WebAssembly in IoT: Beyond Toy Examples

Pyry Kotilainen, Viljami Järvinen, Juho Tarkkanen, Teemu Autto, Teerath Das, Muhammad Waseem, and Tommi Mikkonen

University of Jyväskylä, Jyväskylä, Finland

pyry.kotilainen@jyu.fi, viljami.a.e.jarvinen@jyu.fi, juho.a.tarkkanen@jyu.fi, teemu.a.autto@jyu.fi, teerath.t.das@jyu.fi, muhammad.m.waseem@jyu.fi, tommi.j.mikkonen@jyu.fi

Abstract. WebAssembly enables running the same application code in a range of devices in headless mode outside the browser. Furthermore, it has been proposed that WebAssembly applications can be made isomorphic so that they can be liberally allocated to a set of computers that comprise the runtime environment. In this paper, we explore if WebAssembly truly enables the development of comprehensive IoT applications with the same ease as more traditional techniques would enable.

Keywords: WebAssembly · Web of Things · Internet of Things · IoT.

1 Introduction

There has been a lot of interest in using WebAssembly (Wasm) outside the browser [2]. It has been proposed as a small-memory portable operating system (OS) [34], as well as a solution to improve modularity [17] or performance [16]. In particular, the cloud-edge continuum has been used as a target, either as an OS [13] or by evaluating applicability in serverless computing [19], to support services at the edge. However, many demonstrators have been small-scale implementations, focusing on demonstrating a single claim or feasibility of a proposed approach with WebAssembly.

In industry, WebAssembly has been proposed as a tool for addressing performance [5], size [17], and to some extent security issues [9] that more traditional web technologies – in particular JavaScript – introduce. Numerous enterprises, such as Google, eBay, and Norton have already started using WebAssembly instead of JavaScript in many of the projects with the aim to enhance the performance of services, such as Tenserflow.js applications [4], a barcode reader [26], and pattern matching [27].

Furthermore, WebAssembly could tackle some major challenges of the Internet of Things (IoT), which has become a key technology enabler for several diverse and critical daily life applications such as healthcare, transportation, and industrial automation, among others. The fragmented nature of IoT applications requires a more profound understanding of various programming languages and prior knowledge of technologies, which makes it more complex for developers to implement and manage a typical end-to-end IoT system without impediments.

One of the prominent solutions for addressing these challenges is to adopt an isomorphic IoT system architecture, which provides feasibility in developing the whole system with the same set of technologies [20]. With this in mind, WebAssembly is the way-forward approach that supports the development of isomorphic IoT architecture. The possible benefits of having such an architecture include reducing the complexity of the IoT system, improving the overall performance, and decreasing the overhead of developers from implementing fragmented IoT applications [20].

In this paper, we discuss WebAssembly in the IoT application context. Furthermore, we analyse the feasibility of an isomorphic IoT architecture using WebAssembly based on our experiences gained with a prototype implementation. Finally, we analyze WebAssembly's suitability as a platform for full-stack IoT applications and propose its potential use in IoT in the short term.

2 Background and Motivation

2.1 Isomorphic IoT Systems

While isomorphism is a well-established concept in mathematics, in software development the concept has emerged relatively recently. In the context of web applications, isomorphism refers to the ability to run the same code both on the backend (cloud), and in the frontend (web browser) [29]. More broadly, isomorphic software architectures feature software components that do not have to be modified ('change their shape') when running across the different hardware or software components of the system. Examples of isomorphic systems include Java and its 'write once, run everywhere' promise [1], Unity 3D engine, Universal Windows platform, which enables running the same code in Windows 10, Xbox One gaming machines and HoloLens devices, and liquid web applications [21].

Several different levels of isomorphism can be identified [20]. At the first level, isomorphism refers to the consistent use of the same development technologies across the different computational elements in the entire system. In contrast with such *static*, development-level isomorphism, in *dynamic* isomorphism, a common runtime engine or virtualization solution is used so that the same code can run in different computational elements without recompilation. In an even more advanced system, dynamic migration of code from one computational element to another is enabled.

In the IoT context, the same, isomorphic software can ideally be deployed throughout the end-to-end system to run on edge devices, gateways, mobile clients and cloud services. With its characteristics, WebAssembly is a natural candidate for such use, as discussed below.

2.2 WebAssembly in a Nutshell

WebAssembly is a low level code format designed for efficiency together with hardware- and platform-independence among other things [28]. WebAssembly

offers dynamic isomorphism, as a standard runtime interpreter is used to execute the code. Therefore, WebAssembly can be used as a runtime environment for applications developed using different languages, but compiled for the WebAssembly interpreter.

While the origins of WebAssembly are inside the browser, the developer community has started to realise its significance outside the browser, in particular as a unifying environment for heterogeneous devices [2,31,15,33]. Indeed, WebAssembly's conservative memory usage and somewhat near-native performance make it suitable for constrained environments like IoT devices [6]. Moreover, facilities such as *WebAssembly System Interface* (WASI) [3] have been introduced, to access system resources when running WebAssembly outside the browser.

3 Design and Implementation

3.1 Development Goals and Initial Architecture

The goal of our development approach was straightforward – go all the way to build isomorphic IoT applications with WebAssembly. In more technical sense, the target was to

G1: Demonstrate that isomorphic web applications can be freely located to form a functioning IoT network;
G2: Demonstrate that various WebAssembly modules from different repositories can be used to implement the applications;
G3: Demonstrate that applications can configure themselves upon deployment to liberate developers from rigid, development-time configurations.

The technical framing for the demonstrations was to use a microservice architecture consisting of various IoT devices [10]. Within this architecture, heterogeneous devices could easily be discovered and used in accordance to their characteristics, such as varying processing power and peripheral features, because application code could move inside the system. Computations would then be executed when and where best suited, taking into account the state of the system and the capacity of the different subsystems.

Device discovery would be performed with mDNS for advertising available IoT-devices to the orchestrator. For querying the capabilities of discovered devices, a ReSTful endpoint providing the answers would be placed on each IoT device. For machine-to-machine (M2M) communication between the IoT-devices using CoAP was planned, to demonstrate that IoT specific protocols are feasible. Finally, all functionality running on the different IoT-devices – in particular executing WebAssembly binaries – would be controlled by the host process, which we call *supervisor*.

3.2 Development and the Reality Check

From the beginning, we realized that relying on external components would be necessary, building on earlier experience with IoT systems [22]. Hence, in

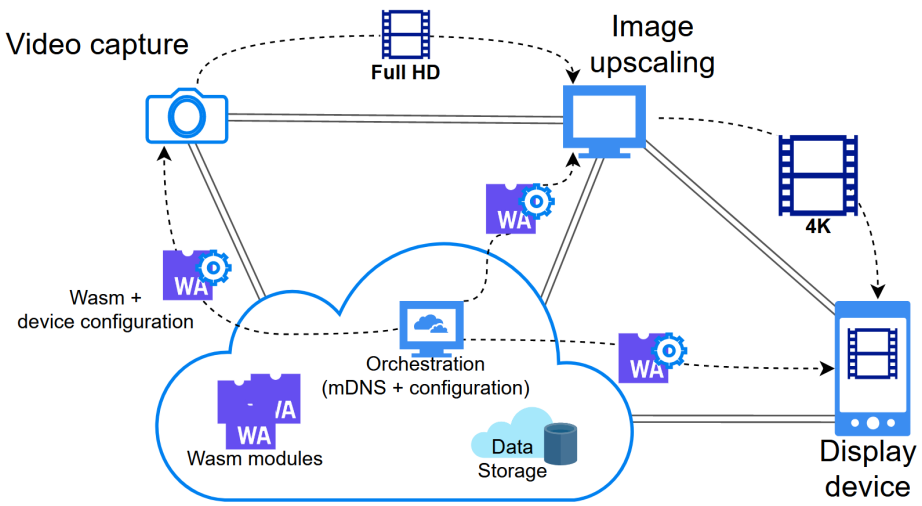

Fig. 1. Overview of the function of the proposed system. In this case, a low quality video is sent from a source to an intermediary device with resources for upscaling the video and then sent to a device with a screen to display the upscaled video.

parallel to composing our first application specific WebAssembly routines, we started with an inventory of suitable 3rd party WebAssembly components we could use. Unfortunately, it turned out that if we wanted to rely on widely used, reliable technologies, we would have to look beyond WebAssembly. For instance, as the ReSTful endpoint we decided to use a Python framework *Flask*, not a WebAssembly derivative. As the development went on, the same thing kept repeating – with almost every feature, some new open source component, not implemented with WebAssembly, crept in because it simply was not technically feasible for the team to implement everything in WebAssembly with a reasonable development time. The most essential learnings are listed below.

During the development, a repeating problem was the features of different WebAssembly runtime implementations. Some of the WASI implementations lacked key functions for our use cases. For instance, networking – or even access to the underlying OS's file system – meant that we had to implement our supervisor function without directly using WebAssembly. Obviously a layer of unnecessary abstraction is a burden when running on constrained IoT-devices, but this was considered the simplest way forward, and we would know where the superfluous function is. In the end, we used Python3 and the WebAssembly runtime *Wasm3* [18], because the selection of libraries they provide fit our planned use cases, e.g. cameras and sensors controlled with laptop and Raspberry PIs, respectively.

Application programming with WebAssembly raised some concerns with respect to data objects and secure and refined interfaces to access the data. Because multiple programming languages can be used to compose WebAssembly code,

it seems that a common and accepted way of using non-primitive datatypes in WebAssembly is based on pointers, as described in [12]. This impression is reinforced by examples of using WebAssembly's memory like an integer-indexable array [23], which is adopted also in practice and business (e.g. [15]). Hence, while WebAssembly might help in dealing with problems such as buffer overflow and running code from untrusted sources in a sandbox [14,31], dangers still seem to exist [11]. Such security concerns are especially unfavorable considering the potential of code reuse with existing C/C++ libraries compiled to WebAssembly.

The application development stage also introduced challenges related to combinining IoT device architecture and general microservice architecture. A way to comprehensively, extensively and even automatically describe the interfaces between different WebAssembly modules was needed. In our work, we took the Web of Things Thing description [7] and OpenAPI [25] as the baseline. Actual descriptions could be made with different interface description languages, such as Smithy, which is used by the WasmCloud [33] project. Even parsing the WebAssembly binaries (i.e., the *modules*) for their exports using Kaitai [8] was considered, but abandoned as we wanted to enforce fundamentally API-like descriptions. Finally, for generating APIs, Swagger [30] was selected, because it supports OpenAPI by its design. However creating a comprehensive, all-purpose implementation was considered an overkill, and we turned to existing research prototypes for inspiration. Unfortunately, earlier work and existing implementations where WebAssembly is used in a way that is aligned with our goals do not support any WebAssembly targeting language in application development, but only one language. The WiProg system as proposed by Li, Dong and Gao [12] uses language-specific constructs tied to C/C++. With WasmCloud [33], in contrast, the currently available languages are Rust and TinyGo.

When considering existing implementations, the closest match with our needs was provided by the Losant [15] project. However we did not wish to commit to using their service to construct our applications. In the end, we wrote our own prototype version of the required functions, targeted to exact use cases we had in mind. The available device server entries and deployment instructions are defined in JSON, and the actual package management and deployment was implemented in JavaScript.

3.3 Results and Observations

At the present phase of the development, there is a running demonstrator, where isomorphic code can be run in WebAssembly environment. However, while WebAssembly is at the core and applications are written in WebAssembly, everything that surround the apps is something else, most often JavaScript, Node.js, or Python, simply because a dominant design already existed. Moreover, building the corresponding function from scratch would have been a major engineering effort, not simple experimentation. Hence, the grand goal to use WebAssembly to implement every part of the architecture was deemed to fail. In hindsight, this could have been overcome by using a monolithic architecture instead of microservices, which by definition embrace technology integration.

4 Discussion

To summarize the experiences from our development, there are numerous aspects that truly fulfilled the WebAssembly promise. For instance, running old code, implementing fast algorithms, was considered feasible, and the new tools and techniques, in particular Rust, form pieces for a really developer-friendly IoT technology stack. Hence, potential for future development truly exists.

However, forming full technology stacks – or even forming one's own – was deemed difficult with WebAsssembly. There was little support for accessing resources or to support integration between different microservices or subsystems, implemented with different technologies. We attribute the above to WASI implementations, which are still immature, and to calling WebAssembly modules from other languages, which is made cumbersome, in particular when dealing with complex data structures. One needs to use generated glue code even with single WebAssembly modules, if these form meaningful business entities, which by definition often is the case with microservices. Moreover, implementation-specific differences in essence imply that it is easy to be bound to a particular virtual machine in a project, instead of being able to use different ones in different devices, based on characteristics of the device. These issues have been overcome by others (e.g. [12,33]) by simply avoiding excessive interfacing, and enforcing a monolithic architecture.

With the above observations in mind, WebAssembly is well suited for small, independent, yet security and/or performance dependent routines that are called when needed, instead of aiming at a full WebAssembly IoT stack. Such use of WebAssembly resembles its role inside the browser where it can have a limited, supporting role in some performance-heavy places, not as a fundamental piece in the tech stack to build on in large scale. This approach has been proposed by [24]. However, even performance advantages have been partially challenged [32].

5 Conclusions

Using WebAssembly outside the browser has gained a lot of interest recently. In this paper, we have studied using it as a comprehensive technology for IoT applications. In conclusion, we were able to use WebAssembly for key functions, but the technology was complemented by readily available subsystems and extensions using some other technology. Therefore, we believe that WebAssembly is presently applicable in the IoT domain, but to speed up executions and to produce security gains, instead of being a comprehensive development stack. This resembles the role of WebAssemly inside the browser, where particular tasks can be run independently inside the WebAssembly virtual machine. However, even in this role several complications exist in the IoT domain, such as lack of comprehensive standards, lack of OS implementation, and the dominance of de-facto implementations composed with dynamic languages and other less rigid techniques than WebAssembly. To this end, in our future work, candidate subjects to study include the use of artificial intelligence and machine learning (AI/ML)

related features, encryption and decryption in general, and domain specific algorithms.

Acknowledgments.This work has been supported by Business Finland (project LiquidAI, 8542/31/2022).

References

1. Arnold, K., Gosling, J., Holmes, D.: The Java programming language. Addison Wesley Professional (2005)
2. Bryant, D.: WebAssembly outside the browser: A new foundation for pervasive computing. Keynote at ICWE'20, June 9-12, Helsinki, Finland (2020)
3. Bytecode Alliance: Welcome to WASI, `https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-intro.md`, retrieved 2022-12-05
4. Daniel Smilkov, Nikhil Thorat, and Ann Yuan: Introducing the WebAssembly backend for TensorFlow.js, `https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html`, retrieved 2020-03-11
5. De Macedo, J., Abreu, R., Pereira, R., Saraiva, J.: On the runtime and energy performance of webassembly: Is WebAssembly superior to JavaScript yet? In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). pp. 255–262. IEEE (2021)
6. Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. pp. 225–236. IoTDI '19, Association for Computing Machinery, New York, NY, USA (Apr 2019). https://doi.org/10.1145/3302505.3310084
7. Kaebisch, S., Kamiya, T., McCool, M., Charpenay, V., Kovatsch, M.: Web of Things (WoT) Thing Description, `https://www.w3.org/TR/wot-thing-description/`, retrieved 2022-12-09
8. Kaitai project: Kaitai home page, `https://kaitai.io/`, retrieved 2023-1-24
9. Kim, M., Jang, H., Shin, Y.: Avengers, assemble! survey of WebAssembly security solutions. In: 2022 IEEE 15th International Conference on Cloud Computing (CLOUD). pp. 543–553. IEEE (2022)
10. Kotilainen, P., Autto, T., Järvinen, V., Das, T., Tarkkanen, J.: Proposing isomorphic microservices based architecture for heterogeneous iot environments. In: International Conference on Product-Focused Software Process Improvement. pp. 621–627. Springer (2022)
11. Lehmann, D., Kinder, J., Pradel, M.: Everything old is new again: Binary security of {WebAssembly}. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 217–234 (2020)
12. Li, B., Dong, W., Gao, Y.: Wiprog: A webassembly-based approach to integrated iot programming. In: IEEE INFOCOM 2021-IEEE Conference on Computer Communications. pp. 1–10. IEEE (2021)
13. Li, B., Fan, H., Gao, Y., Dong, W.: ThingSpire OS: a WebAssembly-based IoT operating system for cloud-edge integration. In: Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services. pp. 487–488 (2021)
14. Long, J., Tai, H.Y., Hsieh, S.T., Yuan, M.J.: A lightweight design for serverless function as a service. IEEE Software **38**(1), 75–80 (2020)
15. Losant IoT, Inc: Embedded Edge Agent, `https://docs.losant.com/edge-compute/embedded-edge-agent/overview/`, retrieved 2022-11-09

16. Mäkitalo, N., Bankowski, V., Daubaris, P., Mikkola, R., Beletski, O., Mikkonen, T.: Bringing WebAssembly up to speed with dynamic linking. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1727–1735 (2021)
17. Mäkitalo, N., Mikkonen, T., Pautasso, C., Bankowski, V., Daubaris, P., Mikkola, R., Beletski, O.: WebAssembly modules as lightweight containers for liquid IoT applications. In: International Conference on Web Engineering. pp. 328–336. Springer (2021)
18. Massey, S., Shymanskyy, V.: wasm3: The fastest WebAssembly interpreter, and the most universal runtime, `https://github.com/wasm3/wasm3`, retrieved 2022-12-09
19. Mendki, P.: Evaluating WebAssembly enabled serverless approach for edge computing. In: 2020 IEEE Cloud Summit. pp. 161–166. IEEE (2020)
20. Mikkonen, T., Pautasso, C., Taivalsaari, A.: Isomorphic Internet of Things architectures with web technologies. Computer **54**(7), 69–78 (2021)
21. Mikkonen, T., Systä, K., Pautasso, C.: Towards liquid web applications. In: International Conference on Web Engineering. pp. 134–143. Springer (2015)
22. Mikkonen, T., Taivalsaari, A.: Software reuse in the era of opportunistic design. IEEE Software **36**(3), 105–111 (2019)
23. Mozilla: WebAssembly, `https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Memory`, retrieved 2023-01-05
24. Oliveira, F., Mattos, J.: Analysis of WebAssembly as a strategy to improve JavaScript performance on IoT environments. In: Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais. pp. 133–138. SBC (2020)
25. OpenAPI Initiative: OpenAPI Specification, `https://github.com/OAI/OpenAPI-Specification`, retrieved 2022-12-09
26. Padmanabhan, S., Jha, P.: WebAssembly at eBay: A Real-World Use Case, `https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/`, retrieved 2019-05-22
27. Raymond Hill. 2019.: gorhill/uBlock, `https://github.com/gorhill/uBlock`, retrieved 2022-12-09
28. Rossberg, A.: Introduction — WebAssembly 1.1 (Draft 2022-04-05), `https://www.w3.org/TR/wasm-core-2/intro/introduction.html`, retrieved 2023-01-12
29. Strimpel, J., Najim, M.: Building Isomorphic JavaScript Apps: From Concept to Implementation to Real-World Solutions. O'Reilly Media (2016)
30. Swagger project: Swagger home page, `https://swagger.io/`, retrieved 2023-1-24
31. Vetere, P.: Why wasm is the future of cloud computing, `https://www.infoworld.com/article/3678208/why-wasm-is-the-future-of-cloud-computing.html`, retrieved 2022-12-09
32. Wang, W.: Empowering web applications with WebAssembly: Are we there yet? In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1301–1305. IEEE (2021)
33. wasmCloud Project: wasmCloud home page, `https://wasmcloud.com/`, retrieved 2022-11-30
34. Wen, E., Weber, G.: Wasmachine: Bring IoT up to speed with a WebAssembly OS. In: 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). pp. 1–4. IEEE (2020)