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:	Teemu Kärkkäinen
Advisor:	Advisor
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

Acknowledgments

Abstract

Contents

1 Introduction

2 Background

2.1 Microcontrollers

Since this thesis is focused on the use of WebAssembly on microncontrollers we'd like to shortly introduce the concept and limitations. A microcontroller (**MCU** for microcontroller unit) is a small computer, meant to fulfill a very specifc requirement without a complex operating system. They are designed for embedded applications from implanable medical devices to toys and very prominently in IoT devices. Devices will often have multiple microcntrollers each responsible for a very specific function, a car for example could include amongst others a MCU to control the mirror adjustemnts, one to handle fuel injection and another one for traction control. Core elements of a MCU are the processor, memory and I/O peripherals. The Processor (CPU) can be though of as the brain of the MCU, it performes basic arithmetic, logic and I/Ooperations. The memory is where aby data is store the processor needs to fullfill it's tasks. Mainly there is program memory, which holds the MCUs instructions and Data memory, which servers as temporary storage while a program is exectued. Lastly the I/O peripherals are the controllers 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 want to specifically focus on the ESP32 system on a chip (SoC). A very popular low-cost, low power series of microcontrollers with integarted WiFi and bluetooth. Developed by the Shanghai based comapny Espressif Systems this successor to the ESP8266 offers a great platform for IoT and embedded projects. Comapred to the the MCU described before the ESP32 has additional processing power and I/O options that make it a great platform for developing secure IoT devcices. 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 version available from ones very well suited for hobbyists to ones usable in industrial manufactures. With a low price point and area footprint they still provide significant performance and many operational features. To reach price and power consumtion targets the ESP32 has

significant hardware limitations. This introduces some constraints when working with the platform. The Operating system for example can hardly be called that. Compared to popular operating systems like windows and linus the used FreeRTOS can be thought of like a thread library. This speciality will be further explained in a following passage.

2.1.2 FreeRTOS

Many MCUs are used in applications where throughput is less importance than a guaranteed performance. This is why the ESP32 uses FreeRTOS, a real time operating system (RTOS) is specifically intedned to be used in time critical situations. A key characteristic of such an operating system is the predictable behaviour of the scheduler, the part of the operating system that decides which task should be run by the CPU at any give 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. Since most applications in which MCUs are used don't warrant the use of a full RTOS, FreeRTOS only provides the core scheduling functionality, timing and synchronisation primitives. It can however be estenden by using addon compontents for example to make use of a specific networking stack. FreeRTOS also built a significant comminuty and support for many platforms in its 15 yaer development.

2.2 WebAssebly

Beginning with static HTML pages the web has since developed into a very popular application paltform, accessible from a many different devices running different operating systems. JavaScript is the only natively supported language on the web. But even though it is universally used and made a lot of progress ion modern implementations it still has some problems as a compilation target. WebAssembly adresses these issues and provides a compilation target for the web. WebAssembly (WASM) was first announced in June 2015 and reached cross-browser consensus in March 2017. It's 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 interested picked up around usage outside of the browser, which is also the main concern of thius thesis. Being meant for the web, WASM has to achieve certain goals that give the platform very interesting properties. It has to be safe, since on the web, code is loaded mainly from untrusted sources. It has to be fast as the main motivation to introduce WebAssembly was to provide a compile target ont he web with reliable performance. Other than the usual low level code such as regular assembly, WebAssembly has to be protable

and work in all the different circumstances the web is currently used. Lastly, because the code is transmitted over the notwork it has to be as small as possible to reduce bandwith and improve latency. WASM is a low-level binary 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 exectutes in near-native speed due to it's low-level design. Still it runs in a memory-safe enviroinment inside the browser and is subject to the same security policies as JavaScipt 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 provies 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.

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 ina variety of ways. The aforemanetioned goals, which WebAssembly achieves make it a very interesting format to explore on embedded devices. Due to it's aim to be universal it would allow the use of languages on MCUs that where 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.

2.3 Interpreters

Interpreters are computer programs that execute a program. They pose a different concept to compiled execution, where a program would be translated to machin 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 an order of magnitude and sometimes more. The oberhead is generated byt the interpreter having to analyze the program code before it can be executed. Interpretes 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 usecase the interpreter executes WASM

instructions, allowing the dynamic loading of modules and running them in the chose environment.

3 Methodology

3.1 Finding a Runtime

3.1.1 Missing documentation

3.1.2 Selection of WASM3

3.2 Designing Tests

3.2.1 Fighting the optimizer

4 Benchmarking

5 Conclusion

List of Figures

List of Tables