Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Memory-Efficient WebAssembly Containers

Author: Maciej Kozub (2776056)

1st supervisor:	Dr. Daniele Bonetta
daily supervisor:	Matthijs Jansen, MSc
2nd reader:	Dr. Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 17, 2024

Abstract

WebAssembly (Wasm) is a binary instruction format originally designed for web browsers, which offers a portable and high-performance execution environment. While Wasm's use on the client side is well-known, its deployment on the server side has recently gained traction, especially within containerized environments managed by Kubernetes. However, integrating Wasm into container runtimes presents challenges related to memory overhead and performance efficiency.

This thesis explores the memory efficiency of Wasm when deployed within containers. Specifically, the research focuses on integrating a lightweight Wasm runtime - WebAssembly Micro Runtime (WAMR) - into the crun container runtime. The goal is to address the memory overhead observed with current Wasm-enabled container runtimes and make Wasm containers a more competitive alternative to traditional non-Wasm containers.

The study begins by analyzing the current support for Wasm within container runtimes and identifying the inefficiencies in memory usage. Following this, a new integration of WAMR into crun is designed and implemented, with careful consideration of the runtime's compatibility with existing container infrastructures and orchestrators. The performance of this new integration is then evaluated through a series of benchmarks, comparing memory usage and startup times against existing solutions.

The results demonstrate that the WAMR integration significantly reduces memory overhead, making Wasm containers more viable for large-scale container deployment. This work contributes to the growing body of research on WebAssembly in server-side applications and provides practical insights into optimizing Wasm runtimes for containerized environments. Future work could explore further optimizations and the potential for broader adoption of Wasm in cloud-native computing.

All related code and detailed reproducibility notes are publicly available for further research and development.

Contents

1

Introduction

In recent years, the landscape of cloud computing has undergone a significant transformation. The adoption of containerized applications has become widespread, largely due to the need for scalable, flexible, and maintainable software deployments (1). Containers are a cost-effective yet high-performing technique for isolating resources. Applications can operate in separate contexts, sharing the host operating system kernel, filesystem, and resources (2). Orchestration platforms like Kubernetes manage these containerized environments. In the last decade, Kubernetes revolutionized the computing landscape and became a popular choice for orchestrating containerized workloads at scale (3). The open-source nature of Kubernetes and its cloud-native approach have made it a widely adopted industry standard for cloud orchestration, significantly impacting computing practices (4). According to recent statistics, over 60% of enterprises have adopted Kubernetes, and it is projected to surge past 90% by 2027 (5). At the same time, SlashData reports that 5.6 million developers globally are using Kubernetes and that the global sale of containerized solutions is projected to reach 1.195 billion dollars in 2022 (6).

Along with the advantages brought by containerization, there is also a need for efficient container execution since running applications in containers creates an overhead that translates to higher resource usage and energy consumption (7). The surge in demand for cloud computing services has further exacerbated data centers' energy consumption and carbon footprint, making energy costs one of the top operational expenses (8). Moreover, in large-scale deployment environments, the high velocity of change in the number of running containers leads to spikes in resource utilization. To maintain high availability and scalability of services, cluster providers need to accommodate more hardware, which further increases the operational costs (9).

WebAssembly is seen as an answer to efficiency challenges posed by traditional technologies, including containerization (10). WebAssembly (Wasm) has emerged as a fast and secure cross-platform compilation target (11). Initially designed for web browsers, WebAssembly is a binary instruction format that provides a portable, high-performance execution environment for code written in multiple higher-level languages like Go, Rust, or C. The performance of WebAssembly compared to native code has been a key focus, with studies showing that applications compiled to WebAssembly typically run only about 10% slower than native code (12). This performance parity with native code and portability makes it an attractive option for implementing cloud-native containerized applications.

One of the key advantages of WebAssembly is its potential to produce smaller container images, which aligns with the trend of reducing container image size in cloud environments (13). To deliver a Wasm workload as a container image, we can create an OCI (Open Container Initiative) compatible image without any underlying layer of the base image. Such OCI image consists only of the Wasm binary file. This reduction in image size translates to reduced storage requirements and faster deployment times (14). Moreover, WebAssembly's security model is built on a sandboxed execution environment, which isolates the running code from the host system and hardware (15). This isolation enhances security by limiting the potential attack surface, making WebAssembly suitable for running untrusted code safely.

1.1 Context

In recent years, containers have become a primary choice for application deployment, especially for environments where a small container image size and low memory footprint are desired (16). Given the extensive popularity of containerization among organizations and the characteristics of WebAssembly, integrating Wasm into the existing container infrastructure presents a compelling opportunity. Such integration should be done with minimal changes to the current deployment infrastructure to minimize introductory costs and complexity.

Enabling support for running Wasm containers in container environments can be achieved using a low-level container runtime that supports Wasm workloads (17). Currently, there are only two OCI-compatible runtimes that support executing Wasm workloads, namely crun (18) and youki (19). Another approach to support Wasm containers advised by CNCF (Cloud Native Computing Foundation) is to use a containerd project called runwasi (20). Runwasi delivers containerd-wasm-shims that execute the Wasm module by invoking the

Wasm runtime directly without relying on the low-level runtime. A more in-depth explanation of container runtimes and support for Wasm can be found in Section 3.1.

Using Wasm-enabled container runtimes under the container orchestrator like Kubernetes allows for seamless Wasm application deployment next to the standard containerized applications. However, we have observed that the memory footprint of a Kubernetes pod containing just one container was higher in the case of a Wasm container than the non-Wasm container. We consider it counter-intuitive, and we have observed the same result when running the Wasm containers using both ways proposed by CNCF, namely the low-level container runtime and runwasi project. Additionally, we have seen the same results regardless of the underlying Wasm runtime used. Both crun, youki runtimes, and the runwasi project support the same limited set of Wasm runtimes, that is, WasmEdge (21), Wasmtime (22), and Wasmer (23). This observation leads us to the problem statement.

1.2 Problem Statement

While Wasm containers are known for their small container image sizes and low startup times, the current integration of Wasm into container runtimes results in memory overhead and startup times exceeding that of traditional containers. This inefficiency invalidates the known advantages of Wasm over non-Wasm containers and slows down Webasmebly's future development towards the containerized world. Therefore, work should be done to lower the memory usage of Wasm containers to match or use less memory than the non-Wasm containers. For this purpose, a new, lightweight Wasm runtime should be integrated into the crun or youki low-level runtimes or integrated with the runwasi project. Thanks to memory-efficient WebAssembly runtime, the overall memory use of Wasm containers should be lower. Such a new feature was already requested on the official GitHub repositories of crun and runwasi in September 2023. However, until this thesis, no successful attempts were published to complete such implementation.

Since WeAssembly is a compilation target for many popular programming languages, solving the high-memory overhead of Wasm containers without compromising other performance metrics could lead to bootstrapping the adoption of WebAssembly in containerized applications. It would become a compelling alternative to standard containers with an execution environment included in an image. Organizations could develop highly portable cross-platform applications that can be distributed using minimal-size images that lower storage requirements and network usage. Moreover, a more memory-efficient Wasm-enabled container runtime would enable professionals to deploy Wasm applications

on currently managed infrastructures while minimizing memory utilization, enabling higher scalability, and lowering the costs and environmental footprint of running the applications.

1.3 Research Questions

Our general objective is to lower the memory footprint of Wasm OCI containers and make them a more competitive alternative to non-Wasm containers an execution environment included in the container image. We aim to achieve that by embedding a new Wasm runtime into the container runtime, following the CNCF guidelines. The selection of this new Wasm runtime should prefer the runtime with minimal memory footprint and performance paired with the one of WasmEdge, Wasmtime, and Wasmer. The formulation and addressing of the following two research questions aid in achieving our goal.

RQ1

How to integrate a new, more lightweight Wasm runtime into container runtimes such as crun, youki, or runwasi that will lower the memory footprint of Wasm containers?

The answer to the RQ1 is this thesis's main novel contribution. It aims to solve the high memory overhead of running a Wasm container, minimizing memory overhead for virtualized applications. To answer this research question, a deep understanding of container execution flow must be obtained. It has to be paired with researching each of the mentioned projects, namely crun, youki, and runwasi, including source code exploration. Research and selection of an appropriate Wasm runtime must also be executed. As mentioned above, the final answer should allow us to embed a more lightweight Wasm runtime into one of the projects and successfully use it to run OCI containers with the Wasm module under a Kubernetes orchestrator.

RQ2

What is the memory footprint and startup time of our new Wasm-enabled container runtime compared to the currently available container runtimes?

Answering RQ2 involves performing a benchmark focused on measuring the memory footprint of Wasm containers. The benchmark should evaluate and compare the memory usage of the container's workloads executed using different underlying Wasm runtimes to showcase if the newly embedded Wasm runtime meets the desired requirements. Moreover,

the benchmark should also reference the results with the base case of a non-Wasm container to gain a broader understanding of the results in the context of other technologies. This should help with further evaluation if the deliveries of this thesis pose promising and beneficial novelty in the state-of-the-art of containerization.

1.4 Research Methodology

The following research methodologies are applied to answer the above research questions systematically:

- **(Methodology M1) Design, abstraction, prototyping** (24, 25, 26);

  A new Wasm runtime was integrated into a container execution runtime (**RQ1**) in an iterative way with a clear distinction between the design and prototyping phases. First, the requirements for the final prototype were set; next, the iterative prototyping began. All required features, like initialization of the Wasm environment, ability to set WASI arguments, ability to redirect the standard output from the Wasm module, etc, were added subsequently. The prototypes were first tested directly on the development environment; later, they were tested on the desired benchmark environment under the Kubernetes orchestrator.

  A similar process was put in place for designing and prototyping the benchmarks (**RQ2**) used to evaluate the new integrated Wasm runtime. Work started with setting the requirements for a wanted design. Later modifications were done to the Continuum Framework (27), which enabled the automated setup of the desired benchmark environment.

- **(M2) Experimental research, designing appropriate micro- and workload-level benchmarks, quantifying a running system prototype** (28, 29, 30)

  The memory overhead of the newly introduced Wasm runtime integration (**RQ1**) was benchmarked with experimental research methods. The benchmarks started with deploying ten pods with one Wasm container each and ranged up to four hundred pods deployed simultaneously, which fully utilized the resources that were at our disposal.

- **(M3) Open science, open source software, community building, peer-reviewed scientific publications, reproducible experiments** (31, 32, 33, 34)

All of our work has been made publicly available in GitHub repositories. A detailed guide to reproducibility can be found in Appendix A. Throughout our thesis, we use open-source software, mainly Kubernetes, crun, WebAssembly Micro Runtime, and Continuum. We have also used publicly available software documentation and community knowledge gathered from sources like GitHub Issues.

1.5 Thesis Contributions

By addressing the previously discussed research questions, this thesis makes the following contributions:

- **(Contribution C1, artifact)** Integration of a lightweight WebAssembly runtime (WebAssembly Micro Runtime) into a low-level container runtime (crun) to lower the memory footprint of running Wasm OCI containers.
- **(C2, artifact)** Guidelines on how to enable deployment of more than 110 pods on one Kubernetes node. This additional contribution results from our need to comprehensively evaluate our work at scale.
- **(C3, experimental)** Extension of the Continuum framework enabling automated deployment and metrics collection of Wasm containers in Kubernetes cluster. This includes the memory footprint analysis of Wasm containers executed with crun, containerd-wasm-shims, and comparison with non-Wasm containers based on a Python Docker image.

1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

1.7 Thesis Structure

This thesis is organized to systematically address the goal of integrating a lightweight WebAssembly runtime into an OCI container runtime. This introductory Chapter 1 sets the stage by discussing the context, problem statement, and research questions. We also declared the methodology used and showcased the main contributions.

Next, Chapter 2 delves into the background knowledge that we find essential to fully understand this thesis's novel contributions and artifacts. We introduce the technology behind the WebAssembly and the Continuum framework we use to validate our work. Chapter 3 presents the design decisions and requirements analysis set for our work. After this, Chapter 4 details the technical implementation, including building, installing, and configuring the necessary infrastructure to support Wasm workloads in Kubernetes. This is followed by evaluating our work in Chapter 5, where we measure and compare our implementation's memory usage and startup performance with current state-of-the-art technologies. In Chapter 6, we review related work in the field, situating this thesis within the broader research landscape. This is followed by Chapter 7, which concludes the thesis by summarizing the findings, once again answering the research questions, discussing limitations, and suggesting future work.

2

Background

This chapter is organized into three main subsections, each building upon the previous to provide the necessary background knowledge to understand the key concepts and technologies relevant to this thesis. The first section introduces WebAssembly 2.1, covering its origins, execution model, and the advantages it offers for cross-platform development. We also discuss the WebAssembly System Interface (WASI), which extends Wasm's capabilities for non-browser environments, enabling broader application scenarios. The second section examines container orchestration 2.2, focusing on Kubernetes and detailing its architecture, components, and the role of container runtimes. Understanding how container runtimes are used by container orchestrators is crucial for grasping how WebAssembly can be integrated and managed within containerized environments. The last section 2.3 explains the Continuum framework used in our experimental setup. It highlights how Continuum facilitates the creation of virtual machines, deployment of applications, and collection of performance metrics, ensuring reliable and reproducible benchmarks.

The motivation behind explaining WebAssembly, container orchestration, and the Continuum framework lies in their interdependent roles in our research. Without a grasp of WebAssembly's execution environment and container orchestration, including container runtimes, it might be challenging to understand the design decisions and technical implementations discussed later in Chapter 3 and Chapter 4. The subsequent chapters build on this background to detail the design, implementation, and evaluation of our new WebAssembly runtime integration.

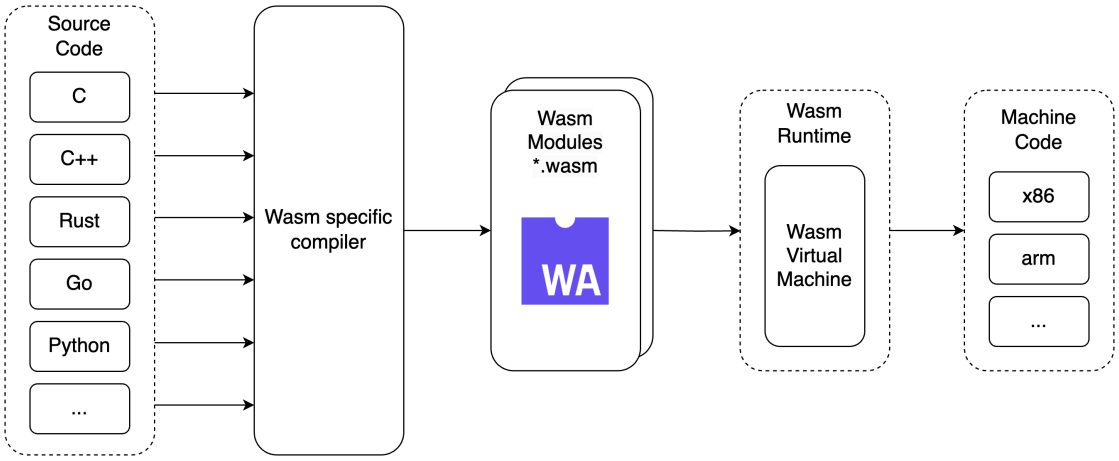

Figure 2.1: WebAssembly module creation and execution process.

2.1 WebAssembly

WebAssembly, also referred to as Wasm, is a stack-based virtual machine, and unlike register-based execution targets, Wasm does not require different instructions to execute on various physical machine architectures; its code runs consistently across all platforms (35). Wasm was initially designed for web browsers, but it quickly found use outside web browsers as it emerged as a fast and secure cross-platform compilation target (36). The WebAssembly provides an optimized and compact binary format that can be quickly decoded and executed. The Wasm module declares all types, functions, tables, memories, and globals required for execution and defines how they interact with one another and the outside world. Moreover, Wasm modules operate within a secure sandboxed environment, which uses fault isolation and linear memory access checks at the region level. Those techniques prevent buffer overflows and out-of-bounds memory access (37).

Figure 2.1 illustrates the process of converting high-level source code into executable machine code through WebAssembly technology. A Wasm-specific compiler must translate the source code into WebAssembly modules, where each module is represented with the ".wasm" file extension. Next, the WebAssembly runtime environment can execute these Wasm modules. This runtime environment incorporates a WebAssembly virtual machine that executes the modules in secure sandboxes. Within the runtime environment, the Wasm binary is interpreted or compiled into native machine code corresponding to the host architecture, such as x86 or ARM. This multi-step pipeline ensures the efficient execution of WebAssembly modules across diverse hardware platforms, enhancing portability.

2.1.1 WASI

The WebAssembly System Interface, or simply WASI, is a crucial component that extends the capabilities of WebAssembly beyond its original design for web browsers. WASI establishes a POSIX-like interface that enables standalone WebAssembly environments to engage in I/O operations and access external resources (38). The WASI is specifically designed to offer functions to WebAssembly modules similar to those available for native applications, empowering Wasm modules to manage files and networking operations (39). In essence, the WASI acts as a mediator between WebAssembly applications and the underlying system, furnishing a standardized interface for interacting with system resources across various computing platforms. This means that WASI further enhances the compatibility and portability of WebAssembly (40). From a security perspective, the WASI is crucial for ensuring the secure and isolated execution of WebAssembly modules by imposing restrictions on system-level interactions. By enforcing these restrictions, WASI prevents unauthorized access to system-level resources and ensures that code executes within previously defined boundaries (41).

2.2 Container Orchestration

Container orchestration tools automate the management of containerized applications, which involves deploying, scaling, and operating containers across a cluster of machines (42). Kubernetes is the most widely used container orchestration platform, providing robust features for managing containerized workloads. Kubernetes architecture comprises several integral components that collaboratively manage the lifecycle of containers. The control plane node oversees the cluster's state through components such as the API server, scheduler, controller, and manager. Worker nodes are responsible for running containerized applications and include components like the kubelet, kube-proxy, and a container runtime like containerd (43).

Kubernetes introduces several key concepts and terminologies essential for understanding its operations. The smallest deployable units in Kubernetes are called pods. They represent a single instance of running processes in a cluster. They can contain one or more containers that share storage and network resources within a pod. To facilitate communication within and outside the cluster, services are used; they abstract and expose a pod or set of pods as network services. Deployments can be used to deploy pods as well; they manage not only the pods and containers inside them but also the scaling of pods, ensuring the desired number of replicas are running at any given time. Finally, the mechanisms provided by

configuration maps and secrets can be used to inject configuration data and sensitive information into containers. Together, these components form the robust foundation of Kubernetes, enabling efficient and scalable container orchestration.

2.2.1 Container Runtimes

Container runtimes are essential components in the container ecosystem. They are responsible for creating, starting, and stopping containers and managing their execution environment. In container orchestration with tools like Kubernetes, container runtimes can be broadly categorized into high-level and low-level. Each category serves distinct roles in container life-cycle management. High-level container runtimes provide a user-friendly interface, abstracting much of the complexity involved in container operations. They typically offer functionalities like image management, container lifecycle management, and network configurations (44).

Originally, Docker Engine was the default container runtime for Kubernetes. However, the Container Runtime Interface (CRI) release in Kubernetes 1.5 changed this landscape. The CRI is a standardized API that was specifically designed to allow Kubernetes to support multiple container runtimes more easily. Starting from Kubernetes 1.23, a fully CRI compatible containerd became a default high-level container runtime (45). CRI-O is another popular and CRI-compatible runtime that was designed explicitly for Kubernetes. CRI-O, similarly to the containerd, fully conforms to the OCI (Open Container Initiative) specification. Thanks to this compatibility with OCI, the containerd, and the CRI-O can work with any OCI-compatible low-level container runtimes. Moreover, thanks to OCI compliance, low-level runtimes, such as runC, kata container, and gVisor, can run side-by-side under the same containerd or CRI-O cluster, providing the foundation for container execution in Kubernetes (46).

Conversely, low-level container runtimes directly interface with the operating system to execute containers. These runtimes handle the low-level details of container creation, including setting up namespaces, cgroups, and file systems. The runC low-level runtime is by default installed with the containerd, it is also a default runtime underneath the Docker. Crun is a lightweight alternative to runC, written in C, to optimize performance. Low-level runtimes are crucial for ensuring the isolation and performance of containers, as they manage the fundamental aspects of container execution.

Figure 2.2 illustrates the container runtime architecture in Kubernetes, showing the interaction between different components. At the top level, an orchestrator tool like Kubernetes manages the cluster and containerized applications using a high-level runtime like

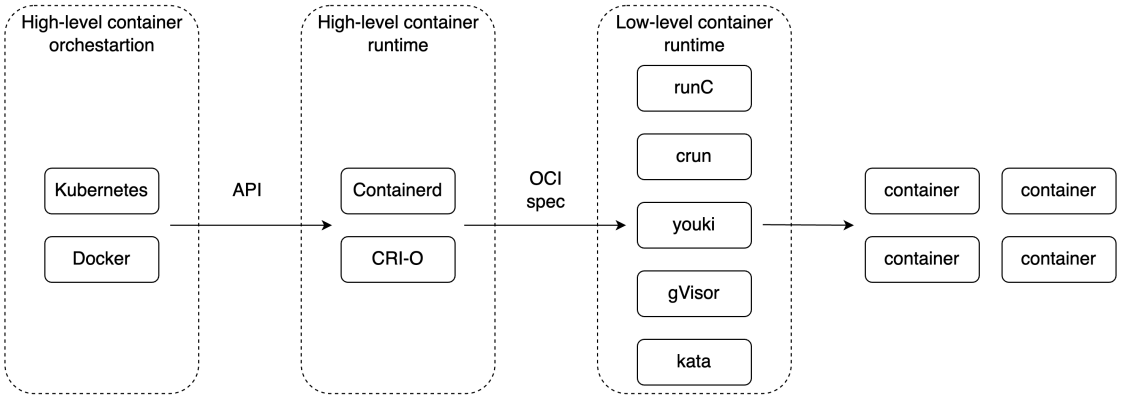

Figure 2.2: The relationship between container orchestrators and container runtimes.

containerd or CRI-O. Below containerd and CRI-O, several low-level container runtimes can be used to handle the actual container execution. Crun and youki, which have been mentioned before in Chapter 1 are the low-level runtimes that come with WebAssembly support. This figure effectively highlights the modular architecture of container runtimes in Kubernetes, emphasizing the role of the Open Container Initiative specification that enables seamless interoperability among low-level container routines.

2.3 Continuum Framework

In this thesis, we extensively use the Continuum framework to set up the VMs and the Kubernetes cluster, which we use for prototyping our implementation and benchmarking and evaluation of our work. The Continuum framework offers a range of features that facilitate automated infrastructure setup, application deployment, and comprehensive benchmarking and monitoring. The framework automates the creation and configuration of virtual machines, equipping them with the necessary software and dependencies required for benchmarking tasks. This setup includes the installation of container runtimes, orchestrators such as Kubernetes, and monitoring tools to ensure a robust infrastructure foundation. The Continuum utilizes predefined templates and configurations to ensure consistent and reliable deployments across diverse environments. Additionally, Continuum excels in benchmarking and monitoring by collecting detailed performance metrics during benchmark tests. It tracks resource usage, startup times, and other relevant metrics, providing an in-depth performance analysis crucial for evaluating the benchmarked system in question's efficiency and effectiveness.

3

Design of New WebAssembly Runtime Integration

Building upon the problem statement outlined in Chapter 1 and the foundational concepts discussed in Chapter 2, this chapter delves into the design decisions and requirements necessary to achieve the overarching goal of reducing the memory overhead of running WebAssembly containers. Achieving this goal involves integrating a lightweight WebAssembly runtime into a container runtime to enhance memory efficiency. This chapter is integral to the thesis as it bridges the conceptual understanding of WebAssembly and Kubernetes with practical design and implementation strategies, laying the groundwork for the subsequent technical and evaluative discussions. We first assess the current support for WebAssembly in Kubernetes 3.1, identifying existing limitations and opportunities for improvement. We then outline our design decisions 3.2, including the selection of container and WebAssembly runtimes, which we use in our further work. This chapter also includes a comprehensive requirements analysis to ensure that our design meets the necessary functional and performance criteria 3.3. The design outline of the pertinent benchmarks that assist us in evaluating our work and responding to our research questions follows this 3.4.

3.1 Current Support of WebAssembly in Kubernetes

Figure 3.1 illustrates the integration of WebAssembly Containers within Kubernetes, following the guidelines set forth by the Cloud Native Computing Foundation (47). At the top, Kubernetes serves as the container management platform, orchestrating containerized applications' deployment, scaling, and operation. It interfaces with containerd service, a daemon process on each worker node, which manages container lifecycle operations.

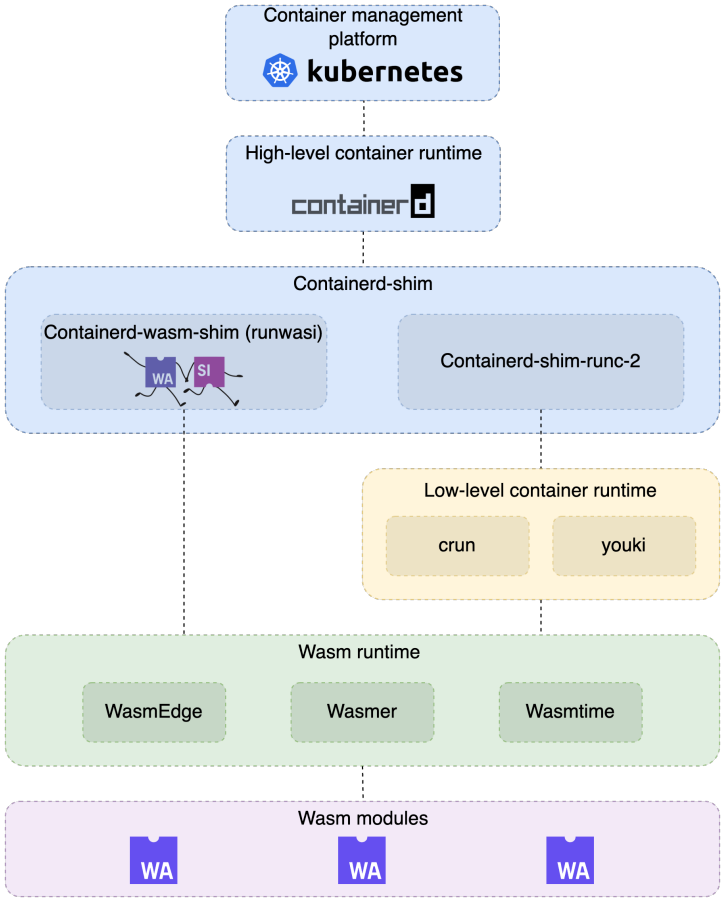

Figure 3.1: How WebAssembly is currently supported in Kubernetes.

Containerd service utilizes shims processes to manage each container instance; a shim is a lightweight intermediary process between the containerd daemon and the low-level container runtime. By using a shim, containerd can ensure that container processes are decoupled from the containerd daemon, which allows the daemon to be restarted or upgraded without affecting running containers, enhancing the system's reliability.

The containerd-shim-runc-v2 is the default shim used by the containerd for running OCI-compliant containers using runC. However, a containerd's runwasi project delivers a set of shims that facilitate the execution of WebAssembly containers by bridging containerd with Wasm runtimes such as WasmEdge, Wasmer, and Wasmtime. On the other hand, Figure 3.1 shows that besides runC, other low-level container runtimes, such as crun and youki, can be managed by containerd-shim-runc-v2 since they are OCI-compliant container runtimes. The usage of crun and youki provides another way of supporting Wasm containers as they can be built and installed with the support of the same set WasmEdge, Wasmer, and Wasmtime Wasm runtimes.

3.2 Design Decisions

Adding an implementation of a new embedded Wasm runtime to an existing container runtime presents a few challenges, primarily due to the complex nature of the container runtime itself (48). The modular nature of container orchestrators and their underlying components presents additional difficulties. As mentioned previously, Kubernetes allows us to use the low-level container runtime of our choice and independently exchange high-level container runtime. We can do so if they conform to the defined Container Runtime Interface (CRI) standard (49). Another consideration has to be taken when choosing a new, desired Wasm runtime for the purpose of this implementation.

Therefore, there are two main design decisions we have to make to be able to work toward our aim of lowering the memory overhead of Wasm containers:

- Should we integrate a new Wasm runtime into the crun, youki, or runwasi project?
- Which Wasm runtime should we choose to embed into a selected runtime?

3.2.1 Container Runtime Selection

We did not find any academic literature comparing or considering all or some of the crun, youki, or runwasi runtimes; thus, we headed toward more empirical decision-making methods. We base our choice of container runtime on the following:

- **Popularity of projects' repositories on GitHub; starts, forks, opened issues, and frequency of commits to the main branch**: The popularity metrics on GitHub provide insights into how widely used and supported a project is by the community. Higher star counts, forks, and active issue discussions indicate a more vibrant and potentially reliable project. Frequent commits suggest that the project is actively maintained, reducing the risk of encountering unaddressed bugs or security vulnerabilities.
- **Usage and mentions of the runtime of interest in documentation and online guidelines regarding container orchestrators and container runtimes**: This factor helps to identify well-recognized and recommended runtimes within the industry. A runtime frequently mentioned in official documentation and community forums is more likely to be compatible with various tools and use cases, ensuring broader support and easier integration.

- **Language in which the runtime of interest is written in**: The language impacts the runtime's performance, the ease of debugging, and the potential for integration with other tools and languages. A well-suited language for systems programming can lead to better performance and lower resource consumption.

- **Availability of documentation and examples of usage of the runtime of interest and availability of other artifacts that would make the implementation process easier**: Comprehensive documentation and examples significantly ease the learning curve and implementation process. They provide essential guidance on effectively using and troubleshooting the runtime, ensuring that developers can quickly and efficiently integrate it into their projects.

Considering all the gathered information, we decided to focus on the crun runtime. At the time of writing, it was less popular than youki repository on GitHub; however, based on set criteria, we found it to be a more mature project, and it is claimed to outperform youki (19). In contrast, in the same repository, crun is showcased as a fully functional runC alternative (50). Moreover, crun is a default container runtime in Red Hat Enterprise Linux (RHEL) since version 9 of RHEL (51).

The runwasi project, despite being an underlying technology of the Docker Beta feature that enables running Wasm containers since Docker version 4.21 (52), seemed the least compelling for our work. Based on open issues on GitHub, we discovered that many advertised features of the runwasi project, such as shared mode, are not functioning. The repository is also less actively maintained and popular than youki and crun repositories. Additionally, runwasi is not a fully featured container runtime; it is a separate shim-level component that can only execute the Wasm workloads, so our preference leaned towards the crun.

3.2.2 WebAssembly Runtime Selection

As far as we know, there is no existing academic literature comparing more than two Wasm runtimes. Therefore, similarly to the process of selecting a container runtime, we set directions to help us. We make our decision based on the following:

- **Publicly available reports and surveys comparing various Wasm runtimes, their popularity, and performance**: These reports and surveys provide an overview of how different Wasm runtimes perform under various conditions and their acceptance within the community. Metrics such as execution speed, memory usage, and

community adoption are critical in assessing the suitability of a runtime for production use.

- **Open source, publicly available benchmarks of the Wasm runtimes**: Benchmarks are essential for understanding the performance characteristics of different runtimes. Publicly available benchmarks allow us to verify performance claims independently and ensure that the chosen runtime meets our specific performance requirements.

- **Language in which the runtime of interest is written in**: The language impacts the runtime's performance, the ease of debugging, and the potential for integration with other tools and languages. A well-suited language for systems programming can lead to better performance and lower resource consumption.

- **Availability of documentation and examples of usage of the runtime of interest and availability of SDKs or other artifacts that would make the implementation process easier**: Good documentation and examples are crucial for reducing development time and avoiding common pitfalls. They provide clear guidance on using the runtime effectively and integrating it with other systems.

Considering the aforementioned directions and findings, we concluded that WebAssembly Micro Runtime (WAMR), developed by the Bytecode Alliance organization, should be selected for further implementation. We base our claim on publicly available reports, including but not limited to The State of WebAssembly 2023 report (53) and the comprehensive open source benchmark that is publicly available (54). This benchmark shows WAMR's promising performance, surpassing all three Wasm runtimes currently integrated with crun or runwasi. The WAMR runtime is claimed to have a small memory footprint, which is desired in our work. Moreover, WAMR was created with embedded use cases in mind, further increasing our belief that it should help us lower the memory footprint of Wasm containers. Interestingly, Bytecode Alliance is an organization that has also developed the Wasmtime runtime, which is currently integrated with crun, youki, and runwasi.

3.3 Requirements Analysis

The definition of clear requirements for a developed product is crucial to properly addressing our research questions. Functional requirements (FR) define the designed system's expected and desired behavior and thus help evaluate when the implementation may be

considered as done and successful (55). On the other hand, non-functional requirements (NFR) are essential aspects of system design that extend beyond the system's core functionality. They encompass performance, security, portability, and usability, ensuring the system's overall quality and effectiveness (56). We begin by specifying the requirements for our main contribution, which is an extended crun runtime with a new WAMR runtime integration. Next, we also define the requirements for our benchmark environment, which should allow us to perform the necessary experiments that evaluate our work.

3.3.1 Modified Container Runtime Requirements

- **(FR1) The crun runtime should successfully compile with WAMR support enabled:** The crun runtime build targets should be extended to include WAMR integration. The crun configured with WAMR support should compile without errors.
- **(FR2) The crun runtime should not lose any existing functionalities:** Adding support for new Wasm runtime should not break any existing crun features, including already supported Wasm runtimes.
- **(FR3) The crun runtime should properly handle exceptions caused by WAMR integration:** Exceptions raised by WAMR or during WAMR shared library loading should be appropriately managed.
- **(FR4) The crun runtime should be able to execute the Wasm module from inside a container using WAMR:** The Wasm module should be executed in embedded WAMR runtime, and a proper exit code should be returned by crun.
- **(FR5) The crun runtime should pass environmental variables and run arguments to the WAMR runtime:** All environmental variables and runtime arguments set in the container should be properly passed to the WAMR runtime and used while running the Wasm module.
- **(FR6) The crun runtime should set proper WASI arguments to enable the Wasm module to access the container files:** The Wasm module should be able to communicate with the outside world within the bounds of its container, using the WASI interface.
- **(NFR1) The crun runtime should use less memory for running a Wasm container when using WAMR:** The memory overhead needed to run the container

with crun and its workload with WAMR should be lower than when using other supported Wasm runtimes.

- **(NFR2) The crun runtime should allow for high volumes of deployed containers when using WAMR without performance degradation:** We should be able to deploy Wasm containers at scale using crun with WAMR without compromising containers startup performance or memory usage.

3.3.2 Benchmark Environment Requirements

We prepare the benchmark environment and execute the experiments by creating the configuration files and introducing changes to the Continuum framework. This section does not consider the requirements already met and delivered by the Continuum framework. We outline the following requirements that our changes to the Continuum and the prepared deployment configuration must meet:

- **(FR8) Continuum should set up benchmark infrastructure with Wasm support enabled:** The VMs that Continuum creates should have WasmEdge, Wasmtime, Wasmer, and WAMR runtimes preinstalled. VMs should also have our modified version of crun runtime and runwasi shims installed.

- **(FR9) Continuum should monitor and collect memory usage data throughout the whole lifespan of Wasm containers:** Currently, the Continuum focuses on the startup performance of Kubernetes deployments. We should extend this functionality to collect relevant metrics for the entire running time of deployed containers.

- **(FR10) Continuum should be able to deploy containers with Wasm workloads successfully:** Containers with Wasm modules should be deployed on Kubernetes cluster using crun runtime with Wasm support or runwasi shim.

3.4 Design of Benchmarks

By answering the research question **RQ1**, we develop a prototype of a crun runtime with a new Wasm runtime embedded in it. To be able to answer our research question **RQ2**, we need to evaluate the memory performance of our prototype properly. Therefore, we must design proper benchmarks focusing on relevant metrics that allow us to position our work accurately compared to the current state-of-the-art technologies.

Our main objective is to reduce the memory overhead caused by the container runtime, which invokes the Wasm runtime to execute the Wasm module from a container. To validate our success, we need to perform a memory usage test to show the memory overhead of running a Wasm container using our implementation of the crun compared to the current possibilities available on Kubernetes. Moreover, a more general startup performance test is necessary to indicate if, together with lowering the memory, we did not sacrifice the performance of deploying and running the containers.

3.4.1 Memory Usage Test

We should use an application with a minimal memory footprint to assess the memory overhead posed by the container runtime. In this way, the memory consumed by the deployment is consumed by the container runtime, not by the application workload. We deploy each container in its own pod; the pods do not have any other container. We intend to measure memory usage using two methods:

- The Linux system memory usage: We measure the system's free memory on the Kubernetes worker node before and after the deployment and calculate the memory used per deployed pod.
- The memory usage reported by the Kubernetes metrics server: We measure the memory used by the deployment and calculate the memory used per deployed pod.

The above measurement should be repeated for different deployment sizes to check the impact of scalability on memory usage and if our integration of Wasm runtime did not compromise it.

3.4.2 Startup Performance Test

For this test, we should use the same application with a minimal memory footprint. In this way, we can measure the time needed for a container runtime to create a pod with a Wasm container and start executing the container's workload. Once again, those measurements should be repeated for different deployment sizes to check the impact of scalability on the overhead performance posed by the container runtime. The obtained results should provide valuable insight into how our work compares to other already supported Wasm runtimes on Kubernetes.

4

Integration of WAMR into crun

Integrating the WebAssembly Micro Runtime into the crun container runtime is the main contribution of this thesis. This chapter details the process of embedding WAMR into crun, providing a comprehensive overview of the necessary steps and technical considerations. The integration aims to leverage the lightweight and efficient nature of WAMR to execute Wasm modules within containers, thereby reducing memory overhead and improving performance.

Figure 4.1 provides an overview of our implementation. It illustrates the process of enabling Wasm container execution through the WebAssembly Micro Runtime embedded within the crun low-level container runtime. The Kubernetes cluster used in our work is automatically set up via the Continuum framework, ensuring a streamlined and reproducible deployment process. The following sections in this chapter delve into the specific components depicted in the figure. Section 4.1 explains how WAMR can be embedded into a C code, including the configuration and installation of WAMR on a machine. Section 4.2 discusses the integration of WAMR into the low-level container runtime, specifically crun. Section 4.3 covers the Continuum configuration files necessary for the Continuum to install and configure all required software on the VMs, including the containerd. Finally, Sections 4.4 and 4.5 detail creating a Wasm container image and deploying it on Kubernetes using the Continuum framework. This step-by-step breakdown ensures that all implementation aspects are clearly understood.

4.1 Overview of How to Embed WAMR into C/C++ Code

Our implementation starts with building the WAMR from its source code. We must build the iwasm executable binary, which encapsulates WAMR's core set of libraries for

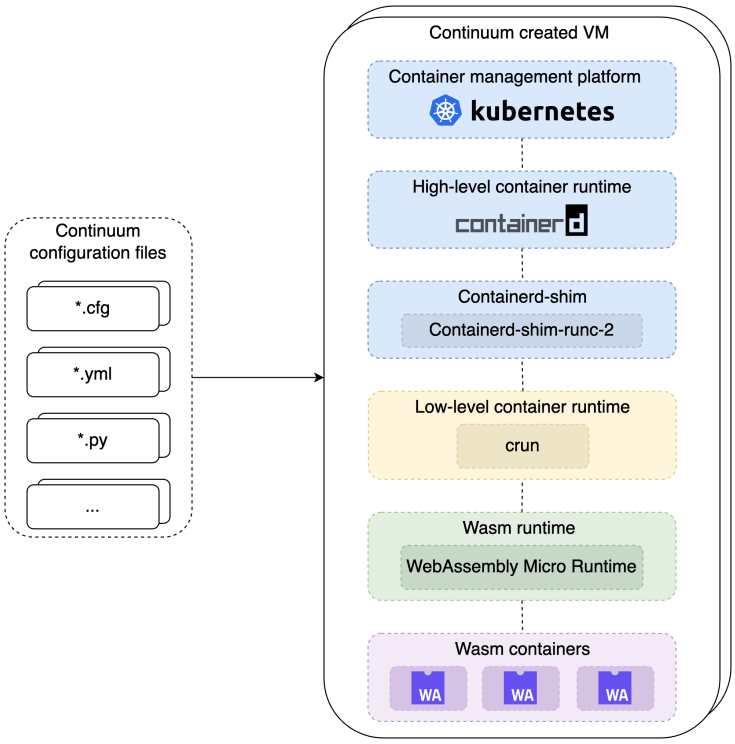

Figure 4.1: Overview of our implementation of WAMR embedded into crun.

loading and running Wasm modules. It also includes Wasm compilers, a command-line tool, and WASI support. The WAMR runtime can be built using the cmake tool, and the configuration can be adjusted by setting the cmake variables in the `CMakeLists.txt` file. We found changing the default values of three available variables important for our further implementation to work correctly and securely. We disabled the variables:

- `WAMR_BUILD_SHARED_MEMORY`: It disables memory sharing among WebAssembly modules. We do not test loading more Wasm modules into one WAMR execution environment instance or use shared heap using API calls such as `shared_malloc` and `shared_free`. We decided to disable it since it introduces additional overhead in memory management due to the complexity of boundary checking to ensure safe and efficient use of the shared memory.

- `WAMR_DISABLE_HW_BOUND_CHECK`: It disables the hardware boundary checks for memory access; instead, software checks are used, which can introduce performance overhead. Although it might be desirable to enable hardware checks since they are typically faster and more efficient than software checks, we found that enabling them

is not stable on our benchmarking infrastructure based on Quemu-managed virtual machines.

- `WAMR_DISABLE_STACK_HW_BOUND_CHECK`: It disables hardware-based boundary checks for native stack access. Again, we disabled this feature due to the instability of WAMR on our virtual machine-based infrastructure.

After building and installing the iwasm executable, we found it also necessary to manually copy the header files that come with the source code of iwasm into `/usr/local/include/` directory and refresh the shared libraries caches in the system. A fork of the WAMR repository with proper build variables set is available, and details about it can be found in Appendix A.

The WebAssembly Micro Runtime was designed to be easily embeddable in other projects written in C or C++ languages and comes with necessary header files `wasm_export.h` and `wasm_c_api.h` that we copied into the system's header files directory in the previous step. The `wasm_c_api.h` is an engine-agnostic API specification aspiring to conform to the official WebAssembly wasm-c-api project (57), which can be used to embed a Wasm engine into another project. Meanwhile, the `wasm_export.h` is a native WebAssembly Micro Runtime API specification. The functionalities of those two head files overlap, and developers should choose only one to use in a project. We chose to use the native API as some wasm-c-api APIs are still not supported in WAMR.

At this point, we have a working WebAsembly Micro Runtime installed in our system, and we are ready to start the process of embedding it into the crun runtime.

4.2 Technical Details of WAMR Integration in crun

The crun repository does not provide any guidelines regarding adding new external handlers of container processes, so we start our implementation with the source code exploration. Inside `/src/libcrun/` location where the core code of crun is located, `custom-handler` C and header files are placed. The `handlers/` directory is also there, where other already implemented external handlers can be found.

Based on what we learned from existing code, we start our implementation by adding a `wamr.c` file to the custom handlers directory; we place the code that embeds the WAMR runtime there. The actual implementation starts with creating the `struct`, which is of type `custom_handler_s` and specifies our new WAMR handler. It includes the handler's name

and required function pointers for loading, unloading, executing, and checking container handling capability, among others.

```
1  static int libwamr_load (void **cookie, libcrun_error_t *err) {
2      void *handle;
3
4      handle = dlopen ("libiwasm.so", RTLD_NOW);
5      if (handle == NULL)
6          return crun_make_error (err, 0, "could not load libiwasm.so: %s",
       dlerror ());
7      *cookie = handle;
8
9      return 0;
10 }
```

Listing 4.1: Function dynamically loading WAMR shared library in crun.

In our implementation, we use dynamic WAMR shared library loading and unloading. Listing 4.1 shows a function used to load the WAMR shared library; in line 4, we use a `dlopen` function to load a library. In line 7, we store the library handle, which is used later to load symbols from the library. We set an error if loading fails. Thanks to dynamic loading, crun can load and unload the library as needed during runtime, reducing the overall memory footprint and lowering the startup time of crun when the WebAssembly handler is not needed.

```
1  ...
2  // Declare function pointers to WASM runtime functions
3  bool (*wasm_runtime_init) ();
4  ...
5  // Dynamically load symbols from the shared library
6  wasm_runtime_init = dlsym (cookie, "wasm_runtime_init");
7  ...
8  // Error handling for missing symbols
9  if (wasm_runtime_init == NULL)
10     error (EXIT_FAILURE, 0,
11         "could not find wasm_runtime_init symbol in ‘libiwasm.so‘");
12 ...
```

Listing 4.2: Dynamic loading of symbols from a shared library.

The function responsible for executing the Wasm binary starts by declaring the function pointers for WebAssembly runtime functions and dynamically loads these symbols from the shared library. An example of this mechanism is presented in Listing 4.2, which shows an extract from our implementation showcasing the dynamic symbol loading. The function

pointer is declared in line 3; then the function is loaded from the library using `dlsym` function in line 6. Using on-demand symbol loading should ensure high performance of crun as it only loads the necessary selected symbols from the library and does it at runtime.

Later, after loading all symbols, the `libwamr_exec()` function creates the Wasm runtime, reads the Wasm binary from the file in the container directory into a buffer, and initializes the Wasm runtime with this buffer. We implement our own function that reads the binary from a file into a buffer and, next to the buffer, returns the buffer's size. After the Wasm runtime is initialized, we set the WASI arguments. This step is crucial if we want our Wasm module to be able to interact with the underlying system. Setting WASI arguments allows us to pass environment arguments, runtime arguments, and pre-open directories for a Wasm module. It also redirects the standard streams from the Wasm module back to crun. Pre-opening directories for a Wasm module ensures isolation by providing controlled access to the file system. Following that, the Wasm module is already sandboxed by the execution in its own mount Linux namespace set up for its container; we pre-open two directories, namely `"."` and `"/"`, without compromising security. The current directory location allows the Wasm module to operate within the context of the directory where it was executed. It is useful for applications that expect to use files relative to their execution location. On the other hand, pre-opening the root directory can be useful for modules that need to navigate the directory structure from a known starting point.

When the WASI environment is set, we instantiate the Wasm runtime by invoking the function called `wasm_runtime_instantiate()`; instantiation involves allocating memory for a Wasm module and setting the runtime state. We look for a Wasm function named `_start` within an instantiated runtime. Applications compiled to Wasm with WASI interface support typically have a single function called `_start` representing the main entry point (58). If the function is found, we execute it in the created execution environment. The exit of the main entry point function is followed by unloading the Wasm module and destroying the runtime environment.

The above implementation of embedded WAMR runtime is placed inside a conditional compilation block that checks whether dynamic loading and WebAssembly Micro Runtime support are available. We implement a `--with-wamr` configuration option using the Autotools `configure.ac` script. If a user configures crun with WAMR support enabled, Autotools validates the presence of the `wasm_export.h` file and sets the preprocessor macro `HAVE_WAMR`, enabling conditional compilation of WAMR-related code.

A complete code is available in the fork of the crun repository and will hopefully be merged with the main branch of the official crun repository soon. Reproducibility details are available in Appendix A.

4.3 Supporting WebAssembly in Continuum Framework

4.3.1 Installing Additional Runtimes

The Continuum framework comes with an extensive set of supported features and configuration possibilities that are available out of the box, such as automated infrastructure creation, application deployment, and benchmark execution in a reproducible manner. To use the Continuum framework while prototyping our new crun implementation with WAMR embedded and later to use it to validate our work, we need to extend the Continuum to support the deployment of WebAssembly containers to the Kubernetes cluster.

We start by installing the runwasi targets onto virtual machines, which the Continuum is creating as our benchmark infrastructure. Three targets can be downloaded from the runwasi GitHub project page: containerd-shim-wasmtime, containerd-shim-wasmer, and containerd-shim-wasmedge. We have downloaded those three binaries in release version v0.4.0 and, for reproducibility reasons, placed them in our fork of the Continuum framework repository. We modify the Ansible playbook `base_install.yml` that the Continuum utilizes to prepare the base operation system images for the virtual machines by copying downloaded runwasi binaries into the `/bin/` directory of the VM's OS. We add executable permissions to the copied files as well.

```
- name: Install crun runtime
  shell: |
    git clone https://github.com/macko99/crun
    cd crun
    git switch wamr_support
    ./autogen.sh
    # here --with-{wasmtime|wasmedge|wasmer|wamr} can be used
    ./configure --with-wamr
    make
    make install
```

Listing 4.3: Ansible task installing crun into VM's OS.

We also need to install WebAssembly runtimes so that the runwasi shims and the crun container runtime can later utilize them. Still, in the `base_install.yml` playbook, we add tasks to install WasmEdge, Wasmer, Wasmtime runtimes, and the Wasmtime C API that

is also required for later compilation of crun. Next, we create tasks that install WAMR prerequisites and build and install WAMR from the source code as already described this process in Section 4.1. Finally, we add tasks that build and install crun runtime from source code; the task is shown in Listing 4.3. The Wasm runtime selection is in line 8 of the aforementioned listing, and it uses the configuration parameter that we introduced and described in Section 4.2. It is worth mentioning that crun can be built and installed with only one embedded WebAssembly runtime at a time. To run Wasm containers using another underlying Wasm runtime, rebuilding and reinstallation of crun are required.

4.3.2 Configuring Containerd

At this point, we need to register installed in the previous step container runtimes with the containerd. We do it in `config.toml` file that is later copied by the Continuum framework onto the VM's OS and used as the containerd service configuration. Lines 6 to 13 in the Listing 4.4 show an example of additional high-level container runtime registration with containerd; it sets up the containerd-shim-wasmedge runtime that should be already present in `/bin/`. It is worth pointing out that the naming of `runtime_type` in line 13 is essential for the runtime to be functional, and it follows the containerd documentation. We register the remaining runwasi shims similarly.

```
1  [plugins."io.containerd.grpc.v1.cri".containerd]
2        default_runtime_name = "crun"
3  ...
4  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes]
5  ...
6    [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.wasmedge]
7      base_runtime_spec = ""
8      container_annotations = []
9      pod_annotations = []
10     privileged_without_host_devices = false
11     runtime_engine = ""
12     runtime_root = ""
13     runtime_type = "io.containerd.wasmedge.v1"
14
15   [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.crun]
16     base_runtime_spec = ""
17     container_annotations = []
18     pod_annotations = ["*.wasm.*", "wasm.*", "module.wasm.image/*", "*.
       module.wasm.image", "module.wasm.image/variant.*"]
19     privileged_without_host_devices = false
20     runtime_engine = ""
21     runtime_root = ""
```

```
22      runtime_type = "io.containerd.runc.v2"
23
24      [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.crun.options]
25        BinaryName = "/usr/local/bin/crun"
```

Listing 4.4: Containerd configuration with additional runtimes registered.

On the other hand, lines 15 to 22 in the Listing 4.4 show how to register a low-level container runtime, in our case, crun. Since crun is a runC-compatible binary compatible with Open Container Initialise (OCI) specification, containerd can still use the containerd-shim-runc-v2 shim process and communicate with crun without any issues; as line 22 shows, the `runtime_type` still indicates the runC. In containerd, pod annotations are key-value metadata added to a pod definition and passed to all underlying components for operational or informational purposes. The `pod_annotations` defined in line 18 instructs crun runtime that the pod contains Wasm containers and should be executed using an embedded Wasm runtime.

4.3.3 Configuring Kubernetes Cluster

The pod annotations specified in the previous subsection notify the crun that containers annotated with them should be executed using the Wasm runtime. To notify the containerd that it should use one of the high-level Wasm-enabled runtimes provided by runwasi, we need additional Kubernetes resources named RuntimeClass. A Kubernetes RuntimeClass is a feature that allows selecting different container runtimes for running pods within one Kubernetes cluster. We must create a separate RuntimeClass for each containerd-shim-wasmtime, containerd-shim-wasmer, and containerd-shim-wasmedge runtimes. Listing 4.5 shows an example of specifying a RuntimeClass for the containerd-shim-wasmedge runtime previously registered with the containerd service; line 5 indicates the handler already known to containerd.

```
1 apiVersion: node.k8s.io/v1
2 kind: RuntimeClass
3 metadata:
4   name: wasmedge-rc
5 handler: wasmedge
```

Listing 4.5: Kubernetes RuntimeClass for containerd-shim-wasmedge runtime.

As mentioned before, we also intend to thoroughly evaluate our implementation and its impact on scaling the Kubernetes deployments. Therefore, we also configured the

Kubernetes cluster to allow and support more than 110 pods on one worker node. Although we could not find any official Kubernetes guidelines on overcoming this strict limit of 110 pods, we found it relatively easy to achieve. Listing 4.6 shows all required configuring to push the maximum number of allowed pods up to 500 on a single Kubernetes node.

```
1  apiVersion: kubeadm.k8s.io/v1beta3
2  kind: ClusterConfiguration
3  ...
4  networking:
5    podSubnet: 10.244.0.0/16
6  controllerManager:
7    extraArgs:
8      "node-cidr-mask-size": "22"
9  ---
10 apiVersion: kubelet.config.k8s.io/v1beta1
11 kind: KubeletConfiguration
12 ...
13 maxPods: 500
```

Listing 4.6: Allowing more than 110 pods on Kubernetes node.

The `podSubnet` configuration parameter shown in line 5 was unchanged since the Continuum framework already configures the subnet for pods with enough hosts for our desired deployment. Initially, we set the `maxPods` parameter of the Kubelet configuration shown in line 11 to 500, but we could not get more than 254 pods deployed and running simultaneously. This result pointed us further into investigating the network configuration of the cluster and Kubernetes control plane components. We learned that the kube-controller-manager has a configuration parameter called `node-cidr-mask-size` that controls the subnet mask for networking within a single cluster node with a default value 24. We set this parameter to 22, as shown in line 8 of the listing.

Similarly to the original Kubernetes capabilities, we allow for significantly more pods per node than the network configuration allows; a CIDR of 22 should allow up to 1016 host addresses, whereas we only intend to support 500 pods. We believe it aims to ensure the proper performance and stability of the Kubernetes cluster operations, as on every cluster node, additional Kubernetes control plane pods are running and require networking resources. Additionally, let's consider a fully utilized Kubernetes cluster with 500 pods deployed and running; Kubelet allows a new pod to start its network initialization process as soon as any of the running pods exits, but it takes time before the exited pod is properly destroyed and its underlying network stack releases the IP address. Thus, we follow the original Kubernetes pattern to eliminate potential networking-related issues.

4.4 OCI Container with WebAssembly Module

To evaluate our work, we need to create a workload, that is, an OCI container with a Wasm module inside that we could deploy on a prepared Kubernetes cluster.

The ability to compile an application to Wasm is largely contingent upon the choice of programming language and Wasm-specific compiler or compilation target. Rust, Go, C, and C++ provide robust support for WebAssembly and WebAssembly System Interface (WASI). To implement a Wasm application that we use to evaluate our work, we chose to utilize Rust. Assuming the Rust and the Rustic compiler are installed onto the system, we need to add a wasm32-wasi compilation target to the compiler. Rustc is a versatile Rust compiler that may be used on several platforms and supports various compilation targets, such as wasm32-wasi. This target can compile Rust code into Wasm modules that adhere to the WASI standard. The simple application we use to evaluate our work is presented in the Listing 4.7.

```
1  use std::env;
2
3  fn main() {
4      println!("Hello, World!");
5      let sleep_time_string = env::var("SLEEP_TIME")
6              .unwrap_or_else(|_| String::from("60"));
7
8      let sleep_time: u64 = sleep_time_string.parse().unwrap();
9      println!("Sleeping for {} seconds", sleep_time);
10
11     std::thread::sleep(std::time::Duration::from_secs(sleep_time));
12     println!("Goodbye, World! #WASM-RUST");
13 }
```

Listing 4.7: Rust application used for evaluation of this work.

As soon as it starts, the application prints a string to standard output, shown in line 4. Then, it reads an environmental variable `SLEEP_TIME` in line 5 and sleeps for a time specified by this variable. Finally, before the application exits, in line 12, it prints a goodbye message. Such an application enables the Continuum framework to capture when the first container from the Kubenrtenes deployment starts, when all desired containers are running, and when they exit. It lets us know when the application deployment begins and finishes, which is crucial for collecting necessary metrics for evaluation benchmarks. Those metrics are used to determine the startup performance of containers for different container runtimes in use.

```
1 FROM scratch
2 COPY --chmod=777 target/wasm32-wasi/debug/app.wasm /main.wasm
3 ENTRYPOINT [ "/main.wasm" ]
```

Listing 4.8: Dockerfile of container with Wasm module.

After the application is built and the binary target file with the `.wasm` extension is created by the compiler, we can proceed to the Wasm container creation. We make a Dockerfile without any base image, as shown in line 1 of Listing 4.8. Next, we copy the Wasm module binary into an image (line 2) and set it as an image entry point (line 3). We can use a Docker Engine with Wasm support enabled to build an OCI image targeting Wasm architecture. At the time of writing, Wasm support can be enabled as a Beta feature on the Docker Desktop, as mentioned before.

We have also prepared a Python application with exactly the same behavior as the one presented in Listing 4.7 Rust code; similarly, we prepared a Dockerfile and a standard Docker image with this Python application.

4.5 Deploying WebAssembly on Kubernetes

With the Wasm container image prepared, we modify the file that defines the Kubernetes deployment used for benchmarking. The file is located in the Continuum framework repository and is called `launch_benchmark_kubecontrol_pod.yml`. In order to execute the Wasm container, we either have to instruct Kubernetes to use one of the installed runwasi runtimes or annotate the pod accordingly to let the crun runtime know that it should use Wasm runtime. The first can be achieved by adding a `runtimeClassName` parameter, the latter by adding an annotation following the crun documentation. An example of pod specification is presented in Listing A.4. Thanks to lines 10 and 11 in the listing, crun is instructed to run the containers using a Wasm runtime. If one wants to run the containers using runwasi runtimes, one of lines 13, 14, or 15 can be uncommented in favor of the current pod annotation.

```
1 apiVersion: batch/v1
2 kind: Job
3 ...
4 spec:
5   parallelism: {{ replicas }}
6   template:
7     metadata:
8       name: {{ app_name }}
```

```
9       # use annotations to run with crun, or comment out and use below
    runtime classes
10      annotations:
11        module.wasm.image/variant: compat
12    spec:
13      # runtimeClassName: wasmtime-rc
14      # runtimeClassName: wasmer-rc
15      # runtimeClassName: wasmedge-rc
16      containers:
17 ...
```

Listing 4.9: Kubernetes deployment with Wasm workload.

5

Evaluation

In this chapter, we experimentally evaluate if our implementation fulfills the aim of lowering memory usage when running Wasm containers. Such validation allows us to fully address our research questions and better position our work among currently available technologies. Our experiments focus on two main metrics: memory used per each running Kubernetes pod with Wasm workload and the time it takes to start the execution of Wasm modules in all deployed containers.

We start with discussing our experiment setup in Section 5.1, and then we present and discuss the obtained results in Section 5.2. We also report any unforeseen negative results in Section 5.3. Finally, we discuss the limitations of our evaluation and how it may affect the validity of this work in Section 5.4. The chapter is closed with a summary of the evaluation and our findings in Section 5.5.

5.1 Experimental Setup

We use the extended version of the Continuum framework to evaluate our work and perform designed benchmarks for this thesis. The Continuum installs dependencies required for Wasm integration with container runtimes, enables Wasm workload deployment on Kubernetes, and collects more extensive metrics throughout the lifespan of deployed pods. All development and evaluation experiments, including operating the Continuum framework, were conducted on a machine with a specification shown in Table 5.1.

The extended version of the Continuum and all other software modified for the purpose of this thesis are publicly available on the corresponding GitHub repositories, and access details are provided in Appendix A. The Continuum Framework is responsible for setting up an experimental environment consisting of 2 virtual machines. One is used as a Kubernetes

Specification	Details
CPU	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
Architecture	x86_64
Endianness	Little Endian
Total CPUs	20
Core(s) per socket	10
Thread(s) per core	1
Virtualization	VT-x
Memory	256 GB
Operating System	Ubuntu 20.04.3 LTS
Kernel	Linux 5.4.0-187-generic

Table 5.1: Experimental machine specifications.

Software	Version
Kubernetes	v1.27.0
Containerd	v1.6.31
RunC	v1.1.12
crun	v1.0.0
runwasi shims	v0.4.0
WasmEdge	v0.14.0
Wasmtime	v23.0.1
Wasmer	v4.3.5
WebAssembly Micro Runtime	v2.1.0

Table 5.2: Software used in the experiments.

control plane node, and one is used as a worker node, where all deployments are scheduled. Both virtual machines have the same specifications and are set up with 8 CPU cores pinned to the physical cores without quota limitations and 100 gigabytes of memory. The Continuum also automatically installs all required software onto those VMs; in Table 5.2, we present the used versions of software in our work.

5.2 Experimental Results

This section presents the results of our experiments, which aim to evaluate the performance and memory usage of Wasm containers executed with our new integration of WAMR. The experiments focus on two main metrics: memory overhead and startup performance. These

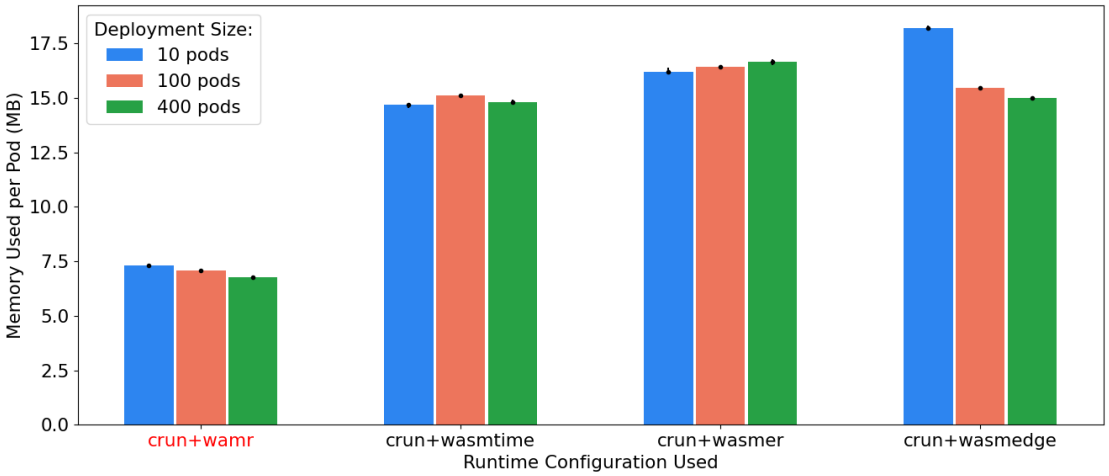

Figure 5.1: Memory usage per pod for different Wasm runtimes in crun, measured by Kubernetes. Our work's results are labeled in red.

metrics are crucial in determining the efficiency and practicality of using Wasm containers in production environments.

We first discuss the memory overhead observed in different scenarios and configurations. The analysis includes measurements obtained through two tools to ensure the accuracy and reliability of the data. Following the memory overhead analysis, we investigate the startup performance, examining how quickly Wasm containers can be initialized and ready for use. This is particularly important for applications that require rapid scaling and responsiveness. The findings from these experiments provide valuable insights into the benefits and limitations of using Wasm containers, helping to answer the research questions posed in this thesis.

5.2.1 Memory Overhead Compared to Wasm containers in crun

As mentioned before, while describing the benchmark design, this work employed two distinct methods to measure the memory usage of a Kubernetes worker node during pod deployment. The first method utilizes the `free` command from the Linux system to obtain memory metrics. In contrast, the second method leverages the Kubectl `top node` command to extract memory usage data from the Kubernetes metrics server. Utilizing these methods in tandem allows for a comprehensive analysis of memory consumption and allows us to double-validate our results.

We begin with measuring the memory overhead of running the Wasm containers in the Kubernetes cluster. The results in Figure 5.1 represent the measurements of memory

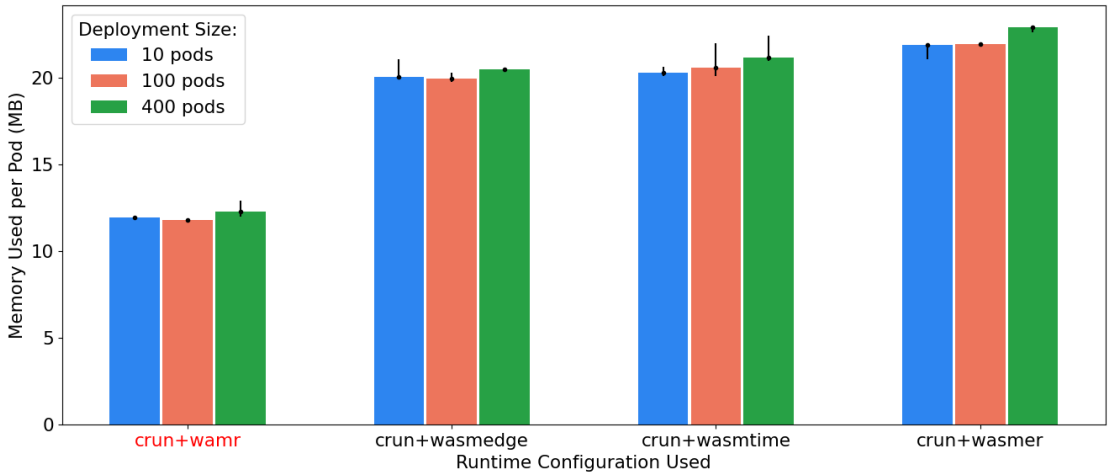

Figure 5.2: Memory usage per pod for different Wasm runtimes in crun, measured by the OS.

used per one pod collected using the Kubectl `top` command. The vertical axis shows the memory used in megabytes, where the lower values are preferred. The horizontal axis indicates the runtime configuration used to execute the containers. We can spot four distinct groups on the horizontal axis. The foremost left with the lowest results, marked in red, corresponds to our implementation of embedded WebAssembly Micro Runtime in crun. The remaining groups show the results for other supported WebAssembly runtimes in crun. The measurements for each runtime were made for three different deployment sizes: 10, 100, and 400 pods with Wasm containers. The results for each deployment size are marked accordingly and indicated by the separate bars for each runtime.

Similarly, Figure 5.2 represents the measurements of memory used per one pod during exactly the same experiments but collected using the `free` Linux command. Comparing the two figures, we notice a significant difference in nominal values of used memory (up to 42%), where higher usage is reported using the `free` command. This difference is expected since the `free` command reports the system's overall memory usage, including buffers, system caches, and processes other than those related to the Kubernetes cluster. Meanwhile, the Kubectl `top` command focuses on resources currently used by the workloads scheduled on the node.

Both Figure 5.1 and Figure 5.2 compare our work with all other Wasm runtimes supported by crun low-level container runtime. From the results, we can derive that our implementation uses at least 50.34% (reported by Kubectl `top` or at least 40.0 % reported by `free` command) less memory to execute the Wasm container than any other Wasm

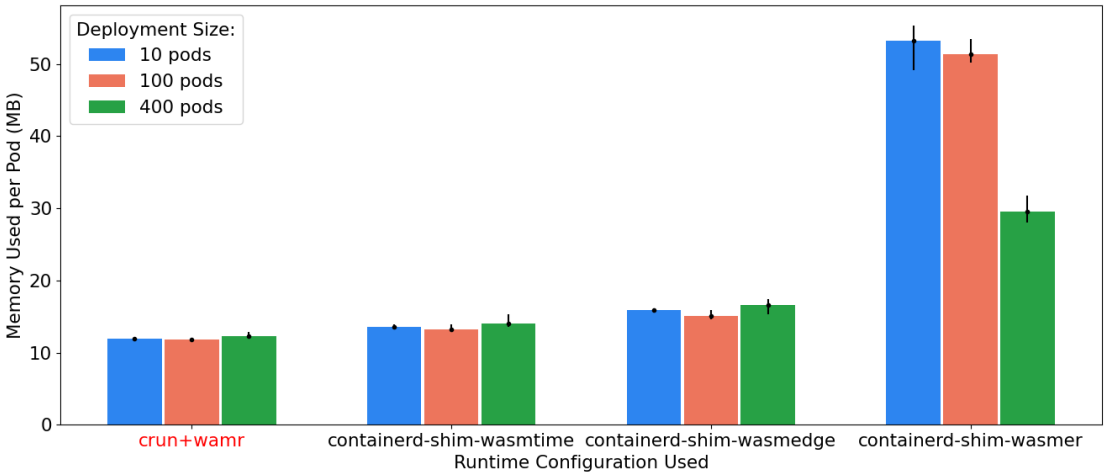

Figure 5.3: Memory usage per pod for different Wasm shims, measured by the OS.

runtime currently supported in crun. We can also observe that the memory the worker node uses to run one pod does not vary significantly depending on the deployment size. This indicates that our implementation provides proper scaling performance.

The obtained results can be explained by further investigation of the crun process executing the container's Wasm workload. We used the `smem` tool, available for Linux, to investigate the memory usage of the crun process in more detail. The `smem` tool, next to reporting the Resident Set Size (RSS) portion of memory held in RAM for each process, also reports the Unique Set Size (USS) and Proportional Set Size (PSS) values. The USS is the portion of memory that is unique to the process and not shared with other processes. Unlike the RSS metric, which can overestimate memory usage for processes sharing memory, the PSS metric splits the shared memory proportionally among the processes using it. The PSS metric indicates similar results compared to what is shown in Figure 5.1. For example, in the case of crun with Wasm support, two processes are needed to execute a container: the containerd shim process and a crun process that loads the needed WebAssembly runtime shared library. Comparing the sum of reported PSS values for those processes when using crun with WAMR and with WasmEdge, we calculated that crun with WAMR used approximately 55% less memory. This observation is consistent with the results of our experiment for 400 pods.

5.2.2 Memory Overhead Compared to runwasi

Following our research questions, we also intend to benchmark our solution against the runwasi-delivered high-level runtimes that support running Wasm containers. Such comparison allows us to better understand the WebAssembly supporting technologies currently available and how this work positions itself among them. Therefore, we conduct more experiments, this time running the same Kubernetes deployments using the runwasi shims as the underlying container runtimes. Figure 5.3 depicts the memory used per one pod reported by `free` command. Similarly to the previous figures, the vertical axis indicates the memory used per pod, and the horizontal axis indicates the runtime used, with distinction for different deployment sizes. Like in the previous figures, the results marked in red correspond to our implementation of embedded WebAssembly Micro Runtime in crun.

Unfortunately, we could not obtain metrics from the Kubernetes metric server when deploying 400 pods for any available runwasi shim. For this reason, we do not include the figure with results from the Kubectl `top` command here. We elaborate on this issue later in Section 5.3. Figure 5.3 contains proper data for all runwasi shims throughout all deployment sizes. From this data, we can derive that our implementation introduces the lowest memory usage per pod compared to all available runwasi shims, regardless of the deployment size. However, the difference is smaller than when we compared our work against other Wasm runtimes embedded in crun. Let's compare crun with newly embedded WAMR against the second-best in this benchmark, the containerd-shim-wasmtime. The memory usage of our implementation is still at least 10.87% lower, as measured using the `free` command. At the same time, compared to the worst performer in this experiment, containerd-shim-wasmer, crun with WAMR, used up to 77.53 % less memory to run one pod (`free` command).

We want to gain a full insight into how our work could be positioned among available containerization technologies when it comes to the memory overhead of running a pod on Kubernetes. Thus, as a reference, we also compare our new implementation of embedded WAMR runtime into crun container runtime against the standard Python container with the Python execution environment available in the container's image. This comparison does not provide direct insights into WebAssembly code distribution and execution inside an OCI container or the state of WebAssembly support in Kubernetes. However, we think this comparison helps us understand the maturity of WebAssembly support in containerized environments such as Kubernetes, the visibility of further development, and possible challenges that WebAssembly still faces.

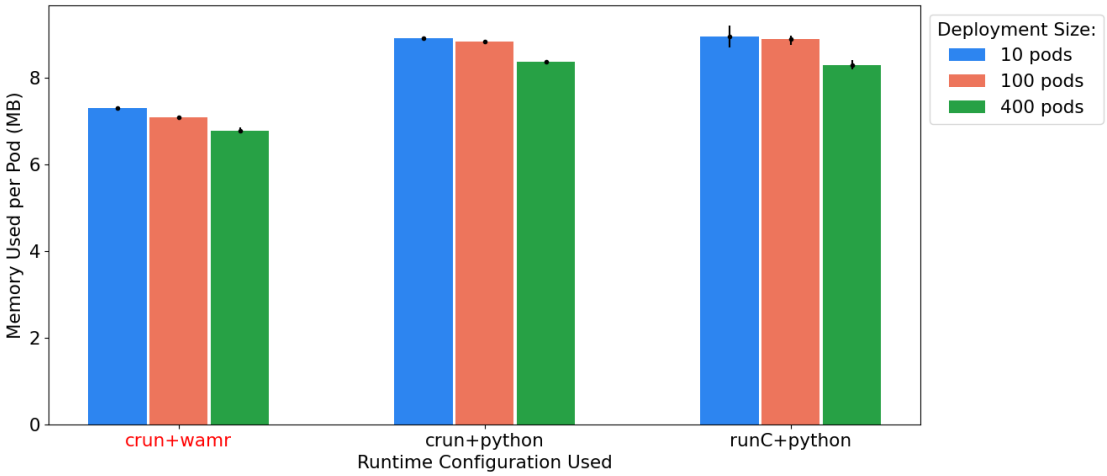

Figure 5.4: Memory usage per pod by our work compared with Python containers, measured by Kubernetes.

5.2.3 Memory Overhead Compared to Non-Wasm Containers

Figure 5.4 and 5.5 represent the measurements of memory used per one pod, collected using the Kubectl `top` command and Linux `free` command, respectively. This time, the figures aim to compare crun with embedded WAMR runtime, marked in red, and well-established standard container runtimes without Wasm support. As shown on the horizontal axis in the figure, we decided to compare our work with Python containers, which were executed using two different runtimes: crun and runC. We included crun because it completes the previous benchmarks since our work utilizes the crun runtime and builds upon it. By including crun, we can gain valuable insight into how crun performs without the need to call an underlying WebAssembly runtime and how it compares with our implementation. We also included runC since it is a default low-level container runtime for containerd and Kubernetes. Therefore, we can compare and challenge our work and the WebAssembly containers against the performance of the out-of-the-box Kubernetes cluster with more popular Python Docker containers.

From Figure 5.4 presenting the measurements from the Kubernetes metric server, we can derive that crun with WAMR runtime embedded uses at least 17.98% less memory when executing the WebAssembly container than crun when executing the Python Linux container. Similarly, crun with WAMR runtime embedded uses at least 18.15% less memory when executing the WebAssembly container than the default runC runtime when executing the Python Linux container. The differences in memory usage when considering the measurements conducted using the `free` command, presented in Figure 5.5 are respectively:

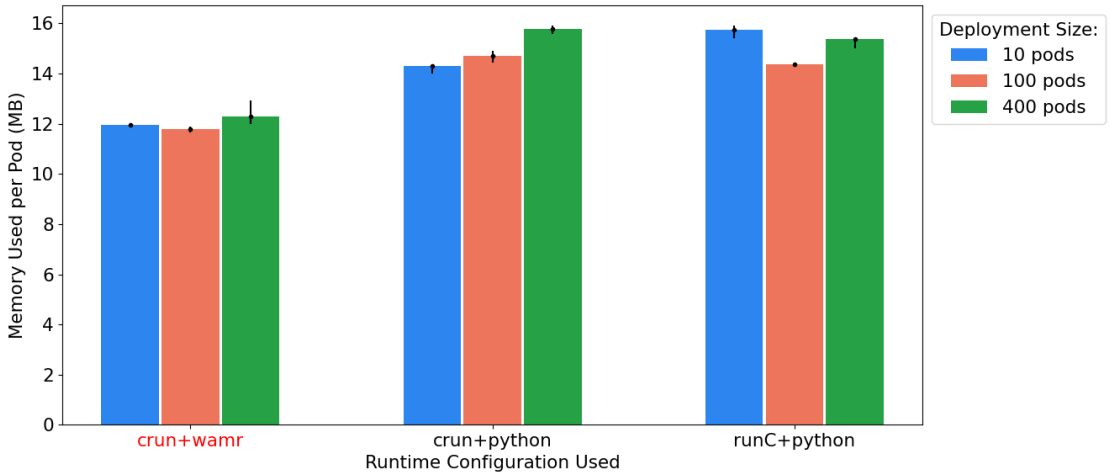

Figure 5.5: Memory usage per pod by our work compared with Python containers, measured by the OS.

16.38% less memory used when comparing our implementation running Wasm container against unmodified crun and 17.87% when comparing it against the default runC runtime.

5.2.4 Startup Performance

In this subsection, we measure and evaluate the startup performance of containers executed with crun with WAMR embedded and compare it with other available runtimes supporting WebAssembly. For reference, we also compare the startup performance against Python containers executed using two distinct low-level container runtimes. This comparison aims to showcase the potential benefits of using WebAssembly in Kubernetes clusters and our work's impact on the aforementioned performance.

Figure 5.6 presents the measurements of time needed to start executing workloads for ten deployed pods. We start the measurements when we apply the Kubernetes deployment file using the Kubectl command. The horizontal axis represents the time elapsed in seconds, and the lower values are preferred, as they indicate a higher startup performance. The vertical axis indicates the container runtime used to execute the container workload. The result obtained by our work, that is, by a crun with WAMR runtime embedded, is marked with orange. Two bars are indicated with a lighter blue color than the remaining ones; those two depict the startup time of Pyhton containers executed using crun without underlying Wasm runtime and runC runtime.

From figure 5.6, we can derive that for a small deployment size, ten pods in this case; our work does not introduce any performance degradation. WebAssembly Micro Run-

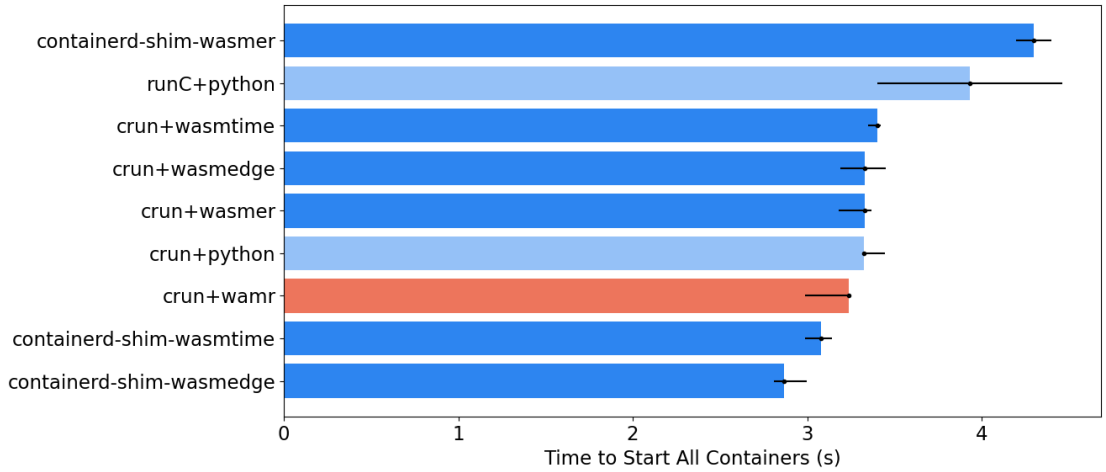

Figure 5.6: Time to start containers' workload execution for 10 deployed pods compared for different container runtimes. Results for non-Wasm containers are marked in a lighter blue color; our work result is marked in orange.

time embedded in crun executes all containers' WASM modules in under 3.24 seconds, which is below the average across all tested runtimes. Obtained results indicate that containerd-shim-wasmedge and containerd-shim-wasmtime runtimes have the best startup performance for small deployments, taking up to 11,45 % less time to start Wasm modules than our implementation. However, as the figure shows, our work performs at least 2.66 % better than any other WebAssembly runtime integrated into crun. Moreover, WebAssembly containers started up faster when executed with crun with WAMR integration than Python containers executed with both crun and runC runtimes.

We also benchmark the startup performance for the larger deployment of 400 pods to better evaluate the behavior of our implantation under heavy load and test its scalability performance. Figure 5.7 depicts the obtained results for measuring the time it takes to start the execution of containers' workload for 400 deployed pods. The horizontal axis again shows the time elapsed in seconds, whereas the vertical axis indicates the runtime used to run the containers. From this figure, we can derive that our implementation's impact on the container startup time scales better than that of any runwasi shims. For a deployment of 400 pods, crun with embedded WAMR took respectively 18.82 % and 28.38 % less time to start the Wasm modules than the containerd-shim-wasmedge and containerd-shim-wasmtime runtimes. However, our implementation shows lower startup performance for larger deployments than currently available WebAssembly runtimes supported in crun. When deploying 400 pods with WebAssembly containers, our work took 6.93 % more time

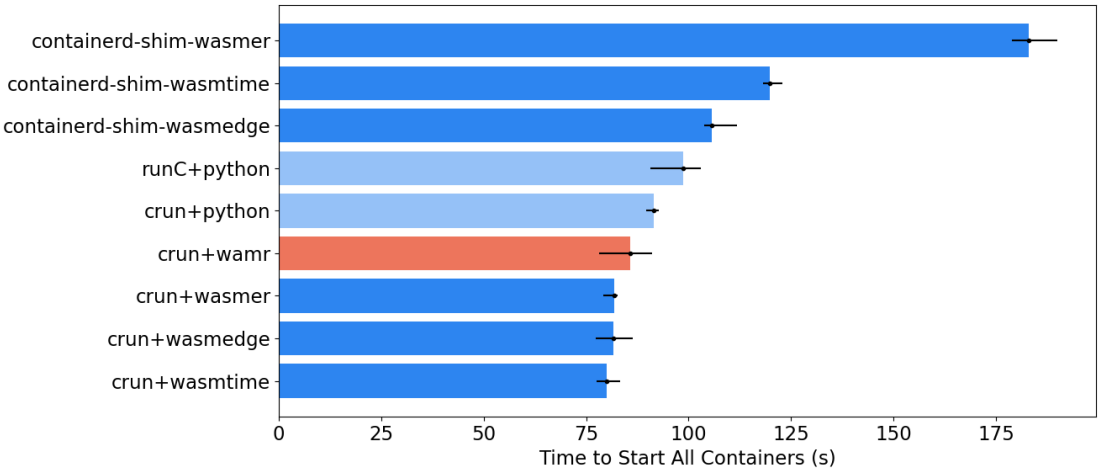

Figure 5.7: Time to start containers' workload execution for 400 deployed pods compared for different container runtimes.

to start the Wasm modules execution than leading in this experiment crun runtime with Wasmtime integration. Nonetheless, our work still presents better startup performance when starting 400 Wasm containers than both crun and runC runtimes when starting 400 pods with Python containers.

5.3 Reporting Negative Results

One of the methods we used to measure the memory usage of the deployed pods was the Kubernetes metric server. We periodically obtained the measurements from this Kubernetes component to gather sufficient data for our benchmarks. However, when we used any of the runwasi shims as the container runtime, and the benchmarked deployment size was larger than, on average, 350 pods, the Kubernetes metric server could not deliver memory or CPU usage values for the worker node. Each time we tried such a setup, we received a message indicating that the metric API was unavailable. This behavior was only observed for a time period between the time of starting the execution of, on average, 260th Wasm container until all containers finished running.

After an extended investigation, we could not solve this issue and provide metrics gathered using this method for runwasi shims as container runtimes. We believe this negative result is present due to the high CPU utilization of the worker node when using containerd Wasm shims. When using those runtimes, we observed a significantly higher CPU utilization of the worker node, frequently topping the value of 100% utilization. We think the

worker node could not report the resource usage data on time on every metrics server API call since it was overloaded with executing the containers' workloads. It is possible that assigning more resources to the VMs used to run the experiments would solve this issue. However, we did not have more resources available at the time of writing this thesis. This observation leads us to the limitations and threats to the validity of our work.

5.4 Limitations and Threat to Validity

The tests we performed focused on the memory overhead introduced by the container runtime and the WebAssembly runtime used to execute the Wasm module from the container. We also included an evaluation of the impact those runtimes have on a startup's performance, that is, the time it takes to start executing the mentioned Wasm modules. However, as already pointed out in the previous section, including the CPU utilization measurements in the conducted tests could also be interesting. We discovered that the underlying runtimes might have different CPU performance overheads. Thus, including CPU performance tests could provide more insights into our work performance, leading to different positioning of this work among alternatives. Moreover, additional metrics could showcase different use cases in which different container runtimes and WebAssembly runtimes excel.

5.5 Summary

In this chapter, we experimentally evaluated our work by performing the measurements of memory overhead introduced by running Wasm containers using crun with newly embedded WAMR runtime. We also compared those results with other Wasm runtimes supported in crun and containerd shims supporting WebAssembly containers execution. Additionally, we referenced the results obtained against Python containers based on Python Docker images executed with unmodified crun runtime and runC runtime. Startup performance tests were also conducted to validate if, together with lowering the memory footprint of running a Wasm container, we did not compromise the container runtime's performance.

We showed that our implementation delivers on the goal of lowering the memory footprint of running Wasm containers. Considering the results obtained from the system using the `free` command, our work uses at least 40.0% less memory to run one pod with Wasm container than other Wasm runtimes integrated with crun. At the same time, our work uses at least 10.87% and up to 77.53% less memory to run one pod than runwasi shims

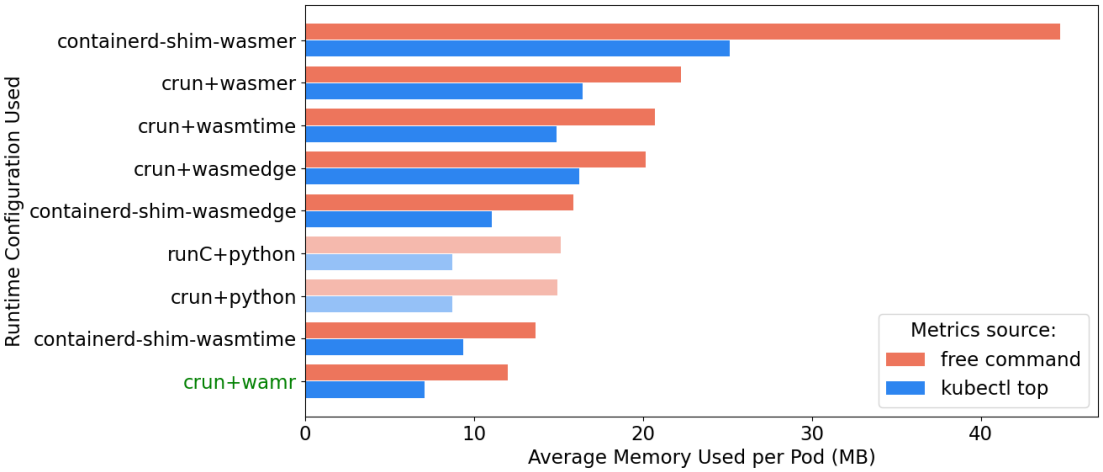

Figure 5.8: Memory usage per container by our work (labeled in green) compared with other container runtimes, averaged over all deployment sizes. Results for non-Wasm containers are marked in lighter colors.

with WebAssembly support, as can be seen in Figure 5.8. We also evaluated that our implementation delivers startup performance comparable to alternative runtimes regardless of the deployment size, ranging from 10 to 400 pods.

6

Related Work

We position this thesis as a novel work investigating and improving the performance of WebAssembly containers in the context of container orchestration systems like Kubernetes. We were unable to find any academic publications treating the extensive WebAssembly runtime performance comparison or container runtimes with Wasm support performance comparison. However, work has been done in the field of WebAssembly in the context of container orchestration. In this chapter, we briefly introduce the related works that we found interesting, which could potentially influence our work's perception.

Shuyao Jiang et al. (59) investigate performance issues in server-side WebAssembly runtimes. They introduce WarpDiff, a differential testing approach to detect performance issues by comparing execution times across different Wasm runtimes. The study applied WarpDiff to five popular Wasm runtimes, tested with 123 cases from the LLVM test suite, and identified seven performance issues. These issues, confirmed by developers, highlight areas needing optimization in Wasm runtimes. The findings aim to inspire improvements in server-side Wasm application performance.

Jasper Alexander Wiegratz (60) concludes that WebAssembly can complement Linux containers in cloud computing, offering specific benefits in security and efficiency. It suggests that future developments could further enhance the integration of WebAssembly in cloud-native applications, potentially leading to broader adoption alongside traditional container technologies.

The Krustlet project (61) by Microsoft tried to implement a Kubernetes Kubelet component in Rust to run WebAssembly workloads alongside Linux containers. It aimed at allowing Kubernetes to manage Wasm modules, offering benefits like faster startup times, smaller sizes, and enhanced security compared to traditional containers. The Krustlet

was supposed to integrate seamlessly with Kubernetes tools; however, the project was abandoned and is no longer being developed.

The Sandbox API (62) is part of the containerd's milestone 2.0, but an experimental version was introduced in release 1.7.0. This new API aims at bringing support for various sandbox types, including those based on virtual machines and WebAssembly, providing a more flexible way to manage different container environments simultaneously. Kuasar (63), a Cloud Native Computing Foundation sandbox project, aims to provide a secure, efficient, and flexible container runtime for modern cloud-native applications using this new API. The Kuasar includes Wasm Sandboxer, which allows containers to be launched within a WebAssembly runtime like WasmEdge or Wasmtime with more runtimes planned for future releases. Although the Sandbox API is in the experimental phase, in our opinion, it might be a game changer when it comes to Kubernetes interoperability and support for external sandboxes. At the same time, it is possible that Kuasar might provide better performance and lower resource use than current state-of-the-art technologies by avoiding double sandboxing Wasm modules.

7

Conclusion

In this thesis, we explored the integration of a lightweight WebAssembly runtime into an OCI container runtime to address the high memory overhead observed when running Wasm containers compared to non-Wasm containers. Our primary goal was to lower the memory footprint of Wasm containers, making them a competitive alternative to well-established containers with execution environments included in the container image. We started by exploring existing container runtimes supporting WebAssembly workloads that can be used under the Kubernetes orchestrator. Later, we chose the most promising runtime that could lead us to achieve our goal of lowering the memory footprint. Alongside implementing the modified container runtime, we also made changes to the Kubernetes cluster configuration and to the Continuum framework, which was used for the automatic setup and execution of the evaluating experiments.

This thesis makes several key contributions to the field of cloud-native computing:

- **The most memory-efficient Wasm-enabled container runtime:** A novel integration of the WebAssembly Micro Runtime into the crun container runtime significantly reduces the memory overhead of running Wasm containers without compromising the container's startup performance.
- **Comprehensive open-source Wasm-enabled benchmarking framework:** Extension of the Continuum framework to support the deployment and benchmarking of WebAssembly containers in Kubernetes, enabling reproducible and comprehensive performance evaluation.
- **Method for increasing Kubernetes node's capabilities:** A clear guideline and discussion on how to enable deployment of more than 110 pods on one Kubernetes worker node.

In conclusion, this work demonstrates the potential of WebAssembly containers as a lightweight, efficient, and secure technology for containerized applications in cloud-native environments. By addressing the memory overhead and maintaining competitive performance, the embedded WAMR in crun offers a viable path forward for the broader adoption of Wasm in Kubernetes clusters. Specifically, when paired with the extremely small size of WebAssembly container images, up to 100 times smaller when compared to the same application delivered as a Python Docker image, and the high cross-platform interoperability of WebAssembly, we are optimistic about the future of Wasm in the cloud-native computing. Future research and development should continue to enhance the capabilities and benefits of Wasm containers.

7.1 Answering Research Questions

In Chapter 1, we posed two research questions that aimed to guide us in lowering the memory footprint of running WebAssembly containers in Kubernetes. We answered those questions successfully throughout Chapters 3, 4, and 5. Below, we provide an extract from answers to each of our research questions.

RQ1: How to integrate a new, more lightweight Wasm runtime into container runtimes such as crun, youki, or runwasi that will lower the memory footprint of Wasm containers?

We successfully embedded the WebAssembly Micro Runtime (WAMR) into the crun low-level container runtime. This integration was achieved through iterative design and prototyping, ensuring the new runtime maintained all existing crun functionalities while implementing the new lightweight Wasm runtime. We started our design by selecting the container runtime and the Wasm runtime we wanted to work with. We made our decision based on available sources and our knowledge and skills. Later, we moved on to the iterative prototyping phase, which finished with a working prototype that conformed to the requirements set beforehand.

On the technical side, to integrate a new, more lightweight Wasm runtime into an OCI container runtime, our implementation utilizes a WAMR shared library to execute the Wasm binaries. The OCI runtime, instead of executing the Docker image's entry point, was instructed to extract the Wasm binary from the Wasm OCI image and execute it using methods from the dynamically loaded shared library.

RQ2: What is the memory footprint and startup time of our new Wasm-enabled container runtime compared to the currently available container runtimes?

Our benchmarks demonstrated that the crun runtime with WAMR embedded uses, on average, at least 52.57% (data from Kubernetes metric server) or at least 40.4% (data from Linux `free` command) less memory per pod compared to other Wasm runtimes integrated with crun. Compared to the runwasi shims, our work uses, on average, at least 18.94% (data from Kubernetes metric server) or at least 11.83% (data from Linux `free` command) less memory per pod. When compared to the non-Wasm containers based on Python Docker image, our implementation showed a memory reduction of at least 18.94%. The modified runtime's startup performance was also evaluated, and it is competitive, especially in smaller deployments. While there is a slight performance degradation compared to the leading Wasm runtime for larger deployments, our implementation still outperforms non-Wasm Python containers.

The above answer to research question **RQ2** indicates that we successfully delivered on our research question **RQ1** and thus on our main goal of lowering the memory footprint of running Wasm containers under Kubernetes orchestrator.

7.2 Limitations and Future Work

While the integration of WebAssembly Micro Runtime into crun showed promising results in terms of memory usage and startup performance, several areas for future research were identified. More metrics, including a CPU utilization test, could be included to provide an even more comprehensive performance analysis. This work uses a generic, very simple workload that does not intend to stress test the WebAssembly runtime performance, which leads us to another idea about future work. It could be interesting to perform similar benchmarks to those conducted but using a real-life application or workload. Such experiments could provide valuable insights into WebAssembly runtimes' performance and optimization. Using a computation or memory-intensive workload, we would gain an insight into how well each available WebAssembly runtime handles more complex tasks, such as matrix multiplication. The last idea comes from the newest advancements and features introduced in containerd. The Sandbox API mentioned in Section 6 could be a better way of running different types of containers on one Kubernetes cluster. It would be interesting to investigate further the Sandbox API introduced by the containerd and compare this

work performance with the native WebAssembly sandboxer managed by the containerd through the Sandbox API.

References

[1] S. Hassan, R. Bahsoon, and R. Buyya. **Systematic scalability analysis for microservices granularity adaptation design decisions**. *Software Practice and Experience*, **52**:1378–1401, 2022. 1

[2] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. **Performance evaluation of microservices architectures using containers**. *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015. 1

[3] Z. Rejiba and J. Chamanara. **Custom scheduling in kubernetes: a survey on common problems and solution approaches**. *ACM Computing Surveys*, **55**:1–37, 2022. 1

[4] B. C. Şenel, M. Mouchet, J. Cappos, T. Friedman, O. Fourmaux, and R. McGeer. **Multitenant containers as a service (caas) for clouds and edge clouds**. *IEEE Access*, **11**:144574–144601, 2023. 1

[5] Datahub Analytics Team. **Kubernetes: Adoption and Market Trends — datahubanalytics.com**. `https://datahubanalytics.com/kubernetes-adoption-and-market-trends/`. [Accessed 18-07-2024]. 1

[6] Edge Delta. **Kubernetes Adoption Statistics: Unveiling Global Trends**. *Edge Delta Blog*, May 2024. Accessed: 2024-07-26. 1

[7] S. Park and H. Bahn. **Performance analysis of container effect in deep learning workloads and implications**. *Applied Sciences*, **13**:11654, 2023. 1

[8] P. E. Mergos. **Seismic design of reinforced concrete frames for minimum embodied co 2 emissions**. *Energy and Buildings*, **162**:177–186, 2018. 1

[9] B. Erdenebat, B. Bud, and T. Kozsik. **Challenges in service discovery for microservices deployed in a kubernetes cluster – a case stud**. *Infocommunications Journal*, **15**:69–75, 2023. 1

[10] M. N. Hoque and K. A. Harras. **Webassembly for edge computing: potential and challenges**. *IEEE Communications Standards Magazine*, **6**:68–73, 2022. 2

[11] WebAssembly. **WebAssembly**. `https://webassembly.org/`, 2024. Accessed: 2024-07-26. 2

[12] A. Jangda, B. Powers, E. Berger, and A. Guha. **Not so fast: analyzing the performance of webassembly vs. native code**. 2019. 2

[13] R. Vaño, I. Lacalle, P. Sowiński, R. S-Julián, and C. E. Palau. **Cloud-native workload orchestration at the edge: a deployment review and future directions**. *Sensors*, **23**:2215, 2023. 2

[14] M. Plauth, L. Feinbube, and A. Polze. **A performance survey of lightweight virtualization techniques**. pages 34–48, 2017. 2

[15] J. Han, Z. Zhang, Y. Du, W. Wang, and X. Chen. **Esfuzzer: an efficient way to fuzz webassembly interpreter**. *Electronics*, **13**:1498, 2024. 2

[16] A. Jeffery, H. Howard, and R. Mortier. **Rearchitecting kubernetes for the edge**. *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, 2021. 2

[17] Seven Cheng. **WebAssembly on Kubernetes: From Containers to Wasm (Part 01)**. *CNCF Blog*, March 2024. Accessed: 2024-07-26. 2

[18] containers. **crun: A Fast and Lightweight Fully Featured OCI Runtime and C Library for Running Containers**. `https://github.com/containers/crun`, 2024. Accessed: 2024-07-26. 2

[19] containers. **Youki: A container runtime written in Rust**. `https://github.com/containers/youki`, 2024. Accessed: 2024-07-26. 2, 18

[20] containerd. **runwasi: WASI Runtime for containerd**. `https://github.com/containerd/runwasi`, 2024. Accessed: 2024-07-26. 2

[21] WasmEdge. **WasmEdge: High-Performance WebAssembly Runtime for Cloud Native, Edge, and Decentralized Applications**. `https://wasmedge.org/`, 2024. Accessed: 2024-07-26. 3

[22] Wasmtime. **Wasmtime: A Fast and Secure WebAssembly Runtime**. 3

[23] Inc. Wasmer. **Wasmer: The Universal WebAssembly Runtime**. `https://wasmer.io/`, 2024. Accessed: 2024-07-26. 3

[24] A. Iosup, L. Versluis, A. Trivedi, E. v. Eyk, L. Toader, V. v. Beek, G. Frascaria, A. Musaafir, and S. Talluri. **The atlarge vision on the design of distributed systems and ecosystems**. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019. 5

[25] Marcus A. Rothenberger Ken Peffers, Tuure Tuunanen and Samir Chatterjee. **A Design Science Research Methodology for Information Systems Research**. *Journal of Management Information Systems*, **24**(3):45–77, 2007. 5

[26] Richard R Hamming. *Art of doing science and engineering: Learning to learn*. CRC Press, 1997. 5

[27] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. **Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum**. In *Proceedings of the First FastContinuum Workshop, in conjunction with ICPE, Coimbra, Portugal, April, 2023*, 2023. 5

[28] Raj Jain. *The art of computer systems performance analysis*. john wiley & sons, 1990. 5

[29] John Ousterhout. **Always measure one level deeper**. *Commun. ACM*, **61**(7):74–83, jun 2018. 5

[30] Gernot Heiser. **Systems Benchmarking Crimes.** `https://gernot-heiser.org/benchmarking-crimes.html`, 2019. [Accessed 19-07-2024]. 5

[31] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. **Is big data performance reproducible in modern cloud networks?** In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 513–527, 2020. 5

[32] Fabien Arnaud, Cécile Pignol, Pierre Stéphan, AL Develle, P Sabatier, O Evrard, Brice Mourier, Maxime Debret, Cécile Grobois, Laurent Millet, et al. **From core referencing to data re-use: two French national initiatives to reinforce paleodata stewardship (National Cyber Core Repository and LTER France Retro-Observatory)**. In *5th PAGES Open Science Meeting*, 2017. 5

[33] Sonja Bezjak, April Clyburne-Sherin, Philipp Conzett, Pedro L Fernandes, Edit Görögh, Kerstin Helbig, Bianca Kramer, Ignasi Labastida, Kyle Niemeyer, Fotis Psomopoulos, et al. **The open science training handbook**. 2018. 5

[34] Matthias Hauswirth Michael W. Hicks Emery D. Berger, Stephen M. Blackburn. **A Checklist Manifesto for Empirical Evaluation: A Preemptive Strike Against a Replication Crisis in Computer Science**, 2019. [Accessed 19-07-2024]. 5

[35] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selçuk. **Sok: cryptojacking malware**. *2021 IEEE European Symposium on Security and Privacy (EuroSamp;P)*, 2021. 10

[36] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan. **Swivel: hardening webassembly against spectre**. 2021. 10

[37] Partha Pratim Ray. **An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions**. *Future Internet*, **15**(8), 2023. 10

[38] S. S. Salim, A. Nisbet, and M. Luján. **Trufflewasm**. *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020. 11

[39] H. Harnes and D. Morrison. **Sok: analysis techniques for webassembly**. *Future Internet*, **16**:84, 2024. 11

[40] S. Song, S. Park, and D. Kwon. **Metasafer: a technique to detect heap metadata corruption in webassembly**. *IEEE Access*, **11**:124887–124898, 2023. 11

[41] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza. **Acctee**. *Proceedings of the 20th International Middleware Conference*, 2019. 11

[42] J. Lee, T. Yoo, E. Lee, B. Hwang, S. Ahn, and C. Cho. **High-performance software load balancer for cloud-native architecture**. *IEEE Access*, **9**:123704–123716, 2021. 11

[43] X. Zhang, L. Li, Y. Wang, E. Chen, and L. Shou. **Zeus: improving resource efficiency via workload colocation for massive kubernetes clusters**. *IEEE Access*, **9**:105192–105204, 2021. 11

[44] M. H. Bhattacharya and H. K. Mittal. **Exploring the performance of container runtimes within kubernetes clusters**. *International Journal of Computing*, pages 509–514, 2023. 12

[45] R. Valavandan, B. Gothandapani, A. Gnanavel, N. Ramamurthy, M. Balakrishnan, S. Gnanavel, and S. Ramamurthy. **Unleashing the power of kubernetes: embracing openness and vendor neutrality for agile container development in an evolving landscape**. *International Journal of Research Publication and Reviews*, **4**:6215–6229, 2023. 12

[46] I. Mavridis and H. D. Karatza. **Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud**. *Concurrency and Computation: Practice and Experience*, **35**, 2021. 12

[47] Seven Cheng. **WebAssembly on Kubernetes: The Practice Guide Part 02**, March 2024. Accessed: 2024-07-27. 15

[48] N. Zhou, Y. Georgiou, M. Pospieszny, L. Zhong, H. Zhou, C. Niethammer, B. Pejak, O. Marko, and D. Hoppe. **Container orchestration on hpc systems through kubernetes**. *Journal of Cloud Computing*, **10**, 2021. 17

[49] E. Truyen, H. Xie, and W. Joosen. **Vendor-agnostic reconfiguration of kubernetes clusters in cloud federations**. *Future Internet*, **15**:63, 2023. 17

[50] containerd. **containerd CRI Configuration**. `https://github.com/containerd/containerd/blob/main/docs/cri/config.md`, 2024. Accessed: 2024-07-26. 18

[51] Inc. Red Hat. **Considerations in Adopting RHEL 9: Containers**. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/considerations_in_adopting_rhel_9/assembly_containers_considerations-in-adopting-rhel-9#assembly_containers_considerations-in-adopting-RHEL-9, 2024. Accessed: 2024-07-26. 18

[52] Inc. Docker. **Docker Desktop Release Notes: 4.21.0**. https://docs.docker.com/desktop/release-notes/#4210, 2024. Accessed: 2024-07-26. 18

[53] Colin Eberhardt. **The State of WebAssembly 2023**. https://blog.scottlogic.com/2023/10/18/the-state-of-webassembly-2023.html, 2023. Accessed: 2024-07-26. 19

[54] Frank Denis. **Performance of WebAssembly Runtimes in 2023**. https://00f.net/2023/01/04/webassembly-benchmark-2023/, 2023. Accessed: 2024-07-26. 19

[55] M. Farzandipour, Z. Meidani, E. Nabovati, M. S. Jabali, and R. D. Banadaki. **Designing a framework of technical hospital information systems requirements and evaluating the systems implemented through this framework**. 2019. 20

[56] L. Leimane and O. Nikiforova. **Mapping of activities for object-oriented system analysis**. *Applied Computer Systems*, **23**:5–11, 2018. 20

[57] WebAssembly. **WebAssembly C API**. https://github.com/WebAssembly/wasm-c-api/tree/main, 2024. Accessed: 2024-07-26. 25

[58] Rob Wong, Zach Shipko, and Gavin Hayes. **WASI Command and Reactor Modules**. https://dylibso.com/blog/wasi-command-reactor/, 2024. Accessed: 2024-07-26. 27

[59] Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazhen Gu, Yangfan Zhou, and Michael R. Lyu. **Revealing Performance Issues in Server-side WebAssembly Runtimes via Differential Testing**, 2023. 47

[60] Jasper Alexander Wiegratz. *Comparing Security and Efficiency of WebAssembly and Linux Containers in Kubernetes Cloud Computing*. Master's thesis, University of Bremen, 2024. 47

[61] Deis Labs. **Krustlet: A Kubelet to Run WebAssembly Workloads in Kubernetes**. https://krustlet.dev/, 2020. Accessed: 2024-07-25. 47

[62] Feng. **Proposal: Sandbox API**. https://github.com/containerd/containerd/issues/4131, 2021. Accessed: 2024-07-25. 48

[63] Kuasar Project Authors. **Wasm Sandboxer**. https://kuasar.io/docs/architecture/wasm-sandboxer/, 2023. Accessed: 2024-07-25. 48

Appendix A

Reproducibility

A.1 Abstract

This project implements a modified crun container runtime with embedded WebAssembly Micro Runtime (WAMR) in order to lower the memory footprint of running WebAssembly (Wasm) containers. Evaluation of this project is conducted using a modified version of the Continuum framework, which is used for automated Kubernetes cluster creation and execution of the designed benchmarks. All of the code is publicly available. For reproducibility directions, please follow the information below.

A.2 Artifact check-list (meta-information)

- **Program:** Continuum framework, crun, WebAssembly Micro Runtime (WAMR)
- **Run-time environment:** Ubuntu 20.04.3 LTS
- **Metrics:** Memory usage, startup time
- **Experiments:** Container runtime performance
- **How much disk space required (approximately)?:** 10 GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 5 to 12 hours
- **Publicly available?:** Yes

A.3 Description

A.3.1 How to access

- The modified version of the Continuum framework can be accessed through the following link: `https://github.com/macko99/continuum_wasm/tree/wasm`.
- The modified version of the crun container runtime can be accessed through the following link: `https://github.com/macko99/crun/tree/wamr_support`.
- The version of the WAMR used in this thesis can be accessed through the following link: `https://github.com/macko99/wasm-micro-runtime/tree/master`
- The Python script plotting all figures used in this thesis can be accessed through the following link: `https://github.com/macko99/plots_master`
- The WebAssembly OCI container image used in this thesis can be accessed through the link: `https://hub.docker.com/repository/docker/macko99vu/wasmrust`
- The Python Docker image used in this thesis can be accessed through the link: `https://hub.docker.com/repository/docker/macko99vu/pythonbase`

A.3.2 Hardware dependencies

For the purpose of this thesis, a machine with a 20-core Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz CPU and 256GB of RAM was utilized. The full specification of the machine is availbe in Table 5.1. The evaluation was performed on two QEMU virtual machines, each with 100GB of RAM and 8 CPU cores pinned to the physical cores. To reproduce these thesis artifacts, we advise using a similar setup; however, the minimal hardware requirements might be much lower.

A.3.3 Software dependencies

- QEMU 6.1.0
- Libvirt 6.0.0
- Docker 20.10.12
- Python 3.8.10
- Ansible 2.13.2

A.4 Installation

To reproduce the artifacts of this thesis, only the Continuum framework and its dependencies have to be installed manually. To do so, please download the GitHub repository and follow the step `Part 1: Install the framework` from the `README.md` file.

After installation, the username has to be adjusted in the following files to match the current user and its directories in the system:

- `configuration/aa_mkozub/*`: In each configuration file, there is a parameter called `base_path`; remove it altogether if the user's home directory has enough space available to store VM images there, or adjust accordingly. See the configuration template for more details about this parameter: `https://github.com/macko99/continuum_wasm/blob/wasm/configuration/template.cfg`.
- `scripts/replicate_kubecontrol.py`: look for `self.username = "mkozub"` and adjust the username accordingly.

Now, the Continuum is ready to be used. From within the Continuum directory, the benchmark can be started by running a configuration file with the below command:

```
1 python3 continuum.py configuration/aa_mkozub/pod_10.cfg
```

A.5 Experiment workflow

Each execution of the Continuum framework is benchmarking one configuration of container runtime and, if applicable, the underlying Wasm runtime. Thus, to replicate this thesis evaluation, at least nine benchmark runs are required. However, for higher reliability and validity of the evaluation, we recommend replying to each benchmark multiple times. All unique benchmarks conducted as part of this thesis included the following configurations:

- Running Wasm containers using crun container runtime and WAMR runtime.
- Running Wasm containers using crun container runtime and WasmEdge runtime.
- Running Wasm containers using crun container runtime and Wasmtime runtime.
- Running Wasm containers using crun container runtime and Wasmer runtime.
- Running Wasm containers using containerd-shim-wasmedge runtime.

- Running Wasm containers using containerd-shim-wasmtime runtime.
- Running Wasm containers using containerd-shim-wasmer runtime.
- Running Python containers using crun container runtime.
- Running Python containers using runC container runtime.

To adjust the Continuum framework for each required run, please localize the phrase `MKB1` in the Continuum source code. This should indicate five distinct locations among three files where appropriate parameters must be adjusted. All required changes are explained in the source code to make this process straightforward.

A.6 Evaluation and expected results

After each experiment has finished execution, there are several log files generated by the Continuum framework. Those logs include all collected metrics needed to reproduce this thesis's findings. The code from the GitHub repository used for plotting all figures for this work contains useful scripts that can help extract the relevant information from log files and plot new figures. Similar results to the ones presented in Chapter 5 are expected.

A.7 Notes

Should any questions regarding this work arise, please contact me via GitHub.