Potential of WebAssembly for Embedded Systems

Stefan Wallentowitz and Bastian Kersting
Munich University of Applied Sciences
Munich, Germany
stefan.wallentowitz@hm.edu, bkersting@hm.edu

Dan Mihai Dumitriu
Midokura (Sony Group)
Lausanne, Switzerland
dan@midokura.com

***Abstract*—Application virtual machines provide strong isolation properties and are established in the context of software portability. Those opportunities make them interesting for scalable and secure IoT deployments.**

WebAssembly is an application virtual machine with origins in web browsers, that is getting rapidly adopted in other domains. The strong and steadily growing ecosystem makes WebAssembly an interesting candidate for Embedded Systems.

This position paper discusses the usage of WebAssembly in Embedded Systems. After introducing the basic concepts of WebAssembly and existing runtime environments, we give an overview of the challenges for the efficient usage of WebAssembly in Embedded Systems. The paper concludes with a real world case study that demonstrates the viability, before giving an outlook on open issues and upcoming work.

***Index Terms*—webassembly, interpreter, runtime, portability, embedded systems**

I. Introduction

Embedded Systems are heterogeneous platforms that are deployed into a variety of settings, ranging from deeply embedded microcontrollers with tough resource constraints to powerful IoT devices with AI capabilities. Hardware platforms and software vary a lot and working with different devices can quickly become challenging.

Modern embedded software development is complex and suffers from this heterogeneity [1]. The tools for embedded software development are often fragmented and bound to the programming language used in a project. To support portable software for a variety of platforms and easily migrate and consolidate complex software stacks, virtualization is increasingly considered for embedded systems. Application virtual machines are an interesting solution to portable software. Such virtual machines run platform-independent applications as bytecode with strong separation properties, such as the Java Virtual Machine (JVM). A successful adoption of Java to a specific domain in embedded systems is JavaCard, which is a Java variant specifically tailored to the requirements of resource-constrained smart cards [2].

Other adoptions of Java have primarily focused on devices with user interfaces. Yet, there is still no dominance of Java for embedded systems, even despite efforts to accelerate the Java VM in hardware [3]. One of the reasons is probably the large amount of legacy code and preferences of developers for other programming languages. While other languages could be compiled to JVM too, this potential was not adopted.

WebAssembly [4] is a relatively new application virtual machine format, partly comparable to JVM. Since WebAssembly was introduced in 2017 it has been quickly adopted in browsers to accelerate websites [5]. It has since then evolved in platforms beyond browsers in servers and desktop computers. The rich and rapidly growing, mostly open source ecosystem makes it an interesting candidate for embedded systems, too.

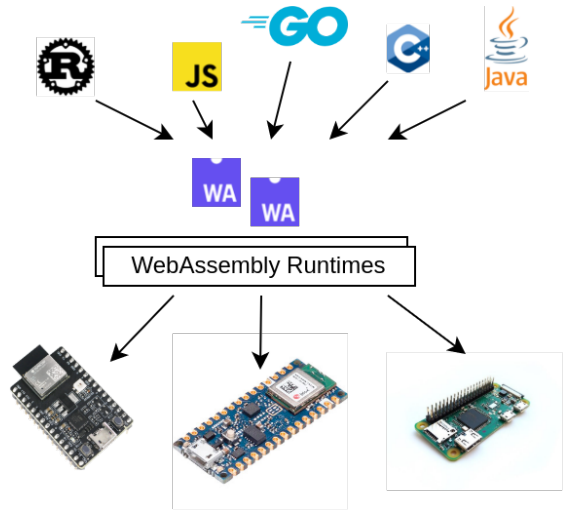

Fig. 1. Portability of WebAssembly for Embedded Systems

Figure 1 illustrates the potential of using manifold source languages to program heterogeneous hardware platforms. A WebAssembly runtime serves not only as a hardware abstraction layer (HAL), but beyond that provides strong isolation properties. WebAssembly applets can be compiled by a variety of source languages, enabled by the LLVM zoo of frontends. After porting a WebAssembly runtime to a platform, those WebAssembly applets can be executed.

In this paper we will briefly introduce WebAssembly and its fundamentals. After elaborating on WebAssembly runtimes, we will dive into the challenges ahead for making WebAssembly a suitable runtime environment for embedded systems. A case study will demonstrate the viability of WebAssembly from a real-world setup. An outlook on future directions concludes this paper.

II. WebAssembly

WebAssembly was initially designed with compilation target JavaScript. In early days, the Web was primarily used for exchanging documents and information in form of simple HTML pages. But with the rising complexity of Websites, JavaScript became the most (and often only) supported language for the Web. With rapid performance improvements in modern

978-1-6654-6828-2/22/$31.00 ©2022 IEEE

2022 11th Mediterranean Conference on Embedded Computing (MECO) | 978-1-6654-6828-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/MECO55406.2022.9797106

VMs, their ubiquity and widespread support, it also became a compilation target for other languages. WebAssembly is maintained by W3C [4]. The Bytecode Alliance [6] stewards runtimes and other software foundations.

WebAssembly can be compiled from a variety of source languages, ranging from script languages like JavaScript, functional languages like Elm, to low-level languages like C and C++. JavaScript's inconsistent performance and other pitfalls that arise when compiling to it motivated the design of a new compilation target. WebAssembly "addresses the problem of safe, fast, portable low-level code on the Web" [5].

WebAssembly has a lot of design goals that are also interesting for domains other than the browser. One of the problems was that the API for interacting with specific software or hardware is not properly defined yet, in contrast to the browser, where the vendors agreed on a public API [7]. The announcement of this WebAssembly System Interface (WASI) therefore sparked a lot of interest fueled by a Twitter statement of one of the Docker inventors, Solomon Hykes: "If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!" [8].

WebAssembly modules (also called applets) are compiled from their source language into a standardized bytecode format. A variety of (open source) compilers and frameworks are available for this. The bytecode format targets a stack-based virtual machine, basically comparable to JVM. As with other application virtual machines there are different modes to execute the bytecode. Running bytecode in an *interpreter* plainly takes the applet and executes the individual commands on the stack machine. *Just-In-Time (JIT)* compilation compiles the bytecode just as its executed to the target architecture on the device, which leads to significantly higher performance. *Ahead-of-Time (AOT)* compilation results in a executable code for the target architecture, which seems like a reasonable compromise for embedded systems while limiting platform flexibility. Each of the approaches has their pros and cons, even interpreters can be sensible, e.g., as often found in smartcards.

III. WEBASSEMBLY RUNTIME ENVIRONMENTS

A large number of WebAssembly runtimes has been released over the last couple of years. In the following we will compare the most prominent runtimes with a focus on popularity and validity for embedded systems during selection. Table I summarizes the findings for the runtimes presented in the following.

One of the criteria for the integration into existing embedded software projects is the possibility to *embed* the WebAssembly runtime. In the scenario of executing a single WebAssembly applet the following steps are executed: (i) initializing the runtime, (ii) register native functions that can be called from the applet, (iii) loading an applet from binary data, (iv) instantiating from that module , (v) call the instance's main function (until it returns), (vi) deinstantiate the module, and finally (vii) destroy the module and the runtime. Starting from those function calls more complex scenarios can be developed. The runtimes that we consider for embedded usage in the following have clear

TABLE I
COMPARISON OF POPULAR WEBASSEMBLY RUNTIMES.

Runtime	Language	AoT	JIT	Embedded
wasmtime [10]	Rust	+	+	-
wasm-micro-runtime [11]	C	+	-	+
wasmer [12]	Rust	+	+	-
wasm3 [9]	C	-	-	+
wasmi [13]	Rust	-	-	+

and concise interfaces, wasm3 [9] is for example available als an Arduino library too.

wasmtime

wasmtime [10] is a runtime maintained by Bytecode Alliance. The runtime is written in Rust but supports other languages such as C, C++ and Python. The Cranelift compiler framework is used as the compiler backend. Furthermore, the Wasmtime project drives new features of WASI, with the official specification of WASI located in the GitHub project files. In its current state it cannot be compiled for embedded targets.

wasm-micro-runtime

The WASM-Micro-Runtime [11] is also maintained by the Bytecode Alliance and is written in C. It fully supports WASI, the latest WebAssembly features on multiple instruction set architectures like x86, Arm, XTENSA and RISC-V. The project's description highlights the small binary size and compiles easily for embedded platforms. It supports AOT compilation.

wasmer

Another runtime that supports many features is Wasmer [12], developed by the eponymous company Wasmer. Wasmer is shipped with WASI and Emscripten support and compliant with the latest WebAssembly proposals (SIMD, Threads, etc.). Furthermore, Wasmer is configurable to support different environments. It comes with three compiler backends, each providing certain advantages, such as execution speed or compilation speed.

wasm3

wasm3 [9] is an extremely lightweight runtime system, particularly targeted at small, resource-constrained devices. It is even available as Arduino module and can be used to interpret a wasm module on an embedded system. It does not support AoT or JIT, but could be an interesting candidate for deeply embedded devices or to run code in other languages on legacy devices that only support C.

wasmi

Wasmi [13] is a runtime developed by the open-source company Parity, which focuses on the blockchain and cryptocurrency implementation "polkadot". The runtime implements an interpreter and is not capable of JiT or AoT compilation. Wasmi also doesn't support WASI and it is unlikely that this is going to happen as the project is in "maintenance mode", where no

TABLE II
MEMORY FOOTPRINTS OF EMBEDDED WEBASSEMBLY RUNTIMES.

Runtime	Code	Data
wasm-micro-runtime Interpreter	94,928	2,068
wasm-micro-runtime Fast Interpreter	103,418	2,076
wasm-micro-runtime AOT	72,040	1,732

TABLE III
PERFORMANCE BENCHMARK OF EMBEDDED WEBASSEMBLY RUNTIMES ON ARM32.

Wasm Runtime	Coremark
wasm-micro-runtime Interpreter	32
wasm-micro-runtime AoT	611
Native	1157

new features are developed. Anyhow, it compiles to baremetal Rust.

wasm-micro-runtime is the runtimes with the most supported embedded platforms and execution modes, so that we have further evaluated it for various use cases.

A. Footprint

Table II compares the footprint of wasm-micro-runtime in different execution modes on a RISC-V 32-bit platforms (ESP32 C3). As expected, the interpreter is generally larger, with performance improving methods leading to a larger code size. AOT can save around 25% of code size of the runtime. It has to be noted that those numbers are upper limits: As modes are embedded into the target code, unused features and further code size optimizations will further reduce the sizes, along with the fact that RISC-V is considered to have a larger footprint then Arm for example.

B. Benchmarking

As shown in Table III we ran the Coremark benchmark on an AllWinner V3S MCU, with 3 different configurations of wasm-micro-runtime. As expected, the wasm interpreter is quite slow. However, in AOT mode, the performance is approximately 50% of native, which is quite acceptable for our use cases.

IV. OPPORTUNITIES AND CHALLENGES OF WEBASSEMBLY IN EMBEDDED SYSTEMS

As mentioned before, there are plenty of opportunities of using application virtual machines in embedded systems, in particular due to the portability and strong isolation properties. Anyhow, as the benchmarks show there are still challenges ahead. In the following we will summarize the key observations.

Opportunity: Portability

Portability is one of the key arguments for virtual machines in general. While porting a container between different x86 computers with similar conditions is ubiquitous, container solutions comparable to Docker are still lacking for embedded systems. Due to platform heterogeneity, much of the code is not portable - if the same functionality is required on an IoT device with a different instruction set or operating system, it must be modified and rebuilt.

Furthermore, embedded systems code tends to be statically linked, meaning that to make any change to an application, the entire system must be rebuilt and reflashed. WebAssembly applets can instead be easily deployed, executed from flash memory and transferred between platforms.

Beyond the target platforms there is a second portability argument to be made: Typical development on embedded systems is done in C, which excludes a significant number of developers. Allowing the (re-)use of code in different source languages with a standardized, clear compilation environment to WebAssembly can be leveraged in terms of productivity.

Opportunity: Isolation

Most embedded systems do not have virtual memory. Fortunately, other memory protection methods are becoming widely adopted, but such methods are often limited and still not available in low end microcontrollers. WebAssembly and other application virtual machines are ideal for such devices, but require AOT and JIT methods to be carefully implemented.

Real Time Operating Systems are often not designed for multi-user scenarios, therefore the system calls are not well secured. Again, running 3rd party code isolated by application virtual machines helps mitigating this problem. WebAssembly and available runtimes can benefit from the experiences of firewalling in JavaCard and similar established isolation methods.

Challenge: Runtime performance

As observed before, a major challenge of application virtual machines is the runtime performance of application VMs. The impact obviously depends on the execution mode of the runtime. Interpreters are often limited by their nature, so that JIT techniques are promising. AOT solves the tradeoff between runtime performance and footprint, but needs to consider underlying protection mechanisms. Isolation properties of hardware platforms and the acceleration by hardware extensions are interesting in this context.

Challenge: Application management

There are many opportunities of portable applets, such as building complex applications that can be easily deployed on a variety of IoT platforms. It gives the opportunity to switch between vendors quickly. But there needs to be a level of trust before installing third party applications. Protocols and methods to load and run trusted applets need to be established. But it is not mandatory to reinvent such methods, standards such as Global Platform [14] or similar can be adopted instead.

Furthermore, deploying large scale fleets of IoT applets onto a variety of platforms needs robust dependency management and standardized 3rd party applets that serve as hardware abstraction layers. Tools and frameworks for the management and maintenance comparable to Toit [15] can deliver scalable, secure IoT management platforms, that significantly improve productivity.

Challenge: Target WebAssembly to Embedded Systems

There are a few hurdles in WebAssembly that can be limiting for embedded systems with strict resource constraints. For example the minimum memory size of 64kB, typing of data on the stack or the absence of small integer types are often cited as limitations [16]. Work on revisions of the spec can address those issues, while non-standard runtime extensions are found nowadays.

V. CASE STUDY

As discussed throughout this paper we believe in the future of WebAssembly as application VM in embedded systems. In the following we demonstrate the applicability to real-work use cases with a case study from the field of deep learning in IoT.

A. Scenario

We introduce a new class of IoT devices called *vision based sensors*. These are meant to use visual sensing modality, but they are not for *imaging* but rather for *sensing* meaning that they extract metadata about the environment. Depending on the exact deployment scenario, such devices have a variety of constraints, such as power consumption, communication, and cost. Privacy may also be a constraint in some cases. These constraints dictate that most or all of the signal processing is done at the edge, on the device, so that less information is transmitted to the cloud. There may be a variety of embedded hardware platforms, each suited for a particular set of constraints. For example, some devices may need to be battery powered, in order to be deployed in the field, while others may require greater computational power, for their particular type of scenario. Furthermore, as we are working with multiple SoC (System on a Chip) vendors, there are a variety of ISAs (Instruction Set Architectures) to deal with.

B. Model

We model the processing as a pipeline of steps, most of which run on the application processor. See V-B. The pipeline starts with the raw sensor input, which is then processed by an ISP (image signal processor), after which the DNN (deep neural network) inference is run, on the application processor or on a dedicated accelerator, a DSP (digital signal processor). Subsequent steps include normalizing the DNN output to a task specific representation, and then running *Business Logic*. The Business Logic step requires the most flexibility, as it can be arbitrarily programmed by a developer.

This type of sensing device needs OTA (over the air) programmability, because it's function and mode of operation may be changed after deployment. For example, changing the device task from counting cats to counting dogs. Wasm gives us a safe way to change the processing pipeline, while maintaining near-native performance.

Using WebAssembly applets for the sensing pipeline stages is solves multiple problems, including isolation, runtime reconfiguration, and portability among platforms.

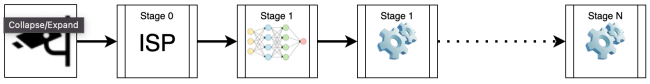

Fig. 2. Example of a pipeline

VI. OUTLOOK

WebAssembly is an interesting application virtual machine for embedded systems. While runtimes still suffer from expected performance limitations, the mostly open source work in the WebAssembly ecosystem puts a spotlight on it for consideration of containers in embedded systems.

Beyond groundwork on runtimes and their performance, we anticipate the standardization of APIs compatible to WASI in the foreseeable future. The adoption of embedded system APIs will be in the focus of our work, for example for image sensor and the ISP, or device and applet management. The interface types `wit` and binding generators are interesting in this context, as they easy multi-language support and compatibility.

Finally, we believe that frameworks and tools for the efficient management of large fleets of heterogeneous systems, including IoT devices, their root-of-trust extensions, edge devices and the cloud will emerge and are actively working on such.

WebAssembly is not the silver bullet for embedded systems, but projects can already benefit when performance is not key. Runtimes focused on embedded systems can be expected to further reduce the gap to native performance. Overall, WebAssembly has the potential to have an impact on the containerized future of embedded systems.

REFERENCES

[1] M. V. Woodward and P. J. Mosterman, "Challenges for embedded software development," in *2007 50th Midwest Symposium on Circuits and Systems*, 2007, pp. 630–633.
[2] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.
[3] M. Schoeberl, "A java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1-2, pp. 265–286, 2008.
[4] "WebAssembly," https://webassembly.org/, accessed: 2022-03-08.
[5] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 185–200. [Online]. Available: https://doi.org/10.1145/3062341.3062363
[6] "Bytecode Alliance," https://bytecodealliance.org/, accessed: 2022-03-08.
[7] "Standardizing WASI: A system interface to run WebAssembly outside the web," https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, accessed: 2022-03-08.
[8] "Tweet by Solomon Hykes," https://twitter.com/solomonstre/status/1111004913222324225, accessed: 2022-03-08.
[9] "wasm3," https://github.com/wasm3/wasm3, accessed: 2022-03-08.
[10] "wasmtime," https://wasmtime.dev/, accessed: 2022-03-08.
[11] "WebAssembly Micro Runtime," https://github.com/bytecodealliance/wasm-micro-runtime, accessed: 2022-03-08.
[12] "wasmer," https://wasmer.io/, accessed: 2022-03-08.
[13] "wasmi," https://github.com/paritytech/wasmi, accessed: 2022-03-08.
[14] "Global Platform," https://globalplatform.org/, accessed: 2022-03-08.
[15] "Toit," https://toit.io/, accessed: 2022-03-08.
[16] "GitHub Issue: Embedded devices: i8/i16 and memory pages less than 64KiB," https://github.com/WebAssembly/spec/issues/899, accessed: 2022-03-08.