The responsiveness and deployment of WebAssembly runtimes in Cyber-Physical Systems

Ian Marsh
RISE, Research Institutes of Sweden, AB
ian.marsh@ri.se

Remo Scolati
KTH, Sweden
scolati@kth.se

***Abstract*—Edge computing is being deployed to satisfy the low-delay requirements of cyber-physical systems (CPS). The network edge consists of heterogeneous and frequently resource-constrained hardware sometimes operating under real-time constraints. Furthermore, the need to develop, support & fix code on different platforms is important when operating CPSs that run tasks continuously (e.g. welding), whence initiated.**

WebAssembly, extracted from browser world, has been standardised, and in some cases, edge-enhanced. WASM generated code from high-level languages can run on several compute architectures. Therefore, a "write once, run everywhere" solution fits CPS perfectly.

We consider the time-critical application of robotics and show that WASM is between *1.2 and 1.5* times slower than natively-compiled binaries. From a 100ms E2E latency requirement, extracted from a commonly used polling interval, we also show where a robotics application fulfils (or not) this requirement in cloud-edge deployment cases.

I. Introduction

Several high-level languages can be compiled to WebAssembly bytecode: C++, Go, Python, and Rust, amongst others [2]. WebAssembly, also known as WASM, was initially developed to provide a lightweight and performant compilation target for Web-based applications [7]. We postulate it fulfils *most* needs for CPS systems, namely, responsiveness, portability, and openness. The research question this paper addresses is what performance penalty WASM introduces. Therefore, as an example of WebAssembly in-situ, consider figure 1, where a robotics application has been introduced as part of a cloud-edge platform for industrial manufacturing.

Hockley and Williamson explored the use of `wasmer`, a WASM runtime, as a sandboxed environment for general-purpose runtime scripting [8]. They focused on micro-benchmarking and one macro-benchmark to compare execution times of JiT WASM & native execution(s). They state a ×5–10 performance penalty for WASM in this case, note, JiT = Just in time compilation. Jangda et. al measured WASM's performance and implemented a framework for emulating the UNIX kernel in browsers[1] for evaluation. They compared WASM to native code [9]. They found a ×1.45–1.55 slowdown compared to native binaries using the SPEC CPU benchmark suite. This correlates quite closely to our findings, even more closely when one considers the performance increase of modern runtimes. Yan et. al found that JiT optimisations

[1]`Browsix-Wasm`.

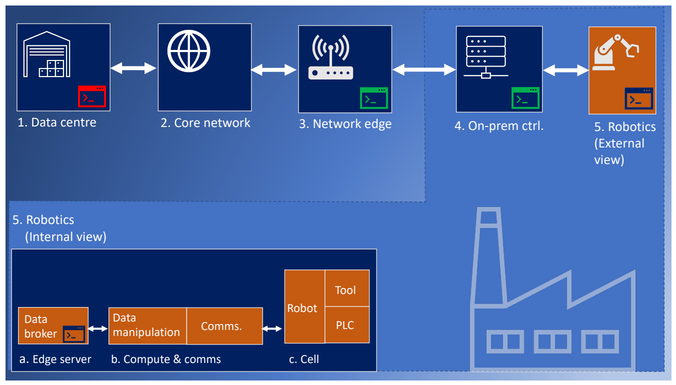

Fig. 1. A *networked* factory with a robotics application. Dark blue blocks represent network operations, whilst orange blocks represent robotics. Red-green-blue prompt-like boxes indicate deployment locations for WASM and green possible edge locations. Note the internals of an industrial robot (5).

were mostly ineffective for WebAssembly, obscured across JS implementations (e.g V8 in Chrome) & the OS-hardware [13]. Note this is not the same as AoT (Ahead of Time compilation). The same authors found a significant memory overhead for WASM applications when compared to Javascript. Napieralla in 2020 compared WebAssembly and Docker for deployment on constrained IoT devices in terms of capabilities and performance [10]. The work uses a number of benchmarks to compare the performance of the Wasmer runtime with both native execution and Docker containers as alternative(s). They state WASM compiled binaries take twice as long to complete tasks compared to native binaries, however the startup time for a WASM runtime is 10% of a Docker application. Given WASM is faster than Docker is no surprise, but that it is ×10 faster is significant, especially where startup speed is needed.

II. A robotics application: welding

In this work, we consider the application of a robotic arm connected to the Edge-Cloud. The arm's task is to weld a metal segment, a common off-the-shelf industrial task. The movements of the robot arm are controlled by known algorithms, one which calculates the position of the arms, **or** one which calculates the angles between the arms, or joints, to move the robot's 'fingers' to a desired location. As with human limbs, there are several different movements that can get to the *same* location. Additionally, there are points we cannot reach, for example touching one's elbow for a moving arm. Reachable or not, calculating these movements, forward (arms) or reverse (joints) kinematics form our *workload.*

In practice, an application loads the robot cell and validates the program using a known checksum and the architectures and code match. Then a two-step process follows i) calculate the motion path, the target locations, the robot speed & acceleration, dependent on the robot's features. A key factor in kinematics is how many arms & joints the robot arm has, called the Degrees of Freedom (DoF). ii) using the angle-orientated inverse kinematics (IK) to establish how the joints should move. The IK per millimeter, positions, speeds & accelerations using the programmed & robot mechanical constraints are then uploaded and run [11]. The task is usually repeated many thousands of times.

An interchange format for robot control is called the Unified Robotics Description Format (URDF), part of the Robot Operating System (ROS). It is ubiquitous in this field and used almost universally. From a programming point of view a Rust application[2] was used to read the specification, calculate the inverse kinematics, and instruct the (software) arm to move.

Task → URDF → Rust → wasm → IK → workload → timings.

III. Webassembly

A. The WASM ecosystem

The top five WebAssembly applications are web development, serverless, containerisation, plugins and IoT [6]. Nearly all are relevant to a CPS deployment [12]. A WASM *module* contains the application code as well as a specification on how much memory the code needs, type declarations, and externally callable functions in the module. A WASM *runtime* is a bytecode interpreter that executes the WebAssembly code. WASM runtimes execute modules in a secure sandbox without access to system services and networking.

CPS application = Robot application + a WASM runtime.

B. The WebAssembly System Interface (WASI)

WebAssembly lacks networking in the core specification. WASI provides a mechanism to communicate using system calls & sockets. The programming interface breaks out of the WASM sandbox and accesses host system resources, sockets and the file system [3], [4]. It is done by an Application Binary Interface (ABI) plus an Application Programming Interface (API) and uses WASM to call POSIX-compliant kernel system calls in a platform independent way via a foreign function interface (FFI).

[2]https://docs.rs/k/latest/k/

C. Three runtimes: Wasm3, wasmer, & wasmtime

Of the 20 active frameworks listed on Github [1], we selected just three, shown in Table I. **i)** `wasmtime` is supported by the Bytecode Alliance, and is the most widely used general purpose runtime, focusing on standards and stability whilst also offering good JiT performance, using the Cranelift JiT from Lucet [6]. **ii)** `Wasmer` is a standalone WebAssembly runtime for running WebAssembly outside of the browser, supporting WASI and Emscripten. It can also utilise either Singlepass's or `llvm`, two distinct compilers. **iii)** `Wasm3` advertises itself as the fastest WebAssembly interpreter, and the most universal runtime[3], and thus sacrifices performance in favour of portability and startup times. It is under the Cloud Native Computing Forum (CNCF) initiative.

WASM runtime	**Vers.**	**Lang. support**	**Frame-work**	**JiT**	**AoT**	**Sys. Int.**
Wasm3	0.5.0	C	-	N	N	-
Wasmer	3.1.1	C++,G,R,P	C / L	Y	Y	WASI
Wasmtime	6.0.0	C, C++, R	C	Y	Y	WASI

TABLE I
SELECTED WASM RUNTIME & FEATURES [1]. C, C#, C++, RUST, PYTHON AND GO ARE INDICATED. PROP=PROPRIETARY. FRAMEWORKS C=CRANELIFT AND L=LLVM TOOLCHAIN. JIT=JUST IN TIME COMPILATION, AOT=AHEAD OF TIME COMPILATION.

IV. Results

Before we present the results, one clarification is that we used two different robot *types.* Labelled as 'arm' and 'torso' below the robots differ slightly in their Degrees of Freedom (DoF) and movement constraints. As mentioned before, the DoF and constraints makes calculating the kinematics differ, a little, but not significantly.

A. Local benchmarks

Figure 2 shows the (average) execution times, grouped by workload and placement, relative to the native implementations. We can see that the Wasm measurements are within ×1.9–×2.2 the times recorded for the native tasks.

The results measured for a local deployment are consistent with the performance ratio of Wasm to native execution times found in similar studies, namely [5], [10]. The gap between our results and those shown by [8], who found a ×5–10 difference, are most likely due to implementation choices since the measured tasks in our project do not require any interaction with resources outside the runtimes' environment.

B. Networked benchmarks

The end-to-end response times, for the chosen implementations, running on the far edge are shown in Figure 3. The box plots show a box extending from the first to the third quartile of the data, with a line at the median and a green point at the mean. The whiskers extend within ×1.5 the inter-quartile range from the box, while outliers beyond that range are plotted as individual points.

[3]https://github.com/wasm3/wasm3

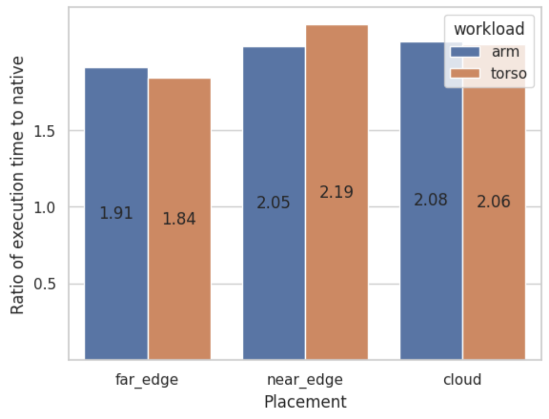

Fig. 2. Local benchmarks, mean Wasm execution times for robotics workloads, relative to mean native execution times.

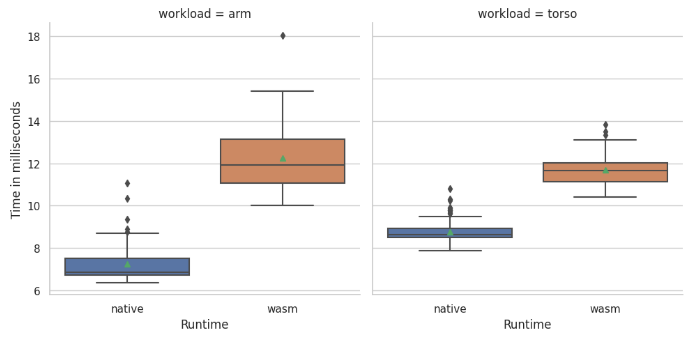

Fig. 3. End-to-end benchmarks, robotics workloads on the far edge device.

Figure 4 shows the (average) response times, grouped by workload and placement, relative to the native response times. As a general trend, we can see that the overall performance overhead introduced by Wasm gets increasingly negligible with growing device performance and network overhead.

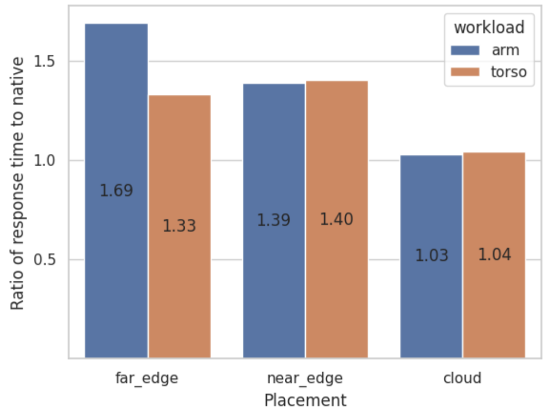

Fig. 4. End-to-end benchmarks, mean Wasm response times for robotics workloads, relative to mean native response times.

C. Just in Time (JiT) & Ahead of Time (AoT) compilation

Figure 5 shows some of the results measured during preliminary tests using non-embedded Wasm artefacts on the far edge device. The results are recorded using the Wasm runtimes to execute several test cases as standalone applications. The results include the execution time for two applications performing a robotics task **i)** compiled and executed as a native binary, **ii)** compiled as a Wasm application and executed in the Wasm3 interpreter and both the Wasmer and Wasmtime runtime in JiT compiled mode, and **iii)** pre-compiled using the Wasmer and Wasmtime CLI tools to compile and execute the applications in AoT compiled mode. The measured execution times for both robot applications executed in native and AoT-compiled mode are below 100ms, with a significant difference between Wasm and native performance, especially for the Wasmer runtime. The measured execution times for both robot samples executed with Wasm3 are, on average, around $\times 20$ longer than the native applications, in both cases exceeding the selected upper response time limit.

The significantly worse JiT performance is most likely due to the complexity of the (non-optimised) Wasm artefacts, which impacts the code analysis and generation. Both Wasm runtimes perform almost $\times 10$ worse in terms of execution time when comparing the JIT execution of an empty application that includes the robotics dependencies with an identical application without any dependencies.

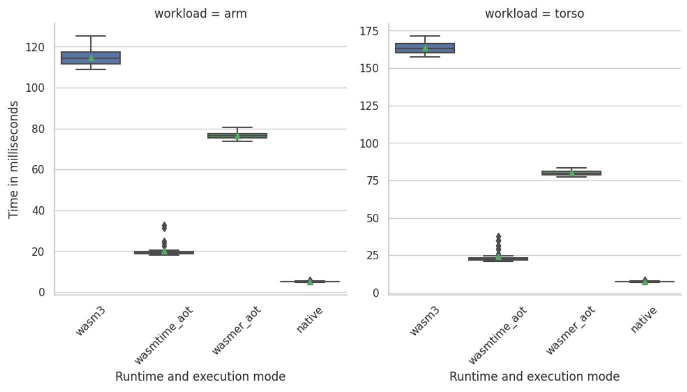

Fig. 5. Standalone benchmarks, selected execution times for robotics.

D. A 100ms requirement

Figure 6 shows a 100ms deadline with respect to our network, WASM and 'other' delays. The times needed for this URDF move on a Raspberry Pi4 in `wasmtime` were 114 ± 4 ms. This is compared to 77ms ± 2 ms in native mode, some $\times 1.4$ slower. As of 2023, Single Board Computers are some 50% faster[4] faster than those of 2019. That is the case for the popular Raspberry Pi 4 and upcoming Pi5.

[4]https://www.elektormagazine.com/news/raspberry-pi-5-vs-raspberry-pi-4-comparison

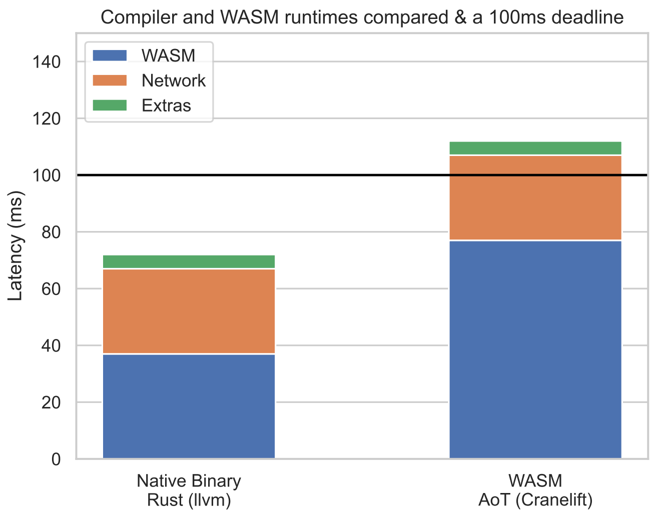

Fig. 6. Total E2E application delays. Blue: the `wasmtime` runtime + IK workload. Orange: non-wasm and non-network delays (e.g. system calls). Green: Cloud-to-Device network measurements, determined by `mtr`. A 100ms requirement given by robotics polling intervals, the horizontal black line.

V. Discussion

The discussion around being able to place apps anywhere is, can one improve the performance of the entire system? However, it is pertinent to add performant software that can only run on the (robot) edge on relatively low-powered hardware might be easily balanced out by using a slower task in the cloud. Where the bottleneck resides is a consideration, e.g. processing large amounts of data which can come from various places then having the app running next to the data source could easily outweigh the performance penalty. Performance-wise a WASM slowdown is not optimal however when it represents a slowdown. We are looking at these issues too.

VI. Conclusions

We have microbenchmarked three wasm runtimes as of Nov. 2023 in this paper. We have considered a robot arm application as a use case, and in particular, the time needed to weld a metal segment autonomously. Moving a robot arm correctly and efficiently requires some calculations, as we have indicated using kinematics. The calculation of the arm's position represents a real workload. We have considered a couple of robot types, which differ in DoFs & movements, but this requires more work.

We have also considered placement of the application and runtimes, as in Cloud-Edge networking movement & placement is often touted as an advantage in a telecommunications network. This is one reason for choosing WebAssembly since it can run on many CPUs, operating systems and hardware, as stated *"write once, run everywhere"*. We have tested WebAssembly in a standalone local environment and in a networked setting with realistic end-to-end delays. We have used a simple benchmark to see if a wasm-enabled application would violate a 100ms requirement.

We concur with the related work, that WebAssembly does incur *some* performance penalty, about $\times$1.2-1.5 slower than a native binary. However, in a networked setting, this is subsumed by the E2E delays. Looking over a number of versions of the runtimes, the performance difference between wasm and the native binaries is closing. This is very much due to compiler improvements and the wasm virtual machine being improved. This compares to the Java world, where the JVM executes at the speed of a native binary today.

Given our results, we show that the overhead introduced by Wasm is not prohibitive for the selected application and that the overall delay of the Wasm implementation is usually within latency requirements. Thus, we conclude that Wasm could benefit and enable several use cases due to its secure architecture, performance, flexibility, and portability.

Acknowledgements

We acknowledge the CELTIC office with project ID C2019/3-2 in Germany and Vinnova AB-Sweden, under project name Automation of Network edge Infrastructure & Applications with aRtificiAl intelligence, ID 2020-00763.

References

[1] S. Akinyemi, "Awesome webassembly runtimes," May 2022. [Online]. Available: https://github.com/appcypher/awesome-wasm-runtimes
[2] Appcypher, "Awesome webassembly languages," Oct. 2022. [Online]. Available: https://github.com/appcypher/awesome-wasm-langs
[3] W. Authors, "Wasi: The webassembly system interface," Oct. 2022. [Online]. Available: https://wasi.dev
[4] L. Clark, "Standardizing wasi: A system interface to run webassembly outside the web," Mar. 2019. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/
[5] F. Denis. Performance of WebAssembly runtimes in 2023. [Online]. Available: https://00f.net/2023/01/04/webassembly-benchmark-2023/
[6] C. Eberhardt, "The state of webassembly," 2022. [Online]. Available: https://blog.scottlogic.com/2022/06/20/state-of-wasm-2022.html
[7] A. e. a. Haas, "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: ACM, 2017, p. 185–200. [Online]. Available: https://doi.org/10.1145/3062341.3062363
[8] D. Hockley and C. Williamson, "Benchmarking runtime scripting performance in wasmer," in *ICPE*, ser. ICPE '22. Association for Computing Machinery, 2022, pp. 97–104. [Online]. Available: https://doi.org/10.1145/3491204.3527477
[9] A. Jangda, B. Powers, A. Guha, and E. D. Berger, "Mind the gap: Analyzing the performance of webassembly vs. native code," *CoRR*, vol. abs/1901.09056, 2019. [Online]. Available: http://arxiv.org/abs/1901.09056
[10] J. Napieralla, "Considering webassembly containers for edge computing on hardware-constrained iot devices," Master's thesis, Faculty of Computing, Blekinge Inst. of Tech., Karlskrona, Swe, 2020. [Online]. Available: http://bth.diva-portal.org/smash/record.jsf?pid=diva2:1451494
[11] L.-C. Wang and C. Chen, "A combined optimization method for solving the inverse kinematics problems of mechanical manipulators," *IEEE Trans. Robotics and Automation*, vol. 7, no. 4, pp. 489–499, 1991.
[12] G. Wikström *et al.*, "6g – connecting a cyber-physical world: A research outlook towards 2030," *Ericsson, White paper, Nov*, 2022.
[13] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the performance of webassembly applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 533–549. [Online]. Available: https://doi.org/10.1145/3487552.3487827