

Performance and Usability Implications of Multiplatform and WebAssembly Containers

Sangeeta Kakati^a and Mats Brorsson^b

*^{a,b}Interdisciplinary Center for Security, Reliability, and Trust (SnT),
University of Luxembourg, Luxembourg
{sangeeta.kakati, mats.brorsson}@uni.lu*

Keywords: WebAssembly, Heterogeneity, Cloud-Edge, Containers, Multi-Platform, Runtimes.

Abstract: Docker and WebAssembly (Wasm) are two pivotal technologies in modern software development, each offering unique strengths in portability and performance. The rise of Wasm, particularly in conjunction with container runtimes, highlights its potential to enhance efficiency in diverse application stacks. However, a notable gap remains in understanding how Wasm containers perform relative to traditional multi-platform containers across multiple architectures and workloads, especially when optimizations are employed. In this paper, we aim to empirically assess the performance and usability implications of native multi-platform containers versus Wasm containers under optimized configurations. We focus on critical metrics including startup time, pull time (both fresh and cached), and image sizes for three distinct workloads and architectures- AMD64 (AWS bare metal) and two embedded boards: Nvidia Jetson Nano with ARM64 and Starfive VisionFive2 with RISCv64. To address these objectives, we conducted a series of experiments using docker and containerd in multi-platform built images for native containers and Wasmtime as the WebAssembly runtime within containerd/docker's ecosystem. Our findings show that while native containers achieve slightly faster startup times, Wasm containers excel in agility and maintain image sizes of approximately 27.0% of their native counterparts and a significant reduction in pull times across all three architectures of up to 25% using containerd. With continued optimizations, Wasm has the potential to emerge as a viable choice in environments that demand both reduced image size and cross-platform portability. It will not replace the current container paradigm soon; rather, it will be integrated into this framework and complement containers instead of replacing them.

1 INTRODUCTION


Containers are essential for native cloud computing, offering efficient and scalable deployment choices for distributed systems. WebAssembly (Wasm) is a compact binary instruction format and portable compilation target that enables seamless execution of code written in multiple languages, delivering high performance across various platforms.


Containerization and WebAssembly represent a convergence of technologies that enhance application deployment and execution across diverse environments. Containerization encapsulates applications and their dependencies in isolated units, facilitating consistent execution regardless of the underlying infrastructure.

WebAssembly, on the other hand, allows applications to run in a secure sandbox environment, allow-

ing near-native performance inside and outside the Web. The WebAssembly System Interface (WASI) has extended Wasm execution beyond web environments, making it increasingly popular in cloud computing. The ability to use the same binary executable on multiple architectures is particularly compelling. A few years ago, Docker introduced support for different runtimes which opened up for choosing a WebAssembly runtime and running Wasm binaries packaged in a container format.

In this paper, we explore the performance implications and usability benefits/issues of WebAssembly in container runtimes using Wasmtime, comparing it to traditional Linux containers. The context is a future cloud-edge compute continuum where we foresee the use of multiple hardware architectures. Indeed, in regular cloud servers, we can already observe both X86-based- and ARM-based architectures of different points in the design space. This proliferation of architectures will only increase with time. Any software

^a  <https://orcid.org/0000-0002-4795-7489>

^b  <https://orcid.org/0000-0002-9637-2065>

developed should be capable of running on multiple architectures. Currently, the options to create cloud-native applications for this scenario is to either build native docker container images that can run on any architecture or to use WebAssembly which is architecture agnostic.

We present a comprehensive performance evaluation and contribution in the field of WebAssembly and multi-architecture native containers across three workloads and three architectures that showcase cloud server and edge devices, using a state-of-the-art runtime: Wasmtime¹, which is a project within the Bytecode Alliance². WebAssembly and multi-architecture containers offer significant advantages, but standardization and consistent performance remain challenges. Resolving these issues is vital for widespread adoption in IoT settings.

Figure 1 illustrates the workflow for developing and executing cross-architecture containers. In the case of native execution, we must build the software for each architecture and support or build a *multi-architecture docker container image*. The Docker engine can execute this image natively through its layers of runtimes, most notably `containerd` and `runc`. The flow for WebAssembly is almost the same, except that the developer only has to build one image. The docker runtime stack is instead `containerd` and `WasmTime` in which the latter will compile the WebAssembly bytecode for native execution.

Cloudnative applications require fast startup times and low resource overhead, especially in serverless and edge computing environments. Native containers offer high performance, but at the cost of portability, and as we will see, larger container image sizes, while Wasm containers offer portability which might come with performance trade-offs. We aim to analyze these trade-offs across different architectures. Multi-architecture containers enhance security but add management complexity and potential performance trade-offs, which must be carefully evaluated in deployment strategies.

Although most of the related work concentrates on the viability of WebAssembly as an alternative to Docker containers for IoT applications (Pham et al., 2023), it has not yet emerged as a fully functioning alternative to Docker. Furthermore, previous work focused on WebAssembly as a mechanism to reduce cold start latency compared to Docker-based runtimes, but did not use optimizations and best practices for building multi-architecture containers (Gackstat-ter et al., 2022).

Hence, to assess the performance characteristics

of Wasm containers, our experiments emphasize collecting key metrics associated with their utilization and management. These metrics encompass image size, startup duration, pull time, and the distinctions between Wasm-based and native Docker’s optimized, multi-platform built containers. Additionally, we explore the use of `containerd` as an alternative to Docker for image management, providing a more direct and efficient means of handling multi-architecture containers.

The findings of our experiments indicate that the Wasm containers are on average 85% smaller than the native containers. In particular, Wasm containers can reduce image pull times across the `amd64`, `arm64`, and `riscv64` platforms up to 25% compared to native containers using `containerd` directly. In particular, the main contributions of this paper are as follows.

1. This study is the first to extensively explore Wasm’s integration with Docker alternative runtimes, like Wasmtime, on multiple architectures.
2. We provide a thorough performance comparison between multi-architecture optimized native and Wasm containers in three distinct workloads across key metrics such as image size, pull time, and startup duration. In particular, our experiments on `arm64`, `amd64`, and `riscv64` architectures using `containerd` (`ctr`) demonstrate clear performance benefits in pull times.
3. We investigate best practices not commonly employed by related works (e.g., non-optimized images) and use of `ctr`, focusing on the usage of `docker` and `ctr` for both native and `wasm` workloads across architectures.

2 BACKGROUND

Docker containers and WebAssembly are two foundational technologies that have greatly impacted the domain of software development. The Docker container technology is a way to package software with its dependencies in a standardized format, making it easy to deploy and execute in any runtime environment of the same architecture for which the software was compiled. WebAssembly is a compact binary format that serves as a versatile compilation target for a variety of programming languages, such as Rust, C, C++, JavaScript, C# and Go.

Docker containers are typically distributed to their runtime environment from a *container registry*, such as Docker hub³. The container runtime is given a

¹<https://github.com/bytecodealliance/wasmtime>

²<https://bytecodealliance.org/>

³<https://hub.docker.com/>

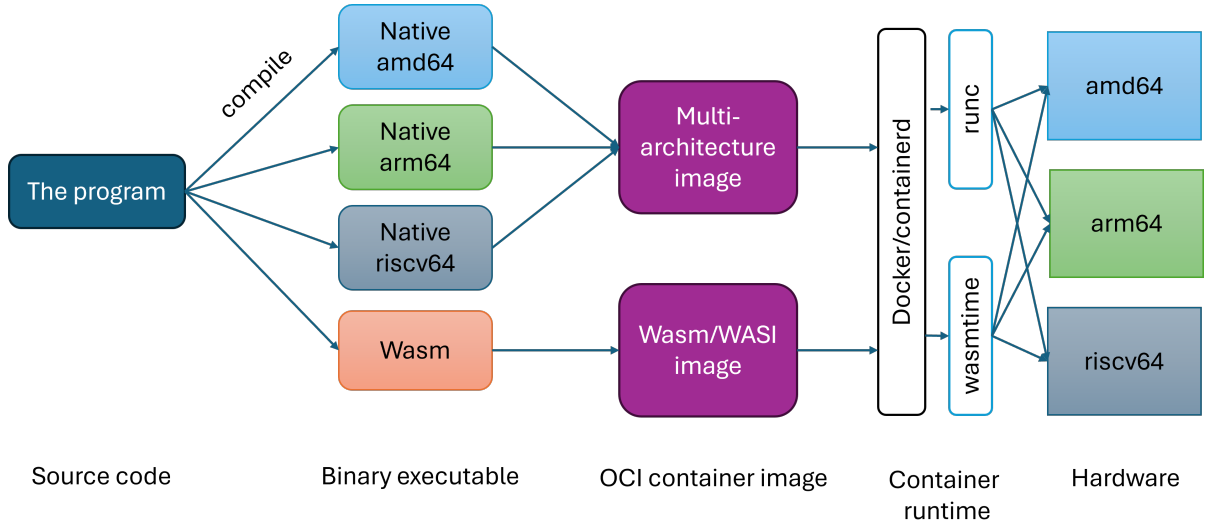


Figure 1: Cross-architecture container execution.

container image specification and will *pull* the image from the registry if it is not found locally.

A WebAssembly runtime performs the bytecode execution in three semantic steps: decoding, validation, and execution, ensuring that code executes at near-native speeds within a secure, sandboxed environment to safeguard against harmful software. Execution can be done through interpretation or after a compilation step done by the runtime. Some WebAssembly runtimes also perform tiered compilation where “hot” functions are further optimized during execution. Wasmtime, which is the runtime used in our study, compiles the WebAssembly bytecode in one step prior to execution.

Dockers’ integration with WebAssembly allows developers to use the performance and security advantages of WebAssembly while benefiting from Docker’s efficient image management and distribution capabilities. The motivation for containerizing Wasm modules is rooted in utilizing existing container orchestration ecosystems. While Wasm provides portability and sandboxing, Docker’s image distribution and management capabilities offer significant operational benefits, such as seamless integration with CI/CD pipelines, image versioning, and registry services. Docker containerizes WebAssembly runtime environments, ensuring uniformity across diverse development and production landscapes. Moreover, packaging Wasm applications streamlines deployment and scaling across different infrastructures. By merging both, developers can navigate the complexities of establishing cross-platform Wasm environments while enjoying the benefits of Docker’s established workflows and comprehensive ecosystem.

3 RELATED WORK

The cloud landscape is evolving, and we increasingly see a development that includes data center servers, regional or on-premise servers, mobile edge stations, and even end-user devices connecting to the edge. Together, we call this the *cloud-edge continuum* (Gkonis et al., 2023). Quite a lot of heterogeneity characterizes this evolving infrastructure. In the context of serverless computing, (Kjorveziroski et al., 2022) discuss the impact of serverless computing on cloud and edge networks, highlighting the startup latency issues associated with current platforms that use containers and microvirtual machines. It proposes WebAssembly as a potential solution to these cold start problems, enabling faster execution of server-side applications and serverless functions. The authors in (Fujii et al., 2024) explored the migration of stateful virtual machines among various runtimes. They aimed to address the complexities of migrating VMs across different Wasm implementations, which lack standardized internal designs.

Previous works have examined WebAssembly for web applications and explored its use in edge computing, however, limited research has focused on comparing Wasm with native containers in multi-architecture scenarios, particularly for cloud-native applications. Several works have explored the use of WebAssembly in the cloud due to its cross-platform and security features. (Ferrari et al., 2021) studied Wasm performance on edge devices, while (Fiedler et al., 2020) analyzed Wasm startup latency in the context of serverless computing. (Felter et al., 2015) focused on docker container performance on x86 ar-

chitectures but did not explore Wasm.

WebAssembly’s ability to address constraints related to quality of service, hardware availability, and networking configurations has been a subject of investigation. The research work of (Hilbig et al., 2021) has explored an update on the usage of WebAssembly in the real world, highlighting vulnerabilities in insecure source languages and a growing diverse ecosystem. Another work by (Kakati and Brorsson, 2024b) presented the performance of the wasmtime runtime in the Sightglass benchmark suite focusing on architecture-specific outcomes.

Another study by (Chadha et al., 2023) explores the utilization of WebAssembly as a distribution format for MPI-based HPC applications, introducing MPIWasm to facilitate high-performance execution of Wasm code. The study demonstrates competitive native application performance, coupled with a significant reduction in binary size compared to statically linked binaries for standardized benchmarks.

(Bosshard, 2020) explore the execution of Wasm in various serverless contexts. It compares server-side Wasm runtime options, such as Wasmer, Wasmtime, and Lucet, and investigates running it within Openwhisk and AWS Lambda platforms. Experiments performed by (Spies and Mock, 2021) and (Wang et al., 2021) demonstrated Wasm’s usage using the benchmark PolyBench. Despite being an effective compiler benchmark suite, PolyBench appears to be unable to reflect applications in a wider range of non-web domains.

A previous investigation by (Hockley and Williamson, 2022) examined the use of WebAssembly as a sandboxed environment for general-purpose runtime scripting. However, the experiments are conducted on specific hardware configurations, which might not fully represent the performance on a broader range of devices and architectures. The performance of WebAssembly is compared to native and container-based applications on ARM architecture by (Mendki, 2020) with benchmarking across various application categories. (Kakati and Brorsson, 2023) highlights a WebAssembly survey as the changing landscape of cloud computing, with a shift toward edge computing, and emphasizes the need for a cross-platform and interoperable solution.

Continuing the exploration of WebAssembly, (Wang, 2022) addressed the gap in previous research by conducting a detailed analysis of standard Wasm runtimes, covering five widely used Wasm runtimes. They introduced a benchmark suite named WaBench, which includes tools from established benchmark suites and whole applications from various domains.

(Kakati and Brorsson, 2024a) provided a thor-

ough analysis of the two most prominent WebAssembly runtimes employing an extensive array of instrumented benchmarks and sheds light on Wasm’s potential to address the requirements of a cross-architecture cloud-to-edge application framework.

There are a multitude of architectures and different performance characteristics, making application development and deployment difficult. We need to ensure that software can run on as many different nodes as possible without specifically adapting to each hardware platform. This is an active research area addressed by several researchers, including (Mattia and Beraldi, 2021), (Nastic et al., 2021), and (Orive et al., 2022). There also exists an opportunity for exploration within the domain of multi-architecture native containers and the incorporation of WebAssembly into containerization.

4 METHODOLOGY

To understand Docker’s support for WebAssembly, it is imperative to understand the operational framework of the Docker Engine. The Docker Engine is predicated on a higher-level container runtime, `containerd`, which furnishes essential functionality for controlling the container lifecycle. Using a shim process, `containerd` can take advantage of the capabilities of `runc`, a low-level runtime used for native containers. Subsequently, `runc` interfaces directly with the operating system to manage diverse facets of container operations. This design offers the capability to develop a shim to integrate various runtimes with `containerd`, including WebAssembly runtimes. Consequently, it becomes feasible to seamlessly utilize different WebAssembly runtimes within Docker, such as Wasmtime, Spin, and Wasmedge. Our experiment in this regard encompassed three distinct architectures:

- **amd64:** The benchmarks were run on an AWS bare metal server (Linux, AMD64).
- **arm64:** The benchmark was run on an Nvidia Jetson Nano server (ARM64 architecture).
- **riscv64:** We used an additional benchmark that was also run on a StarFive VisionFive2 board with RISCv64 architecture.

This multi-architecture setup allowed us to evaluate the performance of WebAssembly containers against native containers across different hardware environments. The general methodology is shown in Fig.2.

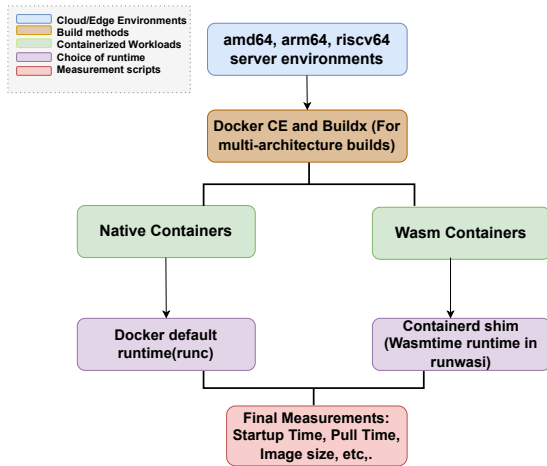


Figure 2: Overview of native and wasm container workflows

4.1 Workloads

Three workloads were designed to evaluate the performance of both native and Wasm containers:

- **Matrix Multiplication Workload:** Implements matrix multiplication in Rust, C++ and TinyGo, with corresponding Wasm versions. This workload tests computational performance in the context of heavy mathematical operations.
- **Sorting Workload:** Reads a text file, sorts its contents, and writes the sorted output to a new file. The sorting algorithm supports both "all at once" and "one at a time" processing and is implemented in Rust, Cpp, TinyGo, and their Wasm counterparts. This workload tests file I/O and memory handling capabilities.
- **Face Detection Workload:** Written in C++, we run this workload on three architectures and use containerd directly, thereby bypassing Docker entirely.

To investigate the performance and resource characteristics, we conducted experiments using three different hardware platforms with different instruction set architectures. Table 1 provides some details on the platforms used.

4.2 Container Configurations

For each workload, we containerized both native and WebAssembly implementations, all approaches are optimized to keep them comparable. Rust programs were built with `--release`, C++ programs with `-O3`, and TinyGo programs with `-opt=2`, because these are the highest optimizations available in their respective compilers, and the respective docker images are kept

to a minimum. Rust `--release` and C++ `-O3` are equivalent optimization levels, both focusing on maximum performance. TinyGo `-opt=2` is the highest optimization available for TinyGo, but it is not as aggressive as Rust's `--release` or C++'s `-O3`. Unfortunately, TinyGo does not have an equivalent to `-O3`, so we are using the best available option.

Native Containers: The Rust, TinyGo and Cpp applications were compiled natively for each architecture. We used Docker *multi-platform builds* (`--platform linux/amd64,linux/arm64`) to generate native containers compatible with both architectures. For the riscv64 target, we need to do it a little bit differently as there is no base image common for amd64, arm64 and riscv64, but the end result is the same, an image which contains a sub-image for each architecture.

The Dockerfiles were optimized through several key measures including a multi-stage build process, which separates the build environment from the runtime environment, ensuring that the final image is minimal and only includes necessary runtime dependencies. The first stage is the build stage during which we use a regular Linux base image such as Alpine (for amd64 and arm64) and an Ubuntu image for riscv64. We ensure that the final image is extremely small and contains only the application binary and its essential dependencies. These measures collectively contribute to an efficient and lightweight Docker image.

Wasm Containers: We compiled the Rust, TinyGo and Cpp applications to WebAssembly and executed them using the wasmtime runtime through `docker run`. To run these containers, the containerd wasmtime shim was installed which is provided by the `runwasi`⁴ project. The containerd runtime can provide support for Wasmtime as an alternative container runtime⁵. The runtime provided a secure execution environment with the added sandboxing offered by WebAssembly. The Dockerfile is kept minimal for running a wasm binary.

Again, by using "scratch" as the base image, we ensure that the final image is lightweight, containing only the Wasm binary and no additional layers or dependencies. This approach minimizes the attack surface and significantly reduces the image size. This setup is ideal for environments where resource efficiency and security are paramount. We used the command `--platform wasm` and

⁴<https://github.com/containerd/runwasi>

⁵<https://docs.docker.com/engine/daemon/alternative-runtimes/>

Table 1: Hardware Architectures Used in Experiments

Platform	CPU Model	Clock frequency	Threads per Core	Cores per Socket	Caches
X86_64 AWS c5.metal	Intel Xeon 8275CL	3 GHz	2	24	L1d: 32 KiB, L1i: 32 KiB, L2: 1 MiB, L3: 71.5 MiB
Nvidia Jetson Nano	ARM Cortex-A57	1.5 GHz	2	4	L1d: 32 KiB, L1i: 48 KiB, L2: 2048 KiB
StarFive VisionFive2	SiFive U74 Core	1 GHz	1	4	L1d: 32 KiB, L1i: 32 KiB, L2: 2048 KiB

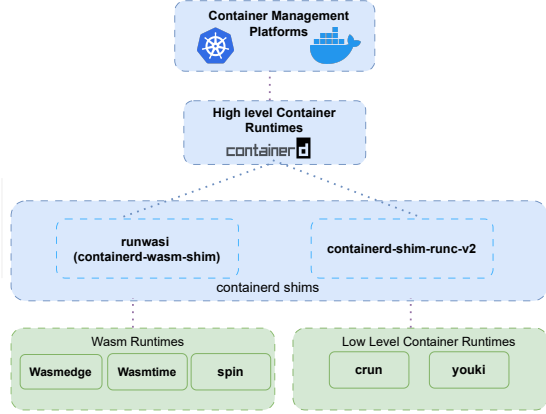


Figure 3: Wasi support with containerd shims.

`runtime=io.containerd.wasmtime.v1` to execute the wasm containers. For native containers, the default docker runtime was used.

The general workflow of WebAssembly and the use of it as an alternative docker runtime is shown in Fig. 3. As illustrated in this figure, when running a program using the `docker run` command, docker will use `containerd` to manage the container. First, it will check the local image store if it is already present locally, and if not, it will pull a fresh image from the used image registry. In our case, the Docker hub.

4.3 Benchmarking Approach

We measured the startup time of each container (both native and Wasm) using a custom script that detects the system architecture, allowing it to run appropriate container images for either AMD64 or ARM64. We instrumented the source code to capture the current time at the beginning of the main function, providing a precise timestamp for when the main function starts to execute. In addition to measuring the time until the main function starts, the script also logs the image pull time, ensuring that we capture the overhead of pulling the container image when a fresh pull is required.

To ensure accurate measurements, we use the ‘date’ command for high-resolution timestamps and employ Docker’s built-in functionality to run containers. Before running each experiment, the script force-removes the image if a fresh pull is specified, ensuring

a cold start, which provides a more accurate measurement of the startup time. The overall approach combines internal timing measurements within the container and external validation through the execution of the Docker container, ensuring consistency and reliability across different environments.

The measured time is categorized into two distinct metrics: *Pull Time*, defined as the time from requesting to run the container until it has been brought in by pulling it from the remote OCI registry, and *Container Time*, which reflects the duration taken to load and initialize the image for execution until the program starts its main function. This reflects the startup time of the container.

The script supports testing with forced fresh pulls and using cached images, allowing us to assess the performance of the containerization approach in different scenarios⁶. This detailed benchmarking method provides insights into the performance characteristics of native versus Wasm containers in multi-architecture cloud environments. The script runs the containers in both architectures (amd64 and aarch64), performs the workload, and calculates the startup time based on the elapsed time between image pull and the execution of the container’s main function.

The AWS instance represents powerful cloud datacenter server instances. The Intel Xeon 8275CL CPU is a powerful processor with advanced parallel pipelines using dynamic instruction scheduling, branch prediction, and cache prefetching mechanisms. In contrast, the Nvidia Jetson and StarFive VisionFive2 boards represent the embedded domain more likely to be found at the edge.

5 RESULTS

To evaluate startup times for native and Wasm containers on file sorting and matrix multiplication workloads, we initially followed best practices by employing multistage Dockerfile setups and optimized base images as shown in Figure 4 for the Matrix Multiplication program and Figure 5 for the Sorting benchmark. The results slightly favors native containers, with smaller differences in image sizes and startup times between native and Wasm, despite the tradition-

⁶https://github.com/sangeeta1998/benchmark_sort/

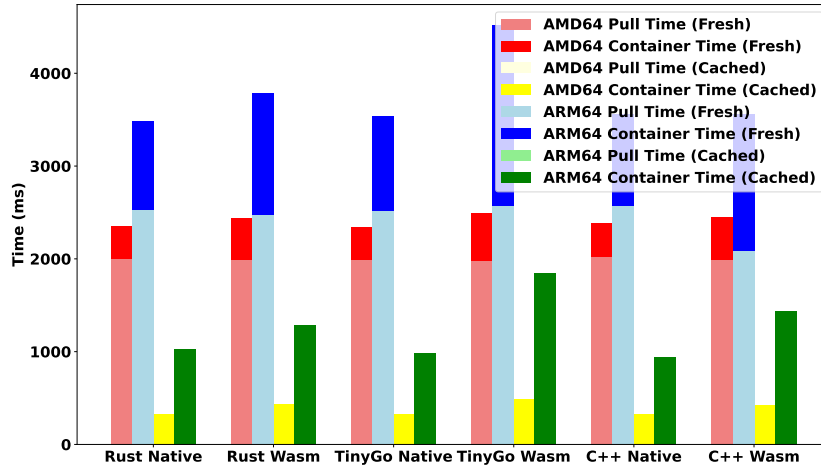


Figure 4: Startup Times (image pull and container time) for Matrix Multiplication.

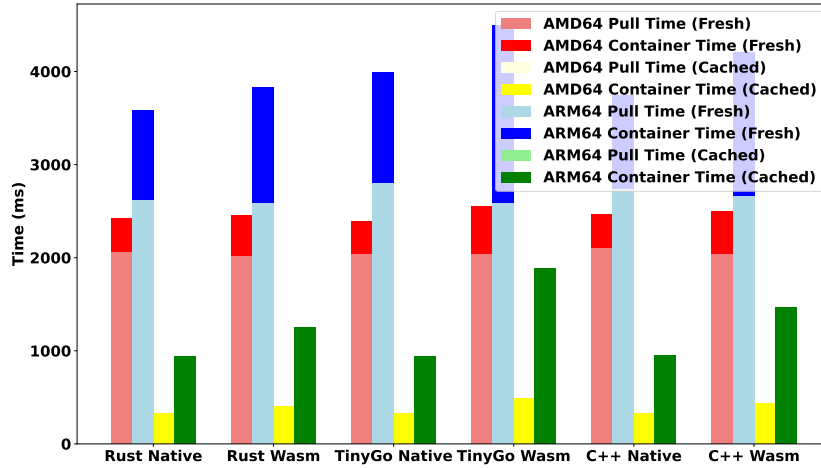


Figure 5: Startup Times (image pull and container time) for Sorting Benchmark.

ally smaller size of Wasm. The main reason is that although WebAssembly execution is fast and competitive, as has been shown by (Kakati and Brorsson, 2024a), there is still some overhead compared to native execution.

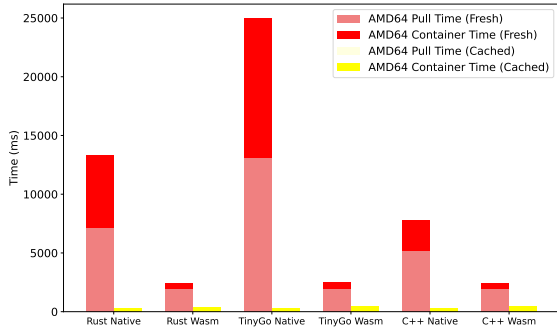
Looking at Figures 4 and 5 we see in the red and blue bars, the image pull time and container time for all architectures and source languages. The yellow and green bars indicate the startup time when the image can be found locally in the Docker image cache.

We first observe that for the AMD64 architecture, the image pull time (light red) is almost the same for both native and Wasm images and also across all source languages. For ARM64, there are some small deviations: The matrix multiplication C++ Wasm image is pulled slightly faster than the native image. The reason can be found in the large difference in image size, see Figure 7. A similar situation occurs for Sort with TinyGo.

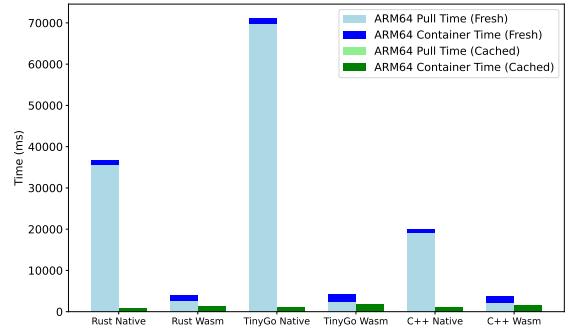
When it comes to the actual startup time after the container image has been pulled from the remote registry, the time is normally longer for Wasm containers than for native. This is normal considering that there is an overhead of compiling the Wasm bytecode that is in the critical path of execution. Some source code languages are more sensitive here, in particular TinyGo which might indicate that the Wasm generation in the TinyGo compiler is not yet as mature as for Rust and C++.

5.1 Evaluation of Startup Times: Impact of Dockerfile Practices

To closely mirror the scenarios often seen in related work, which suggests Wasm has significantly lower startup times than we saw in the previous section, we conducted a revised experiment on matrix multipli-



(a) Average Startup Time in AMD64.



(b) Average Startup Time in ARM64.

Figure 6: Pull and Startup Time for AM64 and ARM64: Without best Practices.

cation workload for native containers without scratch and multistage optimizations. In this configuration, image sizes and startup times were notably impacted, and prominent differences in native vs. wasm startup times can be seen. The Alpine-based C++ (433 MB) produced a minimal image size due to the compact nature of its dependencies. In contrast, Rust (927 MB) required a larger Alpine image to include the Rust toolchain. TinyGo (1.55 GB) relied on the `tinygo/tinygo:latest` image as a base, as TinyGo lacks native Alpine support for arm64.

The results shown in Figure 6 reveal a substantial improvement in the start times of Wasm compared to native containers, particularly on AMD64. For fresh pull and container startup times on AMD64, Native Rust registered a pull time of 7166 ms, with an additional container startup of 6190 ms. Meanwhile, Wasm Rust showed a much faster startup at 1996 ms for both pull and container times, underscoring Wasm’s more efficient startup under these conditions.

Specifically, Wasm Rust’s startup time was approximately 72% faster than the combined pull and startup time of Native Rust. Similar trends emerged for TinyGo and C++ images: TinyGo Native required significantly longer for both pulling and startup compared to Wasm, which achieved an 85% faster overall startup. Native C++ also showed similar results, with Wasm C++ maintaining faster performance, resulting in an 82% faster total startup time.

For ARM64, native images demonstrated greater overhead relative to Wasm, though the performance gap between architectures narrowed slightly. Native Rust showed a higher combined pull and startup time, while Wasm Rust achieved a faster overall start, being 66% quicker. TinyGo on ARM64 followed a similar trend, where Wasm’s total startup time was 77% faster than Native. Native C++ on ARM64 required longer times overall, with Wasm C++ reducing this signifi-

cantly, achieving a 69% faster total startup time.

In summary, this additional experiment reveals that Wasm consistently achieves faster startup times than native containers under *nonoptimized Docker configurations*, particularly on AMD64. While native images in optimized setups can perform similarly to Wasm, removing best practices reveals Wasm’s potential for faster initialization, making it suitable for performance-sensitive applications across both AMD64 and ARM64 architectures.

On the other hand, if we are using the ultimate best practices for native multi-platform built images, Wasm provides no additional benefits in performance and remains within an acceptable range. These results highlight Wasm’s suitability for multi-architecture environments, suggesting that further optimizations can closely match native performance across platforms, offering viable cross-architecture consistency.

While Wasm shows notable advantages in startup time and consistency across architectures, our experiments did not exhaustively compare runtimes for languages with larger dependencies, such as Java, C, Python, and JavaScript. We anticipate that these languages may yield different performance results due to their larger runtime requirements, providing avenues for further study.

For all the other results presented in this paper, we use optimized native docker images.

5.2 Image Size Comparison

Wasm containers offer significantly reduced image sizes compared to native containers, making them optimal for resource-constrained environments. Figure 7 shows the image sizes at the host architecture when they have been pulled and cached locally. Figure 8 shows the compressed sizes at the OCI Registry (DockerHub) for the same images. Note, however, that this is only for one architecture. The real size

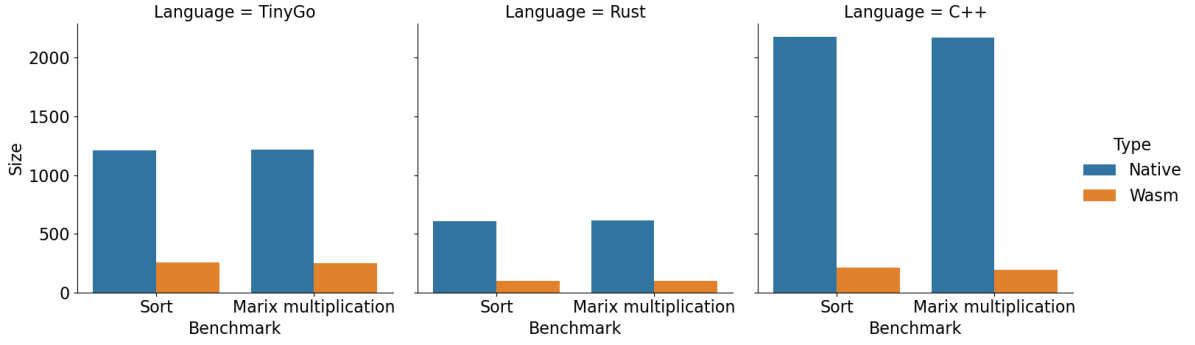


Figure 7: Host-based Image Sizes comparison.

stored at the registry in order to support multiple architectures should be multiplied with the number of architectures (the image size is roughly the same size for each architecture).

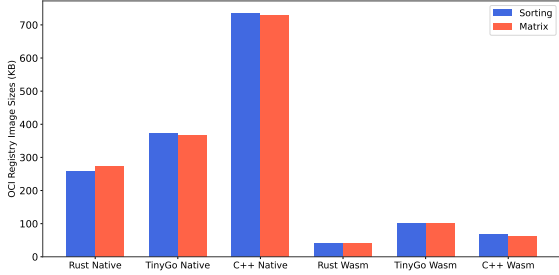


Figure 8: Image Sizes from OCI Registry for the AMD64 architecture.

5.3 Host Architecture Sizes vs. OCI Registry Sizes

Table 2 and 3 summarize the image sizes for both workloads comparing the image size stored in the registry (OCI size) to the one in the image runtime. The registry images are compressed to save storage space and transfer time. The reduction in size from the host architecture to the OCI registry is calculated as shown in Equation 1.

$$\text{Reduction} = \left(1 - \frac{\text{OCI Size}}{\text{Host Size}}\right) \times 100 \quad (1)$$

Table 2: Host Architecture Sizes vs. OCI Registry Sizes for Sorting Benchmark.

Language	Host Size (KB)	OCI Size (KB)	Reduction (%)
TinyGo Native	1210	373.36	69.2
Rust Native	610	259.27	57.5
C++ Native	2180	746.24	65.8
TinyGo WASM	257	100.31	61.0
Rust WASM	96	40.79	57.5
C++ WASM	212	68.5	67.7

Table 3: Host Architecture Sizes vs. OCI Registry Sizes for Matrix Multiplication Benchmark.

Language	Host Size (KB)	OCI Size (KB)	Reduction (%)
TinyGo Native	1220	366.46	69.9
Rust Native	613	273	55.5
C++ Native	2170	735.43	66.1
TinyGo WASM	250	101.26	59.5
Rust WASM	96.1	42.25	56.1
C++ WASM	192	60.38	68.6

When an OCI image is pulled, the compressed version of the image layers is transferred over the network. The decompression happens on the client side after the image layers are downloaded. This approach minimizes the amount of data transferred, making the process more efficient.

Given the small images we are working with here, we do not see much difference in pull time between native images over Wasm images, even when the size difference is large. The docker software introduces some overhead in addition to the overhead of containerd, which is the service that actually does the image management.

The fresh startup time for a Docker container involves several steps:

1. **Calling Docker and checking local image cache:** The initial command is sent to the Docker daemon and it checks if the image is already available locally.
2. **Request time from and in Docker Hub:** If the image is not available locally, Docker requests it from the registry. The registry processes the request and starts sending the image layers.
3. **Getting the image back and Decompressing:** The image layers are downloaded to the local machine. The downloaded layers are decompressed.
4. **Loading image to container runtime and start execution:** The image is loaded into the container runtime, and the container is started.

It is only step 3 and to some extent step 4, which are affected by the image size.

Table 4: Image sizes across architectures.

Architecture	Image Size (MB)
amd64	1.65
arm64	1.55
riscv64	1.55
wasm	0.816

5.4 Using Containerd to handle Images

In this section, we present the performance results for the face detection benchmark, implemented in C++ with `-O3` optimization. The application was containerized for three target architectures: `amd64`, `arm64`, and `riscv64`. Notably, the Dockerfiles for `amd64` and `arm64` are identical, while the Dockerfile for `riscv64` differs only in the base image used to accommodate the architecture. The image sizes across different architectures are summarized in Table 4.

We perform the benchmark using `containerd`, bypassing Docker, to directly evaluate the image pull times. This allowed us to compare the performance when using `containerd` as opposed to Docker.

The Dockerfiles for the native images (`amd64`, `arm64`, `riscv64`) follow a similar structure:

- **Build Stage:** The source code is compiled using `clang++` with optimizations set to `-O3`. This includes compiling several C++ source files which are then linked to the final executable (`facedetection`).
- **Runtime Stage:** The runtime image is based on the `scratch` image to minimize its size. The compiled `facedetection` executable and an input image are copied into the image.

5.4.1 Image Pull Times

We measured the average image pull time over 10 iterations for each architecture using `ctr`. The results are presented in Fig. 9.

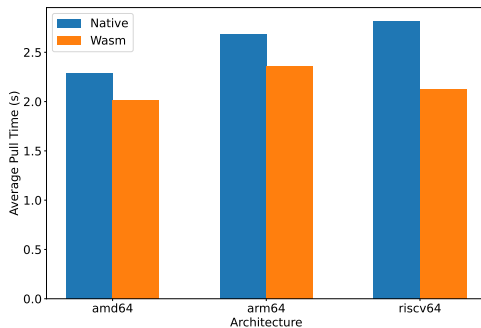


Figure 9: Containerd Image Pull Time

The results show that the Wasm containers has pull times considerably faster than the native containers. The results also demonstrate the effectiveness of using `containerd` for managing containers, as it provides faster image pull times. These findings contribute to understanding the trade-offs between different architectures and container runtimes in the context of performance benchmarks.

6 CONCLUSION

In response to the growing demand for efficient and portable cloud-native computing, we have investigated the performance implications of WebAssembly (Wasm) containers compared to native multi-architecture containers. Through benchmarking across three distinct workloads on ARM64, AMD64 and RISC64 architectures, we found that native containers exhibited slightly faster startup times when using highly optimized build process.

Wasm containers demonstrated a significantly smaller image size, averaging approximately 15% of the size of their native counterparts (see Figure 7), making it a compelling choice for resource-constrained environments, where memory and storage footprint are critical.

An important aspect of this study was the use of `containerd` instead of `docker` to manage containers. This direct approach to container management resulted in up to 25% faster pull times for Wasm images across all architectures. In particular, the use of `containerd` showcased an advantage in efficiency when using container runtimes that bypass the Docker layer.

Our contributions include a comprehensive evaluation of Wasm’s integration with Docker’s alternative runtime, *Wasmtime in runwasi-containerd shim*, across three architectures, along with a detailed performance comparison between native multi-architecture containers and Wasm containers based on key metrics such as startup time, pull time, and image size. The integration of Wasm into containerized environments offers a significant opportunity to optimize performance, and instead of taking the place of the existing container paradigm, it is more likely to be integrated into that framework. Therefore, Wasm will not replace containers; rather, they will function as complementary technologies.

ACKNOWLEDGMENT

This research has been partly funded by the Luxembourg National Research Fund (FNR) under contract number 16327771 and has been supported by Proximus Luxembourg SA. For the purpose of open access, and in fulfillment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

REFERENCES

- Bosshard, B. (2020). On the use of web assembly in a serverless context. In *Agile Processes in Software Engineering and Extreme Programming-Workshops*, page 141.
- Chadha, M., Krueger, N., John, J., Jindal, A., Gerndt, M., and Benedict, S. (2023). Exploring the use of webassembly in hpc. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 92–106.
- Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. IEEE.
- Ferrari, D., Sciascio, E., Gioiosa, R., Krieger, O., Krikellas, K., and Giuliani, G. (2021). Webassembly for cloud-native applications: A performance evaluation. *Future Generation Computer Systems*, 115:335–348.
- Fiedler, M., Alrowaily, H., Menzel, K., and Stiller, B. (2020). Performance evaluation of webassembly as a cloud-edge continuum technology. In *2020 IEEE 6th International Conference on Network Softwarization (NetSoft)*, pages 334–340. IEEE.
- Fujii, D., Matsubara, K., and Nakata, Y. (2024). Stateful vm migration among heterogeneous webassembly runtimes for efficient edge-cloud collaborations. In *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking*, pages 19–24.
- Gackstatter, P., Frangoudis, P. A., and Dustdar, S. (2022). Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149. IEEE.
- Gkonis, P., Giannopoulos, A., Trakadas, P., Masip-Bruin, X., and D’Andria, F. (2023). A survey on iot-edge-cloud continuum systems: Status, challenges, use cases, and open issues. *Future Internet*, 15(12):383.
- Hilbig, A., Lehmann, D., and Pradel, M. (2021). An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*, pages 2696–2708.
- Hockley, D. and Williamson, C. (2022). Benchmarking runtime scripting performance in webassembly.
- Kakati, S. and Brorsson, M. (2023). Webassembly beyond the web: A review for the edge-cloud continuum. In *2023 3rd International Conference on Intelligent Technologies (CONIT)*, pages 1–8. IEEE.
- Kakati, S. and Brorsson, M. (2024a). A cross-architecture evaluation of webassembly in the cloud-edge continuum. *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 337–346.
- Kakati, S. and Brorsson, M. (2024b). An investigative study of webassembly performance in cloud-to-edge. In *2024 International Symposium on Parallel Computing and Distributed Systems (PCDS)*, pages 1–5.
- Kjorveziroski, V., Filiposka, S., and Mishev, A. (2022). Evaluating webassembly for orchestrated deployment of serverless functions. In *2022 30th Telecommunications Forum (TELFOR)*, pages 1–4. IEEE.
- Mattia, G. P. and Beraldi, R. (2021). Leveraging Reinforcement Learning for online scheduling of real-time tasks in the Edge/Fog-to-Cloud computing continuum. In *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, pages 1–9. ISSN: 2643-7929.
- Mendki, P. (2020). Evaluating webassembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*, pages 161–166.
- Nastic, S., Pusztai, T., Morichetta, A., Pujol, V. C., Dustdar, S., Vii, D., and Xiong, Y. (2021). Polaris Scheduler: Edge Sensitive and SLO Aware Workload Scheduling in Cloud-Edge-IoT Clusters. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 206–216. ISSN: 2159-6190.
- Orive, A., Agirre, A., Truong, H.-L., Sarachaga, I., and Marcos, M. (2022). Quality of Service Aware Orchestration for Cloud-Edge Continuum Applications. *Sensors*, 22(5):1755. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.
- Pham, S., Oliveira, K., and Lung, C.-H. (2023). Webassembly modules as alternative to docker containers in iot application development. In *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, pages 519–524. IEEE.
- Spies, B. and Mock, M. (2021). An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10. IEEE.
- Wang, W. (2022). How far we’ve come—a characterization study of standalone webassembly runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 228–241. IEEE.
- Wang, Z., Wang, J., Wang, Z., and Hu, Y. (2021). Characterization and implication of edge webassembly runtimes. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 71–80. IEEE.