KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Measuring the responsiveness of WebAssembly in edge network applications

REMO SCOLATI

Stockholm, Sweden, 2023

Measuring the responsiveness of WebAssembly in edge network applications

REMO SCOLATI

Master's Programme, Software Engineering of Distributed Systems, 120 credits
Date: June 29, 2023

Supervisors: Viktoria Fodor, Ian Marsh
Examiner: Amir H. Payberah
School of Electrical Engineering and Computer Science
Host organization: RISE, Research Institutes of Sweden, AB
Swedish title: Mätning av responsiviteten hos WebAssembly i edge network-applikationer

d|

Abstract

Edge computing facilitates applications of cyber-physical systems that require low latencies by moving compute and storage resources closer to the end application. Whilst the edge network benefits such systems in terms of responsiveness, it increases the systems' complexity due to edge devices' often heterogeneous and resource-constrained nature. In this work, we evaluate whether WebAssembly can be used as a lightweight and portable abstraction layer for such applications. Through the implementation of an edge network robot control scenario, we benchmark and compare the performance of WebAssembly against its native equivalent. We measure WebAssembly's overhead and assess the impact of different placement options in the network. We further compare the overall application responsiveness against the latency requirements of an industrial application to evaluate its performance. We find that WebAssembly satisfies the portability and performance requirements of the selected industrial use case. Our empirical results show that WebAssembly doubles the execution latency in a localized setting, but does not excessively impact the overall responsiveness of a cyber-physical system.

Keywords

Cloud Computing, Edge Computing, Internet of Things, WebAssembly

Sammanfattning

Edge computing underlättar tillämpningar av cyberfysiska system som kräver låga latenser genom att flytta beräknings- och lagringsresurser närmare slutapplikationen. Även om edge-nätverket gynnar sådana system när det gäller reaktionsförmåga, ökar det systemens komplexitet på grund av edge-enheternas ofta heterogena och resursbegränsade natur. I detta arbete utvärderar vi om WebAssembly kan användas som ett lättviktigt och portabelt abstraktionslager för sådana applikationer. Genom att implementera ett robotkontrollscenario för edge-nätverk benchmarkar och jämför vi prestandan hos WebAssembly med dess inbyggda motsvarighet. Vi mäter WebAssemblys overhead och utvärderar effekten av olika placeringsalternativ i nätverket. Vi jämför även den övergripande applikationsresponsen mot latenskraven i en industriell applikation för att utvärdera dess prestanda. Vi konstaterar att WebAssembly uppfyller portabilitets- och prestandakraven för det utvalda industriella användningsfallet. Våra empiriska resultat visar att WebAssembly fördubblar exekveringslatensen i en lokaliserad miljö, men att det inte påverkar den övergripande responsiviteten i ett cyberfysiskt system i alltför hög grad.

Nyckelord

Cloud Computing, Edge Computing, Internet of Things, WebAssembly

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

ABI	Application Binary Interface
AOT	Ahead-of-Time
API	Application Program Interface
CLI	Command-Line Interface
CPS	Cyber-Physical System
DoF	Degrees of Freedom
FK	Forward Kinematics
IIoT	Industrial Internet of Things
IK	Inverse Kinematics
IoT	Internet of Things
JIT	Just-in-Time
WASI	WebAssembly System Interface
Wasm	WebAssembly

Chapter 1

Introduction

Cyber-physical systems (CPS) consist of sensing, connectivity, and control technologies that allow an object or machine to monitor and act in a physical environment. cyber-physical applications play an increasingly important role in industrial processes and decision-making, in particular in industrial Internet of Things (IIoT) implementations. IIoT describes the industrial application of IoT technologies, using networks of connected sensors, devices, and objects. IIoT employment sectors include healthcare, automotive, manufacturing, mining, and many other sectors and associated domains. Example applications are self-driving cars sensing traffic and obstacles, or industrial robots in automated production processes.

Since many IoT devices are typically limited in size and power, many use cases are enabled or enhanced by the availability of cloud resources and improvements in network technology [1]. The centralized cloud approach can, however, be a bottleneck, especially for time-sensitive applications due to network latencies as well as the amount of data collected and processed [2]. Edge computing benefits several use cases by shifting compute and storage away from the cloud, making use of resources available in closer proximity to producers and consumers [3]. The main advantage of moving the compute-storage-communication is lower latency, especially in near real-time applications that need high responsiveness. Distributed edge applications can further make more efficient use of the storage and compute resources that are available in a network of edge devices [4].

The edge network might, depending on the use case, include several heterogeneous devices, ranging from embedded devices in cars and manufacturing machines, appliances, and mobile phones, to an internet service provider's network infrastructure and data centers. In the context of this

document, the network's edge is intended as networked devices in close proximity to the intended industrial application; more specifically, we consider both an on-premise edge server and a resource-constrained device as possible placement options for a robot control application.

1.1 Domain problem

One of the main problems of CPS implementations relying on the edge network is the heterogeneity of the edge devices, which ranges from small 8-bit microcontrollers to rack-mounted servers, where the smaller scale devices are often more limited in processing power due to size and power constraints. The diversity of devices is reflected in several different platforms for which applications need to be developed and maintained. Distributed applications in an edge network must account for the range of capacities and the different platforms of the devices. Portability, that is, the ease with which a program can be run on different platforms, is, therefore, a critical requirement for many distributed edge applications. Many distributed applications, and CPS in particular, require high responsiveness, as well as safe and portable code [4, 5]. Typically, there is a trade-off between the portability and performance of an application when comparing, for example, native binaries and languages targeting portability through an abstraction layer such as interpreters or virtual machines.

1.1.1 WebAssembly as a portable edge runtime

Wasm is a bytecode format designed as a compilation target for multiple programming languages and originally developed for the web [6]. As a lightweight virtualization layer, it is a portable alternative to cross-compiled or interpreted edge applications that allows compiling source code from many different languages into a portable intermediate format. Wasm was specifically designed to execute untrusted applications inside a safe sandbox, protecting the host's environment and resources—possibly shared with other applications—from malfunctioning or malicious code. Its security model provides multiple features for developing safe applications. The portability and flexibility it promises, together with its security model and the stated design goal to offer near-native code performance, make Wasm a promising solution for edge network architectures. Due to its potential in the domain, we intend to assess the usability of Wasm in an edge network CPS.

1.1.2 Research questions

Given the described problem area, this work aims at answering the following research questions.

RQ1 Could Wasm be used to operate an industrial robot application?

RQ2 How much delay does Wasm introduce in an edge-cloud robotic setting in comparison to native binaries?

We hypothesize that Wasm is a viable technology to be employed in a cyber-physical system, and does not introduce an excessive latency overhead in the end-to-end delay, measured against the latency requirements from an industrial use case, and therefore to the responsiveness of a robot arm application.

1.2 Purpose and goals

Through this project, we evaluate whether Wasm can enable and facilitate edge computing use cases. The high-level objective of the project is to evaluate whether Wasm is a viable alternative to native applications in an edge-cloud context. The main goal of this project is to evaluate the responsiveness of a CPS employing Wasm and compare it to a native implementation. We aim to present an overview of the current Wasm ecosystem and describe the usability of the technology as a portable edge network runtime. We further aim to illustrate a possible implementation of an edge network system and to produce recommendations useful to both an industrial application and further research.

1.3 Delimitations

The project forgoes an in-depth comparison and analysis of compilation options since extensively covered in other works. A performance trade-off is expected for Wasm compared to a baseline, but a detailed analysis of the reasons leading to differences in performance is out of scope for this project. This project aims to measure the overall latency Wasm introduces in an example implementation of a CPS and to compare the latency against the performance requirements posed by an industrial application.

1.4 Research methodology

The evaluation of our research questions is based on a quantitative analysis of empirical data collected through experiments on a (simulated) CPS. RQ1 is addressed through an evaluation of the performance, given the experimental results, platform support, and features in general, in terms of the requirements posed by the selected robot arm control scenario. We address RQ2 through the evaluation of the experimental results measured in an implementation of the robotics system. We compare metrics such as execution time and end-to-end responsiveness to a baseline (native binaries), to measure the additional delay introduced by Wasm. We further compare and validate our results with existing literature. A more detailed discussion of the methodology is presented in Chapter 3.

1.5 Ethics and sustainability

Several ethical and sustainability issues can emerge from the use of IIoT. Increasingly large quantities of timely and accurate data drive decision-making and processes in industrial applications. The large amount of collected and transferred information raises privacy concerns, while latency and throughput requirements lead to increased energy and resource demands for network and cloud infrastructure. The locality of edge network architectures can help mitigate some of the privacy and sustainability concerns by transforming and processing information closer to data sources and consumers. It can help to ensure the timeliness of data by reducing latency and easing the strain on core network infrastructure. By facilitating and enabling edge network architectures, Wasm can help increase the efficiency of CPS, allowing device networks to share storage and compute resources and capabilities. The project relies on empirical data recorded from a simulated workload; as such it poses no privacy issues.

1.6 Contributions

This thesis provides the following main contributions:

- We describe different placement and deployment options for Wasm in an edge network architecture.

- We measure the performance overhead introduced by Wasm in a cyber-physical system, comparing it against alternatives and industrial requirements.
- We evaluate the relevance of Wasm as a portable virtualization layer for industrial applications and its usability in the domain.

1.7 Structure of the report

In Chapter 2, we introduce the key concepts encountered in the remainder of this report. We briefly describe both edge computing and Wasm to the extent relevant to this work and describe some of the concepts from robotics used throughout our implementation. We further summarize related and comparable literature and their main findings. In Chapter 3, we outline the design rationale, considered options, evaluation, and key factors of our work. We describe the implementation of our experiments and the major technical choices. In Chapters 4 and 5, we present the main empirical results of this project, discuss the broader implications of the results and evaluate the usability of Wasm in an industrial edge computing application in terms of benefits and requirements. In Chapter 6, we summarize our main conclusions and present several recommendations for future research topics.

Chapter 2

Background

The major technologies and background areas discussed in this project are cyber-physical systems and robotics, edge cloud, and Wasm. In this chapter, we briefly introduce the main concepts and then present a short overview of related works that analyze the performance of Wasm in various contexts.

2.1 Technical background

2.1.1 Cyber-physical systems

A cyber-physical system integrates sensors and actuators, control and connectivity into physical infrastructure and objects. The availability of cost-effective sensors and advances in network technology have enabled several use cases in which networked devices can sense—and interact with—the physical world, creating increasingly complex CPS. Examples of CPS include many IoT and IIoT applications, like self-driving cars and smart devices, as well as smart grids and smart manufacturing. Many implementations leverage the storage, performance, and flexibility of cloud platforms since the capabilities of IoT devices are typically limited due to size and power limitations [1, 7]. The traditional centralized cloud architecture has, however, quickly become a bottleneck, especially for time-sensitive applications, due to low latency requirements [2].

2.1.2 Edge computing

Performance requirements as well as sustainability considerations have led to the development of technologies to replace or complement cloud-based

systems, with varying degrees of decentralization [2, 8]. The distributed architectural paradigm at the base of most of the proposed solutions is edge computing. The core idea, originally introduced in the context of content delivery networks, is to move computation and storage closer to the data sources to reduce latency and save bandwidth [3, 9].

The edge computing paradigm is a promising solution, especially for many novel ubiquitous computing and industrial IoT use cases, but there still exist some hurdles holding back widespread adoption. A recent survey, targeted at network operators and enterprises, shows that service providers and companies expect edge computing to be important or even critical to their business, and Telecommunication companies believe IoT and 5G to be the main drivers for the adoption of edge computing. However, concerns about ecosystem maturity and security are seen as the main inhibitors [10].

Existing studies identify portability, latency, and safety as critical requirements for underlying technologies in edge networks and CPS platforms [4, 5]. The main challenge in an edge network environment stems from the heterogeneity of the devices, which, depending on the applications, ranges from servers in edge data centers and content delivery networks to mobile phones, smart cars, appliances, and embedded robot controllers. The diverse nature and capabilities of devices, paired with the latency requirements in many CPS, make it necessary to write and support fast and portable code with a low operational footprint.

2.1.3 WebAssembly

Wasm has been previously proposed as an enabling technology in edge network architectures based on its features and capabilities. Li *et al.* [11] and Li *et al.* [12], for instance, propose Wasm for Edge and IoT applications describing its main advantages as allowing better portability and execution in sandboxed environments. They propose a programming model, *WiPROG*, and a runtime, *WAIT*, optimized for use on resource-constrained IoT devices, illustrating the benefits of Wasm-based approaches in the domain. Similarly, Nieke *et al.* [13] propose an edge service migration platform, *EDGEDANCER*, that leverages the sandboxed design, efficiency, and portability of Wasm to support secure, portable, and provider-agnostic service relocation.

Wasm is a portable bytecode format for a stack-based virtual machine. The language is designed as a portable compilation target, enabling deployment for web, client, and server applications [14]. Wasm aims to be fast, safe, portable, and hardware-, language-, and platform-independent [6]. It is an alternative to

native compilation or interpreted languages and allows to compile applications from many different languages as well as to reuse of existing tool chains. As a virtualization layer, it offers the portability required by Edge cloud applications [15]. The design and provided features make Wasm a promising solution for creating portable and modular edge cloud architectures that can be supported on a wide range of platforms.

2.1.3.1 Wasm runtimes

Wasm bytecode is designed to be executed in a portable virtual machine that can be embedded in a standalone runtime environment or a host application. A Wasm runtime is a bytecode interpreter that executes the Wasm modules in a secure sandbox without access to system services and networking. Interaction with the outside world is provided by the runtime, originally by Web APIs provided by browsers. Several Wasm runtimes are available for executing Wasm modules as standalone applications or embedded in applications and libraries written in other languages [16]. A Wasm runtime's core functionality is to translate from the portable Wasm code into platform-specific binary encoding. Runtime implementations rely on ahead-of-time (AOT) or just-in-time (JIT) compilation or interpreters to render Wasm code executable, offering one or more options for standalone or embedded execution. The various available runtimes offer a broad spectrum of features, language integrations, execution options, and compilation modes.

2.1.3.2 Security model

The main goals of Wasm's security model are to offer users protection from malicious or malfunctioning code and to provide developers with the means to develop safe applications [6]. The security model of Wasm is centered on the execution of possibly untrusted code inside a sandboxed environment [6]. Code is executed inside a memory-safe sandbox that isolates potentially malicious, or faulty, modules from the host environment and applications and data sharing the same host resources [17, 18]. Fault isolation in Wasm is achieved through memory isolation; each Wasm module is executed in a sandbox with dedicated memory, containing the effects of operations inside the sandbox. Wasm further restricts access to the execution environment on the host platform [6]. Any external functionality, such as access to host functions, I/O, or operating system calls, has to be provided explicitly by the host application or runtime. Further, all callable functions are statically predefined and isolated from an application's memory, allowing strict checks on

the application's control flow [19].

2.1.3.3 WebAssembly System Interface

The WebAssembly System Interface (WASI) is a modular system interface that provides access to features like file system and network socket access, clocks, and other operating-system-like services to Wasm modules [20]. WASI is being standardized by a subgroup of the W3C WebAssembly workgroup [21]. It implements an application program interface (API) and application binary interface (ABI) to provide fine-grained access to host system resources through any compliant runtime, exposing functions analogous in purpose to system calls. The interface aims to allow Wasm to have access to functionality outside of its sandbox in a non-browser environment in a controlled manner.

2.1.4 Use case: robot arm control

The scenario considered in this project is a CPS in which a robot's movement is controlled over a network. The step-wise movement of the robot is given by computing the robot's joint configuration for a target position at each step. A typical robot has multiple degrees of freedom (DoF), meaning one or more joints can be configured independently. Figure 2.1 shows such an example robot arm with three rotational joints that can be positioned independently. The robot can be modeled as a hierarchical chain of links and joints in a parent-child relationship in which a change in a joint is propagated to its children.

The resulting kinematic model can be used to compute the position of a joint or link (e.g., the end effector) from a joint configuration through forward kinematics (FK). That is, if we know the angles of the individual joints, we can compute where an element of the kinematic chain is in space. Inverse kinematics (IK) can be used to compute the opposite, i.e., the joint configuration necessary to place an element of the chain, e.g., the end effector, in a specific position, if such a solution exists [22]. Figure 2.1 illustrates the relationship in a simplified way in a 2-dimensional space.

The inverse kinematics problem can be solved with several approaches, for example by pre-computing all possible solutions using trigonometric methods. A popular and more dynamic solution is the use of the Jacobian matrix to find a solution iteratively [23]. The Jacobian method, used as a solution algorithm in this project, approximates a solution by iteratively introducing changes in the robot's joint positions and minimizing the error between the position of the end effector and the target.

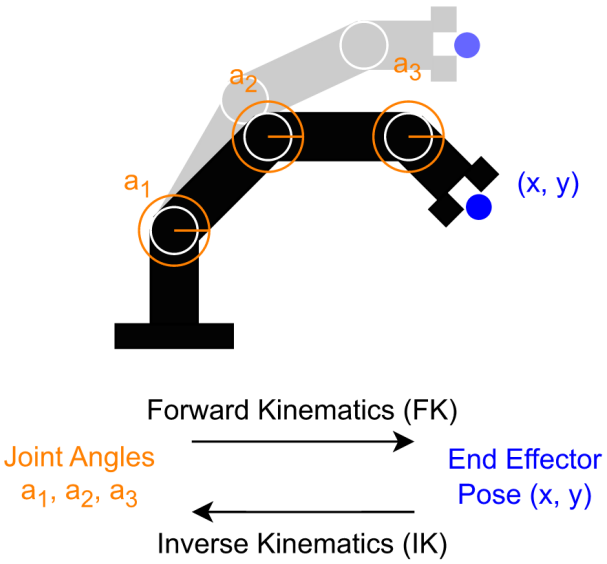

Figure 2.1: Illustration of a robot's joint configuration and the relationships between forward and inverse kinematics.

2.2 Related work

Several studies assess the performance of Wasm through a comparison to alternatives relevant to the domain, such as native code, Docker containers, or JavaScript. In this section, we summarize relevant and recent efforts toward describing the performance of Wasm.

Denis [24] compares the performance of multiple Wasm runtimes using the *libsodium* cryptography library as a benchmark, comparing the execution times of several major Wasm runtimes with native performance on a Cloud compute server instance. The benchmarks show a $\times 2.32$ (median across multiple test samples) performance disadvantage for the best-performing runtime for every single benchmark compared to native execution.

Hockley and Williamson [25] evaluate the performance of Wasm as a general-purpose server-side scripting language. They run micro and macro benchmarks, written in Rust, using the Wasmer runtime, comparing the execution times between JIT and native execution. The authors implement a Web proxy caching simulator as an example application for the macro benchmarks. They find a $\times 5$–$\times 10$ performance loss for Wasm compared to the native implementation. They attribute the performance loss to the overhead

caused by frequent calls into the Wasm runtime's sandbox.

Yan *et al.* [26] compare and analyze the performance of Wasm and JavaScript across different browser environments. The authors find JIT optimizations to be mostly ineffective for Wasm. They further find a significant memory overhead for Wasm compared to JavaScript and that performance is inconsistent for both JavaScript and Wasm between different browsers and engines.

Napieralla [27] compares Wasm and Docker for deployment on constrained IoT devices in terms of capabilities and performance. In the study, they perform several benchmarks using the *PolyBenchC* benchmark suite to compare the performance of the Wasmer runtime with both native execution and Docker containers as an alternative. They state that Wasm binaries take twice as long as native binaries and the startup time for a Wasm runtime is 1/10th that of a Docker container.

Jangda *et al.* [28] measure the performance of Wasm in browser environments. The authors implement a framework for emulating a UNIX kernel in browsers, *Browsix*, allowing a comparison of Wasm and native code. They found a $\times 1.3$ performance advantage of Wasm over JavaScript, and a $\times 1.45$–$\times 1.55$ disadvantage compared to native execution running the *SPEC* benchmark suite.

Haas *et al.* [14] feature an in-depth overview and discussion of Wasm and its design and semantics. They further compare Wasm performance with native execution using the *PolyBenchC* benchmark suite. The authors find execution times of Wasm, executed on both the V8 and SpiderMonkey engines, within $\times 2$ of native execution times.

While some papers focus on a broad set of common workloads using benchmark suites, others measure the impact of Wasm for specific use case implementations. We aim to validate and extend existing results for the selected CPS use case scenario.

State of the art summary

Wasm performance evaluations show, in general, a significant overhead when comparing Wasm with native code implementations. Most studies find that measured Wasm execution times are within a range of $\times 1.5$–$\times 2.5$ the native execution times, depending on the tested workload, execution mode, and implementation details. Table 2.1 summarizes the relevant findings of compared studies.

Study	Focus	Results
Denis [24]	Comparison of Wasm runtimes, *libsodium* cryptography library as benchmark	×2.32 native (median)
Hockley and Williamson [25]	Performance of Wasm as a general-purpose server-side scripting language	×5–×10 native
Napieralla [27]	Comparison between Docker and Wasm, *PolyBenchC* benchmark suite	×2 native
Jangda *et al.* [28]	Comparison between Wasm, native, and JavaScript, *SPEC* benchmark suite	×1.45–×1.55 native
Haas *et al.* [14]	Comparison between Wasm, asm.js and native, *PolyBenchC* benchmark suite	within ×2 native

Table 2.1: Comparison of results in related literature, Wasm execution times compared to native code execution.

Chapter 3

Method

In this study, we collect and analyze empirical data through experiments. This chapter describes the main factors we consider in the experimental design. We further describe the architecture, implementation details, and any relevant software and hardware used in our experiments, and how our results are evaluated.

3.1 The experimental design

We test two main hypotheses, we assume that i) a native implementation significantly outperforms a Wasm implementation on a local level, as shown in similar works, but that ii) the overhead introduced by Wasm in a host application is not excessive in an end-to-end scenario when compared against a latency requirement. To test the hypotheses, we perform a series of benchmarks, using a simulated robot arm control system to evaluate the performance difference between native code (as a baseline) and Wasm. Through the benchmarks' results, we compare and evaluate the different workload implementations, runtimes, application architectures, placement options, and deployment options described below; a summary is shown in Table 3.1.

3.1.1 A robotics workload

The robotics task measured in our benchmarks consists of controlling a step during a robot's movement by calculating its joint angles for a given initial and target position. The input for a task is represented by the initial joint configuration of the robot, i.e., the initial angles of its joints, and the target

position of the robot's end effector. The output is the joint configuration that allows the robot to reach the target position if such a solution exists. All measurements are performed using the same implementation of a workload library which exports the solving functionality for the implemented robot models. The solutions are computed using the Jacobian method, described in Section 2.1.4, as implemented by a third-party library. Through empirical experimentation, we found that the default configuration of the library's solver with an upper iteration limit of 100 iterations works reasonably well for the implemented robot models. We found that setting the iteration limit over that threshold does result in an increase in the time it takes the algorithm to fail for a target the robot cannot reach from a given joint configuration, with only a modest increase of found solutions, since the algorithm simply takes longer to terminate for unreachable targets.

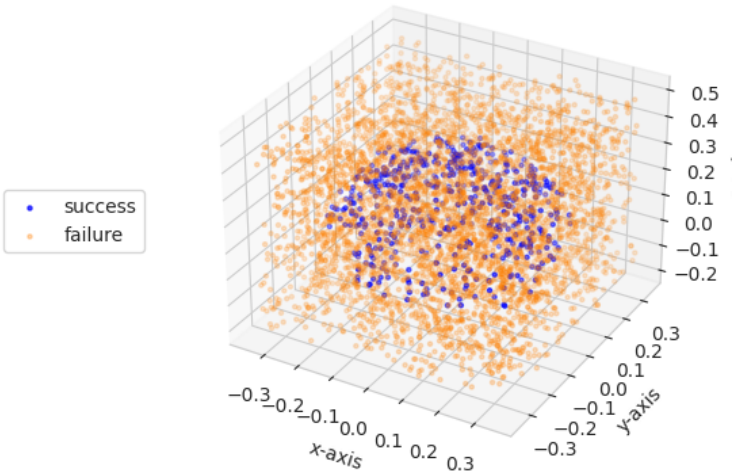

Figure 3.1: 5 DoF Robot reachability plot. Blue indicates a reachable and orange an unreachable target.

To ensure that the initial joint configurations and target positions used as input for the measured samples are evenly distributed and to avoid possibly solving only for targets with the same complexity, we generate pairs of initial joint configurations and target points randomly over the whole action radius of the robots. A set of 100 reachable and 100 unreachable combinations is used as input for the experiments. An unreachable combination, or failure, is intended as a combination of initial joint angles and target position for which the algorithm does not find a solution with the given solver configuration. That

is, the robot model cannot reach the specified target from its initial position due to joint constraints or the limits of the robot's action radius. Figure 3.1 shows for example the distribution of the target points for one of the implemented arm models in the three-dimensional space. The sample size is determined through a small number of preliminary measurements used to estimate the variance*.

3.1.2 Wasm runtimes

We select a subset of available runtimes based on their "traction", that is, user adoption and reasonable development activity [30]. The runtimes are further selected for compatibility with the WASI standard, language support, and to represent different execution options, including AOT and JIT compilation, or interpreted execution. The experiments are performed using the runtimes described below; all projects are in active development.

- Wasmer (version 3.1.1), an open-source Wasm runtime, offers both AOT and JIT compilation. Supports different compiler backends. Throughout this project, the *Cranelift* compiler is used [31].
- Wasmtime (version 6.0.0), similar to Wasmer, with comparable features. The runtime is built on the *Cranelift* code generator [32].
- Wasm3 (version 0.5.0), a Wasm interpreter, designed for small footprint and wide platform support [33]. Used only in preliminary measurements.

The Wasmer and Wasmtime runtimes offer a command-line interface (CLI) tool for running Wasm in standalone mode as well as libraries for integration in several programming languages, including Rust, C/C++, Python, and .Net. Wasm3 does not fulfill the use case requirements in terms of performance in preliminary test results (described in Section 4.1.3) and is thus excluded in the implementation of the final local and end-to-end benchmarks.

3.1.3 Application architecture

All runtimes considered in our implementation offer both a standalone CLI utility as well as a library available for many programming languages. The runtimes' CLI tools are used to compile (in AOT mode, if available) and

*Determined through the method outlined in Jain [29], assuming a confidence interval of 95% and ±5% accuracy.

either directly invoke functions on Wasm binaries or execute complete Wasm applications through the command line. Given the significant overhead caused by access of resources outside the Wasm sandbox shown in related research, we opt for embedding a Wasm runtime in a host application, even though we record some preliminary measurements using the runtimes as standalone environments to compare AOT and JIT performance [25]. The implemented architecture allows the execution of only the core functionality, that is, solving a given robotics workload, inside a guest Wasm environment.

For the host applications, we implement a guest abstraction layer as a common interface for the embedded Wasm runtimes. The Wasm guest implementation exposes the required functionality, that is, the initialization of Wasm modules and function calls. The embedded runtime reads the pre-compiled Wasm workload library during application start-up and performs instantiation steps, such as code validation, optimization, and generation, that result in an in-memory version of the Wasm library. Each function call is then executed in a separate, sandboxed instance of the module. Ideally, a host application should offer the functionality to read and update Wasm modules at runtime, but some simplification is required to have predictable interfaces for both the native and the Wasm function calls we time and compare. Section 3.2.1 illustrates the host applications' architecture and implementation in more detail.

3.1.4 Compute placement

Figure 3.2 illustrates the conceptual scenario selected for the series of benchmarks we performed. The use case represents an industrial robotics scenario, set in a factory, in which we compare the responsiveness of a Wasm implementation of a robotics task with a native implementation. The three placement options we consider for our experiments are i) a remote cloud server, ii) a near edge device, e.g., an on-premise rack server, and iii) a far edge device near the client, e.g., a control device close to the robot. The terms near and far edge are used here from the perspective of cloud providers and denote the proximity to the core network; near edge refers to the infrastructure closer to the cloud while far edge refers to the infrastructure closest to the user or end application.

The placement options are shown in Figure 3.2 as red, green, and blue markers, where green and blue markers indicate possible near and far edge locations. To assess the impact of the placement choice, we conduct measurements on three devices representing the following placement options.

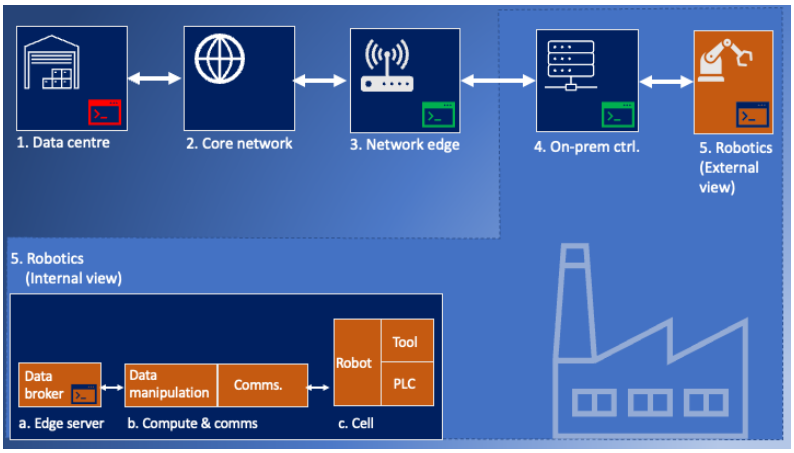

Figure 3.2: Networked factory scenario defined for the end-to-end benchmarks with placement options.

Far edge A resource-limited device located at the far edge, in close proximity to (or possibly integrated with) the client device. The far edge experiments are performed on a Raspberry Pi 4 Model B running *DietPi* (version 8.14.2), a minimal operating system based on Debian and optimized for use on SBCs [34]. The device used for the experiments comes with an ARMv8 CPU and 4GB RAM.

Near edge Represents a small server or comparable device located at the near edge, on-premise in proximity (in network terms) to the client device. The near edge experiments are performed on a local device running Ubuntu 20.04 LTS. The local device used during the experiments is a Linux notebook that runs on an x86-64 2.67GHz CPU and 4GB RAM.

Cloud A cloud device in a remote data center. The cloud experiments are performed on a VPS from DigitalOcean* running Ubuntu 20.04 LTS. The VPS used is an instance of the ”basic” offering, which comes with a (virtual) x86-64 CPU and 512MB RAM.

Client Represents the client device in the robot control system. An Odroid XU4 with an ARMv7 CPU and 2GB RAM is used as a client device during remote measurements.

The local devices (near and far edge, client device) are connected to the same, RISE-internal test network. The (average) TCP round-trip times from

*Website: https://www.digitalocean.com.

the client to the near and far edge device are measured at around 1ms and 8ms, respectively. The cloud device is located in the DigitalOcean Frankfurt region, in Germany. The average round-trip time is measured at 32ms. TCP round-trip times are measured using the *Nmap Nping* (version 0.7.80) utility [35]. All experiments are performed on an unloaded system (less than 5% CPU usage), with the bare minimum of system and user services running, to minimize the impact of other processes on the benchmark results.

3.1.5 Deployment

Even though the focus of the project lies on a networked robot control application, we perform measurements for both a local and a remote workload deployment for all placement and workload options. The latency measured in local benchmarks allows us to compare our results with similar results from related literature, while the remote application is implemented as a simple proof-of-concept for a CPS consisting of a client-server pair. We implement the same overall host application architecture with an embedded Wasm guest environment for both the local and remote benchmarks. The local and remote host applications and the remote client implementations are described in more detail in Section 3.2.1. The remote CPS is implemented through an HTTP client-server pair.

Summary

Table 3.1 summarizes the main factors we consider for the implementation of our experiments.

Factor	Considered options	Notes
Workload	Native or Wasm	Robotics tasks
Wasm Runtimes	Wasmer, Wasmtime, Wasm3	
Placement	Cloud, near edge, far edge	
Application architecture	Embedded or standalone	Applies to Wasm only
Deployment	Remote or local	
Network	LAN/WAN	Does not apply to local deployment, depends on placement

Table 3.1: 6 factors considered for the experimental design.

3.2 Implementation

All workloads and Wasm host applications are implemented in Rust. The language choice is based on Rust's affinity to the Wasm ecosystem and performance [30]. Both Wasm and native measurements are performed using two versions of the same workload library, one is compiled as a Wasm artifact, and one is imported as a native dependency. All workloads are compiled using the Rust default toolchain (version 1.67) for both the native and the Wasm artifacts. All workloads and applications are compiled using the default target architecture on the tested system (native) and the `wasm32-unknown-unknown` target architecture (Wasm) as compilation targets, and the default `release` compilation optimization. We rely on the *time* implementation in the Rust Standard library to measure the time elapsed while solving a given arm movement task. Preliminary measurements include results recorded using the *hyperfine* tool on the far edge device using the runtimes' CLI utilities to execute standalone Wasm artifacts [36].

3.2.1 Host Applications

To measure both the end-to-end response times and the Wasm overhead for solving the tasks, we implement a local and a remote host application. The remote application is used to measure the response times between the client and the server for the three placement options described previously. The local application is used to measure the execution times on the devices locally. The remote and local host applications share a similar architecture and the same Wasm guest implementation. Since the timing utility shares code between the local host application and the client, it is implemented as part of the local application.

3.2.1.1 A remote host setup

The remote CPS Wasm host application implements an HTTP API that exposes both native and Wasm function calls to a client. A simple HTTP client application is used to iterate over the set of target and joint configuration combinations and measures the response times of the solver API. Figure 3.3 gives a high-level overview of the components of the remote host application.

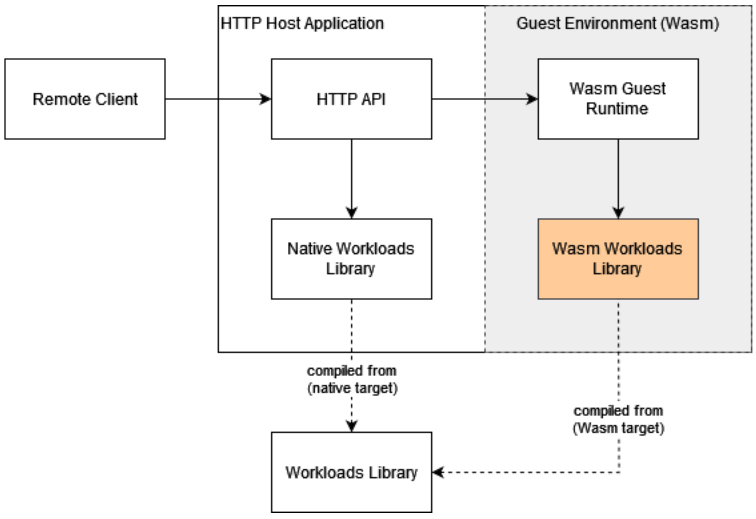

Figure 3.3: Overview of remote host application architecture used in end-to-end benchmarks.

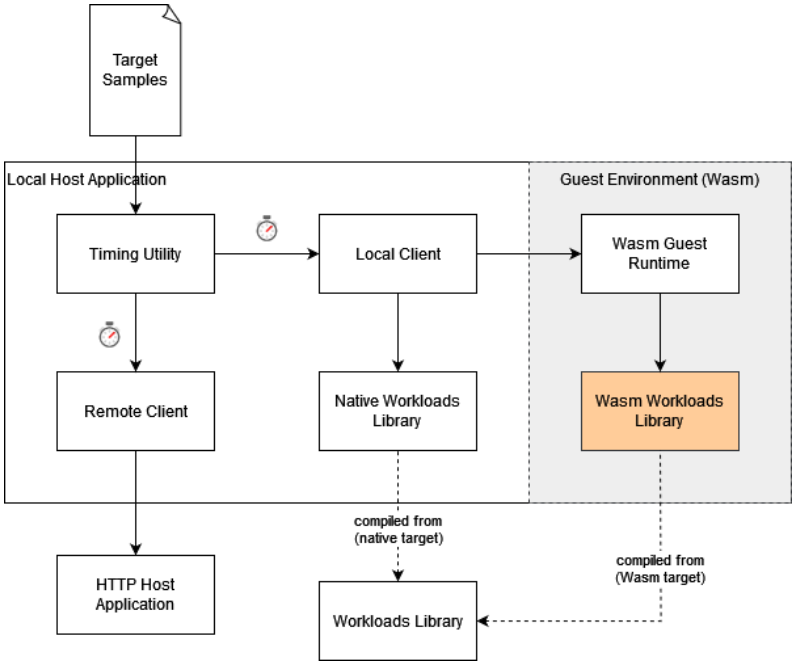

Figure 3.4: Overview of local host application architecture used in local benchmarks.

3.2.1.2 A local host setup

The local host application iterates over the set of target and joint configuration combinations and measures the time it takes to solve a given combination through both the function call exported by the native workloads library and the call to the same function exported by the compiled Wasm workloads module via the guest library for a given robot model. The components of the local host application are shown in Figure 3.4.

3.2.2 Robot models

For the implementation of the robotics tasks, we rely on the *k* project (version 0.29.1) by the Open Rust Robotics platform [37]. The project is a Rust Kinematics library that offers a straightforward API for the implementation of forward and inverse kinematics functionality and robot models. Two robot models are used in our benchmarks. The first represents a robot designed for didactic purposes for the Arduino platform, the *Tinkerkit Braccio*[*] robot. The second robot model is a robot definition provided as an example in the *k* library and represents a torso with a left and right arm. The robots are defined through available files in the *URDF* format; the implemented robots are illustrated in Figures 3.5 and 3.6, respectively [37, 38].

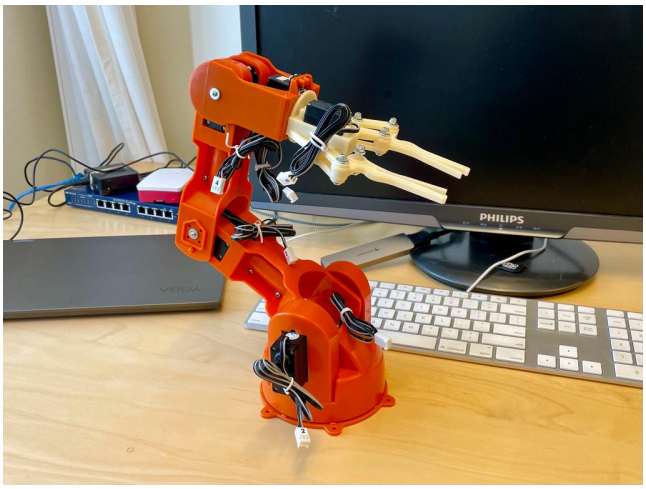

Figure 3.5: The Braccio Arduino robot arm (during assembly).

We do not consider more complex, composite movements, e.g., grabbing and moving an object. The goal of the implemented solver library is to

[*]Website: https://store.arduino.cc/products/tinkerkit-braccio-robot.

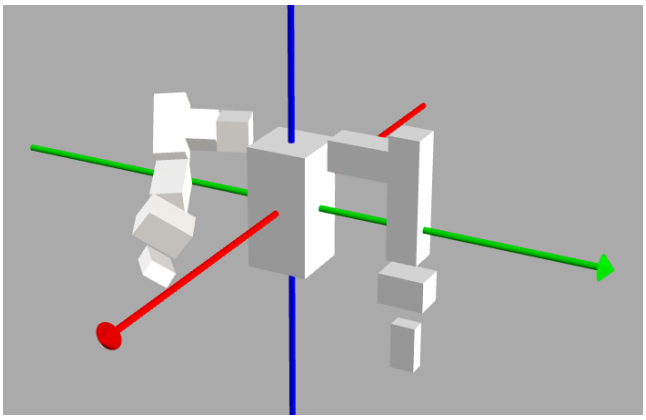

Figure 3.6: *k* sample robot torso model (rendering).

compute the necessary joint configurations to reach a given target point with the end effector. For the robot arm model, the end effector is one of the gripper joints; for the torso robot model, the aim is to reach a point with the right wrist joint. The resulting kinematic chains representing the robots have 5 DoF and 6 DoF, respectively.

3.3 Evaluation

To address our research questions, we analyze our empirical results and evaluate whether Wasm satisfies the responsiveness requirements posed by a CPS in an edge network environment. We further assess whether Wasm offers the platform and language support to achieve the portability goal posed by a diverse edge network system.

3.3.1 RQ1 - Usability in an edge network CPS

We compare our empirical results with performance requirements identified from industrial demands to address RQ1. Although the exact demands vary depending on the domain and application, necessitating a more detailed analysis and possibly a comparison with existing alternatives, we evaluate whether Wasm provides developers and maintainers with the features necessary to achieve reasonable portability and performance goals, as described below.

3.3.1.1 Portability

Due to the diverse nature of devices and platforms, applications and artifacts must provide a high degree of portability to help reduce implementation efforts and increase the flexibility of edge network systems [4, 5]. Portable and backward-compatible code can help reduce development and maintenance costs and orchestration complexity in distributed systems. Due to its open design, broad support, and its design goals, we expect Wasm to be compatible with most major languages and common platforms.

3.3.1.2 Performance

A CPS task feasibility depends on how its performance compares against industrial requirements. Some examples, with latency limits varying between <1–500ms, are shown in Table 3.2. The latency requirements can vary depending on the details and limitations of the application. To evaluate whether Wasm satisfies the performance requirements, we compare the metrics recorded during our experiments with a pre-defined latency limit, selected from the requirements of possible applications. As a use case, we consider the control of a robot's movement through input signals from infrared gates or similar sensors. Thus, for this project, we assume an overall latency limit of 100ms, corresponding to a common signal polling interval, to be a reasonable and realistic choice.

Sector	Latency	Source
Industrial Applications	5–500ms	Lasi *et al.* [39]
Industrial Robotics	50ms	HAL Robotics Ltd., polling for tool path calculation for an arc welding task
Industrial Robotics	100ms	HAL Robotics Ltd., a common interval for electrical signal polling
6G	<1ms	Wikström *et al.* [15]
5G-Edge	1–10ms	Marsh *et al.* [4]

Table 3.2: Selection of latency requirements for industrial applications, industrial robotics, and networking standards.

3.3.2 RQ2 - Wasm overhead

To address RQ2, we compare the response times measured for our Wasm implementation with a native implementation for the same robotics task. We further compare and validate the local and remote results for different placement options in an edge network with results from existing literature, described in Section 2.2. The performance is measured strictly in terms of the latency given by the solving time (local deployment) and response time (remote deployment); a more detailed evaluation based on other metrics is dependent on a variety of specific characteristics of possible use cases and thus out of scope for this project.

Chapter 4

Results and analysis

In this chapter, we present the main empirical results of this project. Further, we describe how Wasm can help solve portability issues posed by an edge network CPS and compare the empirical results to the use case's latency requirements.

4.1 Empirical results

To answer RQ2, we perform benchmarks using a CPS implementation. We measure the response times for the robotics tasks on three different devices, representing the considered placement options. The Wasm results shown in the following figures are those measured for the Wasmer runtime; the Wasmtime runtime results are similar but omitted here for clarity. The complete results can be found in Chapter A of the Appendix. The Appendix further includes measurements performed with a no-op task, i.e., an empty function call, and a homepage request (native remote results only, implemented mainly for testing purposes). The results shown in this section include the response times measured for the successful solution of a robotics task through the remote and local host applications as described in Chapter 3, as well as some relevant results from preliminary standalone measurements. In the following results, we consider exclusively the time measured for tasks for which a solution can be successfully found since response and execution times in case of failure depend mostly on the parametrization of the solver algorithm and error handling. Recorded times for the tasks failed on unreachable combinations are shown in Chapter A of the Appendix.

4.1.1 End-to-end measurements

Figure 4.1 shows the elapsed time, in milliseconds, for solving the two robotics tasks (arm and torso robot models) on the Raspberry Pi (far edge device). The average solving times for the robot arm task are around 7ms and 12ms for the native and Wasm implementation, respectively, and around 9ms for the native and 12ms for the Wasm implementation of the torso robot model. The box plots in Figure 4.1 and successive figures 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 $\times 1.5$ the inter-quartile range from the box, while outliers beyond that range are plotted as individual points. The response times, for both implementations, are well below the requirements of the selected scenario, even though there is a difference in response times between the native and Wasm implementation.

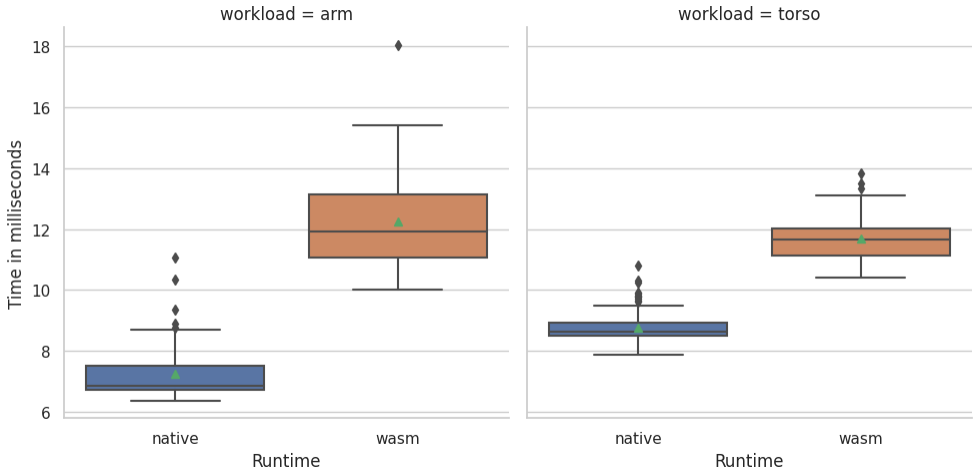

Figure 4.1: End-to-end benchmarks, robotics workloads on the far edge device.

The same results are shown for the near edge placement in Figure 4.2, where the measured response times have an average value of 9ms and 13ms for the native and Wasm implementations of the arm robot model, and 11ms and 15ms for the native and Wasm implementations of the torso robot model. Figure 4.3 shows the results similarly measured on the cloud device. The mean values of the measured response times are between 29ms and 32ms for both tasks, both for the native and the Wasm implementations.

The difference between Wasm and native execution times is significant even though execution with near-native performance is a stated Wasm design

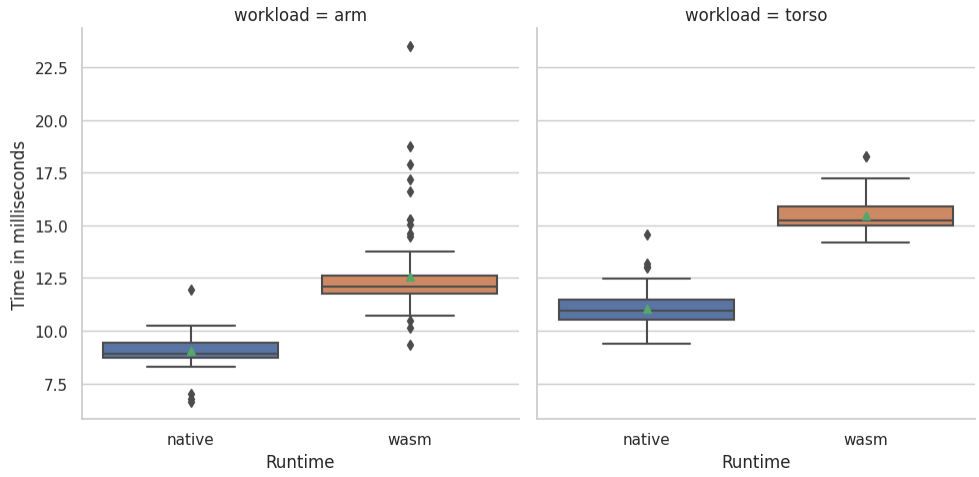

Figure 4.2: End-to-end benchmarks, robotics workloads on the near edge device.

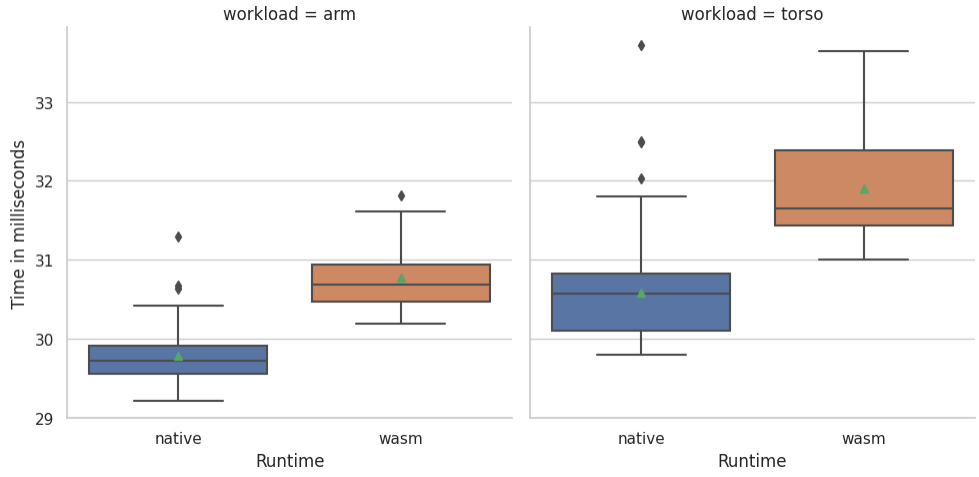

Figure 4.3: End-to-end benchmarks, robotics workloads on a cloud device.

goal [6]. The results across all placement and runtime options show that the native implementation consistently outperforms the Wasm version, as shown in comparable works. Missing optimizations and code generation issues during the conversion from Wasm bytecode into machine code have been identified as the main causes for the performance reduction [26, 28].

Figure 4.4 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.

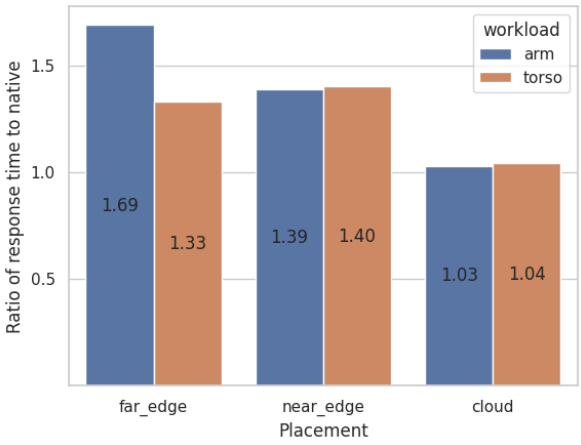

Figure 4.4: End-to-end benchmarks, mean Wasm response times for robotics workloads, relative to mean native response times.

The results found in related works, described in Section 2.2, suggest that the performance ratio of Wasm to native code execution does not vary substantially across different runtimes and implementations [14, 24, 27, 28]. Assuming that this holds across devices and platforms, this means that the overall latency difference between Wasm and native implementations in an end-to-end scenario is mostly dependent on the network latency and the device's performance. Increasing the device performance or the network latency decreases the impact of the performance trade-off introduced by Wasm.

However, given that, when comparing the considered cloud and edge placement options, the network latency is the more dominant factor, we find that moving the computation closer to the target application is beneficial in terms of overall application responsiveness, even when considering a greater performance decrease due to device limitations. The results show that in our implementation we can roughly half the application latency by deploying the workloads at the network's edge, moving closer to the target application, thus validating the edge computing approach.

4.1.2 Local benchmarks

The results measured for a local deployment are consistent with the performance ratio of Wasm to native execution times found in similar studies,

described in Section 2.2, across all considered runtime and placement options [14, 24, 27, 28]. Figure 4.5 shows the (average) execution times, grouped by workload and placement, relative to the native implementations. We can see that the Wasm measurements are within $\times 1.9$–$\times 2.2$ the times recorded for the native tasks.

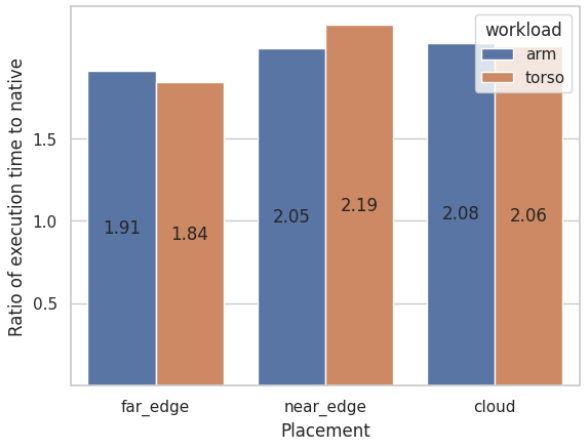

Figure 4.5: Local benchmarks, mean Wasm execution times for robotics workloads, relative to mean native execution times.

The gap between our results and those shown by Hockley and Williamson [25], who find a $\times 5$–$\times 10$ 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. The complete results for the local measurements can be found in Chapter A of the Appendix.

4.1.3 Non-embedded measurements

Figure 4.6 shows some of the results measured during preliminary tests using non-embedded Wasm artifacts 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. Results measured for both Wasmtime and Wasmer in JIT compiled mode are not included in this plot since they are both at around 1s, considerably exceeding the chosen latency limit.

The significantly worse JIT performance is most likely due to the complexity of the (unoptimized) Wasm artifacts, 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.

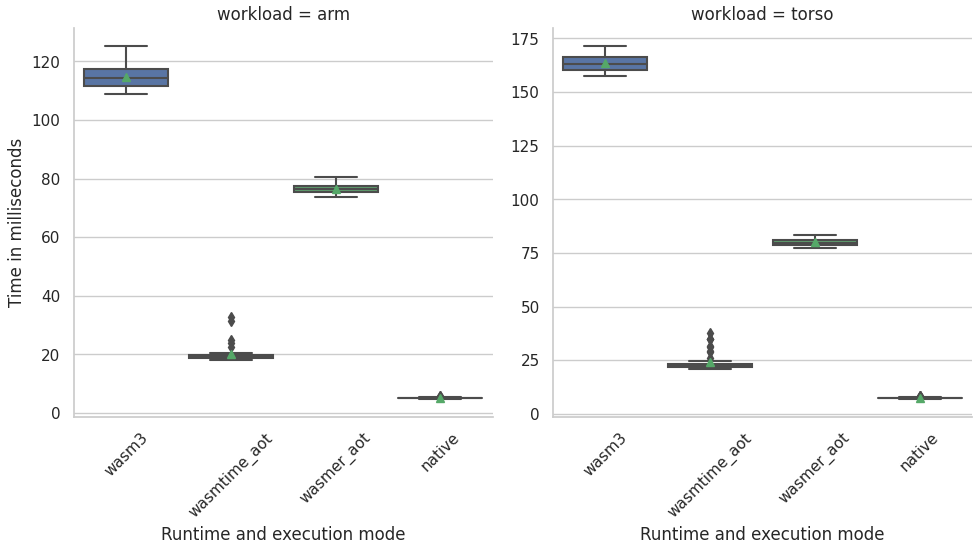

Figure 4.6: Standalone benchmarks, selected execution times for robotics applications.

4.2 Portability and performance

To answer RQ1, in this section, we take a closer look at the portability and flexibility offered by Wasm and compare our empirical results in an end-to-end application against the chosen latency requirement for an industrial application.

4.2.1 Portability

Table 4.1 illustrates the language and platform support offered by the Wasm runtimes we consider for this project [31–33]. Wasm bytecode is a compilation target for all major languages, can be embedded in all popular languages, and supports most common platforms, offering considerable portability and language flexibility [16, 40]. The offered language and platform independence make it a suitable choice for applications targeting portability and flexibility. Wasm is an open standard and the ecosystem includes a great number of compilers, runtimes, and tools provided by third-party projects. Both Wasm and WASI further aim to be backward-compatible, reducing maintenance and development efforts even though both are evolving standards [6, 20].

Runtime	Supported Languages	Supported Platforms
Wasmer	Rust, C/C++, Go, Python, PHP, Ruby, Go, C#, R, Elixir, Java, JS	AMD64, ARM64, RISC-V64
Wasmtime	Rust, C, Python, .NET, Go, Bash, Ruby, Elixir	AMD64, ARM64, RISC-V64, S390X
Wasm3	Python3, Rust, C/C++, GoLang, Zig, Perl, Swift, .Net, Nim, Arduino, PlatformIO, Particle, QuickJS	Multiple architectures, including x86, AMD64, ARM, RISC-V, PowerPC, MIPS, Xtensa, ARC32

Table 4.1: Language and platform support, selected Wasm runtimes.

4.2.2 Performance

Our empirical results show that, for all tested placement options, the Wasm execution times for the implemented robotics tasks are within $\times 1.9$–$\times 2.2$ the native execution times. The results are consistent with comparable results shown in other studies. The overall overhead introduced by Wasm into an end-to-end system varies with the devices' placement and performance, growing with decreasing network latency or performance. The results show that, even though there is a significant performance overhead when comparing Wasm with native code, the overall performance is still in line with the requirements of the chosen CPS application. That is, the response times for all tested placement and runtime options are below the chosen latency limit of 100ms. The results show considerable latitude in all three placement options we test;

the overall latency would still be below a more conservative latency limit, e.g., 50ms. The results further validate the edge network architecture, illustrating how proximity can help increase the responsiveness of a CPS application despite the overhead introduced by Wasm and devices with more restricted performance.

Chapter 5

Discussion

In this section, we discuss the relevance of our findings for edge networks and describe possible placement options considering our empirical results and the functionalities and capabilities offered by Wasm runtimes.

5.1 Wasm usability in edge networks

While edge network architectures can help reduce the latency in CPS, they can pose significant issues due to the diversity of devices and platforms. The language and platform flexibility offered by Wasm can help implement an edge network system that allows services to be executed and migrated in a heterogeneous network. While there is a trade-off between portability and performance, our results show that the performance overhead introduced by Wasm in the system is outweighed by the reduced latency found in edge systems. We describe two possible placement options inside an edge network with similar effects on the overall responsiveness of a cyber-physical application. Based on our empirical results, we believe Wasm to be a viable choice for implementing edge and cyber-physical systems. The features and advantages Wasm offers in terms of portability and flexibility, particularly benefit use cases such as service migration and scaling.

While our results do not show a significant advantage for one of the tested edge placement options, the choice might be influenced by several other factors, such as network topology, system architecture, and resource availability. In our experimental design, we assume that the restrictions on the devices' performance increase with proximity, while it should be easier to scale resources with increasing network latency; however, this does not necessarily hold for all use cases and applications. In general, the results

from our benchmarks and comparable studies, suggest that the performance degradation Wasm displays when compared to native code does not vary substantially, with the following major caveats:

- Studies suggest that accessing resources and functionality outside the Wasm sandbox is costly in terms of performance [25]. We opt for an application architecture that embeds Wasm functionality in a host application to minimize overhead due to frequent host-guest interaction while allowing us to exploit Wasm's portability for the core functionality of our application.
- While efforts are underway to standardize access through WASI, Wasm cannot take advantage of native hardware acceleration features, which are relevant to applications such as cryptography and machine learning [24, 41, 42]. The performance difference between Wasm and native implementations might be prohibitive for such use cases.
- The proposed application architecture and implementation might not be viable for all use cases. Platform and device limitations as well as application requirements might dictate, for example, different runtime, placement, or deployment choices, leading to worse results due to runtime execution and optimization overhead [24].
- Studies found that the performance degradation is in large parts imputable to missing optimizations and problems during code generation [26, 28]. Other applications and use cases might be impacted to a higher degree, even though our results are consistent with similar benchmarks performed in previous works, i.e. [14, 24, 27, 28].

The main goal of this project is to assess the responsiveness of Wasm in an edge network CPS and evaluate whether the performance and portability offered by the language can benefit applications in heterogeneous edge networks. One further key requirement we do not consider in our implementation and viability evaluation is safety. One of the main safety challenges in edge network systems stems from untrusted code execution. Systems and resources, e.g., memory and compute resources, need to be safeguarded from malicious or malfunctioning code. Edge network systems require the ability to isolate applications and data, particularly in situations where applications from multiple tenants and entities share the same environment and resources. Wasm offers developers security primitives and risk mitigation features for the implementation of a *safe* edge network system.

However, this means that many of the safety guarantees rely on correct host and runtime implementation. While we find that Wasm offers reasonable risk mitigation mechanisms, the safety demands vary with the domain and application, requiring a more detailed evaluation on a case-by-case basis.

Summary of research question answers

We find that Wasm is a viable option as a portable edge network runtime, with some limitations. The main findings and answers to our original research questions stated in Section 1.1.2, are the following.

RQ1 Could Wasm be used to operate an industrial robot application?

We find that Wasm offers the required portability to facilitate the implementation of edge network systems. We further find that the response times measured in the implementation of an edge network CPS are within the 100ms latency limit chosen for the implemented robot control application. Our results show that an edge network placement can significantly reduce the response times through closer proximity to the client, despite the considerable impact of Wasm on the application's overall performance and lower device performance.

RQ2 How much delay does Wasm introduce in an edge-cloud robotic setting in comparison to native binaries?

We find that Wasm introduces a significant performance overhead when compared to native implementations. For tasks executed locally, we find that Wasm execution times are within $\times 1.9$–$\times 2.2$ native execution times. In an end-to-end scenario, we measure average Wasm response times within 15ms for both edge network placement options and Wasm response times within 32ms for the cloud placement option, while the impact of Wasm on the overall latency decreases with increasing device performance and network overhead.

Chapter 6

Conclusions

Edge network solutions can reduce the latencies in cyber-physical systems such as in IIoT applications. The heterogeneity of edge networks and industrial demands make it critical to implement and deploy portable, fast, and safe code. In this project, we evaluate the usability of Wasm as a portable runtime in an edge network robot control system. To assess the usability of Wasm for industrial applications, we evaluate whether it provides the portability required by edge network systems and satisfies the performance demands of an industrial use case.

We select and implement a robot arm control scenario and perform a series of remote and local benchmarks. We compare our results against a baseline, represented by a native implementation of the same task. This measures the overall impact of introducing Wasm in the application stack. We further compare the responsiveness of the implemented application against a latency limit from selected industrial requirements.

We find that Wasm satisfies the portability requirement due to its language and platform independence. Additionally, given our empirical 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 well within the chosen latency requirements. Thus, we conclude that Wasm could benefit and enable several use cases due to its secure architecture, performance, flexibility, and portability.

6.1 Future work

To keep the scope of the project manageable, we consider a restricted set of technical and architectural choices, thus limiting the external validity of our

results. The implemented application is a simplification of a remote robot control system meant to assess primarily the performance differences between a native and a Wasm implementation of a robotics task. In the remainder of this section, we highlight some of the possible extensions to this project and options that might warrant further research:

- The implemented task and architecture is a simplified model for one of several possible applications that might be found in an IIoT scenario. Many ubiquitous computing use cases depend on more complex models and tasks, such as systems relying on computer vision, or federated learning applications. Further research should validate the results for increased complexity and different architecture and implementation choices dictated by other use case requirements. Concretely, we could extend the cyber-physical system presented from a single robot arm to, for example, an automated factory floor.

- We do not consider all available options and features provided by the Wasm ecosystem. For instance, we test two runtimes using the same compiler backend and do not compare available optimization options and tools. While our measurements seem to show a trade-off between JIT performance and executable size (or, more precisely, complexity), favoring AOT compilation and interpretation, further work is required to evaluate available features and options, and their impact on the performance of a system.

- In our experiment, we consider a simplified robot control application in which a step in a robot's movement is represented only by the computation of the joint configuration for a given target. Further factors, such as robot movement delays, error handling, and the effective control of the robot, impact the overall responsiveness of a CPS. We think that considering a more complete CPS implementation would be a useful extension to this work.

- As can be seen in the plots in Chapter 4, we register several outliers in some of the experiments which we cannot easily explain in a consistent manner. The near edge model device shows the most severe examples, which are excluded from the results, thus we assume that the outliers are due to issues for that particular device. In other cases, the problems might be caused by either hardware or software, the network, libraries, or our implementation. Therefore, there is further room for investigating these anomalies. One suggestion is to use a more granular analysis of the

causes and components of the performance overhead and the variability in some of the presented results.

Chapter 7

Lessons learned

When starting with this project, I had no experience with WebAssembly and a superficial knowledge of edge computing. Throughout the work on this thesis, I had the opportunity to tinker with many different technologies and paradigms, some outside of my comfort zone, and this project helped me apply knowledge in new ways and widen my scope. But apart from know-how and skills around the key technologies and a deep dive into (an admittedly small) subset of robotics and system architecture, the main takeaway from this work will be how it shaped my way of approaching and working on such a project. Early on, it was clear that we were in front of a great number of different paths and choices throughout the project, and the support from my supervisors was invaluable in helping me keep the scope reasonable and approachable, and consider things from different perspectives. What helped most in managing the extent of the project was starting early with prototyping and to perform small, incremental changes, evaluating at each step how we could narrow down the options for the next iteration while working towards our original goal. Having a short feedback loop was key in maintaining focus and steady progress throughout the project. Another device I found helpful, particularly when writing the report for this project, was maintaining organized and up-to-date notes about the progress, thoughts, ideas, issues, and backlogs. This project, all things considered, was an invaluable lesson, not only about the featured technologies and concepts, but also in project management, and taught me how to plan and execute a research project from start to finish through the whole process.

References

[1] L. Hou *et al.*, "Internet of things cloud: Architecture and implementation," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 32–39, Dec. 2016, ISSN: 0163-6804, 1558-1896. DOI: 10.1109/MCOM.2016.1600398CM. [Online]. Available: https://ieeexplore.ieee.org/document/7785887/ (visited on 06/07/2023).

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016, ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2579198. [Online]. Available: http://ieeexplore.ieee.org/document/7488250/ (visited on 06/07/2023).

[3] A. Davis, J. Parikh, and W. E. Weihl, "Edgecomputing: Extending enterprise applications to the edge of the internet," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters - WWW Alt. '04*, New York, NY, USA: ACM Press, 2004, p. 180, ISBN: 9781581139129. DOI: 10.1145/1013367.1013397. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1013367.1013397 (visited on 06/07/2023).

[4] I. Marsh *et al.*, "Evolving 5g: ANIARA, an edge-cloud perspective," 2022. DOI: 10.48550/ARXIV.2205.03098. [Online]. Available: https://arxiv.org/abs/2205.03098 (visited on 02/03/2023).

[5] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate IoT edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, Jan. 2018, ISSN: 0890-8044, 1558-156X. DOI: 10.1109/MNET.2018.1700175. [Online]. Available: https://ieeexplore.ieee.org/document/8270640/ (visited on 09/25/2022).

[6] A. Rossberg, "WebAssembly core specification," W3C, W3C Working Draft, Apr. 2022. [Online]. Available: https://www.w3.org/TR/2022/WD-wasm-core-2-20220419/ (visited on 03/16/2023).

[7] S. Hamdan, M. Ayyash, and S. Almajali, "Edge-computing architectures for internet of things applications: A survey," *Sensors*, vol. 20, no. 22, p. 6441, Nov. 11, 2020, ISSN: 1424-8220. DOI: 10.3390/s20226441. [Online]. Available: https://www.mdpi.com/1424-8220/20/22/6441 (visited on 06/07/2023).

[8] S. Wang, "Edge computing: Applications, state-of-the-art and challenges," *Advances in Networks*, vol. 7, no. 1, p. 8, 2019, ISSN: 2326-9766. DOI: 10.11648/j.net.20190701.12. [Online]. Available: http://www.sciencepublishinggroup.com/journal/paperinfo?journalid=131&doi=10.11648/j.net.20190701.12 (visited on 09/25/2022).

[9] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, Aug. 17, 2010, ISSN: 0163-5980. DOI: 10.1145/1842733.1842736. [Online]. Available: https://dl.acm.org/doi/10.1145/1842733.1842736 (visited on 06/07/2023).

[10] Heavy Reading. "Strategies for connecting the edge: 2019 heavy reading survey," www.infinera.com. (2019), [Online]. Available: https://www.infinera.com/white-paper/strategies-for-connecting-the-edge-2019-heavy-reading-survey/ (visited on 09/25/2022).

[11] B. Li, W. Dong, and Y. Gao, "WiProg: A WebAssembly-based approach to integrated IoT programming," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, Vancouver, BC, Canada: IEEE, May 10, 2021, pp. 1–10, ISBN: 9781665403252. DOI: 10.1109/INFOCOM42981.2021.9488424. [Online]. Available: https://ieeexplore.ieee.org/document/9488424/ (visited on 01/25/2023).

[12] B. Li, H. Fan, Y. Gao, and W. Dong, "Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, Portland Oregon: ACM, Jun. 27, 2022, pp. 261–272, ISBN: 9781450391856. DOI: 10.1145/3498361.3538922. [Online]. Available: https://dl.acm.org/doi/10.1145/3498361.3538922 (visited on 01/19/2023).

[13] M. Nieke, L. Almstedt, and R. Kapitza, "Edgedancer: Secure mobile WebAssembly services on the edge," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, Online United Kingdom: ACM, Apr. 26, 2021, pp. 13–18, ISBN: 9781450382915. DOI: 10.1145/3434770.3459731. [Online]. Available:

https://dl.acm.org/doi/10.1145/3434770.3459731 (visited on 03/22/2023).

[14] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 185–200, Sep. 14, 2017, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/3140587.3062363. [Online]. Available: https://dl.acm.org/doi/10.1145/3140587.3062363 (visited on 05/28/2023).

[15] G. Wikström *et al.*, "6g – connecting a cyber-physical world," *Ericsson White Paper*, Nov. 2022. [Online]. Available: https://www.ericsson.com/en/reports-and-papers/white-papers/a-research-outlook-towards-6g (visited on 03/16/2023).

[16] S. Akinyemi. "Awesome WebAssembly runtimes." (2023), [Online]. Available: https://github.com/appcypher/awesome-wasm-runtimes (visited on 03/17/2023).

[17] D. Salim, *Securing Trigger-Action Platforms With WebAssembly*. 2022. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-321994 (visited on 06/14/2023).

[18] J. Bosamiya, W. S. Lim, and B. Parno, "{Provably-safe} multilingual software sandboxing using {WebAssembly}," presented at the 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 1975–1992, ISBN: 9781939133311. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya (visited on 06/08/2023).

[19] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, Delft Netherlands: ACM, Dec. 7, 2020, pp. 265–279, ISBN: 9781450381536. DOI: 10.1145/3423211.3425680. [Online]. Available: https://dl.acm.org/doi/10.1145/3423211.3425680 (visited on 06/14/2023).

[20] "WASI: The WebAssembly system interface." (2023), [Online]. Available: https://wasi.dev/ (visited on 03/17/2023).

[21] L. Clark. "Standardizing WASI: A system interface to run WebAssembly outside the web – mozilla hacks - the web developer blog," Mozilla Hacks – the Web developer blog. (Mar. 27, 2019), [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface (visited on 03/17/2023).

[22] R. Nilsson, *Inverse kinematics*. 2009. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:ltu:diva-45528 (visited on 05/10/2023).

[23] M. Meredith and S. C. Maddock, "Real-time inverse kinematics: The return of the jacobian," 2004. [Online]. Available: https://www.semanticscholar.org/paper/Real-Time-Inverse-Kinematics%3A-The-Return-of-the-Meredith-Maddock/85dac7f7da71078853e59352530c2ffe22c7a30b (visited on 05/10/2023).

[24] F. Denis. "Performance of WebAssembly runtimes in 2023," Performance of WebAssembly runtimes in 2023 | Frank DENIS random thoughts. (2023), [Online]. Available: https://00f.net/2023/01/04/webassembly-benchmark-2023/ (visited on 01/19/2023).

[25] D. Hockley and C. Williamson, "Benchmarking runtime scripting performance in wasmer," ser. ICPE '22, New York, NY, USA: Association for Computing Machinery, Jul. 19, 2022, pp. 97–104, ISBN: 9781450391597. DOI: 10.1145/3491204.3527477. [Online]. Available: https://doi.org/10.1145/3491204.3527477 (visited on 01/19/2023).

[26] 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*, Virtual Event: ACM, Nov. 2, 2021, pp. 533–549, ISBN: 9781450391290. DOI: 10.1145/3487552.3487827. [Online]. Available: https://dl.acm.org/doi/10.1145/3487552.3487827 (visited on 02/01/2023).

[27] J. Napieralla, *Considering WebAssembly Containers for Edge Computing on Hardware-Constrained IoT Devices*. 2020. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:bth-20112 (visited on 02/08/2023).

[28] A. Jangda, B. Powers, E. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," 2019. DOI: 10.48550/ARXIV.1901.09056. [Online]. Available: https://arxiv.org/abs/1901.09056 (visited on 02/01/2023).

[29] R. Jain, *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. New York: Wiley, 1991, 685 pp., ISBN: 9780471503361.

[30] C. Eberhardt. "The state of WebAssembly 2022," Scott Logic. (Jun. 20, 2022), [Online]. Available: https://blog.scottlogic.com/2022/06/20/state-of-wasm-2022.html (visited on 03/22/2023).

[31] Wasmer. “Wasmer, the universal WebAssembly runtime.” (**n.d.**), [Online]. Available: https://wasmer.io/ (visited on 05/10/2023).

[32] Bytecode Alliance. “Wasmtime, a fast and secure runtime for WebAssembly.” (**n.d.**), [Online]. Available: https://wasmtime.dev/ (visited on 05/10/2023).

[33] Wasm3. “Wasm3.” (**n.d.**), [Online]. Available: https://github.com/wasm3/wasm3 (visited on 05/10/2023).

[34] “Lightweight justice for your SBC!” DietPi. (**n.d.**), [Online]. Available: https://dietpi.com/ (visited on 05/25/2023).

[35] Nmap.org. “Nping — network packet generation tool & ping utility.” (**n.d.**), [Online]. Available: https://nmap.org/nping/ (visited on 05/25/2023).

[36] D. Peter, *Hyperfine*, version 1.16.1, original-date: 2018-01-13T15:49:54Z, Mar. 2023. [Online]. Available: https://github.com/sharkdp/hyperfine (visited on 05/25/2023).

[37] Open Rust Robotics. “K,” K, Kinematics library for rust-lang. (**n.d.**), [Online]. Available: https://github.com/openrr/k (visited on 05/05/2023).

[38] J. Balzer. “URDF file for arduino braccio toy robot arm,” URDF file for Arduino Braccio toy robot arm. (**n.d.**), [Online]. Available: https://github.com/jonabalzer/braccio_description (visited on 05/10/2023).

[39] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, “Industry 4.0,” *Business & Information Systems Engineering*, vol. 6, no. 4, pp. 239–242, Aug. 2014, ISSN: 1867-0202. DOI: 10.1007/s12599-014-0334-4. [Online]. Available: http://link.springer.com/10.1007/s12599-014-0334-4 (visited on 05/23/2023).

[40] “The top programming languages,” The State of the Octoverse. (**n.d.**), [Online]. Available: https://octoverse.github.com/2022/top-programming-languages (visited on 06/14/2023).

[41] “WASI cryptography APIs,” WASI Cryptography APIs. (**n.d.**), [Online]. Available: https://github.com/WebAssembly/wasi-crypto (visited on 06/03/2023).

[42] “A proposed WebAssembly system interface API for machine learning (ML).,” A proposed WebAssembly System Interface API for machine learning (ML). (**n.d.**), [Online]. Available: https://github.com/WebAssembly/wasi-nn (visited on 06/03/2023).

Appendix A

Additional results

A.1 End-to-end measurements

Figure A.1 illustrates the results recorded for the remote host application during the end-to-end tests, grouped by placement (rows), workload (columns), and task success status (color). Table A.1 summarizes the same results, showing the average response times in milliseconds with the standard error (SEM). The results include response times for the solution of the robot tasks, an empty (remote) function call, and a homepage request for all considered placement and runtime options.

A.2 Local measurements

Figure A.2 shows the response times measured for the local host application. The results are grouped by placement (rows), workload (columns), and task success status (color). Table A.2 summarizes the same results, showing the time in milliseconds and the SEM. The results include execution times for the solution of the robot tasks and an empty function call for all considered placement and runtime options.

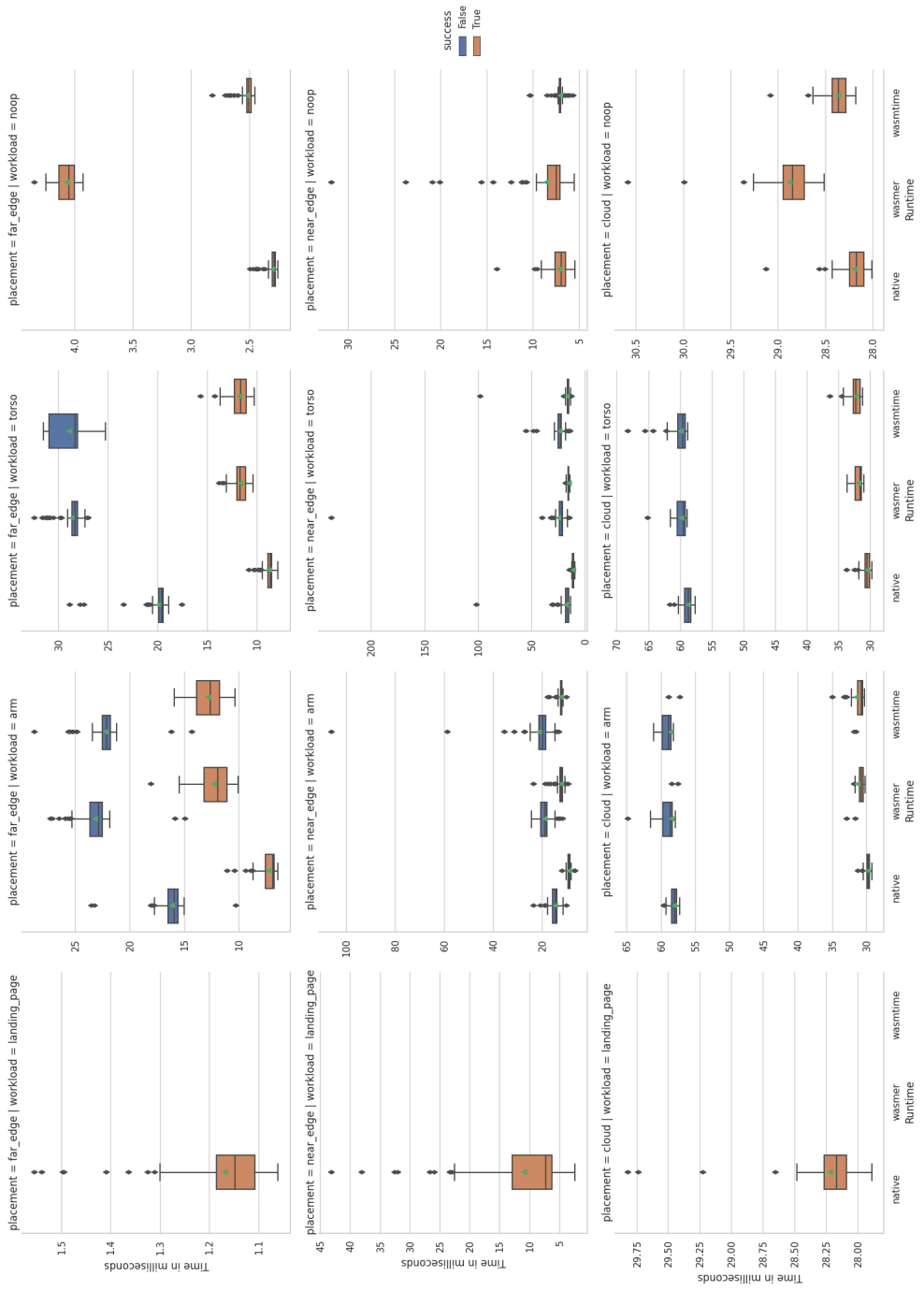

Figure A.1: End-to-end benchmarks, response times.

Placement	Workload	Runtime	Mean ±SEM [ms]	
			Success	Failure
far_edge	wasmer	arm	12.25 ±0.14	23.14 ±0.16
		torso	11.69 ±0.06	28.7 ±0.11
		noop	4.07 ±0.01	
	wasmtime	arm	12.81 ±0.13	22.21 ±0.15
		torso	11.77 ±0.09	28.94 ±0.15
		noop	2.52 ±0.01	
	native	arm	7.26 ±0.09	16.13 ±0.14
		torso	8.77 ±0.05	19.96 ±0.16
		noop	2.3 ±0.01	
		landing_page	1.17 ±0.01	
near_edge	wasmer	arm	12.56 ±0.18	18.94 ±0.23
		torso	15.46 ±0.08	24.86 ±2.22
		noop	8.59 ±0.37	
	wasmtime	arm	12.38 ±0.1	21.32 ±1.0
		torso	16.67 ±0.86	23.68 ±0.56
		noop	7.08 ±0.06	
	native	arm	9.05 ±0.07	14.85 ±0.22
		torso	11.06 ±0.08	17.89 ±0.89
		noop	7.14 ±0.11	
		landing_page	10.91 ±0.77	
cloud	wasmer	arm	31.33 ±0.39	58.47 ±0.38
		torso	31.9 ±0.06	59.86 ±0.09
		noop	28.88 ±0.03	
	wasmtime	arm	31.51 ±0.4	58.6 ±0.39
		torso	32.28 ±0.08	60.0 ±0.14
		noop	28.37 ±0.01	
	native	arm	29.78 ±0.03	58.07 ±0.05
		torso	30.58 ±0.06	58.78 ±0.07
		noop	28.19 ±0.01	
		landing_page	28.22 ±0.03	

Table A.1: End-to-end benchmarks, summary of results.

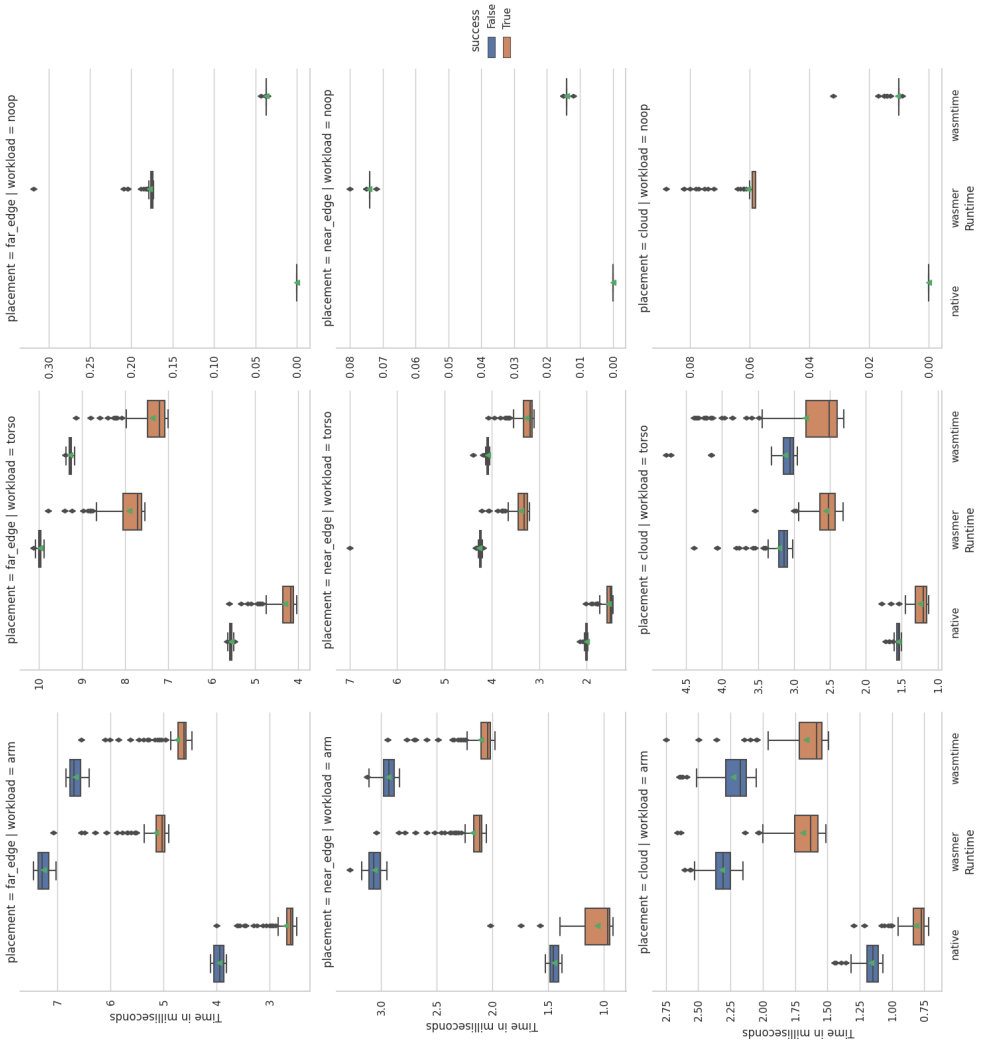

Figure A.2: Local benchmarks, execution times.

Placement	Workload	Runtime	Mean ±SEM [ms]	
			Success	Failure
far_edge	wasmer	arm	5.15 ±0.0	7.25 ±0.01
		torso	7.91 ±0.0	9.97 ±0.0
		noop	0.18 ±0.0	
	wasmtime	arm	4.74 ±0.0	6.66 ±0.01
		torso	7.37 ±0.0	9.27 ±0.0
		noop	0.04 ±0.0	
	native	arm	2.7 ±0.0	3.96 ±0.01
		torso	4.3 ±0.0	5.55 ±0.0
		noop	0.0 ±0.0	
near_edge	wasmer	arm	2.18 ±0.0	3.05 ±0.01
		torso	3.39 ±0.0	4.27 ±0.03
		noop	0.07 ±0.0	
	wasmtime	arm	2.11 ±0.0	2.93 ±0.01
		torso	3.28 ±0.0	4.09 ±0.0
		noop	0.01 ±0.0	
	native	arm	1.06 ±0.0	1.44 ±0.0
		torso	1.55 ±0.0	2.01 ±0.0
		noop	0.0 ±0.0	
cloud	wasmer	arm	1.69 ±0.0	2.31 ±0.01
		torso	2.55 ±0.0	3.21 ±0.02
		noop	0.06 ±0.0	
	wasmtime	arm	1.67 ±0.0	2.23 ±0.01
		torso	2.84 ±0.0	3.13 ±0.03
		noop	0.01 ±0.0	
	native	arm	0.81 ±0.0	1.16 ±0.01
		torso	1.24 ±0.0	1.55 ±0.0
		noop	0.0 ±0.0	

Table A.2: Local benchmarks, summary of results.

TRITA-EECS-EX- 2023:0000