *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. FreeRTOS. URL: https://www.freertos.org/ (visited on 03/02/2020).

[11] *FreeRTOS - Real-time operating system for microcontrollers - AWS*. Amazon Web Services, Inc. Library Catalog: aws.amazon.com. URL: https://aws.amazon.com/freertos/ (visited on 03/02/2020).

[12] I. Grokhotkov. *esp32-idf example fails when -DESP32 is defined · Issue #28 · wasm3/wasm3*. GitHub. Library Catalog: github.com. URL: https://github.com/wasm3/wasm3/issues/28 (visited on 03/12/2020).

[13] *Instructions — WebAssembly 1.0*. URL: https://webassembly.github.io/spec/core/syntax/instructions.html#syntax-instr-control (visited on 03/13/2020).

[14] A. Jangda, B. Powers, E. Berger, and A. Guha. "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code". In: *arXiv:1901.09056 [cs]* (May 31, 2019). arXiv: 1901.09056. URL: http://arxiv.org/abs/1901.09056 (visited on 03/13/2020).

[15] F. Lardinois. *Amazon FreeRTOS is a new operating system for microcontroller-based IoT devices*. TechCrunch. URL: http://social.techcrunch.com/2017/11/29/amazon-freertos-is-a-new-operating-system-for-microcontroller-based-iot-devices/ (visited on 03/02/2020).

[16] A. Maier, A. Sharp, and Y. Vagapov. "Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things". In: *2017 Internet Technologies and Applications (ITA)*. 2017 Internet Technologies and Applications (ITA). Wrexham: IEEE, Sept. 2017, pp. 143–148. ISBN: 978-1-5090-4815-1. DOI: 10.1109/ITECHA.2017.8101926. URL: http://ieeexplore.ieee.org/document/8101926/ (visited on 03/02/2020).

[17] M. Musch, C. Wressnegger, M. Johns, and K. Rieck. "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren. Cham: Springer International Publishing, 2019, pp. 23–42. ISBN: 978-3-030-22038-9.

[18] *Non-Web Embeddings - WebAssembly*. URL: https://webassembly.org/docs/non-web/ (visited on 03/13/2020).

[19] A. Rossberg, B. Titzer, A. Haas, D. Schuff, D. Gohmann, L. Wagner, A. Zakai, J. Bastien, and M. Holman. "Bringing the web up to speed with WebAssembly". In: *Communications of the ACM* 61 (Nov. 20, 2018), pp. 107–115. ISSN: 00010782. (Visited on 03/13/2020).

[20] *Runtime Structure — WebAssembly 1.0*. URL: https://webassembly.github.io/spec/core/exec/runtime.html#store (visited on 03/13/2020).

[21] V. Shymanskyy. *WASM3 Performance*. Library Catalog: github.com. Feb. 3, 2020. URL: https://github.com/wasm3/wasm3/blob/master/docs/Performance.md (visited on 03/06/2020).

[22] V. Shymanskyy. *wasm3/Interpreter*. GitHub. Library Catalog: github.com. Jan. 23, 2020. URL: https://github.com/wasm3/wasm3/blob/master/docs/Interpreter.md (visited on 03/13/2020).

[23] L. Wagner. *WebAssembly | Luke Wagner's Blog*. June 17, 2015. URL: https://blog.mozilla.org/luke/2015/06/17/webassembly/ (visited on 03/13/2020).

[24] L. Wagner. *WebAssembly consensus and end of Browser Preview from Luke Wagner on 2017-02-28 (public-webassembly@w3.org from February 2017)*. Feb. 28, 2017. URL: https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html (visited on 03/13/2020).

[25] L. Wagner. *WebAssembly Will Finally Let You Run High-Performance Applications in Your Browser - IEEE Spectrum*. IEEE Spectrum: Technology, Engineering, and Science News. Library Catalog: spectrum.ieee.org. Sept. 21, 2017. URL: https://spectrum.ieee.org/computing/software/webassembly-will-finally-let-you-run-highperformance-applications-in-your-browser (visited on 03/12/2020).

[26] *wasienv/wasienv*. original-date: 2019-10-16T19:19:48Z. Mar. 10, 2020. URL: https://github.com/wasienv/wasienv (visited on 03/13/2020).

[27] *wasm3/wasm3*. original-date: 2019-10-01T17:06:03Z. Mar. 6, 2020. URL: https://github.com/wasm3/wasm3 (visited on 03/06/2020).