A Standalone WebAssembly Development Environment for the Internet of Things

István Koren[0000−0003−1350−6732]

Chair of Process and Data Science, RWTH Aachen University, Aachen, Germany
koren@pads.rwth-aachen.de

Abstract. In Industry 4.0, there is a growing demand to perform high-performance and latency-sensitive computations at the edge. Increasingly, machine data is not only collected but also processed and translated into actionable decisions influencing production parameters in real-time. However, heterogeneous hardware in the Internet of Things prevents the adoption of consistent development and deployment structures as known from service containers. WebAssembly is a recent hardware-agnostic bytecode format that is capable of running not only in the browser, but also on microcontrollers and in cloud environments. In this article, we argue that this web technology can have a real impact by leveraging tools and programming languages that web engineers are familiar with. As a first step, we present a proof-of-concept integrated development and deployment environment to execute WebAssembly modules on microcontrollers. Its key feature is a built-in web server that provides a self-contained browser-based IDE to directly develop, compile and flash AssemblyScript code to a device. In this sense, the Web of Things will unfold a streamlined development and deployment context for the agile and low-latency operationalization of adjustable data streaming and action-oriented process adaptations for industrial devices.

Keywords: WebAssembly · Internet of Things · Industry 4.0.

1 Introduction

Digitalization of industrial assets comes with many promises like higher productivity, less rejects and smaller lot sizes. For instance, recent developments in the application of machine learning in the production context allow for data-driven decisions like parameter tuning in order to obtain a desired quality in the manufactured part. However, there are a number of challenges towards a fully connected smart production. First, the asset-heavy manufacturing industry is characterized by long-term investments with many legacy devices on the shop floor; while state-of-the-art industrial assets are able to push data over protocols like OPC-UA, many legacy machines only feature serial communication ports. Even programmable logic controllers are often restricted to a proprietary programming language that only specialized developers can work with. Furthermore, many industrial processes are time-critical, thus parameter adaptions need to be carried out within a narrow window. For instance, compensations for disruptions in steel production must be made within a few milliseconds.

The concept of *Retrofitting* [10] refers to augmenting industrial assets with computational equipment able to forward data from serial interfaces via modern Internet of Things (IoT) protocols. It can be achieved by using microcontrollers, industrial-grade versions of boards like Raspberry Pi, or specialized edge cloud hardware. By bringing computation close to the devices, Retrofitting is therefore also useful for addressing latency issues. Overall, in the highly heterogeneous system landscape of the shop floor, different computational hardware like proprietary programmable logic controllers prevail together with custom modules. Yet, following the idea of agile manufacturing, software on the edge underlies frequent incremental changes [9]. To tackle this highly complex environment, a development and deployment structure is required that can handle this heterogeneity, i.e. produces service components that run on microcontrollers, on edge devices as well as in the cloud.

Over the last decade, containerization of microservices through technologies like Docker achieved low coupling of heterogeneous technologies with clearly cut functionalities. However, these containers are not lightweight enough to be run on low-powered IoT devices. To this end, WebAssembly (WASM) is a recent web standard. Originally, it was conceived for computationally intensive tasks within browser clients that need near-native performance, of that the interpreted JavaScript environment is not capable of, like image processing. Initiatives like WASI (WebAssembly Systems Interface) aim to make system calls available to WASM modules, making them a suitable platform to tackle the above challenges on the shop floor. In this article, we present a proof-of-concept that allows to run the same code on different hardware platforms. It features a built-in web server that delivers a simple IDE to develop code, compile it to bytecode, and flash it to the device. After its initialization, the binary code is capable of accessing device in- and output pins, for instance, to forward data from the device, or to directly change control parameters as the result of stripped-down machine learning algorithms. While essentially powered by web technologies, this opens many use cases, e.g. for the low-latency operationalization of real-time decisions.

This article is organized as follows. Section 2 motivates WebAssembly usage in service-oriented architectures and discusses related work. Section 3 presents the conceptual design and implementation of our proof-of concept. Section 4 discusses our findings, experiences and limitations. Finally, Section 5 concludes the article with an outlook on future work.

2 Towards Code Mobility on the Web of Things

Microservices and the containerization paradigm introduced by technologies like Docker and Kubernetes changed development and deployment structures in the last decade. By running isolated software containers, processes are sandboxed from each other, i.e., services cannot access each other's memory. However, because of the large overhead of software containers, this approach is not feasible

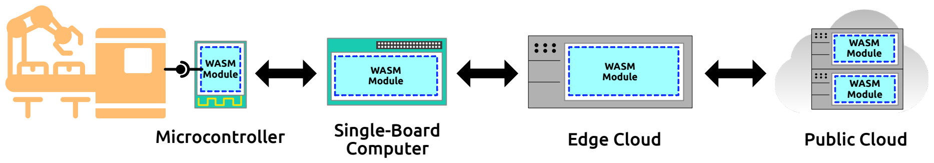

Fig. 1. Migration of Unmodified WebAssembly Modules From Edge to Cloud

for resource-constrained devices like single-board computers or even microcontrollers. Even though WebAssembly has only been introduced recently, there is already a vast adoption on frontend and backend as it harmonizes well with the serverless computing paradigm [3,6]. A curated list of use cases in the browser can be found on the list of *Awesome Wasm*[1]. Besides, open source and commercial offerings recognize the potential of WebAssembly for deploying functionalities on the backend. *Fastly Inc.*, one of the world's largest edge cloud providers, feature a dedicated WebAssembly runtime in their content delivery network. WebAssembly has an inherent sandbox based on design decisions such as linear memory management, ensuring that program code cannot break out its dedicated memory addresses. Its bytecode format can be cross-compiled from various programming languages and interpreted on platforms like IoT devices and the cloud. The format is governed by an open alliance of industry and research partners. For instance, AssemblyScript, a derivate of TypeScript, can be compiled into WASM bytecode. TypeScript, in turn, is very similar to JavaScript, opening access to a large number of web developers. It adds variable type declarations while staying syntactically close to JavaScript. The compiled WebAssembly modules can move freely between different hardware architectures. We argue, that it is therefore suited as code mobility [2] framework for small-scale functions in the Web of Things, adopting principles of *Liquid Web Applications* [5] on the backend. Our use case are deployments in the heterogeneous device landscape of industrial machines as illustrated in Figure 1.

A number of researchers evaluated WebAssembly in serverless contexts outside the browser. Hall et al. run serverless functions and compare the execution as WebAssembly with Docker containers [3]. In their system, a node.js context executes WebAssembly modules. As primary advantage, they identify the absence of a large cold start penalty, as opposed to Docker. Tiwary et al. confirm that spawning WebAssembly modules in a containerized environment suffers from cold-start problems [11]. Murphy et al. compare the performance of different native runtimes and find executing in a pure node.js environment to be the fastest [6].

3 Proof-of-Concept WASM on the Edge

In this section, we discuss our proof-of-concept, by demonstrating the viability of WASM modules on microcontrollers.

[1] cf. `https://github.com/mbasso/awesome-wasm`

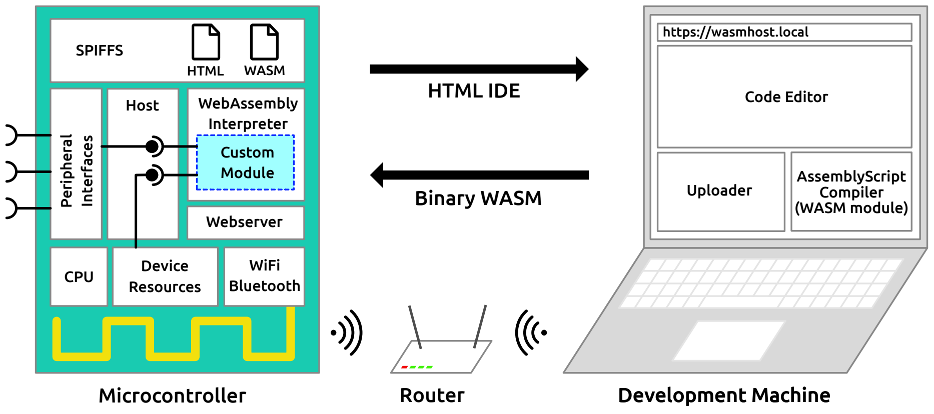

Fig. 2. Overall System Architecture With Microcontroller and Development Laptop

3.1 Conceptual Design

The main conceptual idea is to provide a code execution layer on top of the microcontroller firmware whose code is exchangeable. Our prototype consists of three parts: the host environment, the runtime, as well as the development environment. Figure 2 presents the basic architecture. On the left, the components of a microcontroller are shown. On its top, SPIFFS (Serial Peripheral Interface File System) is a lightweight system for storing files. Other peripheral interfaces connect to external hardware over serial connections. The host is the firmware that gets activated once the controller is booted. It connects to the WiFi network and ensures availability via Bluetooth, if needed. Then, if a WebAssembly module has been loaded, it gets instantiated and provided access to the device's in- and output pins. Any external dependency needs to be explicitly declared beforehand to be available from within the module.

The host features a built-in web server. It delivers a simple HTML page that has a code editor and a button. The HTML page references the external compiler. Once the button is clicked, the compiler gets called. If the code can be compiled without errors, the resulting binary WebAssembly module is uploaded to the host environment on the microcontroller. For this purpose, the same web server that provides the code editor also has an HTTP endpoint for accepting and storing the binary module. After deployment, the microcontroller reboots and executes the new module. In the next section, we discuss the implementation and present the libraries used.

3.2 Microcontroller Implementation

As microcontroller, we chose an ESP32 board. The successor of the ESP8266 is extremely popular for tiny IoT projects, and comes with a dual-core processor, as well as WiFi and Bluetooth connectivity. Most of the development boards come with 4 MB of flash size. We specifically used modules from TTGO and the DEVKIT V1, which are in a price range between 4-9 Euros. The host firmware is

Listing 1.1. AssemblyScript Code Passing Sensor Value Every Third Time

```
 1  // the pass function is imported from the host
 2  declare function pass(data: f32): void;
 3
 4  // a global variable to store the number of cycles
 5  let cycle:f32 = 0;
 6
 7  // the process function gets called from the host
 8  export function process(a: f32): f32 {
 9    cycle++;
10    if (cycle == 3) { // only send value upstream on every third cycle
11      pass(a);
12      cycle = 0;
13    }
14    return a;
15  }
16
17  export function _start(): void {
18    while (true) { // keep module active
19    }
20  }
```

developed using the Arduino framework that is compatible to the native ESP development kit. As WebAssembly interpreter, we decided to use the open source Wasm3 library[2]. The web server is available via a Multicast DNS hostname over the local WiFi network. It is powered by the ESPAsyncWebServer library[3]. The WebAssembly module is developed with AssemblyScript[4]. It is a variant of TypeScript that is popular in the web development community. The compiler that translates AssemblyScript into the binary bytecode format is available as WebAssembly module itself, it can thus be called from within an HTML5 application context. We load it from the *unpkg.com* CDN to free as much memory as possible for the application code. After its compilation, the WASM file is uploaded via a HTTP multi-part form upload to the microcontroller. On the microcontroller, it is saved within the flash that is formatted as SPI file system.

Listing 1.1 shows an example AssemblyScript code. It is a very basic module with a `process` function, that forwards every third call to the `pass` method. With `declare` (line 2) we import functions of the host into the module. In the other direction, functions that are called from the host are marked with `export` (line 8 and 17). The `_start` function is called for initializing resources; it is also responsible for keeping the module active (cf. the `loop()` function in Arduino code). Due to the sandbox, it is not possible to call other environment functions from within the developed module. Consequently, all external functions must be declared at compile time. WebAssembly is using a linear memory management with allocated memory locations. In our demo code, we import and export for now only integer and float types, as the management of elaborate types involving strings and objects is more complicated and error-prone.

[2] cf. `https://github.com/wasm3/wasm3`

[3] cf. `https://github.com/me-no-dev/ESPAsyncWebServer`

[4] cf. `https://www.assemblyscript.org/`

4 Applicability of WebAssembly on Microcontrollers

Our proof-of-concept allows to run user-contributed code on microcontrollers and to exchange it during runtime. It brings advantages known from container-based deployments to the Internet of Things with its resource-constrained devices and variety of architectures. Our approach allows for the fast exchange of code, but promises a certain level of security that is achieved through sandboxing. At the same time, the standardized bytecode format allows to leverage various programming languages. We are convinced that these characteristics make WASM applicable for realizing the potential of the Web of Things in the industrial area.

Running replaceable user-contributed code on the edge, both in the browser and as in our case on IoT devices, opens a number of security issues. Attack vectors of running code in web browsers are discussed by Papadopoulos et al. [7]. The authors analyze the use of service workers for leveraging the processing power of client devices, for instance to mine cryptocurrencies. Service workers can execute code in the background, even if the user is not actively using the originating website. WebAssembly adds a controllable sandbox. However, through the bytecode format, it is much harder to anticipate what code is running. Security measures therefore need to be undertaken, and access to the outer world needs to be limited to the absolutely necessary.

4.1 Preliminary Evaluation

The firmware size of our host, including the WASM and server libraries is around 1.1 MB. On a typical ESP32 chip, this leaves around 2.9 MB of flash that has to be shared with RAM and the SPI file system. The AssemblyScript compiler has a size of around 1 MB; to save SPIFFS space, we load it as external dependency. To compare the performance of our WebAssembly framework with running code natively on the microcontroller, we ran evaluations that confirmed the performance penalty described in Section 2. For instance, Jangda et al. calculated an average decrease between 1.5x and 2x [4]. The authors note that WebAssembly is still a rather new technology that is constantly optimized. The range of possible functionalities in the module versus natively on the host, however, is not limited, as it depends on what device resources are linked into the module.

4.2 Limitations of the Prototype

Conceptually, our proof-of-concept shares limitations with WebAssembly. For instance, it does not support threads, even though the ESP32 has a dual-core processor. WebAssembly does not allow for hardware-specific abstractions [3] like graphics card based matrix multiplication for machine learning algorithms. This could be a limiting factor when running (even light-weight) machine learning models. Moreover, researchers have highlighted issues inherent to WebAssembly, like decreased memory safety when compiling from other languages [1].

Our built-in IDE currently only supports AssemblyScript. We are planning to add further languages, like Python or Rust, depending on the availability of compilers that can be executed in a browser context. Besides, modules can already be written in various languages, as described in Section 2. For this, the modules need to import and export the specified methods. Regarding data exchange with the underlying host, we only support integer and float types. Also, no security, like access control, is part of our firmware. We are evaluating different mechanisms, like private-key signatures and identification via OpenID Connect to overcome this. Specific to the ESP32, we experienced a challenge with processor interrupts caused by timeouts. To overcome this, we regularly call Arduino's `delay()` function. In this regard, we do also not handle errors besides catching exceptions and closing the module gracefully. To overcome this, we are planning to introduce a fallback firmware. However, broken AssemblyScript modules are detected at compile-time and thus cannot be flashed onto the device.

5 Conclusion and Outlook

Data-driven insights are currently driving the fourth industrial revolution. Industrial assets are equipped with sensors that are able to provide fine-grained properties in a high frequency. However, there are many challenges towards agile software development structures in this highly complex area. Heterogeneous hardware architectures on the edge, from Arduino-driven microcontrollers up to specialized edge hardware make it hard to uniformly develop functionalities. Also, updating code is cumbersome and varies across boards. Following the ideas of agile manufacturing [9], frequent updates are part of the approach.

Our proposal is to introduce a common development methodology and framework powered by web technologies, that can run on edge-deployed hardware as well as in the cloud. For this, WebAssembly is an ideal candidate. In this article, we introduced a working proof-of-concept. It features a web server that delivers a complete IDE to develop, compile and deploy new firmware modules within a browser. In the future, we plan to support a peer-to-peer firmware propagation between modules. On the ESP platform, we can use built-in functionalities that build a mesh via Bluetooth or WiFi. Similarly, we want to enable a dynamic context-dependent code mobility from IoT device to edge, cloud and vice versa. For instance, if the processor load becomes to high, the device should be able to move its module to a near-by edge device. Both peer-to-peer deployment and context adaptation require higher security measures. This can be achieved by a security layer, e.g., by signing flashed modules.

We are currently equipping industrial laboratories of the engineering department at our university to trial industrial use cases. As described, agile updates of developed modules can be leveraged not only for data collection, but also for the operationalization of, e.g., business processes. We are convinced that faster development and deployment methodologies can finally enable the age of the *Internet of Production* [8], where data collection powers artificial intelligence algorithms that in turn achieve action-oriented data insights.

Acknowledgement. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC-2023 Internet of Production - 390621612.

References

1. Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A., Stefan, D.: Position Paper: Progressive Memory Safety for WebAssembly. In: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 1–8. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3337167.3337171
2. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding Code Mobility. IEEE Transactions on Software Engineering **24**(5), 342–361 (1998). https://doi.org/10.1109/32.685258
3. Hall, A., Ramachandran, U.: An Execution Model for Serverless Functions at the Edge. In: Landsiedel, O., Nahrstedt, K. (eds.) Proceedings of the International Conference on Internet of Things Design and Implementation. pp. 225–236. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3302505.3310084
4. Jangda, A., Powers, B., Berger, E.D., Guha, A.: Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 107–120. USENIX Association, Renton, WA (2019), `https://www.usenix.org/conference/atc19/presentation/jangda`
5. Mikkonen, T., Systä, K., Pautasso, C.: Towards Liquid Web Applications. In: Cimiano, P., Frasincar, F., Houben, G.J., Schwabe, D. (eds.) Engineering the Web in the Big Data Era, Lecture Notes in Computer Science, vol. 9114, pp. 134–143. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-19890-3_10
6. Murphy, S., Persaud, L., Martini, W., Bosshard, B.: On the Use of Web Assembly in a Serverless Context. In: Paasivaara, M., Kruchten, P. (eds.) Agile Processes in Software Engineering and Extreme Programming – Workshops, Lecture Notes in Business Information Processing, vol. 396, pp. 141–145. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58858-8_15
7. Papadopoulos, P., Ilia, P., Polychronakis, M., Markatos, E.P., Ioannidis, S., Vasiliadis, G.: Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation, `http://arxiv.org/pdf/1810.00464v1`
8. Pennekamp, J., Glebke, R., Henze, M., Meisen, T., Quix, C., Hai, R., Gleim, L., Niemietz, P., Rudack, M., Knape, S., Epple, A., Trauth, D., Vroomen, U., Bergs, T., Brecher, C., Buhrig-Polaczek, A., Jarke, M., Wehrle, K.: Towards an Infrastructure Enabling the Internet of Production. In: 2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS). pp. 31–37. IEEE (06052019 - 09052019). https://doi.org/10.1109/ICPHYS.2019.8780276
9. Schuh, G., Reuter, C., Prote, J.P., Brambring, F., Ays, J.: Increasing data integrity for improving decision making in production planning and control. CIRP Annals **66**(1), 425–428 (2017). https://doi.org/10.1016/j.cirp.2017.04.003
10. Stock, T., Seliger, G.: Opportunities of Sustainable Manufacturing in Industry 4.0. Procedia CIRP **40**, 536–541 (2016). https://doi.org/10.1016/j.procir.2016.01.129
11. Tiwary, M., Mishra, P., Jain, S., Puthal, D.: Data Aware Web-Assembly Function Placement. In: Seghrouchni, A.E.F., Sukthankar, G., Liu, T.Y., van Steen, M. (eds.) Companion Proceedings of the Web Conference 2020. pp. 4–5. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3366424.3382670