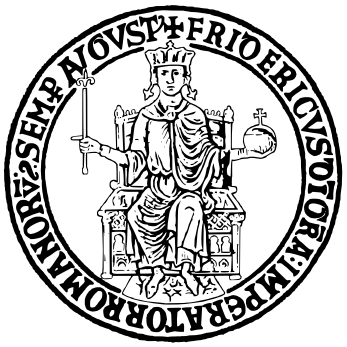
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Classe di Laurea LM-32
Tesi di Laurea in High Performance and Quantum Computing
in collaborazione con il Centro Aerospaziale Tedesco (DLR)

A Systematic Evaluation of MPSoC-enabled Real-Time Containers for Space Environment Applications

Relatori
Ch.mo Prof. Alessandro Cilardo
Dr. Zain A. Haj Hammadeh

Candidata
Carla Coppola
M63001571

Correlatore
Ing. Vincenzo Maisto

Anno Accademico
2024 - 2025

Università degli Studi di Napoli Federico II
Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Classe di Laurea LM-32

Tesi di Laurea in High Performance and Quantum Computing
in collaborazione con il Centro Aerospaziale Tedesco (DLR)

A Systematic Evaluation of MPSoC-enabled Real-Time Containers for Space Environment Applications

Relatori
Ch.mo Prof. Alessandro Cilardo
Dr. Zain A. Haj Hammadeh

Correlatore
Ing. Vincenzo Maisto

Candidata
Carla Coppola
M63001571

Anno Accademico
2024 - 2025

"Eat this life 'til your heart is full
If you want, you can have it all"

— BTS
(*Like Animals*)

Aknowledgments

I would like to express my sincere gratitude to the *German Aerospace Center* (DLR) for granting me the opportunity to undertake my graduation internship and subsequently my master's thesis in such an inspiring and international environment. These experiences have been invaluable for both my professional and personal growth.

I would especially like to thank my company supervisor, *Dr. Zain A. H. Hammadeh*, whose unwavering guidance and sincere encouragement have been fundamental to the completion of this work and have deeply inspired my approach to research.

I am profoundly grateful to *Professor Alessandro Cilardo* from the University of Naples Federico II for his kindness, mentorship, and support during challenging times.

I would also like to extend my gratitude to all the *amazing people I met during my time at DLR*: you have deeply inspired and motivated me to always give my best. In particular, I would like to thank *Armin Purle-Kopacz* for his valuable technical and motivational support from afar throughout these months.

I am also truly thankful to all the *members of the Embedded Systems Lab* for the insightful conversations and advice, as well as for the moments of laughter and support during the hours spent there. In particular, I would like to thank *Vincenzo Maisto* for his patience and guidance during those days.

Finally, I would like to acknowledge the outstanding support of *my parents, my boyfriend, my friends and university colleagues*, without whom this work and the achievement of my degree would not have been possible.

Abstract

In recent years, the growing number of satellites has motivated space agencies to open spacecraft software development to third parties, highlighting the need for ways to abstract away the complexity linked to the aerospace domain, in order to speed up the development of experiments that will run in space. For this reason, containerization is emerging as a promising solution to simplify deployment and provide isolation among applications, but its compatibility with real-time requirements - essential in space applications - remains largely unexplored, especially on embedded ARM platforms used in space agencies like DLR.

This thesis investigates the feasibility of enabling real-time containers on MPSoC-based boards representative of on-board computers used in space. A complete software stack was designed and implemented, combining a Linux kernel extended with Xenomai 4's EVL real-time core and container-based isolation provided by Podman. The system was first implemented on an emulated ARM Cortex-A53 Zynq UltraScale+ MPSoC platform using PetaLinux and QEMU, and then it was also deployed on the real (same) ZCU102 board.

An extensive experimental evaluation has been conducted to assess three main aspects: the ability to run real-time applications inside containers, their concurrent execution, and the scalability of the system on the target hardware. The results have shown that real-time execution within containers is feasible, although it introduces overhead and reduces determinism compared to bare-metal execution, and that multiple concurrent containers are also able to run while guaranteeing real-time capabilities for the applications inside them. An interesting aspect has been the feasibility of executing rootless containers with real-time capabilities, since they are more valued for safety reasons.

Overall, this work demonstrates that real-time containers on embedded platforms are a viable solution for space-oriented applications where isolation, performance and predictability are the most important guarantees to achieve.

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
BSP	Board Support Package
CFS	Completely Fair Scheduler
DLR	Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center)
DTB	Device Tree Blob
DTG	Device Tree Generator
DTS	Device Tree Source
EVL	Embedded Virtual Linux
FDT	Flattened Device Tree
FIFO	First-In-First-Out
IB	In-Band
OCI	Open Container Initiative
OOB	Out-Of-Band
OS	Operating System
PID	Process Identifier
PMU	Processing Management Unit
RPM	Red Hat Package Manager
RT	Real Time
SoC	System on Chip
TID	Thread Identifier
UAPI	User Application Programming Interface
UID	User Identifier
VM	Virtual Machine
WCET	Worst Case Execution Time

Chapter 1

Project Context and Background

The following chapter introduces the context and motivations of this thesis.

In particular, the 1.1 paragraph focuses on the paradigm change that is being observed in recent years in spacecraft software development, due to the increasing number of orbiting satellites and how space agencies are addressing it. The 1.2 paragraph is focused on the specific approach provided by *DLR* (*German Aerospace Center*) with which this thesis was realized. Finally, the 1.3 paragraph gives an overview of the steps followed to reach the goal of the thesis.

1.1 Introduction

In recent years, the space sector has experienced a profound transformation driven by the rapid growth of satellite missions and the rapid expansion of small satellite platforms such as *CubeSats* and *nanosatellites*. Indeed, according to *United Nations Office for Outer Space Affairs* (UNOOSA) [1], by 2025 nearly 10,000 active satellites were orbiting the Earth, reflecting an unprecedented change of paradigm and the diversity of actors participating in space activities.

In this context, reliable and high-performance *On-Board Computers (OBCs)* have become increasingly important.

Space missions often involve operations thousands of kilometers from Earth, where limited communication windows and stringent real-time requirements make the timely and safe execution of software *critical*. A single failure in a minor application can potentially disrupt major system functions, highlighting the importance of *safety, security and application isolation*.

This paradigm shift is largely fueled not only by the proliferation of small satellites and the deployment of large constellations, but also from the growing involvement of private companies, research institutions and universities in the development of space technology, and from the

innovative technical approach that big space agencies are adopting.

Space agencies indeed, like *ESA* and *DLR*, are trying to open up the development of space software to a broader community of developers, including researchers and engineers who may not have deep expertise in astrodynamics or spacecraft engineering, to speed up the development of experiments that need to run in space, with the final goal of significantly accelerating innovation and experimentation.

This more-open approach, even if enables a larger ecosystem of contributors to design and deploy algorithms for onboard processing and scientific analysis, also sets a not-coming-back point in terms of how safety and security have to be guaranteed on spacecraft. When third-party applications are enabled to run on a spacecraft indeed, it is mandatory to ensure that they are not malicious and to prevent the harm that their unexpected behavior can cause to other experiments running alongside, or to the spacecraft and the mission themselves.

Moreover, the paradigm is increasingly shifting toward *in-situ data processing*, as already highlighted by several research initiatives and as reported in the *Nebulae study* [2] presented at the 2020 IEEE Aerospace Conference. This study indeed, demonstrated that the future of scientific space missions will rely increasingly on processing scientific data directly onboard, rather than transmitting all of them back to Earth for analysis. Such approach will no longer be practical due to the quantity of them. Consequently, spacecraft will need to *handle advanced onboard data processing*, requiring a safe and secure environment capable of supporting high-performance and mixed-criticality applications in space.

At the same time, today *on-board computers* in spacecrafts are also expected to host multiple experiments [3] and applications running concurrently, sometimes even developed from different organizations, and this behavior is the root of various problems.

Applications developed from different organizations may have heterogeneous strict performance and temporal requirements, necessitating predictable execution times and controlled access to shared hardware resources, hence managing their concurrent execution on the same OBC can result in a challenge. Furthermore, heterogeneous provenience can also set a significant safety and security problem, addressing which may require additive precautions to avoid malicious software or unwanted behavior to put the satellite or the mission at risk.

As a consequence of all of this, if traditionally space missions relied on tightly integrated and monolithic software stacks, developed just for a specific mission, today *on-board computers* face the need to support multiple independent applications, potentially developed by different stakeholders, causing spacecraft software architectures to become increasingly complex from an *embedded systems* point of view.

To address these challenges, space agencies and scientists are moving in the direction of a

more flexible and modular software architecture approach, based on *virtualization technologies* which can enable multiple applications to run in *isolation* on shared computing hardware while also maintaining guaranties on performance and resource allocation.

Indeed, virtualization techniques are already widely adopted in *cloud and edge computing environments* because, by isolating workloads and abstracting the underlying hardware platform, they can also help preventing malicious behaviors in an application propagate to others and stay confined. Furthermore, when combined with appropriate *real-time scheduling mechanisms*, such solutions may also support applications with strict timing and performance requirements.

Hence, within this evolving landscape, the development of robust mechanism for *virtualizing applications in space systems* represents a key enabling technology. Moreover, these mechanisms need to address both the flexibility of modern computing platforms and the stringent constraints imposed by space environments, including limited computational resources and strict temporal guaranties.

The present thesis is situated within this context and aims to contribute to this ongoing research, proposing a software solution to achieve real-time isolation on an embedded platform representative of the on-board computers for spacecrafts, through *Xenomai 4* and *Linux Containers*.

1.2 Research at the *DLR*

In this context and among the big space agencies that are renewing their technical approach for spacecraft software development, there is the *German Aerospace Center* (*DLR – Deutsches Zentrum für Luft und Raumfahrt*), where researchers and engineers are developing a new onboard application framework for space missions, known as *Stellar Apps*.

DLR is Germany's national research center for aerospace, energy, transport, digitalization, and security, and it also acts as the country's official space agency. It consists of more than 50 institutes and facilities.

Among these, the Institute of Software Technology with its *Flight Software Department* is dedicated to creating advanced and innovative software solutions for space and aerospace missions.

Specifically, the Flight Software Department at DLR – in collaboration with which this thesis was carried out – concentrates on software quality assurance for safety-critical missions and on the development of dependable flight software, ranging from embedded and hardware-related systems to sophisticated payload control.

In recent years, the the Flight Software Department has already implemented a project named *ScOSA* (see Par. 1.2.1), whose goal was to realize an interconnected architecture of computing nodes capable of running parallel experiments on a multipurpose spacecraft, while ensuring safety and stability.

Now, researchers are working on the next step in this direction through the development of the *Stellar Apps* project. This initiative was conceived to address the limitations and shortcomings of *ScOSA* and to further support the broader effort to open-up satellite software development to a wider community, including students and small companies.

The work presented in this thesis is carried out within the context of the *Stellar Apps* project and represents a step toward its development.

1.2.1 ScOSA

ScOSA stands for "Scalable On-board Computing for Space Avionics" Flight Experiment" and it represents an important project carried out at the DLR.

It consists in a distributed, heterogeneous On-Board Computers (OBC) architecture composed of multiple computing nodes that are interconnected in a network.

In recent years in fact, the demand for reliable, high-performance OBCs for space missions grew immensely. The reason relies mainly in the fact that modern satellite and spacecraft computer systems require ever-increasing processing power, in order to support demanding Earth observation missions and autonomous exploration of other celestial bodies.

The ScOSA Flight Experiment indeed, is designed to provide this enhanced computing capability.

The system's heterogeneity results from the deliberate combination of different processor types. Processors widely used in terrestrial applications are powerful and cost-efficient, but they are highly vulnerable to cosmic radiation, which can interfere with their operation and lead to their failure. Consequently, radiation-hardened processors are also required, even though they offer lower performance. By integrating commercial off-the-shelf (COTS) and radiation-tolerant nodes via a SpaceWire network, ScOSA provides a system software stack that supports parallel computing and dynamic system reconfiguration.

In particular, the ScOSA Flight Experiment includes eight Xilinx Zynq 7000 system-on-chips with dual-core ARM-based processors, along with a LEON3 radiation-tolerant processor. In this experiment, all 18 cores, as well as the programmable logic on the Zynq FPGAs, are exploited for high-performance on-board data processing, i.e., for hardware-accelerated computation.

The central concept is that the ScOSA middleware detects failures and transparently migrates

parallel tasks from failing nodes to nodes that are still operating correctly, thereby increasing the overall reliability of the system. Once a node is restored, it can be automatically rejoined to the network and tasks can again be scheduled on it.

One of the main limitations of the current design is that all applications intended to run on the architecture are compiled into a single monolithic binary. As a result, the middleware is responsible for mapping all application tasks and communication channels to the various computing nodes. While this enables high utilization of the system, it reduces flexibility during the development and integration of different applications. In addition, a failure in a single application can compromise the entire system. Stellar Apps, the successor to ScOSA, is intended to address and improve upon these shortcomings.

By the end of 2024, the ScOSA project had successfully operated on ESA's OPS-SAT satellite. The plan is to continue its development so that it can also be deployed on the DLR CubeSat in 2026, together with additional AI applications.

1.2.2 Stellar Apps

Stellar Apps is an on-board application platform for space missions, conceived as an ecosystem that enables the safe and secure execution of third-party applications in space. It includes not only the software running on board, but also a supporting infrastructure on the ground.

As the successor to the ScOSA Flight Experiment Project, its goal is to address and resolve that project's shortcomings, offering an architecture shaped by the lessons learned from the ScOSA activities.

The core objective is to enhance the flexibility and reusability of multipurpose spacecraft (such as satellites, rovers, space stations, and even full satellite constellations). It allows users—including those from universities and the private sector—to add, replace, remove, and update (third-party) applications running on the spacecraft, thereby simplifying and speeding up the development and deployment of future space software.

In fact, most applications are not initially designed with space usage in mind; therefore, when an application later proves useful for a space mission, it typically requires considerable time and effort to adapt it to the harsh space environment and to guarantee compliance with strict safety and security constraints. In contrast, Stellar Apps seeks to shorten the development cycle for space applications by reusing existing code and libraries, while also enabling applications to be updated and tested directly in situ.

Whereas in ScOSA all applications are compiled into a single binary, Stellar Apps requires applications to run independently from one another so that a malicious or faulty application

does not compromise the stability of the entire system. This isolation is implemented using *containers*. Specifically, a container image is a portable, executable package containing a software application and all its dependencies, used to instantiate containers that provide a consistent and reliable runtime environment for the application.

Among the main requirements and core characteristics of the project are:

- Executing applications in an isolated environment with configurable access to spacecraft and instrument data, thereby ensuring mission safety;
- Supporting incremental updates and in-orbit replacement of applications to enable multi-mission spacecraft operations;
- Enabling multi-tenant operation and the execution of untrusted code in orbit by relying on a robust security framework that regulates resource access and prevents faulty applications from compromising the mission;
- Satisfying the timing constraints of real-time applications and delivering real-time guarantees through the integration of the Xenomai 4 co-kernel and PREEMPT-RT in Linux-based nodes;
- Offering a software development environment that closely mirrors the actual execution environment, reducing the time required to deploy applications in space;
- Supporting AI applications by providing AI-specific libraries and accelerators;
- Allowing a broad ecosystem of application developers to build software for space systems, through an accessible API-based communication interface between applications and the underlying hardware;

Stellar Apps is structured into three primary components: the *space segment*, the *ground interface* and the *application development*, as illustrated in Fig. 1.1.

Together, these components enable the secure execution of potentially untrusted third-party applications in orbit. The Stellar Apps ground interface is a trusted entity that links application developers with the spacecraft. It offers a container-based development environment. Applications developed in this environment are submitted to the ground interface, which verifies their authenticity and checks their requested permissions on the spacecraft. After validation, the ground interface generates an app bundle enriched with information on granted entitlements and other specific details, and then transmits it to the spacecraft. Consequently, applications are deployed as *app bundles*.

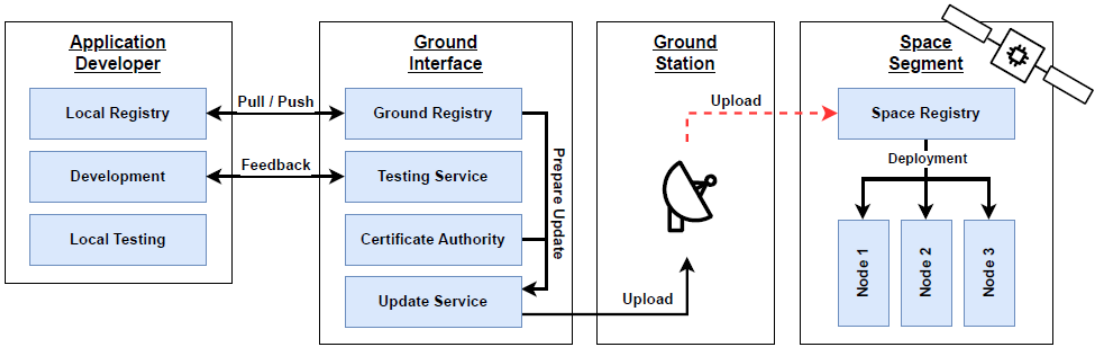
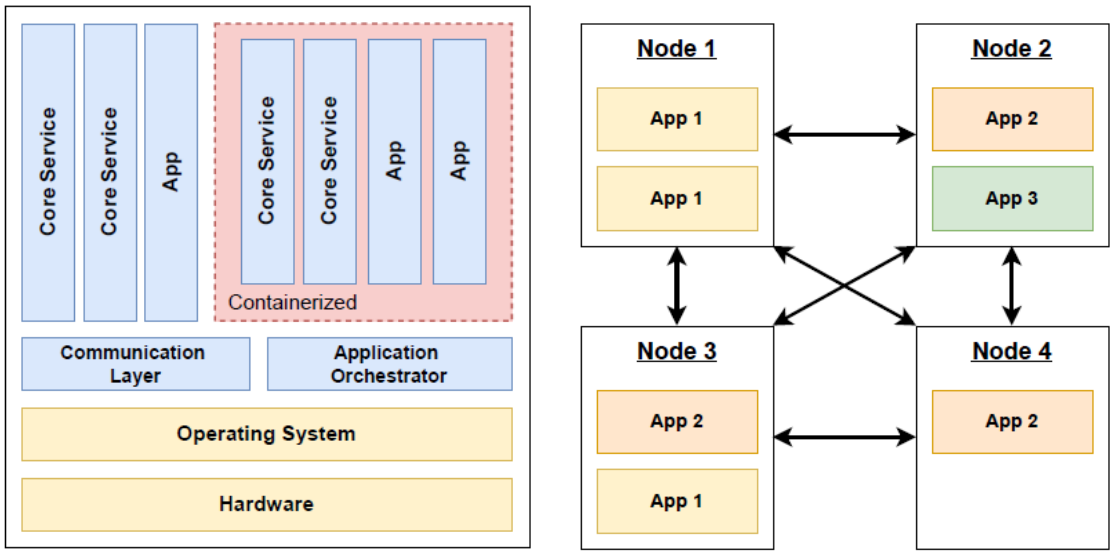


Figure 1.1: Architecture of Stellar Apps [3]

Once deployed, the applications are run by the Stellar Apps software platform in the space segment.



(a) *Layered software stack for a single node* (b) *Nodes' infrastructure*

Figure 1.2: Stellar Apps architecture [3]

As depicted in Fig. 1.2a, the software stack for a single node includes, among others, the following layers:

- the *communication layer*, which manages secure communication between applications;
- the *application orchestrator*, which is responsible for starting, stopping, installing, updating, and removing applications on the platform.

Specifically, the orchestrator manages the containerization engine, which isolates applications and restricts their interaction with the underlying system. When non-nominal behavior is detected, the application orchestrator is able to terminate the affected applications. In the event that a malfunctioning application is identified, a detailed report is forwarded to the ground segment, and the application is shut down. A registry maintains the available applications and offers a collection of supported frameworks and libraries in multiple versions.

As illustrated in Fig. 1.2b, applications may run on a single computing node or be distributed across several nodes. Communication links between nodes enable data exchange among applications. Data is only shared when the participating applications consent.

Stellar Apps is built for broad compatibility across diverse hardware architectures and offers a robust development environment grounded in the Linux Operating System.

1.3 Proposed contribution of the thesis

Within the context of ensuring isolation between applications running on on-board computers in satellites and spacecraft in general, the goal of this thesis is to enable such isolation among real-time applications running on the same embedded platform through *containerization* — specifically using *Podman containers* — and to investigate the overhead introduced by such real-time containers.

In particular, Since the operating system running on these kind of platforms is usually *GNU/Linux* that is a *general-purpose operating system*, it needs to be made *real-time* to ensure real-time guaranties to applications that require them.

For this reason, in this thesis, to transform Linux into a real-time operating system, the *Xenomai 4* approach - with the *EVL core* - was explored and applied. This is not the only existing solution to enable real-time capabilities in Linux, but it was not extensively studied yet and, above all, it was never used to enable real-time containers.

Moreover, since *isolation* can be achieved not only through containers but also through *hypervisors or virtual machines*, an alternative approach would have been represented by them, if it was not for the fact that such techniques have been already widely explored and are known to incur significant performance overhead. *Containers* instead, appear to be noticeably lighter in terms of memory and performance overhead, a desirable requirement in strict space environment.

To accomplish this goal, a *software stack* was designed and implemented by carrying out the following steps:

1. Patching the Linux kernel with the *EVL core* provided by *Xenomai 4* to enable real-time capabilities;
2. Emulating the target hardware platform (*AMD Xilinx Zynq MPSoC UltraScale+ ZCU102 platform*) using *QEMU* and *Petalinux* integrating the custom-kernel image;
3. On the QEMU-emualted board, enabling Podman containers with real-time libraries, in *Rootful* mode;
4. Exploring the feasibility of enabling real-time capabilities in rootless containers;
5. Examining the possibility of running the real-time application inside a *Rootless* container;
6. Assessing the *overhead* - in terms of *latency execution time* - introduced by the containerization, through comparison with a native execution of the same real-time application on the *bare-metal system* and inside a *Rootful* and *Rootless* container.
7. Testing the possibility of running multiple concurrent real-time containers on the QEMU-emulated system;
8. Testing the behavior change of the system when running a different number of concurrent real-time containers;
9. Further testing the QMEU-emulated system to find the limit in the number of multiple concurrent real-time running containers supported.

Chapter 2

State of the Art and Used Technology

The growing adoption of Linux-based embedded platforms in modern space systems, together with the increasing complexity and modularity of on-board software, is driving a convergence between research areas that were traditionally treated separately. On the one hand, space software is progressively moving toward more flexible and software-defined paradigms, inspired by cloud and edge computing approaches and supported by lightweight virtualization mechanisms such as containers. On the other hand, the stringent requirements of space systems in terms of predictability, reliability, temporal isolation, and mixed-criticality execution continue to demand strong foundations in real-time systems engineering.

This convergence raises a central challenge: although container-based virtualization offers significant advantages in terms of software portability, modularity, deployment flexibility, and reduced overhead, its adoption in safety-critical and time-sensitive environments is only meaningful if the underlying operating system is able to provide real-time guarantees. For this reason, the transition toward container-enabled space systems is tightly connected to the evolution of Linux from a general-purpose operating system into a real-time execution platform, through approaches such as PREEMPT_RT, hierarchical scheduling based on cgroups, and co-kernel architectures such as Xenomai.

This chapter reviews the state of the art and introduces the technological foundations that motivate the work developed in this thesis. It begins by analyzing the evolving landscape of space software, showing how recent missions and research initiatives are progressively adopting container-based and cloud-inspired paradigms to reduce entry barriers and enable more flexible on-board software (see Par.2.1).

It then examines the enabling technologies required to support such a transition, focusing on the main approaches used to provide real-time behavior in Linux-based systems: the PRE-

EMPT_RT patch, cgroup-based hierarchical scheduling, and co-kernel architectures. Particular attention is devoted to Xenomai and, more specifically, to Xenomai 4 and its EVL core, since this framework constitutes one of the key technological pillars of this thesis (see Par.2.2).

Building on these foundations, the chapter then discusses the transition from real-time Linux mechanisms to container-based virtualization in mixed-criticality and space systems, highlighting both the opportunities offered by real-time containers and the limitations that still characterize the current literature (see Par.2.3).

Finally, the chapter identifies the specific research gap addressed by this work, namely the lack of integrated real-time containerized execution environments for embedded ARM-based Linux platforms in the context of space-oriented mixed-criticality systems (see Par.2.4).

2.1 The evolving landscape of Space Software

Space missions represent for sure one of the most critical, delicate and advanced contexts in which software and hardware properly need to integrate between themselves. Indeed, the main goal is to ensure reliability, precision, safety and security and, in the case of real-time components, predictability and determinism.

Historically, only large organizations with specialized engineering teams and multimillion-dollar budgets could build and deploy satellite software, since the complex knowledge required - ranging from satellite-specific hardware to orbital mechanics - made entry into the field nearly impossible [4].

In recent years instead, with advances made in the direction of small satellites such as *CubeSats* and *nanosatellites* [5], space became within reach for many small agencies, groups of researchers and university students, making the number of objects orbiting Earth grow to between 10,000 and 12,000 by 2025 [1]. This shift triggered a transition from highly specialized, monolithic on-board software toward more flexible, software-defined payload architectures for spacecrafts.

The *European Space Agency*'s *OPS-SAT* mission [6] is a significant example of this transition. It consists in a triple-unit CubeSat built by the Graz University of Technology for ESA around an ARM-based SoC with FPGA fabric and launched in December 2019, which hosted experiments from over 100 entities across 18 countries and explicitly conceived to break the *"has never flown, therefore it will never fly"* barrier that prevents many ideas from ever reaching orbit due to excessive cost or risk.

This approach breaks the traditional validation bottleneck and enables a wider community -

including academia, startups, and even individual developers - to test novel ideas directly in space.

Similarly, the emergence of *satellite-as-a-service* models further reduces the entry barrier. For instance, in order to increase the number and diversity of people who can access space and to speed-up the advancement in the space field, some commercial platforms like *Exodus Orbitals* [4] have taken a further step by adopting *Docker containers* as a deployment mechanism, allowing developers with no prior satellite expertise to build and deploy satellite applications in a matter of days, reducing development timelines from years to months.

This democratization is also supported by the increasing availability of shared satellite infrastructures and hosted payloads, as highlighted in *CubeSat ecosystems* [7], where users can deploy software without building the entire spacecraft.

In parallel, research efforts such as DLR's *Stellar Apps* [3] propose modular on-board software platforms where third-party applications can be deployed, updated, and executed in isolated environments, enabling continuous evolution of mission capabilities even after launch.

Other works, such as [2], envision spacecraft as distributed computing nodes capable of hosting data processing pipelines and cloud-like services directly in space.

Moreover, the introduction of virtualization concepts in avionics architectures, such as *Plug & Fly approaches* based on hypervisors [8], highlights the need for flexible and reconfigurable software environments in modern missions.

Similarly, *sandboxing approaches for CubeSat applications* [9] demonstrate how isolation mechanisms can enable the safe execution of heterogeneous software components on shared platforms.

Overall, recent works show that space agencies and industry stakeholders are progressively shifting toward *container-based* and cloud-inspired paradigms, enabling faster development cycles, increased flexibility, and broader participation in space missions.

This evolution is also consistent with a broader trend observed in *mixed-criticality computing*, where traditional hypervisor-based partitioning is progressively being complemented, and in some cases challenged, by lighter virtualization mechanisms based on containers. In fact, the literature highlights that *containers* are increasingly regarded as an attractive solution for consolidating workloads with different criticality levels on the same hardware, thanks to their lower overhead, simplified deployment and improved scalability with respect to full virtual machines.

At the same time, these advantages come with a fundamental requirement: containerized applications can only be used for time-critical workloads if the underlying operating system is able to provide predictable timing behavior and strong temporal isolation [10, 11, 12]. For this

reason, the transition toward software-defined and container-enabled space systems is tightly connected to the evolution of Linux into a *real-time execution platform.*

Enabling *real-time behavior* on the general-purpose operating system Linux, requires approaches such as the *PREEMPT_RT patch*, co-kernel architectures like *Xenomai or RTAI*, or *hierarchical scheduling mechanisms based on Linux control groups* [10].

In this perspective, embedded and space platforms can benefit from the same technological direction: lightweight OS-level virtualization can make software deployment more modular and reusable, but only if supported by a Linux-based system equipped with real-time capabilities and by kernel mechanisms such as namespaces and cgroups, which constitute the foundation of container isolation and resource control [10, 13]. Therefore, before introducing real-time containers for mixed-criticality and space scenarios, it is necessary to examine the enabling technologies that make this possible, namely real-time Linux frameworks and the kernel mechanisms underlying OS-level virtualization.

2.2 Real-Time Linux and foundations for Container-based systems

2.2.1 Real-Time Linux Frameworks

In recent years, significant research efforts have focused on transforming the Linux kernel into a platform capable of providing real-time guarantees such as predictability, bounded latency and temporal isolation. This evolution is crucial to support modern workloads in domains such as industrial automation, robotics, and, more recently, space systems, where flexible software deployment must coexist with strict timing constraints.

Several approaches have been proposed, including the single-kernel solution based on the PREEMPT_RT patch (see Par.2.2.2), the *resource management mechanisms - control groups (cgroups)* - which enable hierarchical resource allocation (see Par. 2.2.3, and the more radical dual-kernel (or co-kernel) architecture (see Par. 2.2.4).

These approaches are widely recognized in the literature as the main techniques to enable real-time capabilities in Linux systems [10], and they represent the technological foundation for supporting more advanced paradigms such as real-time containerization and mixed-criticality execution.

2.2.2 PREEMPT_RT

To enhance the real-time capabilities of the Linux kernel, the PREEMPT_RT patch can be applied. This patch modifies the kernel to minimize latency and improve system responsiveness. Its main goal is to reduce the non-deterministic behavior of standard Linux by enabling the preemption of almost all kernel tasks. As a single kernel approach, PREEMPT_RT allows real-time tasks and non-real-time tasks to coexist within the same Linux kernel, providing a unified execution environment for both types of applications.

More specifically, the PREEMPT-RT patch enables *full kernel preemption*, allowing higher-priority tasks to preempt lower-priority ones even during critical sections of code execution. To support this behavior, the Linux scheduler is modified to implement priority-based scheduling policies that prioritize real-time tasks over non-real-time tasks. This ensures that critical processes are executed in a timely manner and meet their deadlines, leading to reduced latency and enhanced responsiveness.

The most significant changes introduced by PREEMPT_RT are represented by the replacement of locking primitives, such as *spinlock_t*, with a preemptible and priority-inheritance-aware mechanism known as *rtmutex*. Moreover, interrupt handling is enforced through the use of threaded interrupts.

As a result, the kernel becomes fully preemptible, with the exception of a few critical code sections, such as entry routines, the scheduler itself, and low-level interrupt handlers.

Unlike dual-kernel approaches, PREEMPT_RT maintains a *single-kernel architecture*, where real-time and non-real-time tasks coexist within the same execution environment. This simplifies system integration and preserves compatibility with standard Linux tools and applications.

With the release of Linux 6.12, PREEMPT_RT has been included into the mainline kernel enabling real-time scheduling on x86, ARM64 and RISC-V architectures, without requiring modifications to the original kernel source.

Although PREEMPT_RT may not provide the strict determinism required for ultra-low-latency or hard real-time applications [10], it offers a considerable improvement over the vanilla Linux kernel. For this reason, it is widely adopted in scenarios where soft real-time guarantees are sufficient, such as multimedia processing, certain industrial automation processes, and robotics or it is complemented by additional mechanisms such as hierarchical scheduling or container-based isolation when stronger guarantees are required.

2.2.3 RT-cgroups and Hierarchical Scheduling

Another fundamental mechanism for enabling real-time behavior in Linux systems is the use of *control groups (cgroups)* that enables the hierarchical organization of processes and the controlled distribution of system resources among them.

In particular, a cgroup is a collection of processes that are bound to a set of limits or parameters defined via the cgroup filesystem [14]. cgroups are organized in a tree-like structure and every process in the system belongs to exactly one and only cgroup. All threads of a process belong to the same cgroup. Upon creation, a process inherits the cgroup of its parent, although it may later be migrated to a different one. Such migration does not affect any already existing child processes.

A cgroup framework consists of two main components: the *core* and a set of *controllers*.

cgroups are primarily used to:

- Set limits for system resource allocation;
- Prioritize the allocation of hardware resources to specific processes;
- Restrict certain processes from accessing some hardware resources.

Real-time tasks, like any other type of task, are auto-placed under a cgroup, determined mostly by *systemd* or by an application managing its own cgroup sub-tree. These tasks can run until preempted by higher priority tasks or voluntarily yield due to I/O waits or sleep operations. However, such monopolization of CPU resources by real-time tasks may end up destabilizing the system, as it can prevent essential Linux Kernel housekeeping and periodic tasks from executing.

When it comes to *real-time scheduling*, it is essential that a group can rely on a constant amount of bandwidth, such as CPU time. In order to schedule multiple groups of real-time tasks, each group must therefore be assigned a fixed portion of the available CPU time. However, without a guaranteed minimum, a real-time group may fail to meet its timing constraints, while an imprecise upper bound is insufficient for deterministic scheduling. Consequently, the only possible solution is to allocate a fixed CPU time slice to each group. This means that the CPU time is partitioned into periods, specifying how much time can be spent running in a given period. This allocated runtime is reserved for each real-time group, and the other real-time groups are not permitted to use it. The time not allocated to a real-time group will be used to execute normal priority tasks (best-effort tasks) scheduled under the *SCHED_OTHER policy*. Likewise, any unused real-time runtime is also picked up by SCHED_OTHER tasks.

Such mechanisms are particularly relevant for *real-time containers*, since containers rely on cgroups for resource control. Indeed, several works identify cgroups as a key building block for enabling real-time behavior in containerized environments [10].

However, achieving *deterministic behavior* requires more than simple resource partitioning. As discussed in the literature, *real-time scheduling for cgroups* often relies on deadline-based policies, such as extensions of the Linux SCHED_DEADLINE scheduler. For example, in [13], a two-level hierarchical scheduling framework has been proposed, where cgroups are scheduled using deadline-based policies, while tasks within each group are scheduled using fixed priorities.

This approach aligns with broader research on hierarchical resource orchestration [15], where system resources are dynamically allocated among containerized workloads to improve predictability and efficiency.

2.2.4 Co-Kernel Architectures: Xenomai

A fundamentally different approach from the previously discussed solutions is based on the introduction of a dedicated real-time execution core that runs alongside the general-purpose Linux kernel at a higher priority level. This core is capable of intercepting hardware interrupts and deferring them when necessary, ensuring that real-time execution is not affected by the non-deterministic behavior of the standard kernel.

This architecture is commonly referred to as a *co-kernel* or *dual-kernel* approach, and it is adopted by systems such as *RTAI* and *Xenomai* [16, 17].

In a dual-kernel system, the design principle is to minimize the functional overlap between the general-purpose kernel and the real-time core. The latter is responsible for handling time-critical workloads and is intentionally kept simple and decoupled from the rest of the system in order to ensure trustworthiness and predictability.

This separation enables extremely low latency and minimal jitter, making co-kernel architectures particularly suitable for applications with stringent real-time requirements. However, achieving these guarantees requires that applications interact exclusively with the system call interface provided by the real-time core. If a real-time task invokes a standard Linux system call while executing in the high-priority stage, it is automatically migrated to the low-priority (in-band) stage, thus losing its real-time guarantees.

As highlighted in the literature, co-kernel solutions generally provide stronger real-time guarantees than single-kernel approaches such as PREEMPT_RT, at the cost of increased system complexity and reduced transparency for application developers [17, 10].

In a typical dual-kernel (co-kernel) architecture, two main components coexist (see Fig. 2.1):

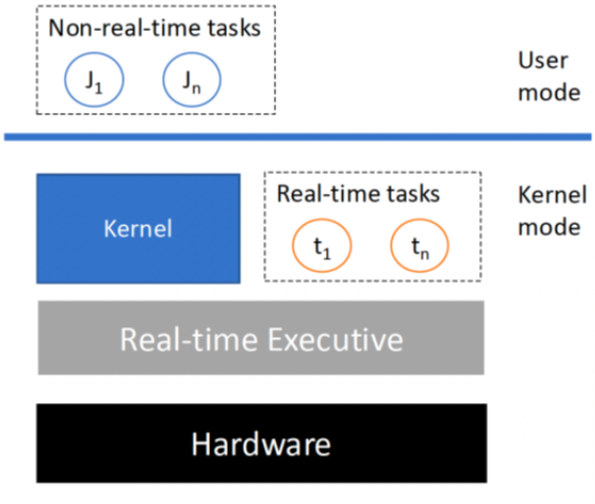


Figure 2.1: Dual-kernel architecture

- a low-level *real-time executive*, which directly manages hardware interactions and interrupts;
- a *general-purpose Linux kernel*, which handles non-critical tasks and deferred interrupts.

The key idea is that interrupts are first intercepted by the real-time executive instead of being immediately delivered to the Linux kernel. Critical interrupts are handled immediately, while non-critical ones are delayed and later forwarded to Linux. In this way, real-time tasks are never blocked by non-deterministic kernel activities.

As a result, the Linux kernel becomes fully preemptible from the perspective of real-time execution, allowing time-critical tasks to interrupt kernel activities at any time and achieve immediate response. The two kernels operate quasi-independently, each serving its own class of workloads, with absolute priority always granted to the real-time core.

This approach was initially introduced by RT-Linux and later adopted and extended by frameworks such as RTAI and Xenomai.

Among these, *Xenomai* represents one of the most mature and widely adopted co-kernel frameworks. It extends Linux by providing real-time capabilities through a companion real-time core (historically known as the *Xenomai nucleus*), enabling deterministic execution, low latency, and strict timing guarantees for real-time tasks [16].

The Xenomai project was launched in 2001 and evolved through several iterations. Initially, it merged with the RTAI project to form RTAI/fusion, and later became an independent framework, continuously evolving toward improved integration with the Linux kernel.

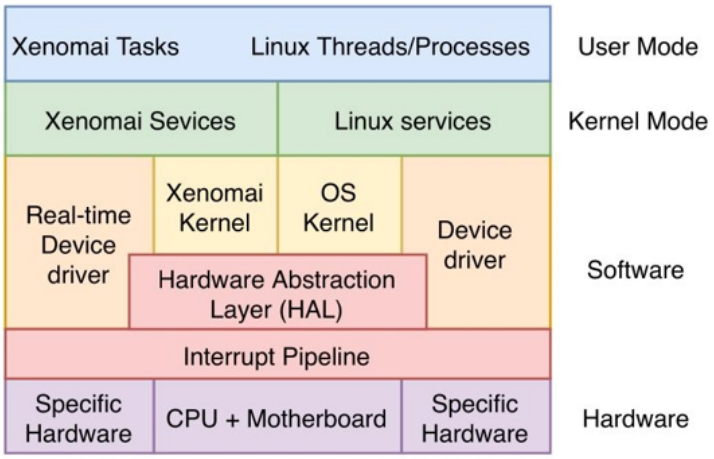


Figure 2.2: Integration of Xenomai within the Linux architecture [18]

The latest generation, *Xenomai 4*, is adopted in this thesis and is described in the following section.

2.2.5 Xenomai 4 and the EVL Core

The latest evolution of the Xenomai framework, *Xenomai 4*, introduces a modern, scalable, and maintainable dual-kernel architecture built on top of the *Dovetail* interface [16, 19].

Dovetail provides a mechanism for integrating a high-priority execution stage into the Linux kernel, enabling the coexistence of real-time and general-purpose workloads within a unified system while preserving strong timing guarantees.

As shown in Fig. 2.3, the Xenomai 4 architecture consists of the following main components:

- The *Dovetail interface*, which acts as a communication layer between the Linux kernel and the real-time execution stage. It introduces an additional high-priority execution context, often referred to as *out-of-band* execution, where real-time tasks can run independently from standard kernel constraints such as interrupt masking and spinlocks.
- The *EVL (Embedded Virtual Layer) core* [21], which replaces the former Cobalt core used in Xenomai 3. EVL is integrated into the Linux kernel as a lightweight subsystem responsible for executing time-critical tasks with deterministic behavior and bounded latency.
- The *libevl* user-space library, which exposes the services of the EVL core to C/C++ applications, providing a set of low-latency primitives designed to operate in the out-of-band context.

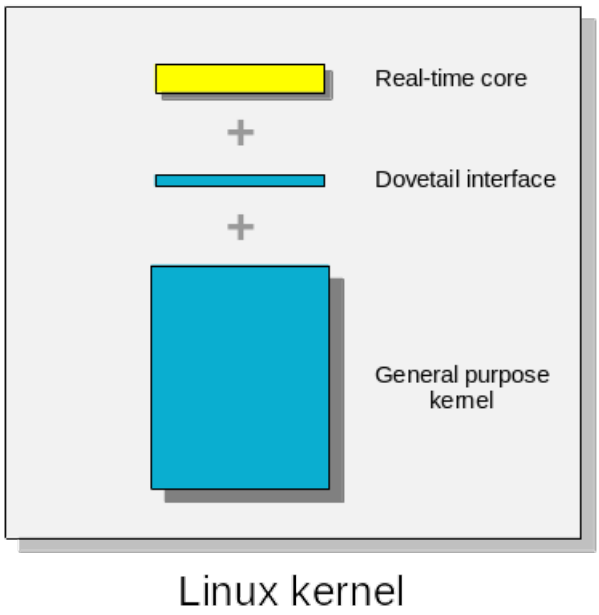


Figure 2.3: Xenomai 4 architecture [20]

- The *revl* Rust crate, which provides similar functionality for Rust-based applications.

Interrupt Pipeline (I-Pipe)

A key mechanism enabling the dual-kernel behavior in Xenomai is the *interrupt pipeline* (I-Pipe), which is part of the Dovetail infrastructure.

The I-Pipe introduces a layered interrupt handling model, where interrupts are first delivered to a high-priority domain (the real-time core) before reaching the Linux kernel. In this model:

- the real-time core has the first opportunity to process interrupts;
- time-critical interrupts are handled immediately in the high-priority stage;
- non-critical interrupts are deferred and later propagated to the Linux kernel.

This mechanism ensures that real-time tasks are not delayed by kernel activities, such as interrupt disabling or critical sections protected by spinlocks, which are common sources of latency in standard Linux systems.

Conceptually, the I-Pipe [22] creates two execution stages:

- an *out-of-band stage*, where real-time tasks execute with highest priority;
- an *in-band stage*, corresponding to the standard Linux kernel execution.

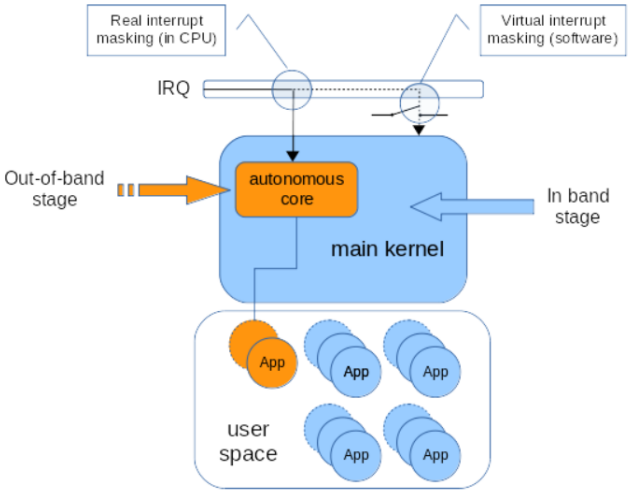


Figure 2.4: Interrupt pipeline (I-Pipe) mechanism in dual-kernel architectures

When a *real-time task* is active, it can preempt the in-band stage at any time. Conversely, when no real-time activity is present, Linux executes normally. This cooperative yet prioritized execution model allows both domains to coexist efficiently.

It is important to note that Dovetail itself only provides the infrastructure for hosting a real-time execution stage, but does not implement a real-time core. This functionality is provided by EVL, which builds on top of Dovetail to deliver full real-time capabilities.

Compared to Xenomai 3, the Xenomai 4 architecture significantly simplifies integration with the Linux kernel, improves maintainability, and provides a more flexible programming model, while still preserving the strong real-time guarantees typical of co-kernel solutions.

2.3 From Real-Time Linux to Container-Based Virtualization in Mixed-Criticality and Space Systems

While the approaches described in the previous sections enable Linux to meet real-time requirements, they also provide the foundation for higher-level virtualization mechanisms (*container-based virtualization*).

Containers rely on two fundamental Linux kernel features:

- *namespaces*, which provide isolation of system resources such as processes, networking, and filesystems;

- *cgroups*, which enable fine-grained control and limitation of resource usage (e.g., CPU, memory, I/O).

Together, these mechanisms implement *OS-level virtualization*, allowing multiple isolated applications to run on the same kernel with minimal overhead. Compared to traditional hypervisor-based virtualization, containers offer near-native performance, faster deployment, and more efficient resource utilization [10].

However, standard containers are not designed to provide real-time guarantees and, in fact, they lack mechanisms to ensure deterministic execution and temporal isolation under resource contention. This limitation has motivated extensive research on *real-time containers*, aiming to combine the flexibility of containerization with the predictability of real-time systems.

Several approaches have been proposed in the literature to enable real-time behavior in containerized environments. For instance, extensions of Linux scheduling policies allow containers to be managed with deadline-based mechanisms, providing temporal guarantees to containerized workloads [13]. In this context, hierarchical scheduling frameworks have been introduced to allocate resources across containers in a predictable manner, ensuring that each workload receives a guaranteed share of system resources [15].

In parallel, co-kernel-based approaches integrate containers with real-time execution cores (e.g., Xenomai or RTAI), achieving strong temporal isolation and low-latency execution even in the presence of mixed workloads [23]. These solutions demonstrate that the evolution of Linux into a real-time operating system is a key enabler for deploying container-based applications in safety-critical domains.

This trend is particularly relevant in the context of *mixed-criticality systems* (MCS), where applications with different levels of criticality must coexist on shared hardware. Traditionally, such systems have relied on hypervisor-based virtualization to ensure spatial and temporal isolation. Hypervisors provide strong guarantees, but at the cost of significant overhead, as each virtual machine requires a full operating system instance.

Recent research highlights a growing shift toward lightweight virtualization solutions based on containers. In fact, containers are increasingly considered a viable alternative to hypervisors, as they enable efficient workload consolidation while reducing computational and memory overhead [**Towards_Lightweight**]. This aspect is particularly important in *embedded and resource-constrained environments*, such as those found in *space systems*.

At the same time, ensuring predictability in container-based mixed-criticality systems remains a key challenge. To address this, several works propose integrating containers with advanced real-time scheduling techniques. For example, hierarchical scheduling approaches

have been used to guarantee temporal isolation among containers with different criticality levels [15]. Similarly, lightweight isolation frameworks aim to replace hypervisor-based architectures with container-based solutions while preserving both temporal and fault isolation [11].

Additional research efforts focus on enabling real-time scheduling directly within containerized environments. For instance, Linux kernel extensions have been proposed to support deadline-based scheduling of containers, demonstrating that temporal guarantees can be achieved without relying on hypervisors [13]. Moreover, orchestration frameworks have been introduced to dynamically allocate and adjust resources among containers, improving real-time performance under varying workloads [15].

Other approaches combine containerization with co-kernel architectures and monitoring mechanisms to ensure temporal separation and fault isolation in mixed-criticality environments [23]. These solutions highlight the feasibility of using containers as a building block for next-generation real-time systems.

Despite these promising developments, the adoption of real-time containers in *space systems* still presents several challenges. Indeed, space platforms impose strict requirements in terms of reliability, determinism and resource constraints, while also increasingly relying on embedded architectures, such as ARM-based systems.

Although real-time container technologies have been extensively studied and validated in industrial and cloud environments, most experimental evaluations have been conducted on *x86 platforms*, with limited evidence of their deployment on embedded architectures typically used in space missions [10].

This gap indicates that, although the technology is maturing, its adoption in real space applications is still at an early stage. In particular, further research is required to:

- ensure real-time guarantees under resource contention;
- provide strong temporal and spatial isolation between workloads of different criticality;
- validate container-based solutions on embedded and space-grade platforms.

Hence, it is possible to affirm that, as the literature suggests, real-time containers represent a promising direction for future space systems due to flexibility and isolation they provide. However, enabling these kind of technologies in practice is still a challenge, especially on ARM-based and resource-constrained platforms.

2.4 Research gap addressed by this thesis

Despite significant progress in real-time Linux and container-based virtualization, a clear gap remains between existing research and their practical adoption in space systems. In particular, most real-time container solutions have been designed and validated in industrial or cloud environments, predominantly targeting x86-based platforms, while their applicability to embedded and space-grade architectures, such as ARM-based systems, is still largely unexplored [10]. Moreover, although several approaches have been proposed to combine containers with real-time scheduling techniques or co-kernel architectures, there is still a lack of integrated frameworks that can simultaneously guarantee temporal predictability, isolation, and efficient resource utilization under the stringent constraints of space missions.

This thesis aims to address this gap by investigating how real-time Linux frameworks and container-based virtualization can be effectively combined on embedded platforms - especially ARM-based ones like the ones used by space agencies - to support mixed-criticality workloads in space systems. In particular, this work focuses on enabling real-time execution in containerized environments by using co-kernel architectures like Xenomai 4, along with Linux mechanisms such as cgroups and namespaces for resource control and isolation.

The main contribution of this work lies in the design and implementation of a real-time capable containerized execution environment on an embedded platform, demonstrating the feasibility of deploying modular and reusable software components while preserving real-time guarantees. By bridging the gap between real-time operating system extensions and container-based virtualization, this thesis provides a step toward the adoption of lightweight, software-defined architectures for next-generation space missions.

Chapter 3

Project Implementation

The following chapter finally delves into the details of the implementation of the project proposed in this thesis.

First, the target hardware and the motivation of the choices are presented 3.2. Then, an overview of the steps followed and reasons behind them is depicted in Par. 3.3. The actual building of the EVL-kernel and the emulation of the target board through QEMU and Petalinux are reported instead in Par. 3.4). Then, the *testing part* of the project started in Par. 3.5, with the final goal of checking if it was possible to run a real-time application (which complete implementation is available in Appendix A) inside Podman containers on the emulated system. The test phase addressed different questions, from the feasibility of running the real-time application inside Podman *Rootful* and *Rootless* containers using *EVL devices* to exploit the real-time capabilities of the system (see Par. 3.5.1), to the possibility of running multiple concurrent Rootless containers with a real-time application executing inside them (see Par. 3.6), to the final question regarding the scalability of the system that brought to discover a limit to the number of concurrent containers running on this kind of architecture (see Par. 3.7).

3.1 ZCU102 Zynq ARM A53 UltraScale+ Board

The hardware platform that has been emulated and employed in this thesis is the *Xilinx AMD Zynq UltraScale+ MP-SoC ZCU102* board (see Fig. 3.1). This board incorporates a quad-core Arm Cortex-A53, dual-core Cortex-R5F real-time processors, and a Mali-400 MP2 graphics processing unit, all built on AMD's 16nm FinFET+ programmable logic fabric [24].

The ZCU102 supports all major peripherals and interfaces, enabling development for a wide range of applications, from automotive to industrial, from video to communications applications.

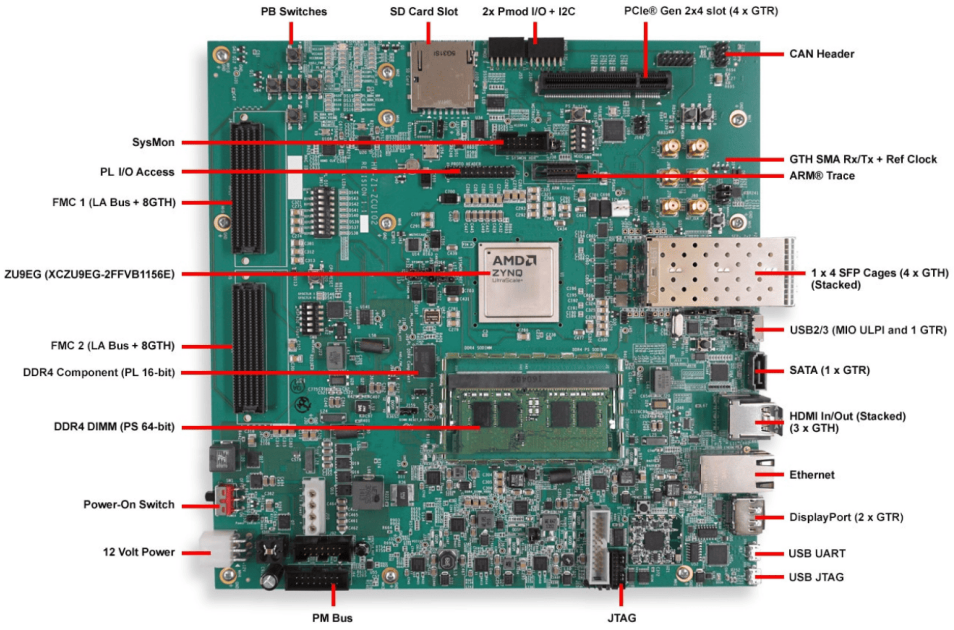


Figure 3.1: AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Platform

The ZCU102 was selected as reference hardware because its family is widely adopted in fields such as *aerospace on-board computers, industrial controllers, and robotics*. Specifically, at DLR the closely related *ZCU104* and *Zynq 7000* are used as on-board computers, including in the *Stellar Apps* and *ScoSA* projects, whereas the ZCU102 was physically available and usable in the laboratory at the University Federico II where this thesis was conducted.

3.2 Tools and technology used in the project

3.2.1 Yocto Project

The *Yocto Project* is an open-source collaboration project that provides tools and templates for creating custom Linux-based systems for embedded platforms. According to the official Yocto Project documentation [25], the framework allows developers to *"create a custom Linux distribution for embedded systems regardless of the hardware architecture"*.

It is not a Linux distribution itself, but rather a framework that enables developers to generate tailored distributions through a reproducible build process.

At its core, Yocto is based on the *OpenEmbedded build system*, which uses metadata and recipes to describe how software components should be configured, compiled, and assembled into a complete system image. This process is orchestrated by the BitBake build engine, which

resolves dependencies and executes build tasks.

The modular structure enables fine-grained control over kernel configuration, root filesystem composition, toolchain generation, package management - making Yocto particularly suitable for complex embedded systems where reproducibility, customization, and cross-compilation are essential.

In reality, in the context of this thesis, the Yocto environment was not directly used because no Linux distribution had to be built from scratch, but rather it was used as a *cross-compilation infrastructure* for targeting the ARM64 ZynqMP platform. Specifically, *Petalinux 2022.1* - Xilinx's embedded development framework built on top of Yocto - was used to bundle a pre-configured Yocto environment that was leveraged to cross-compile the container runtime stack required for the project.

3.2.2 Petalinux

PetaLinux is a Linux-based embedded development framework provided by AMD (formerly Xilinx), built on top of the *Yocto Project* [25, 26]. It simplifies the process of generating embedded Linux systems targeting programmable SoCs such as *Zynq, Zynq UltraScale+* and *Versal* devices.

By leveraging Yocto internally, PetaLinux abstracts much of the complexity of the build system while still benefiting from its flexibility and scalability.

In particular, it provides pre-configured metadata layers for Xilinx platforms, integration with hardware descriptions exported from Vivado, and simplified configuration interfaces. As described in the official *PetaLinux Tools Reference Guide* [26], the toolchain enables the creation of a complete embedded Linux image including bootloaders, kernel, device tree, and root filesystem, starting either from a Board Support Package (BSP) or from a custom hardware design.

In the context of this thesis, PetaLinux 2022.1 was used in a non-standard way, not to perform a full system build, but selectively, to extract specific components that the final emulated system depended upon. Additionally, PetaLinux's Yocto environment was used as a cross-compilation infrastructure to build the container runtime stack as ARM64 RPM packages, since the PetaLinux root filesystem lacked a package manager with feeds containing these tools. Finally, PetaLinux provided the ZynqMP-specific QEMU binaries used to launch the emulated platform, which include Xilinx-specific patches not present in standard distribution QEMU builds.

3.2.3 QEMU

QEMU (Quick Emulator) is an open-source system emulator widely used for *virtualization and hardware emulation.* It enables the execution of software compiled for one architecture on a different host system. It supports multiple modes of operation, including *full-system emulation* — where an entire machine including CPU, memory, and peripherals is emulated — and *user-mode emulation* - where individual applications are executed across architectures.

Its core mechanism relies on *dynamic binary translation (DBT)*, which converts guest instructions into host instructions at runtime, allowing efficient cross-platform execution [27, 28].

QEMU is widely used in both academia and industry as a flexible platform for system prototyping and validation, providing models for CPUs, memory subsystems, and peripherals that enable unmodified operating systems to run in a virtual environment [29].

Within the *AMD/Xilinx* ecosystem, QEMU is extended to support specific SoC platforms, providing accurate representations of hardware components and boot flows [30].

In this thesis, the Xilinx-patched QEMU provided by PetaLinux 2022.1 was used to emulate the ZCU102 board — a Zynq UltraScale+ MPSoC based on four ARM Cortex-A53 cores.

The detailed implementation is reported in Par. 3.4.

Anyway, it is important to note that while QEMU provides functional emulation, it does not guarantee cycle-accurate timing [31], which directly impacted the real-time latency measurements collected in this thesis.

3.2.4 EVL core and libevl

The *EVL core* is the real-time framework at the heart of Xenomai 4, designed to provide deterministic, low-latency execution on Linux systems through a *dual-stage interrupt pipeline architecture* called *Dovetail* [20, 19].

Unlike its predecessor *Xenomai 3 (Cobalt)*, which implemented a true *dual-kernel model* where a separate co-kernel treated Linux as its idle task [32], *EVL* integrates directly into the Linux kernel as an *out-of-band (OOB) execution stage* that takes priority over Linux's in-band stage at the interrupt pipeline level. This means there is only *one kernel binary* — Linux extended with Dovetail and the EVL core — but with two distinct scheduling domains: the OOB stage, managed by EVL, which handles real-time threads with deterministic guarantees, and the in-band stage, managed by Linux's CFS scheduler, which handles all general-purpose workloads including container infrastructure.

According to the official EVL documentation [16], the framework enables highly predictable execution latencies by running time-critical workloads in the OOB stage, isolated from the unpredictability of Linux's in-band execution.

EVL threads are genuine Linux threads — fully visible to the kernel, with real *PIDs* and *TIDs* — that migrate to the OOB scheduling domain upon calling `evl_attach_thread()`.

This architectural property was directly relevant to container integration *in this thesis*: since EVL threads retain their host TIDs regardless of container boundaries, the container isolation layer is completely transparent to the EVL core, allowing containerized RT-threads to attach to the OOB stage exactly as native processes would.

Furthermore, in this thesis, the `linux-evl` *source tree* — the pre-integrated repository maintained by the Xenomai project at [33], containing *Linux 6.12 with Dovetail and the EVL core already merged* — was cross-compiled for ARM64 using the Xilinx ZynqMP defconfig from `linux-xlnx` as the hardware configuration baseline. The resulting kernel image was the kernel running on the emulated ZCU102 throughout all experiments.

This process will be further explained in the Par.3.4.4.

`libevl` is the *userspace library* that provides the API for interacting with the EVL core. Through libevl, applications can create and manage real-time threads and handle synchronization primitives. Furthermore, it allows to perform time-sensitive operations with deterministic guarantees.

Unlike standard POSIX interfaces, libevl maps its calls to EVL-specific kernel mechanisms, ensuring that threads executing through it are scheduled by the OOB stage rather than Linux's CFS.

In *this thesis*, libevl also was cross-compiled (as the kernel image) for ARM64 on the host and then installed into the root filesystem.

Inside *containers*, libevl was made available through *bind mounts*, and the real-time benchmark application used it to attach its periodic task thread to the EVL OOB SCHED_FIFO scheduler at priority 80, achieving the real-time execution whose latency characteristics were measured and analyzed throughout the experimental evaluation.

3.2.5 Podman Containers

Podman is an open-source container engine developed by *Red Hat* for building and running containers compliant with the *Open Container Initiative (OCI)* standards. It provides a complete toolchain for container lifecycle management, from the image creation to the orchestration of containerized workloads.

According to the official Podman documentation [34], it is a *"daemonless, open source, Linux native tool"* designed to simplify the deployment and execution of applications using OCI-compatible containers.

Unlike traditional container engines such as *Docker* though, Podman does not rely on a central background service (daemon): indeed, each container is executed as a direct child process of the user, improving both system transparency and security [35, 36]. From an architectural point of view, Podman is based on the *libpod library*, which provides APIs for managing containers, images, volumes, and groups of containers called pods, maintaining full compatibility with existing container ecosystems - including Docker-compatible command-line interfaces and image formats [37].

Containers managed by Podman follow the paradigm of *OS-level virtualization* [38, 39], where isolated userspace instances are created using two fundamental Linux kernel mechanisms: *namespaces* and *control groups (cgroups)*. These two mechanisms serve complementary roles and together define what a container is at the kernel level.

In more detail, *Namespaces* provide isolation by giving each container a private view of specific system resources, independently from what the rest of the system sees. Indeed, Linux provides several namespace types, each isolating a different resource:

- *PID* namespace, that gives the container its own private process ID space;
- the *network* namespace, that provides isolated network interfaces and routing tables;
- the *mount* namespace, that gives the container its own filesystem tree;
- the *user* namespace, that maps container UIDs/GIDs to host UIDs/GIDs;
- the *UTS and IPC* namespaces, that isolate hostname and inter-process communication resources respectively.

In reality, namespaces control *visibility* but not consumption: indeed, a process inside a namespace cannot see resources outside it, but is not inherently limited in how much CPU or memory it uses.

For this reason, *cgroups* complement namespaces by providing resource control, namely limiting and accounting for how much CPU time, memory, block I/O bandwidth, and number of processes each container group can consume. This is why, when Podman creates a container, it simultaneously creates a new cgroup for that container's processes and configures resource limits accordingly. The kernel then transparently enforces these limits on every process running inside that cgroup, including any processes spawned after container startup.

This approach provides lightweight isolation compared to full virtual machines, enabling efficient resource utilization and rapid deployment.

Another distinctive feature of Podman exploited throughout this thesis was its support for *Rootless container* execution ,

In particular, in the context of *this thesis*, the interaction between Podman and the Linux kernel mechanisms of namespaces and cgroups required careful attention in order to correctly enable real-time container execution on the EVL kernel. First, the system had to be configured to operate in pure *cgroup v2 mode*, since the container runtime refused to operate in the hybrid cgroup configuration that the PetaLinux root filesystem defaulted to. Cgroup v2 was also essential for enabling safe *Rootless execution* which allows containers to run without elevated privileges, reducing the attack surface and aligning with the security requirements relevant to multi-tenant space software platforms such as DLR's *Stellar Apps* [35].

Rootless execution was the primary target configuration of this thesis, with *Rootful containers* used only as a baseline for comparison in the experimental evaluation.

On the *namespace side*, the *user namespace* mechanism required specific handling to allow *Rootless containers* to access EVL devices, which by default were not accessible to remapped unprivileged users inside the container. This was resolved by appropriately adjusting device permissions and preserving user identity mapping between host and container.

Finally, a fundamental architectural property that made real-time container execution possible is that container processes remain *ordinary Linux processes from the kernel's perspective*. This means that the namespace and cgroup boundaries are completely transparent to the EVL core, which can schedule containerized RT threads in its OOB domain exactly as it would native ones. This is also what distinguishes *OCI containers* from hypervisor-based virtualization approaches, where an additional layer between the RT thread and the kernel would destroy any determinism guarantee.

The details of the implementation of Podman containers in the system is provided in Par. 3.2.5.

3.2.6 Role in the Development Workflow

So, in conclusion of this paragraph about the tools and technology used, it is possible to affirm that the combined use of *Yocto/PetaLinux, QEMU, and EVL/libevl* defined the complete development and validation workflow of this thesis.

In particular, *PetaLinux* provided the hardware-specific firmware and tooling infrastructure for the ZynqMP platform.

Then, *QEMU* enabled full system emulation without requiring physical hardware, allowing iterative development and large-scale scalability experiments — up to 64 concurrent containers — that would have been significantly more complex to conduct on real hardware.

At that point, *EVL* introduced the real-time capabilities through its OOB scheduling domain, while *libevl* exposed those capabilities to userspace applications running both natively and inside Podman containers.

Finally, *Podman containers* were enabled through cgroups v2 and namespaces mechanism of Linux kernel, with the EVL devices exposed, in order for the containers to be able to execute a real-time application inside themselves.

It is important to note though, that QEMU's lack of cycle-accurate timing represents a fundamental limitation on the validity of the real-time latency measurements collected: the emulated environment introduced artificial latency floors and jitter that would not be present on real ZCU102 hardware, and all quantitative results in this thesis should be interpreted in this context.

3.3 Understanding the basics of the Software-Stack Implementation

In order to enable the execution of real-time applications inside containers on a platform representative of the on-board computers used by DLR for space missions - such as *ScOSA* (see Par. 1.2.1) and *Stellar Apps* (see Par. 1.1) - many different steps were carried out.

Before detailing the adopted workflow, which will be described throughout this chapter, it is useful to first introduce the various *components* involved.

This project mixes multiple architectural and software domains. In more detail, it combines three different architectural "spaces" (*host environment, target filesystem*, and *QEMU-emulated hardware*), two kernels (`linux-xlnx` (Xilinx vendor fork of the Linux kernel) and the `linux-evl` (*an already EVL-patched Linux kernel provided by Xenomai 4*), two build paradigms (*cross-compilation* and *native compilation*), and three software layers (*kernel, EVL core* and *containers*).

3.3.1 Overview of workflow and achievements

Before delving into the technical details, it is useful to provide an overview of the achieved goals and the main steps undertaken to accomplish them.

Therefore, these are summarized as follows:

1. On the host side (x86_64), an EVL-enabled ARM64 kernel (*linux-evl* with *linux-xlnx* used as a source of the defconfig for the ZynqMP platform) was built, producing a functional bootable *Image*.

2. An emulation of the Xilinx ZCU102 board was set up using the Xilinx/Petalinux QEMU flow, ensuring a QEMU boot stable pipeline through a custom-made shell file *run-qemu-evl.sh*.

3. The pre-built *rootfs.ext4* image provided by Petalinux used to boot QEMU was resized to guarantee enough space for development and containers.

4. The ARM64 userspace - including *libbpf* and *libevl* - for the emulated-board was *cross-compiled*.

5. The resulting userspace was injected into the rootfs.ext4 image to ensure persistency.

6. The modified rootfs.ext4 was mounted inside the QEMU emulation.

7. QEMU was launched using the Xilinx/Petalinux QEMU workflow, executing the run-qemu-evl.sh shell file, replacing the default kernel image with the previously built EVL-enabled Image and pointing to the edited rootfs.ext4.

8. EVL tests - such as *hectic* and *latmus* - were executed on QEMU to test the correct functionality of the EVL kernel.

9. A *Podman container stack* was built as a set of RPM packages using *Yocto/BitBake* on the host x86_64 system.

10. A local RPM repository was created on the host and exposed via HTTP for installation on the QEMU guest.

11. Podman was installed on QEMU using *dnf* from the local repository.

12. A basic *busybox* Podman container was launched on QEMU to test the proper functioning of the container engine.

13. A *Rootful* Podman container based on a Debian image was created with real-time capabilities enabled.

14. The target real-time benchmark application (*Real_Time_Execution_monitor.c*) provided externally was successfully executed on QEMU outside containers.

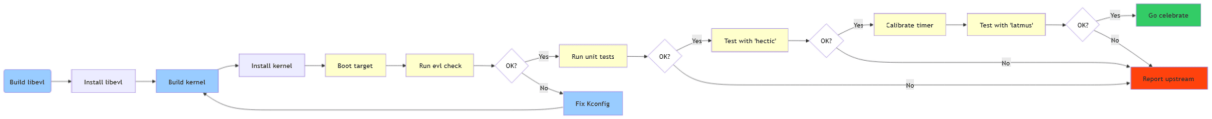


Figure 3.2: Process for getting the EVL core running on a target system [16]

15. The same benchmark application was successfully executed inside the Rootful Debian Container.

16. A *Rootless* Podman container based on a Debian image with real-time capabilities was created.

17. The same application was executed inside the Rootless Debian Container.

18. Concurrent execution of multiple instances of the same Real-Time Application inside different real-time Podman containers in different configurations (Rootful-Rootful, Rootless-Rootless and mixed) was successfully realized.

19. An evaluation of the overhead introduced by containerization was performed, comparing containerized execution of a real-time application with the bare-metal execution.

20. More tests were conducted to evaluate the feasibility of running concurrent real-time Rootless containers on the QEMU-emulated board.

21. The scalability of the system in terms of multiple concurrent real-time containers running was tested resolving into a founded-limit.

22. The realized software stack was implemented on a real ZCU102 board and new measurements were taken.

The first eight steps followed are also pictured in Fig. 3.2 as reported on the official website of *Xenomai 4* [16, 20] for the integration and usage of the EVL core to turn a general-purpose Linux kernel into a real-time capable one.

The final outcome is a bootable QEMU-based ZCU102 system running an EVL-enabled kernel, with *libevl* correctly working and EVL tests successfully passing. A Podman + *crun* container stack was installed inside the guest system via a host-served Yocto-built RPM repository, without using the Petalinux build system. Finally, a real-time benchmark application was correctly executed both inside and outside Podman containers - both Rootful and Rootless - with real-time capabilities on the QEMU-based system.

3.3.2 Architectural spaces involved

Delving into technical details, the software stack designed to enable real-time applications to run inside containers on the embedded board ZCU102 involved multiple different architectural spaces. These domains had to be carefully integrated with each other in order to develop a properly functional and coherent system.

1. The first architectural space was the operating system on which the entire development took place, i.e. the *host environment*. In this thesis, the host was implemented as a Virtual Machine running on VMWare with Ubuntu 20.04 and a general-purpose Linux 5.15 vanilla-kernel. Consequently, the host system was based on the *x86_64* architecture.

2. The second domain was the *target filesystem*, represented by the *ARM64 root filesystem*. This did not correspond to a physical hardware: rather, it constituted the *ARM64 userspace environment* that was meant to run on the target kernel once it had been properly configured. Since this environment targeted the ARM64 architecture, all its components (e.g., binaries, libraries, pkg-config files, etc.) had to be ARM64 as well.

3. The third and final space involved - differently from the second one - was some actual hardware. Indeed, the final goal of the broader project this thesis is part of, was to deploy this same software stack on a real embedded board (ZCU102 or similar). However, at the development stage - which accounts the most of the work presented in this thesis - the target hardware was replaced by an emulated platform. Specifically, a QEMU-based emulation of the very same board was used (i.e., it relied on the *Xilinx-zcu102 machine model*) to conduct experiments with real-time containers and only in the end the system was also deployed on the real board.

3.4 Implementation of the project step-by-step

The implementation process required many steps and, due to some encountered errors, different paths were tried before getting to the successful one. In fact, a more linear path would probably have been possible to carry out the same output (emulating the target board with the EVL kernel for real-time purposes). Hence, here are first reported the exact steps followed, and at the end of the chapter some notes highlight what could have been done differently in order to get to the same output faster.

In particular, the steps followed to emulate the ZCU102 board have been taken from the *official Petalinux UserGuide UG1144* [26].

3.4.1 Host machine preparation

Starting from the host machine (a VMWare Virtual Machine running Ubuntu 20.04 with a Linux general-purpose 5.15 vanilla-kernel), in order to emulate the Xilinx ZCU102 board with a Linux EVL kernel, the *Xilinx/Petalinux QEMU flow* was used, ensuring a QEMU boot stable pipeline through a custom-made shell file *run-qemu-evl.sh*.

To prepare the machine, some dependencies needed to be solved and some *host tools* to be installed, as:

```
1: sudo apt update
2: sudo apt install -y iproute2 gawk python3 build-essential gcc git make net-tools
      netstat libncurses5-dev zlib1g-dev zlib zlib1g:i386 libssl-dev flex bison
      libselinux1 gnupg wget diffstat chrpath socat xterm autoconf libtool tar unzip
      texinfo gcc-multilib libsdl1.2-dev libglib2.0-dev meson gcc-aarch64-linux-gnu
      g++-aarch64-linux-gnu binutils-aarch64-linux-gnu libelf-dev bc
      device-tree-compiler
```

The list above also contains some of the requirements that were subsequently found missing and that were so matched.

Before being installed, *Petalinux* also required to reconfigure the *Dash*, ensuring that `/bin/sh` pointed to `bash`, with the command:

```
1: sudo dpkg-reconfigure dash # Select "No"
```

3.4.2 Petalinux Installation

Then, Petalinux was installed downloading it from the official AMD/Xilinx website [40] and, in more detail, the version 2022.1 was chosen for compatibility reasons with the 5.15 kernel of the host machine which Petalinux need to be run on.

For this thesis, it was installed under the path `/home/carla/petalinux/2022.1` as reported in the following commands:

```
carla@ubuntu:~/zcu102_project/xilinx-zcu102-2022.1$ source /home/carla/petalinux/2022.1/settings.sh
PetaLinux environment set to '/home/carla/petalinux/2022.1'
WARNING: This is not a supported OS
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "UG1144 2022.1 PetaLinux Tools Documentation Referen
ce Guide" for its impact and solution
```

Figure 3.3: Source Petalinux setting to enable the environment to use Petalinux commands

```
1: chmod +x petalinux-v2022.1-*.run
2: ./petalinux-v2022.1-*.run
```

From now on, the ~ will stand for `/home/carla`.

Furthermore, to prevent any possible errors with the Petalinux's build system and the several tools it invokes internally as Python scripts and BitBake, also *setting the locale* was needed:

```
1: sudo locale-gen en_US.UTF-8 && sudo update-locale LANG=en_US.UTF-8
```

3.4.3 Petalinux Project Creation from BSP

At this point, with Petalinux installed and the host machine set, it was possible to actually create a minimal project from which to start emulating the ZCU102 board.

In particular, first the *BSP (Board Support Package)* for the ZCU102 board was downloaded and installed from the official Xilinx website [41], with respect to the same version of Petalinux (2022.1).

It is also important to observe that Petalinux needs his path to be sourced any session, so any time a new terminal was open, it was necessary to run the following command in order to use the Petalinux commands (see also Fig. 3.3):

```
1: source ~/petalinux/2022.1/settings.sh
```

Then, it was finally possible to create the project from the BSP:

```
1: mkdir ~/zcu102_project && cd ~/zcu102_project
2: petalinux-create -t project -s xilinx-zcu102-v2022.1-04191534.bsp -n
       xilinx-zcu102-2022.1
```

```
3: cd xilinx-zcu102-2022.1
```

The creation-project step unpacked the BSP archive and created a structured project directory, namely `xilinx-zcu102-2022.1`. Indeed, it created a path inside the folder with a series of binaries that were later used by QEMU run script to properly emulate the system. The files are also shown in Fig. 3.4.

```
carla@ubuntu:~/zcu102_project/xilinx-zcu102-2022.1/pre-built/linux/images$ ls
bl31.bin                  pxelinux.cfg            system.bit
bl31.elf                  ramdisk.cpio.gz         system.dtb
BOOT.BIN                  ramdisk.cpio.gz.u-boot  system.dtb.bak
bootgen.bif               ramdisk.manifest        u-boot.bin
boot.scr                  ramdisk.tar.gz          u-boot-dtb.bin
config                    rootfs.cpio.gz          u-boot-dtb.elf
Image                     rootfs.cpio.gz.u-boot   u-boot.elf
Image.gz                  rootfs.ext4             vmlinux
image.ub                  rootfs.ext4.BACKUP      zynqmp_fsbl.elf
openamp.dtb               rootfs.ext4.bak         zynqmp-qemu-arm.dtb
petalinux-sdimage.wic.xz  rootfs.jffs2            zynqmp-qemu-multiarch-arm.dtb
pmufw.elf                 rootfs.manifest         zynqmp-qemu-multiarch-pmu.dtb
pmu_rom_qemu_sha3.elf     rootfs.tar.gz
```

Figure 3.4: pre-built files unpacked from the zcu102 BSP file and later used to launch QEMU with the proper configuration

In particular, among the pre-built files that were unpacked and later used there were:

- Firmware binaries loaded by the QEMU run script:
  - `pmu_rom_qemu_sha3.elf`, the PMU ROM loaded as the MicroBlaze QEMU kernel;
  - `pmufw.elf`, the Platform Management Unit firmware;
  - `bl31.elf`, the ARM Trusted Firmware (ATF);
  - `u-boot.elf`, the U-Boot bootloader.
- Boot infrastructure files:
  - `boot.scr`, the U-Boot script that tells U-Boot how to load the kernel, DTB and rootfs;
  - `ramdisk.cpio.gz.u-boot`, the initial ramdisk used during early boot.
- Device tree files:

  - `system.dtb`, the hardware device tree for the ZCU102, later patched to add cgroup v2 bootargs for Podman Containers;
  - `zynqmp-qemu-multiarch-arm.dtb`, the QEMU machine model DTB for the AArch64 process, passed to QEMU via -hw-dtb;
  - `zynqmp-qemu-multiarch-pmu.dtb`, the QEMU machine model DTB for the MicroBlaze PMU process.

- Root filesystem:
  - `rootfs.ext4`, the base root filesystem that was mounted on the host to install *libevl, libbpf, kernel modules and Podman* and that also ensured persistency in the emulated system. It was resized multiple times in the end (before to 4GB, then to 32GB) to host multiple Podman Containers (debian-based images) at the same time on the system.

- QEMU binaries (not inside `pre-built/` but accessible after the PetaLinux installation was sourced via settings.sh):
  - `qemu-system-aarch64`, the ZynqMP-patched AArch64 QEMU;
  - `qemu-system-microblazeel`, the MicroBlaze QEMU for the PMU process.

These *pre-built files* were then used to create a shell script to directly launch the QEMU emulation of the board. Since in the end, the used files were the pre-built ones, it was not necessary to run the command `petalinux-build` (as suggested from the *Petalinux User Guide* [26]), that, anyway, always resulted into some errors. Moreover, the project's `pre-built/linux/images/` directory already contained all the firmware components necessary for QEMU The final shell script obtained at the end of the whole process is reported in Par. 3.4.10.

3.4.4 Building the EVL Kernel (linux-evl) on the host

The main goal of this configuration consisted in getting a Linux kernel image featuring *Dovetail* (*Xenomai 4*) and the *EVL core* on top of it;

The building of the EVL kernel from the source code was a two-step process: first, it required enabling the EVL core from the `linux-evl` kernel sources and integrating it with the configuration for the ZynqMP board, then to enable the user API (userspace) through the library *libevl*, as shown in Fig. 3.5 where the exact toolchain was reported.

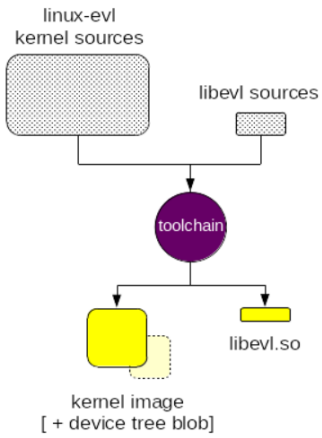


Figure 3.5: Building EVL process [20]

```
carla@ubuntu:~$ git clone --depth 1 -b v6.12.y-evl-rebase https://source.denx.de
/Xenomai/xenomai4/linux-evl.git
Cloning into 'linux-evl'...
remote: Enumerating objects: 92090, done.
remote: Counting objects: 100% (92090/92090), done.
remote: Compressing objects: 100% (88273/88273), done.
remote: Total 92090 (delta 7957), reused 25413 (delta 2836), pack-reused 0 (from
 0)
Receiving objects: 100% (92090/92090), 256.53 MiB | 4.71 MiB/s, done.
Resolving deltas: 100% (7957/7957), done.
Updating files: 100% (86881/86881), done.
carla@ubuntu:~$
```

Figure 3.6: Clone `linux-evl` from Github repository

All the kernel work was done standalone on the host (a VMWare Virtual Machine running Ubuntu 20.04 with a Linux general-purpose 5.15 kernel) and not through Petalinux.

In more detail, a generic Linux kernel, already patched with *Dovetail* and *EVL*, but not configured for any specific board, is available for cloning on the official GitHub repository of *Xenomai 4* [33].

There are different branches with different kernel versions: in this thesis, the latest-LTS kernel version 6.12 was cloned. The cloning was executed as a shallow copy from GitHub (see Fig. 3.6), through the following command:

```
1: cd ~/zcu102_project
2: git clone --depth 1 -b v6.12.y-evl-rebase
       https://source.denx.de/Xenomai/xenomai4/linux-evl.git
```

```
carla@ubuntu:~$ git clone https://github.com/Xilinx/linux-xlnx
Cloning into 'linux-xlnx'...
remote: Enumerating objects: 11756642, done.
remote: Counting objects: 100% (3054/3054), done.
remote: Compressing objects: 100% (392/392), done.
Receiving objects: 100% (11756642/11756642), 3.45 GiB | 5.42 MiB/s, done.
remote: Total 11756642 (delta 2730), reused 2664 (delta 2662), pack-reused 11753
588 (from 4)
Resolving deltas: 100% (9925395/9925395), done.
Updating files: 100% (87144/87144), done.
carla@ubuntu:~$
```

Figure 3.7: Cloning `linux-xlnx` from Github repository

This kernel represented a complete Linux kernel tree already patched with the EVL patch fully integrated and based on the Xenomai-4 real-time core. However, it was not configured for the Xilinx ZCU102 board, hence it was necessary to also clone the AMD/Xilinx's Linux kernel tree specific for the board, already including drivers and SoC options for the said board. The cloning was performed again from a Github repository (see Fig. 3.7):

```
1: cd ~/zcu102_project
2: git clone https://github.com/Xilinx/linux-xlnx
```

The Xilinx's known-good configuration for the board was imported into the folder of `linux-evl`, as follows:

```
1: cp linux-xlnx/arch/arm64/configs/xilinx_defconfig linux-evl/.config
```

This step was critical because such configuration was necessary to build the EVL kernel for the exact ZynqMP-class SoC.

To compile the EVL kernel in order to configure and obtain the image, it was necessary to *Cross-Compile* it, since the compilation happened on the host - with x86_64 type of architecture - and considering that the target architecture was the ARM64. Hence, there was previously a *configuration step*:

```
1: cd linux-evl
2: make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- olddefconfig
3: ARCH=arm64 make menuconfig
```

The *make olddefconfig* ensured that the existing .config (or default base) was updated to match the current (new) kernel tree, while the *make menuconfig* opened a configuration menu of the kernel (see Fig. 3.8).

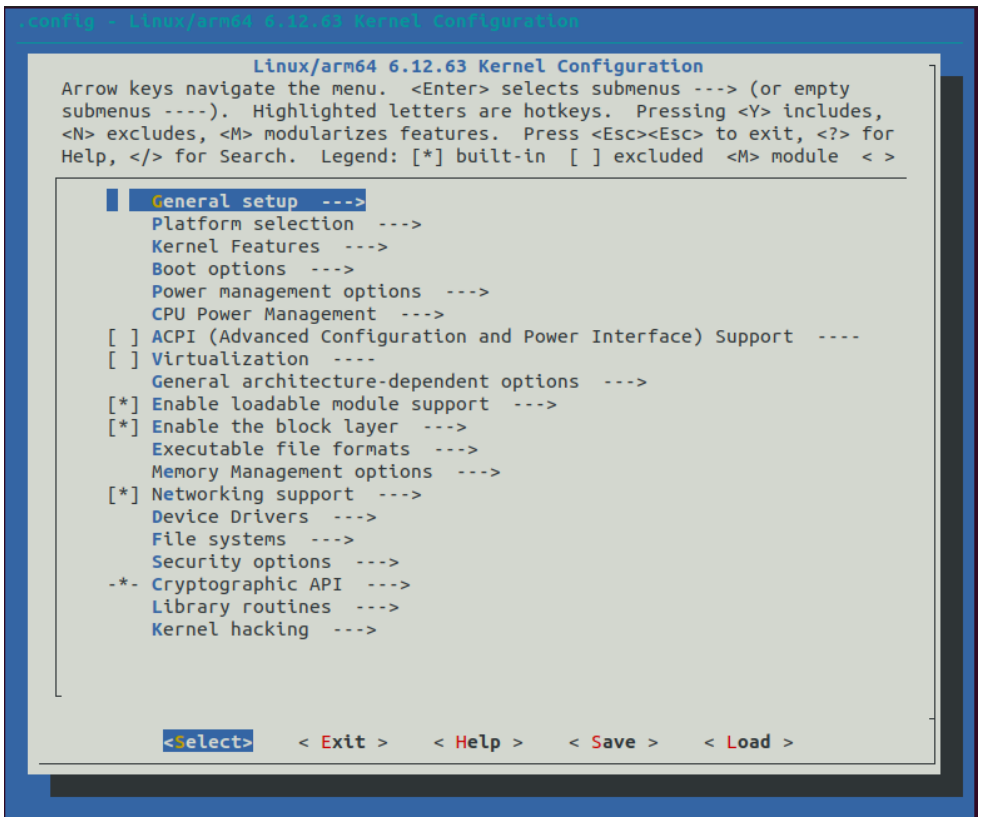


Figure 3.8: General Configuration linux-evl menu

In the *configuration menu*, a variety of options were enabled or disabled to ensure that all the EVL and real-time options would function properly.

The options enabled were:

- General setup —>
    * Local version
    * Kernel .config support
    * Enable access to .config through /proc/config.gz
    * Namespaces support –>
        * UTS namespace
        * IPC namespace

        * PID namespace
        * Network namespace
        * User namespace
    * Control Group support –>
        * Memory controller
        * CPU controller
        * PIDs controller

- Kernel Features —>
    * Dovetail Interface
    * EVL real-time core
        * Enable quota-based scheduling (EVL_SCHED_QUOTA)
        * Enable temporal partitioning policy (EVL_SCHED_TP)
        * Debug support
- Timer Frequency –> 1000 Hz
- CPU Power Management –>
    - CPU Frequency Scaling
        * Default CPUFreq governor –> performance
- File systems –>
    * Overlay filesystem support
- Device Drivers –>
    - Out-of-band device drivers –>
        * OOB context switching validator (EVL_HECTIC)
        * Timer latency calibration and measurement (EVL_LATMUS)
    * Network device support –>
        * Network core driver support
        * Virtual ethernet pair device (veth)

At that point, it was possible to start the *Kernel Compilation* through the commands:

```
carla@ubuntu:~/linux-evl$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- olddefconfig
#
# configuration written to .config
#
carla@ubuntu:~/linux-evl$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j"$(nproc)"
Image dtbs modules
  SYNC    include/config/auto.conf.cmd
  WRAP    arch/arm64/include/generated/uapi/asm/kvm_para.h
  UPD     include/generated/uapi/linux/version.h
  WRAP    arch/arm64/include/generated/uapi/asm/errno.h
  WRAP    arch/arm64/include/generated/uapi/asm/ioctl.h
  WRAP    arch/arm64/include/generated/uapi/asm/ioctls.h
  WRAP    arch/arm64/include/generated/uapi/asm/ipcbuf.h
  WRAP    arch/arm64/include/generated/uapi/asm/msgbuf.h
  UPD     include/generated/compile.h
  WRAP    arch/arm64/include/generated/uapi/asm/poll.h
  WRAP    arch/arm64/include/generated/uapi/asm/resource.h
  HOSTCC  scripts/dtc/dtc.o
  WRAP    arch/arm64/include/generated/uapi/asm/sembuf.h
  HOSTCC  scripts/dtc/flattree.o
  WRAP    arch/arm64/include/generated/uapi/asm/shmbuf.h
  WRAP    arch/arm64/include/generated/uapi/asm/siginfo.h
  WRAP    arch/arm64/include/generated/uapi/asm/socket.h
  WRAP    arch/arm64/include/generated/uapi/asm/sockios.h
```

Figure 3.9: EVL Kernel Compilation with Real-Time options enabled

```
1: make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j"$(nproc)" Image dtbs modules
```

This way, the kernel image was built with the new options enabled and keeping the ones not modified as default (as specified in the *old configuration* of the previous command (see Fig. 3.9).

This took about 20 minutes to run.

This process created the `linux-evl/arch/arm64/boot/Image` of the EVL Kernel that was later loaded by the `run-qemu-evl.sh` script to launch the QEMU-emulated system.

It was then also necessary to generate the *kernel headers* for user-space builds:

```
1: make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- headers_install \
2: STALL_HDR_PATH=$PWD/_headers
```

3.4.5 Building `libbpf` for ARM64 (dependency for `libevl`)

In order to properly configure the userspace library that provides the API for interacting with the EVL core, namely libevl, it was required to first resolve an additional dependency represented by the EVL framework: compilation and configuration of *libbpf*.

libbpf is the official user-space library maintained within the Linux kernel ecosystem that provides the necessary abstractions to load and manage eBPF (extended Berkeley Packet Filter)

programs and maps [42]. In particular, EVL's *latency measurement* and *tracing tools* - `latmus` and `hectic` - use eBPF mechanisms to instrument kernel behavior. So, this step was necessary to enable the correct loading and execution of EVL-related BPF components, thereby allowing libevl to provide deterministic real-time services.

The problem here was represented by the fact that the compilation of said library was not elaborated inside the QEMU-emulated board, i.e. on an ARM64 architecture, but on the host, where the Ubuntu host's system libbpf was x86_64-only. Hence, this step did not only require a cross-compilation as in the case of linux-evl, but it demand to build an ARM64 version from scratch and to install it in the *custom sysroot*, so that when libevl's build system would have looked for it, it would have found the correct architecture.

So, first of all it was necessary to enable the multi-architecture for ARM64 target, to restrict the main repositories to *amd64* to avoid conflicts with the architecture of the host and, finally, to add the ARM64 ports to the list of allowed-ports (`/etc/apt/sources.list.d/arm64-ports.list`):

```
1: sudo dpkg --add-architecture arm64
2: sudo sed -i 's|^deb http://us\.archive\.ubuntu\.com|deb [arch=amd64]
      http://us.archive.ubuntu.com|g' /etc/apt/sources.list
3: sudo apt update
4: sudo apt install -y libelf-dev:arm64 zlib1g-dev:arm64 linux-libc-dev:arm64
```

At that point, it was possible to clone `libbpf` from the repository Github and to cross-compile it:

```
1: cd ~/zcu102_project
2: git clone https://github.com/libbpf/libbpf.git
3: PREFIX=$HOME/zcu102_project/_sysroot-aarch64/usr
4: make -C libbpf/src -j$(nproc) CC=aarch64-linux-gnu-gcc AR=aarch64-linux-gnu-ar
5: make -C libbpf/src CC=aarch64-linux-gnu-gcc AR=aarch64-linux-gnu-ar PREFIX=$PREFIX
      install
```

3.4.6 Cross-building *libevl* for ARM64

libevl is the userspace library that provides the API for interacting with the EVL core. Through libevl, applications can create and manage real-time threads, handle synchronization primitives, and perform time-sensitive operations with deterministic guarantees. In particular, while the

EVL core lives in kernel space and manages real-time scheduling, interrupt handling, and domain switching, applications running in userspace cannot talk to it directly, hence they need libevl as the bridge. For this reason, libevl exposes the EVL API, ensuring that threads executing through it are scheduled by the OOB stage rather than Linux's CFS.

In more detail, libevl uses *Meson* as its build system. Meson is an open-source build system meant to be fast, user-friendly and good for cross-compiling [43]. The only *problem* was represented by the fact that it needs to know,unambiguously, which compiler, linker, and `pkg-config` to use for the target architecture. Since on the host there was an x86_64 machine with ARM64 cross-compilation tools installed alongside native x86_64 tools, Meson needed to be told explicitly to use the ARM64 toolchain and not the native one. In practice, getting that instruction correct, with all its dependencies resolved, took many iterations and failures.

First, it was necessary to install Meson and Ninja (a build system related to execute the `build.ninja` file generated from Meson):

```
1: sudo apt update
2: sudo apt install -y meson ninja-build pkg-config libbsd-dev libreadline-dev
```

Then, `libevl` was cloned from the Github repository:

```
1: cd ~/zcu102_project
2: git clone https://gitlab.com/Xenomai/xenomai4/libevl.git
3: cd libevl
```

Then, in order to assure that Meson could also find the correct path to `libbpf` (necessary to cross-compile libevl) and considering that Meson uses *pkg-config* (a small utility that looks up where libraries are installed by reading -pc files), it was necessary to explicitly tell to pkg-config to use the previously cross-compiled libbpf, instead the default one. Hence, since the cross-compiled libbpf was present into a custom sysroot at `~/zcu102_project/_sysroot-aarch64/usr`, it was necessary to redirect the pkg-config through the following commands:

```
1: export SYSROOT_A64=$HOME/zcu102_project/_sysroot-aarch64/usr
2: export PKG_CONFIG_PATH=$SYSROOT_A64/lib/pkgconfig:$SYSROOT_A64/lib64/pkgconfig
3: KHEAD=$HOME/zcu102_project/linux-evl/_headers/include
```

This way, the pkg-config looked in the custom ARM64 sysroot instead of the system path to

resolve the dependency represented by libbpf. Furthermore, as reported from the commands above, libevl also needed access to the EVL kernel headers — the *UAPI (User API) headers* that defined the data structures and constants used to communicate between userspace and the EVL core. These were not standard Linux headers but specific to the linux-evl tree built. So, the flag `-Duapi=$HOME/zcu102_project/linux-evl` told Meson's build scripts exactly where to find them. This is also why building linux-evl first and running headers_install was not optional: indeed, without those headers, libevl's Meson configuration would have failed immediately because the EVL-specific type definitions would have been missing.

Still, since Meson had to perform a cross-compilation, it was necessary to also specify - through a *Meson cross file* - which binaries to use for the compilation and to describe the target machine in order for it to know that it was not building for the host. For this reason, the cross-file `cross-arm64.txt` was created in the following way:

```
cat > cross-arm64.txt <<'EOF'
[binaries]
c = 'aarch64-linux-gnu-gcc'
cpp = 'aarch64-linux-gnu-g++'
ar = 'aarch64-linux-gnu-ar'
strip = 'aarch64-linux-gnu-strip'
pkg-config = 'pkg-config'

[host_machine]
system = 'linux'
cpu_family = 'aarch64'
cpu = 'aarch64'
endian = 'little'
EOF
```

Finally, it was possible to configure and build libevl through the following commands:

```
rm -rf build
meson setup build --cross-file cross-arm64.txt \
    -Duapi=$HOME/zcu102_project/linux-evl \
    --prefix=/usr \
    -Dc_args="-I$KHEAD -I$SYSROOT_A64/include"
ninja -C build
```

However, as specified in the previous commands, even with pkg-config pointing to the right *sysroot* and the *UAPI path* set correctly, the compiler itself still needed to be told where to find the header files at compile time. So, the flag `-Dc_args="-I$KHEAD -I$SYSROOT_A64/include"` passed two extra include directories directly to `aarch64-linux-gnu-gcc`:

- $KHEAD pointed to the installed EVL kernel headers (linux-evl/_headers/include)
- $SYSROOT_A64/include pointed to the custom sysroot's include directory, where the libbpf headers lived

3.4.7 Installing `libevl` and `libbpf` into the *rootfs.ext4*

In order to properly emulate the ZCU102 board through QEMU, it was necessary to point a root filesystem as a sort of virtual hard drive stored as a file. In particular, the file mounted by QEMU was the `rootfs.ext4`, exactly as a real board would mount a physical SD card. Furthermore, this file is different from the *ramdisk* `rootfs.cpio.gz.u-boot` since the ramdisk lives entirely in RAM and it is destroyed on reboot, while the `rootfs.ext4` from `images/linux` (built by Petalinux and not the pre-built one) was chosen for *persistency reasons* and because it was resizable. Indeed, in order to host all the Debian images for containers, it was necessary to resize it since the rootfs.ext4 generated by Petalinux's Yocto build was minimal by design and just large enough to hold a basic embedded Linux system, without enough room for libevl, libbpf, kernel modules, kernel headers and Podman.

The resize was realized on the host. The original dimension was `/dev/mmcblk0 320 MB` and, afterward, it became 4 GB (it needed to be a power of 2 to be accepted by QEMU without crashing).

```
1: cd ~/zcu102_project/xilinx-zcu102-2022.1/images/linux
2: cp rootfs.ext4 rootfs.ext4.bak
3: truncate -s 4G rootfs.ext4
4: e2fsck -f rootfs.ext4
5: resize2fs rootfs.ext4
```

In reality, in the last part of the project, to test the scalability of the system, some tests with a number of containers up to 100 were created and, in order to host all of them, it was necessary to resize the filesystem again. In more detail, the resize was performed in the same way and the final size was 32 GB.

Subsequently, to install files into rootfs.ext4 from the host without booting QEMU, a *loop device* was used. Specifically, a loop device is a Linux kernel mechanism that allows a regular file to be treated as a block device — exactly as if it were a physical disk.

```
1: sudo modprobe loop
2: sudo mkdir -p /tmp/zcu102-rootfs
3: LOOPDEV=$(sudo losetup -Pf --show rootfs.ext4)
4: sudo mount $LOOPDEV /tmp/zcu102-rootfs
```

In more detail:

- `modprobe loop` ensured the loop device kernel module was loaded;
- `losetup -Pf -show rootfs.ext4` found the next available loop device, associated the image file with it and printed the device name. The -P flag told it to also scan for partitions inside the image;
- `mount $LOOPDEV /tmp/zcu102-rootfs` mounted that block device at a temporary directory, making the filesystem accessible as a normal directory tree on the host.

After this, the `/tmp/zcu102-rootfs` looked and behaved exactly like the root filesystem of the emulated board. This means that writing a file to /tmp/zcu102-rootfs/usr/lib/ had the same effect to placing that file at `/usr/lib/` on the running QEMU system.

At that point, `libevl` could finally be properly installed as:

```
1: sudo env "PATH=$HOME/.local/bin:$PATH" \
2:     DESTDIR=/tmp/zcu102-rootfs \
3:     ninja -C ~/zcu102_project/libevl/build install
```

Such commands not only installed libevl on the rootfs.ext4 but they also told *Ninja* to redirect all output paths to be relative to `/tmp/zcu102-rootfs` instead of the real system root. So, when the install script tried to put `libevl.so` at `/usr/lib/libevl.so`, it actually landed at `/tmp/zcu102-rootfs/usr/lib/libevl.so`, i.e. the correct location inside the image. The PATH manipulation also ensured Ninja found the correct Meson version from the local install rather than any system-wide one.

To make libevl run properly, obviously also `libbpf` had to be installed on the rootfs.ext4:

```
1: SYSROOT_A64=$HOME/zcu102_project/_sysroot-aarch64/usr
```

```
2: sudo cp -av $SYSROOT_A64/lib64/libbpf.so* /tmp/zcu102-rootfs/usr/lib/
```

To be able to use tools like *evl-hectic* and *evl-latmus*, also Linux kernel modules were needed in the rootfs.ext4 and, in particular, it was critical that these modules came from the same kernel build as the *Image* file that QEMU booted, because a module built against a different kernel version would have been rejected with a version mismatch error:

```
1: sudo make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- \
2:     INSTALL_MOD_PATH=/tmp/zcu102-rootfs modules_install
```

Then, the entire `linux-evl` source tree (minus git history, documentation, object files and built modules) had to be copied inside the rootfs at `/usr/src/linux-evl/`. This was necessary to enable *native builds inside QEMU*, necessary to compile the real-time application used for test purposes directly inside the emulated system, correctly accessing to the EVL kernel headers:

```
1: sudo rsync -a --delete \
2:     --exclude '.git' --exclude 'Documentation' --exclude 'tools' \
3:     --exclude '*.o' --exclude '*.ko' \
4:     ~/zcu102_project/linux-evl/ \
5:     /tmp/zcu102-rootfs/usr/src/linux-evl/
```

Finally, a synchronization command was necessary to force all pending writes to be flushed from the kernel's page cache to the actual image file. This was mandatory to avoid a possible corrupted filesystem. Then the the filesystem could be detached cleanly, updating all metadata, releasing the loop device and dissociating it from the image file, as shown below:

```
1: sync
2: sudo umount /tmp/zcu102-rootfs
3: sudo losetup -d "$LOOPDEV"
```

After these commands the rootfs.ext4 file on the host contained the complete modified filesystem, ready to be handed to QEMU.

3.4.8 Patching system.dtb for cgroup v2 (for Podman Containers)

The last step to emulate the target system was to enable Podman Container with real-time capabilities (they had to be able to exlploit EVL devices for that), ultimate goal of the thesis. At the base of containers working in Linux, there is the mechanism of cgroups (*control groups*), a Linux kernel feature that allows processes to be organized into hierarchical groups, each with controlled access to resources like CPU time, memory, and I/O.

Cgroups exist in two incompatible versions. In particular, *cgroup v1* represents the original design, where each resource controller (CPU, memory, PIDs, etc.) has its own independent hierarchy. Instead, *cgroup v2* represents the modern unified design, introduced in Linux 4.5, where all controllers share a single unified hierarchy.

Then, in this case, the problem was that Podman, as most of modern container runtimes, prefer cgroup v2 because it has a cleaner, more consistent interface and better support for rootless containers and delegation to unprivileged users. In general, the kernel can operate in three modes: *pure v1, pure v2*, or *hybrid mode* where both coexist.

The stock *PetaLinux DTB* booted the emulated system into hybrid mode, but Podman explicitly refused to support it. Indeed, an error raised and in the end it was necessary to completely disable v1 and create a pure-v2 system.

Since the *cgroup mode* was controlled by *kernel boot arguments* (bootargs) passed to the kernel at boot time by the bootloader, it was necessary to edit the *Device Tree Blob* (DTB) directly. Indeed, on an ARM embedded system booting through *U-Boot*, these bootargs are stored inside the DTB. In more detail, two were the arguments needed:

- `systemd.unified_cgroup_hierarchy=1`, that told systemd to mount only the unified cgroup v2 hierarchy
- `cgroup_no_v1=all`, which told the kernel not to mount any cgroup v1 controllers at all

This way, a pure cgroup v2 mode was assured.

Since the DTB was a binary file and not human-readable, the `dtc` (*Device Tree Compiler* - a tool that can compile a human-readable DTS (*Device Tree Source*) file into a binary DTB, and viceversa, had to be used.

```
1: cd ~/zcu102_project/xilinx-zcu102-2022.1/pre-built/linux/images
2: cp -a system.dtb system.dtb.bak
3: dtc -I dtb -O dts -o /tmp/system.dts system.dtb
```

The result was a text file (`/tmp/system.dts`) that could be open in any editor in order to be modified. Precisely, the fix was realized as a *Regex substitution* through *Perl* as follows:

```
1: perl -pi -e 's/(bootargs\s*=\s*"\s*[^"]*?)";/$1 systemd.unified_cgroup_hierarchy=1
      cgroup_no_v1=all";/' /tmp/system.dts
```

Finally, the DTS could be recompiled to a DTB:

```
1: dtc -I dts -O dtb -o system.dtb /tmp/system.dts
```

Hence, in the end, the result was a binary `system.dtb` functionally identical to the original except that its bootargs now included the cgroup v2 flags. This file was what the `run-qemu-evl.sh` script passed to QEMU at boot as explained in Par. 3.4.10.

3.4.9 Installing Podman Containers

By this point, a working EVL kernel and a rootfs containing libevl were available. So, the next requirement was Podman, the container runtime needed to run and orchestrate containers on the emulated board. To accomplish this step, the straightforward approach could have been one of the following:

- `apt install podman`, but it was impossible because PetaLinux's rootfs was not a Debian system and had no apt;
- `opkg install podman`, this also was not possible because PetaLinux's opkg package manager had no feeds configured that included Podman;
- `petalinux-build with Podman added to the image` but this was also impossible, because petalinux-build always crashed with the DTG xilinx-zcu102.dtsi error before it could produce anything.

Hence, this left as only viable path using *BitBake* directly.

BitBake represents the underlying build engine that PetaLinux itself is built on top of and it allowed to compile Podman and its dependencies as ARM64 packages, bypassing the broken parts of PetaLinux entirely. It reads recipe files, resolves dependencies between them and executes build steps. Since the `petalinux-build` command is essentially a wrapper around `bitbake petalinux-image-minimal` that sources the Yocto environment and calls BitBake,

only adding some Xilinx-specific setup, then, this meant that the failing of petalinux-build due to a broken recipe, could be solved sourcing the *Yocto environment* and *calling BitBake* directly, targeting only the specific packages needed and completely bypassing the recipe that was causing the failure.

In more detail, the reason why `petalinux-build` always failed was the DTG recipe, responsible for generating the device tree source files for the board. The problem was caused by the missing file `xilinx-zcu102.dtsi` from the 2022.1 PetaLinux installation. Now, because virtually everything in PetaLinux's build depends on the device tree being generated first, this single missing file caused a cascade failure that prevented BitBake from building anything. However, having already a perfectly good `system.dtb` file — the one from PetaLinux's pre-built images, already patched for cgroup v2, the solution consisted into creating a fake recipe that told BitBake to use the provided pre-built DTB instead of generating another one running DTG:

```
1: mkdir -p project-spec/meta-user/recipes-bsp/dtb-prebuilt/files
2: cp pre-built/linux/images/system.dtb \
3:     project-spec/meta-user/recipes-bsp/dtb-prebuilt/files/
```

This way, a *Yocto recipe file* `dtb-prebuilt.bb` was created and it declared itself as providing virtual/dtb, namely the abstract target that other recipes depend on when they need a device tree. Then BitBake was told to prefer this fake recipe over the broken DTG:

```
1: echo 'PREFERRED_PROVIDER_virtual/dtb = "dtb-prebuilt"' >> build/conf/local.conf
```

Still, to make BitBake fetching reliable during the build from the Git repositories, another command line was added:

```
1: echo 'FETCHCMD_git = "/usr/bin/git -c core.fsyncobjectfiles=0 -c
       gc.autoDetach=false"' >> build/conf/local.conf
```

Then, the Yocto environment had to be sourced directly (as it also happened with Petalinux in Par. 3.4.3):

```
1: source components/yocto/layers/core/oe-init-build-env build
```

After this command, `bitbake` was available in the shell and finally, it was possible to build the

packages needed for Podman:

```
1: bitbake podman crun conmon slirp4netns fuse-overlayfs
```

In reality, not only podman but five specific packages for the ARM64 target were needed. This was due to the fact that each one had a specific role:

- podman is the main container runtime and CLI tool that manages container images, creates and runs containers and handles the orchestration logic. Unlike Docker, Podman is daemonless — it doesn't require a background service to be running.
- `crun` is the OCI container runtime that does the low-level work of creating Linux namespaces, setting up cgroups, and executing the container process. Specifically, Podman calls crun to do the actual container creation. Furthermore, tt was preferred over runc because it is written in C (making it lighter and faster) and has better support for real-time workloads and cgroup v2.
- `conmon` is the container monitor, i.e. a small process that sits between Podman and crun and monitors the lifecycle of each container. It is the one in charge to manage the communication channel between Podman and the running container.
- `slirp4netns` provides user-space networking for containers. It implements a virtual *TCP/IP* stack that allows containers to have network access without requiring root privileges or kernel-level network namespace configuration.
- `fuse-overlayfs` implements the overlay filesystem in user space using FUSE (*Filesystem in Userspace*). It was needed because the kernel's native overlayfs requires certain privileges that may not be available in all configurations, so fuse-overlayfs provides a compatible alternative.

So, BitBake resolved all the dependencies of these packages, downloaded their source code, cross-compiled them for ARM64, and packaged them as *RPM files* in its output directory. The problem was that such RPM packages were on the host machine. To properly use them inside the QEMU-emulated system then, it was necessary to install them through a two-step process.

First, a *local RPM repository* was created on the host through the `createrepo-c-native` BitBake-built tool that generated a self-contained repository metadata named `repodata/`. Then, a simple HTTP server (like *Python*'s `http.server` module) was started on the host to serve the RPM directory as shown in Fig. 3.10. At that point, inside the QEMU session, `dnf` was

```
carla@ubuntu:~/zcu102_project/xilinx-zcu102-2022.1/build/tmp/deploy/rpm$ ls -lah /tmp/podman-repo/repodata | head
total 11M
drwxrwxr-x  2 carla carla 4.0K Jan 11 13:23 .
drwxr-xr-x 11 carla carla 4.0K Jan 11 13:23 ..
-rw-rw-r--  1 carla carla 3.3M Jan 11 13:23 2a904d78c8b3aa9edd1315b07a475aa3d7d23d3e236cae541abd82a879ed9a6a-filelists.sqlite.bz2
-rw-rw-r--  1 carla carla 2.9M Jan 11 13:23 78cf88720e2ad027425b0431754c3326e0c4d1e04ec186b0ca92b58d3c4a2abe-primary.sqlite.bz2
-rw-rw-r--  1 carla carla 1.4M Jan 11 13:23 93b0c73736290842420675576 64c636363dad23236e7a1ab86ac38df1e2d48bd-primary.xml.gz
-rw-rw-r--  1 carla carla 561K Jan 11 13:23 a273cec9af34acb35afd5130c2762e7f8bcc749fcf2381adb314fc4723534d6a-other.xml.gz
-rw-rw-r--  1 carla carla 852K Jan 11 13:23 cd69b44c1ae3dadfc2b93a3594decd18fdc5ed8e8e7e1f7c37081b672c0a3eab-other.sqlite.bz2
-rw-rw-r--  1 carla carla 2.2M Jan 11 13:23 dea163ffb4bb5d804b0ddaa270ea2a4f84f8bc90e7c704c9c18c259d0cf7a0d1-filelists.xml.gz
-rw-rw-r--  1 carla carla 3.1K Jan 11 13:23 repomd.xml
carla@ubuntu:~/zcu102_project/xilinx-zcu102-2022.1/build/tmp/deploy/rpm$ cd /tmp/podman-repo/
carla@ubuntu:/tmp/podman-repo$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
127.0.0.1 - - [11/Jan/2026 13:37:42] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [11/Jan/2026 13:40:51] "GET /repodata/repomd.xml HTTP/1.1" 200 -
127.0.0.1 - - [11/Jan/2026 13:40:52] "GET /repodata/93b0c73736290842420675576 64c636363dad23236e7a1ab86ac38df1e2d48bd-primary.xml.
gz HTTP/1.1" 200 -
127.0.0.1 - - [11/Jan/2026 13:40:52] "GET /repodata/dea163ffb4bb5d804b0ddaa270ea2a4f84f8bc90e7c704c9c18c259d0cf7a0d1-filelists.xm
l.gz HTTP/1.1" 200 -
127.0.0.1 - - [11/Jan/2026 13:45:36] "GET /cortexa72_cortexa53/cni-v0.8.0%2bgitb5ab16f010e822936eb974690ecec38ba69afc01-r0.0.cort
exa72_cortexa53.rpm HTTP/1.1" 200 -
127.0.0.1 - - [11/Jan/2026 13:45:37] "GET /cortexa72_cortexa53/conmon-2.0.29%2bgit0%2b1ef246896b-r0.0.cortexa72_cortexa53.rpm HTT
P/1.1" 200 -
```

Figure 3.10: Serving RPM repository through Python's HTTP server from host machine to QEMU-emulated board

configured to use this HTTP server as a repository source and, finally, it was possible to actually install on the system Podman:

```
1: dnf install podman crun conmon slirp4netns fuse-overlayfs
```

This way, `dnf` fetched the packages from the host over the virtual network interface that QEMU provided between the host and the guest and installed them into the running rootfs. Because QEMU's network connects the guest to the host by default, the guest could reach an HTTP server on the host without any special network configuration.

Truthfully, this entire phase was a workaround for what could have been a single command both in the case of a Debian-based system (`apt install podman`) or of a working `petaLinux-build` command.

3.4.10 Booting QEMU through the `run-qemu-evl.sh` script

The culmination of every decision made across all the previous phases was in the end reflected into the shell script `run-qemu-evl.sh` to boot QEMU easily with all the dependencies solved and making sure to link and mount exactly all the files needed at the correct paths.

The script, saved as `/home/carla/zcu102_project/xilinx-zcu102-2022.1/run-qemu-evl.sh`, required a normal shell execution, as also shown in Fig. 3.11.

The detail of it is reported below:

```
#!/usr/bin/env bash
set -euo pipefail

PROJ="$(cd "$(dirname "$0")" && pwd)"

# Use PetaLinux QEMU binaries (NOT /usr/bin)
QEMU_PMU="/home/carla/petalinux/2022.1/components/yocto/buildtools \
/sysroots/x86_64-petalinux-linux/usr/bin/qemu-system-microblazeel"
QEMU_AARCH64="/home/carla/petalinux/2022.1/components/yocto/buildtools \
/sysroots/x86_64-petalinux-linux/usr/bin/qemu-system-aarch64"

# Create a stable machine-path directory
MPATH="$(mktemp -d /tmp/zcu102-qemu.XXXXXX)"
mkdir -p "$MPATH"

# Start PMU QEMU (microblaze)
"$QEMU_PMU" -M microblaze-fdt \
  -display none \
  -serial null -serial null \
  -monitor none \
  -kernel "$PROJ/pre-built/linux/images/pmu_rom_qemu_sha3.elf" \
  -device loader,file="$PROJ/pre-built/linux/images/pmufw.elf" \
  -hw-dtb "$PROJ/pre-built/linux/images/zynqmp-qemu-multiarch-pmu.dtb" \
  -machine-path "$MPATH" \
  -device loader,addr=0xfd1a0074,data=0x1011003,data-len=4 \
  -device loader,addr=0xfd1a007C,data=0x1010f03,data-len=4 \
  >"$MPATH/pmu.log" 2>&1 &

sleep 1

# Start AArch64 QEMU (EVL kernel swap happens here)
"$QEMU_AARCH64" -M arm-generic-fdt \
  -nographic \
  -serial stdio \
  -monitor none \
  -device loader,file="$PROJ/pre-built/linux/images/bl31.elf",cpu-num=0 \
  -device loader,file="$PROJ/pre-built/linux/images/ramdisk.cpio.gz.u-boot",
  addr=0x04000000,force-raw=on \
  -device loader,file="$PROJ/pre-built/linux/images/u-boot.elf" \
  -device loader,file="/home/carla/zcu102_project/linux-evl/arch/arm64/
  boot/Image",addr=0x00200000,force-raw=on \
  -device loader,file="$PROJ/pre-built/linux/images/system.dtb",
```

```
43:   addr=0x00100000,force-raw=on \
44:   -device loader,file="$PROJ/pre-built/linux/images/boot.scr",addr=0x20000000,
45:   force-raw=on \
46:   -net nic -net nic -net nic -net nic,netdev=eth0 \
47:   -netdev user,id=eth0,tftp=/tftpboot \
48:   -hw-dtb "$PROJ/pre-built/linux/images/zynqmp-qemu-multiarch-arm.dtb" \
49:   -machine-path "$MPATH" \
50:   -global xlnx,zynqmp-boot.cpu-num=0 \
51:   -global xlnx,zynqmp-boot.use-pmufw=true \
52:   -drive if=sd,format=raw,index=1,file="$PROJ/images/linux/rootfs.ext4" \
53:   -m 4G
```

```
carla@ubuntu:~/zcu102_project/xilinx-zcu102-2022.1$ ./run-qemu-evl.sh
qemu-system-aarch64: warning: hub 0 is not connected to host network
PMU Firmware 2022.1     Apr 11 2022   09:29:50
PMU_ROM Version: xpbr-v8.1.0-0
NOTICE:  BL31: v2.6(release):v1.1-9207-g67ca59c67
NOTICE:  BL31: Built : 03:46:40, Mar 24 2022


U-Boot 2022.01 (Apr 04 2022 - 07:53:54 +0000)

CPU:   ZynqMP
Silicon: v3
Model: ZynqMP ZCU102 Rev1.0
Board: Xilinx ZynqMP
DRAM:  4 GiB
PMUFW:  v1.1
EL Level:       EL2
Chip ID:        unknown
NAND:  0 MiB
MMC:   mmc@ff170000: 0
Loading Environment from nowhere... OK
In:    serial
Out:   serial
Err:   serial
Bootmode: JTAG_MODE
Reset reason:
Net:
ZYNQ GEM: ff0e0000, mdio bus ff0e0000, phyaddr 12, interface rgmii-id
```

Figure 3.11: Booting QEMU to emulate zcu102 board with EVL Kernel and resized rootfs.ext4

In particular, it is possible to observe from the script that it launched *two separated QEMU processes* and not just one: the *PMU process* and the *AArch64 process*. This was due to the architecture of the target hardwar (ZynqMP UltraScale+ MPSoC) that has two distinct processing subsystems: the *Application Processing Unit* (APU), which is the the *quad-core ARM*

Cortex-A53 cluster where Linux runs, and the *Platform Management Unit* (PMU), which is a separate *MicroBlaze microcontroller*, responsible for power management, clock control and hardware initialization. Therefore, even if on real hardware these two subsystems communicate over dedicated internal buses, in QEMU, they are emulated as two separate processes that communicate through a shared machine path represented by a temporary directory that acts as an inter-process communication channel: the `/tmp/zcu102-qemu.XXXXXX` passed as MPATH in the script. Furthermore, the PMU process was started first, given a one-second head start (sleep 1), and ran in the background while the AArch64 process started. This mirrored the real boot sequence where the PMU firmware initializes the hardware before the APU is released from reset. Still, it is also possible to observe from the script that the QEMU binaries passed as *QEMU_PMU* and *QEMU_AARCH64* were not the standard Ubuntu system QEMU binaries, but Xilinx-specific patches provided by Petalinux that implemented the ZynqMP machine model, the PMU subsystem emulation, the inter-process machine path protocol and the `xlnx,zynqmp-boot` device behavior. The standard `/usr/bin/qemu-system-aarch64` from Ubuntu could have never been used because it knew nothing about any of these, hence it would either have failed to boot or have produced a non-functional emulation.

Then, the `system.dtb` file patched in Par. 3.4.8 to add cgroup v2 boot arguments was loaded as a device to be consumed by the Linux kernel. In particular, it described the *ZynqMP SoC peripherals* to allow Linux to read at boot the correct existing hardware.

In reality, also another DTB file was passed to the script: the `zynqmp-qemu-multiarch-arm.dtb` passed via `hw-dtb`. This step was necessary because this DTB was consumed by QEMU itself and not by Linux. In fact, it described the virtual machine topology for QEMU to construct the emulated platform, specified by the flags `-M arm-generic-fdt` (for the *generic ARM machine whose topology was described by an FDT*) and `zynqmp-qemu-multiarch-arm.dtb` (the specific FDT). The union of these two-DTB architecture was what truly permitted to emulate the specific ZCU102 board and not a generic ARM platform.

Subsequently, the complete *boot sequence* was encoded in the script, with AArch64 QEMU process that loaded several files at specific memory addresses, respecting the order in the ZynqMP boot flow. Such files were loaded as *device loader* and they were:

- `bl31.elf`, the ARM Trusted Firmware (ATF) BL31;
- `ramdisk.cpio.gz.u-boot`, the initial ramdisk, namely the minimal RAM-based filesystem that U-Boot uses during early boot;
- `u-boot.elf`, the bootloader;

- `.../linux-evl/arch/arm64/boot/Image`, the EVL kernel swap that made the entire EVL integration possible. It was the customized image provided by the steps followed in Par. 3.4.4. It is important to specify that this was a *boot-time swap* and not a *Yocto rebuild*;

- `boot.scr`, the U-Boot boot script that told exactly to U-Boot where the kernel and DTB are and which argument to use;

- `$PROJ/images/linux/rootfs.ext4`, the modified rootfs from `images/linux` rather than the pre-built one from `pre-built/linux/images/`. This modification was what made all the system persistent.

Moreover, four virtual network interfaces were instantiated, mirroring the ZCU102's multiple Ethernet ports and 4 GB of RAM was allocated to the emulated system.

Finally, it should be also noted that the flags `-global xlnx,zynqmp-boot.cpu-num=0` and `-global xlnx,zynqmp-boot.use-pmufw=true` configured the Xilinx-specific ZynqMP boot device emulated by PetaLinux's QEMU. Indeed, `cpu-num=0` told the boot device to release CPU 0 from reset first (the primary core), while `use-pmufw=true` told it to coordinate with the PMU firmware process running in the MicroBlaze QEMU instance through the shared machine path. Without this flag the AArch64 QEMU process would have not synchronized with the PMU process and the boot would have hanged waiting for platform initialization that would have never been arrived.

After the booting was completed, the emulation of the ZCU102 board appeared as in Fig. 3.12.

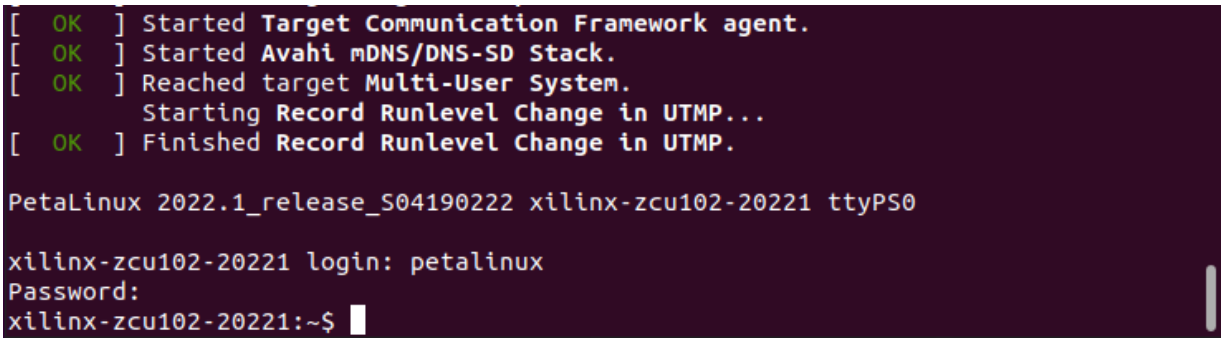


Figure 3.12: Boot of emulated ZCU102 board through QEMU with EVL Kernel integrated

3.5 Testing the feasibility of running real-time Podman containers

When the QEMU-emulated system was finally complete and running, it was possible to move on with the second part of the project: testing the feasibility of such system to execute Podman containers with *real-time capabilities*, hence exploiting the *EVL devices* exposed by the EVL core integrated in the Linux kernel.

In particular, to do that, three were the main questions addressed:

1. *Feasibility*: is it possible to run real-time applications inside *Rootful* and *Rootless* Containers on the platform? (see Par. 3.5.1)

2. *Concurrency*: is it possible to run *multiple concurrent Rootless Containers* side by side? (see Par. 3.6)

3. *Scalability*: how far can the system go before it fails? (see Par. 3.7)

In order to address these questions, a mechanism to launch Podman Containers had to be enabled on the system and a *Real-Time Benchmark Application* (provided by *DLR*) was used to test the real-time capabilities of them.

The complete code of the RT-application is available in Appendix A. In more detail, the application executed 5000 times a dummy workload represented by a simple ADD-MUL operation, in order to collect the jitter and variability of the time required to run it and to evaluate the responsiveness of the system (with the additive layer of containers) in a real-time environment.

The C code of the *Real_Time_Evaluation_monitor.c* benchmark real-time application is the following:

The compilation of the RT-application was performed directly on the QEMU-emulated system and, from now on, all the shell commands are intended to have been run in the QEMU-emulated system and not on the host anymore, unless differently specified:

```
1: gcc -O2 -Wall -Wextra -pthread $(pkg-config --cflags evl)
      Real_Time_Execution_monitor.c -o rt_monitor $(pkg-config --libs evl)
```

3.5.1 RT-Application outside Podman Containers

Hence, to address the first of the previous questions, the one about *feasibility*, a first test was executed on the QEMU-emulated ZCU102 board. It consisted into the execution of the real-time application outside any containers with EVL Kernel in order to first understand if the EVL integration worked fine and if the EVl devices were reachable by the application itself. The test was launched with the following commands:

```
1: sudo -s
2: ulimit -r 99
3: ulimit -l unlimited
4: cd rt_app
5: ./rt_monitor
```

Specifically, the real-time execution required to set the real-time scheduling priority of the process to the maximum (99) in order to guaranty the correct creation of EVL threads and the use of a deterministic real-time scheduling policy such as SCHED_FIFO (default). Furthermore, also the amount of memory that the process can lock in RAM (`ulimit -l`) was set to maximum (unlimited), in order to avoid eventual page faults that could destroy real-time guarantees.

The outcome is shown in Fig. 3.13

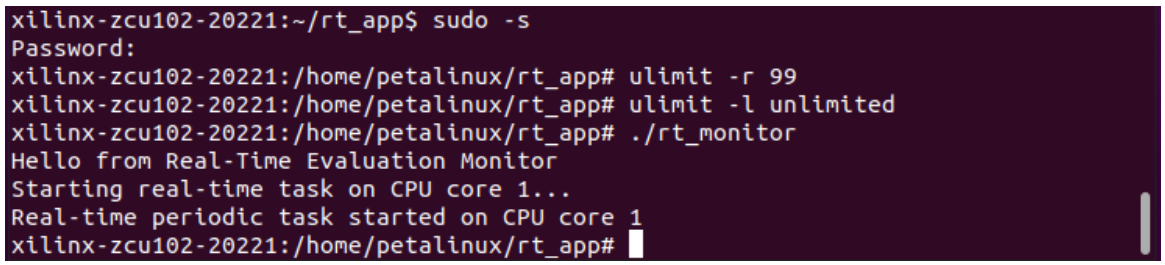


Figure 3.13: Real-Time Evaluation Monitor (RT-Benchmark-App) outside Containers but inside the ZCU102 board emulation with EVL Kernel

The timings measured by the execution of the Benchmark application are shown in Fig. 3.14.

3.5.2 RT-Application inside *Rootful* Container

Then, to test effectively the Podman Containers and their real-time capabilities through the EVL kernel, the following shell script was used to launch a real-time Podman *Rootful* Container in which, then, executing the RT-application:

```
xilinx-zcu102-20221:/home/petalinux/rt_app# tail -n 5 /tmp/ss_timing_log_with_lo
ad_evl_bash.csv
Iteration 4995 - Execution Time: 0.646 ms
Iteration 4996 - Execution Time: 0.275 ms
Iteration 4997 - Execution Time: 0.272 ms
Iteration 4998 - Execution Time: 0.628 ms
Iteration 4999 - Execution Time: 0.616 ms
```

Figure 3.14: Iterations of the Real-Time Evaluation Monitor Application bare metal

```
root@92e96c7605fa:/bench# ./rt_monitor
Hello from Real-Time Evaluation Monitor
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
root@92e96c7605fa:/bench#
```

Figure 3.15: Execution of Real-Time Evaluation Monitor inside a Rootful Podman Container

```
1: sudo podman run --rm -it --network=none \
2:   -v /home/petalinux/rt_app:/bench:rw \
3:   -v /home/petalinux/logs:/tmp:rw \
4:   --device /dev/evl/control \
5:   --device /dev/evl/thread/clone \
6:   --device /dev/evl/clock/monotonic \
7:   --device /dev/evl/clock/realtime \
8:   -v /usr/lib/libevl.so.6:/usr/lib/libevl.so.6:ro \
9:   -v /usr/lib/libbpf.so.1:/usr/lib/libbpf.so.1:ro \
10:  -v /usr/lib/libelf.so.1:/usr/lib/libelf.so.1:ro \
11:  docker.io/library/debian:stable-slim bash
```

In particular, it is possible to observe that such script had the network disabled, because it could cause problems, it was not required by the application and, furthermore, without it the *jitter* should have been also less, good for real-time purposes. Moreover, a *Debian stable image* of ~100 Mb was used and it *bind-mounted* the *EVL libraries*, in order to use the real-time capabilities offered by the *EVL kernel*.

This script opened an interactive Bash shell so, to effectively run the real-time application, it was necessary to:

```
1: cd bench
2: ./rt_monitor
```

The outcome is shown in Fig. 3.15.

```
root@92e96c7605fa:/bench# head -n 10 ss_timing_log_with_load_evl_bash.csv
Iteration 0 - Execution Time: 0.836 ms
Iteration 1 - Execution Time: 0.624 ms
Iteration 2 - Execution Time: 0.378 ms
Iteration 3 - Execution Time: 0.995 ms
Iteration 4 - Execution Time: 0.274 ms
Iteration 5 - Execution Time: 1.246 ms
Iteration 6 - Execution Time: 0.965 ms
Iteration 7 - Execution Time: 0.469 ms
Iteration 8 - Execution Time: 1.617 ms
Iteration 9 - Execution Time: 0.645 ms
root@92e96c7605fa:/bench#
```

Figure 3.16: Iterations of the Real-Time Evaluation Monitor Application inside a Rootful Podman Container

And the outcome of the evaluated times is visible in the .csv file created by the application, shown in Fig. 3.16.

Finally, in order to exit the container and return to the QEMU-emulated system (without interrupting it), it was sufficient to terminate the shell with `Ctrl+D`.

Substantially, this step demonstrated that it was truly possible to run a real-time application inside a Podman *Rootful* container with real-time capabilities.

3.5.3 RT-Application inside *Rootless* Container

The next step was to prove the same, but on a *Rootless* container.

Rootless Real-Time Containers are harder to realize, since they cannot *grant themselves* privileges such as the usage of EVL devices. Hence, they can only use the privileges already granted to the user account.

For this reason, the idea which the Rootless Containers were built on was to change the point of view and, instead than granting real-time permissions and privileges to the containers, they were granted to a specific user. This way, only the user with these permissions was able to run Rootless container *safely*.

To realize this, some modifications inside the real-time application intended to run were needed first:

- the `ret = evl_attach_thread(EVL_CLONE_PUBLIC,"rt-main");` had to become PRIVATE:

```
1: ret = evl_attach_thread(EVL_CLONE_PRIVATE, evl_main_name);
```

```
xilinx-zcu102-20221:~$ id
uid=1000(petalinux) gid=1000(petalinux) groups=1000(petalinux),29(audio),
44(video),1001(evl)
xilinx-zcu102-20221:~$ ls -l /dev/evl/control /dev/evl/thread/clone
crw-rw---- 1 root evl 242, 0 Jan 27 10:28 /dev/evl/control
crw-rw---- 1 root evl 242, 1 Jan 27 10:28 /dev/evl/thread/clone
```

Figure 3.17: id command to show groups and users

```
petalinux@af097b8c2bcb:/$ ls -l /dev/evl/control /dev/evl/thread/clone
crw-rw---- 1 nobody nogroup 242, 0 Jan 27 10:28 /dev/evl/control
crw-rw---- 1 nobody nogroup 242, 1 Jan 27 10:28 /dev/evl/thread/clone
```

Figure 3.18: Users and Groups for EVL devices inside Rootless Container

PUBLIC attachment indeed, was restricted by policy, because only root or a specific group could create "public" EVL threads. Furthermore, this also ensured that third-party tasks could not create globally visible Real-Time entities, unless explicitly authorized. Moreover, *PRIVATE* also kept the tasks contained, guaranteeing a better isolation and security model.

- It was observed that the EVL device nodes — such as `/dev/evl/control`, `/dev/evl/thread/clone` etc. — were associated with the *evl* group and configured with permissions `0660` (see Fig. 3.17). This configuration implied that read and write access to these devices was restricted to the owner and members of the *evl group*. Since the user *petalinux* was a member of this group, any process executed under the same user identity —including those running inside a rootless container — could access the EVL devices without requiring additional privileges.

  However, when executing the application inside the *Rootless Container*, the user and group mappings appeared really different:

  - `uid=0(root) gid=0(root) groups=0(root)`
  - `crw-rw-- 1 nobody nogroup 242, 0 Jan 27 10:28 /dev/evl/control`
  - `crw-rw-- 1 nobody nogroup 242, 1 Jan 27 10:28 /dev/evl/thread/ clone` (see Fig. 3.18)

  This behavior was a consequence of user namespaces which remapped *user and group identifiers* inside the *Rootless* containers. Hence, although the process appeared as `root` within the container (`uid=0`), it was actually mapped to an unprivileged user on the host. As a result, the EVL device nodes were seen as owned by `nobody:nogroup`, and the containerized process did not belong to the `evl` group. Moreover, root privileges

```
xilinx-zcu102-20221:~$ ls -l /dev/evl/control /dev/evl/thread/clone
crw-rw---- 1 root petalinux 242, 0 Jan 27 10:28 /dev/evl/control
crw-rw---- 1 root petalinux 242, 1 Jan 27 10:28 /dev/evl/thread/clone
```

Figure 3.19: From EVL to Petalinux group for EVL devices

```
petalinux@af097b8c2bcb:/$ id
uid=1000(petalinux) gid=1000(petalinux) groups=1000(petalinux)
```

Figure 3.20: User namespace mapping for Rootless Container

inside a user namespace do not grant the ability to bypass file permissions on host device nodes. This mismatch, indeed, resulted into an `"failed to attach to xenomai evl, -13ret"` kind of error, because the EVL devices appeared not accessible by the RT-application. To solve this issue then, it was necessary to modify on the host the permissions and group ownership of the EVL devices nodes, in a way that they were truly accessible to the user *petalinux*. This was achieved through the commands (see Fig. 3.19):

```
1: sudo chmod -R g+rwX /dev/evl
2: sudo chgrp -R petalinux /dev/evl
```

And, this way, the processes executed within the container and mapped to the corresponding host user were finally able to access the EVL devices correctly.

- Subsequently, Podman was configured to run in *Rootless mode* with *user namespace mapping*. This was ensured with the flag `-userns=keep-id` into the script that ensures that the user and group identifiers inside the container are preserved consistently with those on the host system - specifically, `UID=1000 (petalinux)` and `GID=1000 (petalinux)` — instead of being remapped to anonymous identities such as `nobody`. As a result, processes running inside the container maintain the same effective identity as on the host, enabling correct group membership resolution and access to EVL device nodes. The outcome was the one shown in Fig. 3.20.
- In the end, the shell commands used to launch the container were:

```
1: podman run --rm -it --network=none \
2:   --userns=keep-id \
3:   -v /home/petalinux/rt_app:/bench:rw \
4:   -v /home/petalinux/logs/rootless.csv:
        /tmp/ss_timing_log_with_load_evl_bash.csv:rw \
```

```
5:  --device /dev/evl/control \
6:  --device /dev/evl/thread/clone \
7:  --device /dev/evl/clock/monotonic \
8:  --device /dev/evl/clock/realtime \
9:  -v /usr/lib/libevl.so.6:/usr/lib/libevl.so.6:ro \
10: -v /usr/lib/libbpf.so.1:/usr/lib/libbpf.so.1:ro \
11: -v /usr/lib/libelf.so.1:/usr/lib/libelf.so.1:ro \
12: docker.io/library/debian:stable-slim bash
```

Before running this script though, it was also necessary to create the log file *rootless.csv* in the `/home/petalinux/logs/` path and to change its permissions and groups, in the same way as with the previous /dev/evl:

```
1: cd logs
2: sudo chmod -R g+rw rootless.csv
3: sudo chgrp -R petalinux rootless.csv
```

This way, the application inside the *Rootless* container could properly write the execution times inside the log, terminating correctly the execution.

From the *Bash shell* of the container, it was possible to simply run the *Real_Time_Evaluation_monitor* application, as:

```
1: cd bench/
2: ./rt_monitor
```

and get the output shown in Fig. 3.21, while the timing and iterations got are shown in Fig. 3.22.

```
petalinux@bb898b5c4233:/bench$ ./rt_monitor
Hello from Real-Time Evaluation Monitor
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
```

Figure 3.21: Real_Time_Evaluation_Monitor execution inside Rootless Container

```
petalinux@bb898b5c4233:/bench$ head ss_timing_log_with_load_evl_bash.csv
Iteration 0 - Execution Time: 0.836 ms
Iteration 1 - Execution Time: 0.624 ms
Iteration 2 - Execution Time: 0.378 ms
Iteration 3 - Execution Time: 0.995 ms
Iteration 4 - Execution Time: 0.274 ms
Iteration 5 - Execution Time: 1.246 ms
Iteration 6 - Execution Time: 0.965 ms
Iteration 7 - Execution Time: 0.469 ms
Iteration 8 - Execution Time: 1.617 ms
Iteration 9 - Execution Time: 0.645 ms
```

Figure 3.22: Iterations of the Real_Time_Evaluation_Monitor application inside a Rootless Podman Container

Anyway, it is important to specify that, even if the execution of the *Real_Time_Execution_monitor.c* application was possible inside a *Rootless* container, this did not mean that any real-time application could run without any problems. Indeed, many real-time applications require `libbpf`, that is usually a privileged library. The used application did not use such library but trying to run a RT-application that loads BPF programs could result into a failure because of the missing privileges.

3.6 Concurrent execution of Real-Time Containers

After successfully running a single *Rootful* and a single *Rootless* container instance at a time, some tests involving multiple containers executing concurrently were tried.

The ultimate goal of the project this thesis is part of is, indeed, to test and prove that multiple *Real-Time Containers* (with the least of the privileges possible) can run simultaneously on the same embedded platform, still guaranteeing low latency and avoiding mutual interference. In particular, one of the objectives was to verify that multiple real-time workloads can execute concurrently while sharing kernel and hardware resources - such as CPU cores, clocks, and EVL kernel services - without causing conflicts or *namespace collisions* on *EVL* objects.

However, when it came to running multiple real time container instances, some new problems rose up differently from the case where only one container was active. Indeed, the problematic part in this scenario lied in the fact that, since EVL devices exposed global kernel interfaces and were inherently shared among the processes, it was in the care of each container to create its own distinct EVL objects to work with. Otherwise, an error rose up because the corresponding EVL object files were already considered in use.

For this reason, running some experiments with two or more containers with overlapping execution times (concurrent execution) required verification that the Real-Time application

intended to run complied with the precautions related to the EVL objects and devices previously mentioned and did not get stuck into a namespace conflict.

In the case of the Real_Time_Execution_monitor.c real-time application provided by DLR and used for testing purposes, such modifications were also required. The code, already modified and used for each experiment conducted in this thesis, is reported in full in Appendix A.

In more detail, when it comes to multiple *Podman* containers using *EVL devices*, it was necessary to assign a unique identifier (`EVL_INSTANCE=A`, `EVL_INSTANCE=B`, etc.) to each container, in order to generate unique names for EVL threads and tasks inside the application for each instance. This way, namespace-conflicts were avoided and more containers could run concurrently, properly sharing system resources and EVL devices through different EVL objects.

Furthermore, it was also necessary to ensure that the name of the real-time thread created by any instance of the real-time application running was unique and linked to the container it was running in, otherwise a conflict between two instances could arise because of the same name-thread that more containers could try to execute at the same time.

Finally, to run multiple instances of the same application that exploited *EVL real-time capabilities*, it was mandatory to check if the flag used to attach the thread to the *EVL core* was *PRIVATE* and not *PUBLIC* because it was necessary that each instance created a private EVL thread context, independent from other threads, in order for the thread to get its own real-time control block, its own scheduling parameters and, in general, to become a fully independent EVL real-time entity. Hence, in the case of the *Real_Time_Execution_monitor.c* application, it was necessary to modify the following lines from:

```
1: if (evl\_attach\_thread (EVL\_CLONE\_PUBLIC, evl\_task\_name) < 0)
```

to:

```
1: if (evl\_attach\_thread (EVL\_CLONE\_PRIVATE, evl\_task\_name) < 0)
```

Another relevant aspect of these experiments was represented by the *synchronization mechanism*, to ensure a real-concurrent execution of the containers.

The easiest idea indeed, even if not adaptable in case of scalability, would have been to launch multiple containers in separate terminals, in order to then start the real-time application at the same time in each of them. However, this was not possible for two main reasons: first of all, these experiments were executed in the QEMU-emulated system that could be open in just

one terminal and did not allow to run more terminal in parallel inside the same execution of the system itself, and secondly, to be sure that the two containers could truly run concurrently (at the same time) in a detached way, a more quickly way to start the containers was needed, especially to test the scalability of the system where tens of containers had to start at the same time.

For these reasons, it was decided to adopt as a synchronization mechanism among containers, a *timing barrier*. Such barrier was implemented through a synchronization start file (mechanism of a *GO file*), that made both containers start apart of less than 4 ms after having already created them. The shell commands used to create the timing barrier are shown below:

```
1: mkdir -p /home/petalinux/sync
2: rm -f /home/petalinux/sync/GO /home/petalinux/sync/start_times.txt
3: rm -f /home/petalinux/sync/rootful_A.debug /home/petalinux/sync/rootful_B.debug
4: touch /home/petalinux/sync/start_times.txt
5: chmod 666 /home/petalinux/sync/start_times.txt
```

The `start_times` file is a file filled with the real starting time of each container, in order to after-prove that they truly ran in parallel.

In more detail, the containers were launched in *detached mode* and configured to wait on a shared synchronization file. Then, after having created all the required container instances, the synchronization and start of the containers were triggered by the creation of a *GO file*, acting as a sentinel for releasing the waiting processes (containers). In this way, even if the launching itself of each container required some seconds, it was possible to confirm that the actual start of the real-time application inside each instance occurred within only a few milliseconds of each other. This was also confirmed by the shared log file (*start_times.txt*) written by the containers, which recorded their respective start times. Moreover, it was observed that a single execution of the Real-Time application used to test the containers required at least five seconds. Comparing this with the recorded starting times of the containers demonstrates that the multiple instances of the application were truly overlapping in time.

This proved that not only can a real-time application with EVL devices on an embedded system run, but that multiple concurrent instances can also run, while properly sharing resources and EVL devices.

Various experiments were performed to prove what previously affirmed. The shell commands and output are reported in the following paragraphs:

1. Two *Rootful* Containers at the same time (see Par. 3.6.1);

2. Two *Rootless* Containers at the same time (see Par. 3.6.2);
3. One *Rootful* Container and one *Rootless* Container at the same time (see Par. 3.6.3);
4. Two *Rootless* and one *Rootful* at the same time (see Par. 3.6.4);

3.6.1 Concurrent execution of two *Rootful* Containers

As already introduced in Par. 3.6, in order to make multiple concurrent (*Rootful*) containers run, the pre-steps and precautions shown before had to be taken.

After that, knowing that each execution of the real-time application inside a container created some *CSV files* to store the 5000 execution times of the 5000 execution of the same application, some folders to store such files at different paths needed to be created. Otherwise, the different executions of the application inside the different containers would have overwritten the files because of the path engraved inside the application itself. Hence, it was necessary to change the path for such files externally, as shown below:

```
1: mkdir -p /home/petalinux/logs/rootful_dir_A /home/petalinux/logs/rootful_dir_B
2: chmod 777 /home/petalinux/logs/rootful_dir_A /home/petalinux/logs/rootful_dir_B
3: rm -f /home/petalinux/logs/rootful_dir_A/ss.csv
       /home/petalinux/logs/rootful_dir_B/ss.csv
```

Then, the two *Rootful* containers could finally be created:

- First *Rootful* Container (A) in detached mode:

```
 1: sudo podman run -d --name rt_rootful_A \
 2:   --log-driver=k8s-file \
 3:   --network=none \
 4:   --hostname rt_ful_A \
 5:   -v /home/petalinux/rt_app:/bench:rw \
 6:   -v /home/petalinux/sync:/sync:rw \
 7:   -v /home/petalinux/logs/rootful_dir_A:/logs:rw \
 8:   --device /dev/evl/control --device /dev/evl/thread/clone \
 9:   --device /dev/evl/clock/monotonic --device /dev/evl/clock/realtime
10:   -v /usr/lib/libevl.so.6:/usr/lib/libevl.so.6:ro \
11:   -v /usr/lib/libbpf.so.1:/usr/lib/libbpf.so.1:ro \
12:   -v /usr/lib/libelf.so.1:/usr/lib/libelf.so.1:ro \
13:   docker.io/library/debian:stable-slim \
14:  bash -lc '\{
```

```
  while [ ! -e /sync/GO ]; do sleep 0.001; done
  echo "rootful_A start $(date +\%s.\%N)" >> /sync/start_times.txt
  echo "rootful_A whoami=$(id -u):$(id -g)" >> /sync/start_times.txt
  ln -sf /logs/ss.csv /tmp/ss_timing_log_with_load_evl_bash.csv
  EVL_INSTANCE=A /bench/rt_monitor_EVL
  echo "rootful_A rc=$?"
 \} >> /sync/rootful_A.debug 2>\&1
'
```

- Second *Rootful* Container (B) in detached mode:

```
sudo podman run -d --name rt_rootful_B \
  --log-driver=k8s-file \
  --network=none \
  --hostname rt_ful_B \
  -v /home/petalinux/rt_app:/bench:rw \
  -v /home/petalinux/sync:/sync:rw \
  -v /home/petalinux/logs/rootful_dir_B:/logs:rw \
  --device /dev/evl/control --device /dev/evl/thread/clone \
  --device /dev/evl/clock/monotonic --device /dev/evl/clock/realtime
  -v /usr/lib/libevl.so.6:/usr/lib/libevl.so.6:ro \
  -v /usr/lib/libbpf.so.1:/usr/lib/libbpf.so.1:ro \
  -v /usr/lib/libelf.so.1:/usr/lib/libelf.so.1:ro \
  docker.io/library/debian:stable-slim \
 bash -lc '\{
  while [ ! -e /sync/GO ]; do sleep 0.001; done
      echo "rootful_B start $(date +\%s.\%N)" >> /sync/start_times.txt
      echo "rootful_B whoami=$(id -u):$(id -g)" >> /sync/start_times.txt
      ln -sf /logs/ss.csv /tmp/ss_timing_log_with_load_evl_bash.csv
      EVL_INSTANCE=B /bench/rt_monitor_EVL
      echo "rootful_B rc=$?"
    \} >> /sync/rootful_B.debug 2>\&1
  '
```

At that point, it was possible to release both containers in parallel (at the same time) through the *GO file barrier*:

```
touch /home/petalinux/sync/GO
```

Then, after the `rt_monitor` application was executed in both containers, the debug output provided by the containers was examined in order to check that the execution had been flawless and did not result in an error, using the following commands:

```
1: cat /home/petalinux/sync/start_times.txt
2: sed -n '1,120p' /home/petalinux/sync/rootful_A.debug
3: sed -n '1,120p' /home/petalinux/sync/rootful_B.debug
4: sudo podman inspect rt_rootful_A --format '{{.State.ExitCode}} {{.State.Error}}'
5: sudo podman inspect rt_rootful_B --format '{{.State.ExitCode}} {{.State.Error}}'
```

The expected output of the previous commands is shown in Figs. 3.23 and 3.24.

```
xilinx-zcu102-20221:~/rt_app$ touch /home/petalinux/sync/GO
xilinx-zcu102-20221:~/rt_app$ cat /home/petalinux/sync/start_times.txt
rootful_B start 1769845848.417360328
rootful_A  start 1769845848.420817835
rootful_A whoami=0:0
rootful_B whoami=0:0
```

Figure 3.23: The two Rootful Containers start at ~3.4ms apart (they truly run in parallel) through the timing barrier mechanism

```
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/rootful_A.deb
ug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/rootful_B.deb
ug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
xilinx-zcu102-20221:~/rt_app$ sudo podman inspect rt_rootful_A --format '{{.Stat
e.ExitCode}} {{.State.Error}}'

0
xilinx-zcu102-20221:~/rt_app$ sudo podman inspect rt_rootful_B --format '{{.Stat
e.ExitCode}} {{.State.Error}}'
0
```

Figure 3.24: Clean state and execution of the rt_monitor application inside two Rootful Containers running in parallel

Before also examining the CSV produced by the two executions of the application, a preventive check of the length of the files was considered useful in order to verify that the application in the two containers truly wrote the expected timing values:

```
1: wc -l /home/petalinux/logs/rootful_dir_A/ss.csv
      /home/petalinux/logs/rootful_dir_B/ss.csv
```

```
xilinx-zcu102-20221:~/rt_app$ wc -l /home/petalinux/logs/rootful_dir_A/ss.csv /h
ome/petalinux/logs/rootful_dir_B/ss.csv
  5000 /home/petalinux/logs/rootful_dir_A/ss.csv
  5000 /home/petalinux/logs/rootful_dir_B/ss.csv
 10000 total
```

Figure 3.25: Word Count of words written in the log files by both Rootful Containers to check if the execution of the `rt_monitor` application was flawless

---

The expected output of this command is shown in Fig. 3.25.

Finally, to check the results the following commands were used:

```
1: tail -n 5 /home/petalinux/logs/rootful_dir_A/ss.csv
2: tail -n 5 /home/petalinux/logs/rootful_dir_B/ss.csv
```

The outcome was the one shown in Fig. 3.26.

```
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/rootful_dir_A/ss.cs
v
Iteration 4995 - Execution Time: 7.378 ms
Iteration 4996 - Execution Time: 1.958 ms
Iteration 4997 - Execution Time: 2.540 ms
Iteration 4998 - Execution Time: 0.362 ms
Iteration 4999 - Execution Time: 0.278 ms
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/rootful_dir_B/ss.cs
v
Iteration 4995 - Execution Time: 0.304 ms
Iteration 4996 - Execution Time: 0.638 ms
Iteration 4997 - Execution Time: 0.810 ms
Iteration 4998 - Execution Time: 1.051 ms
Iteration 4999 - Execution Time: 0.967 ms
```

Figure 3.26: Output timing values of the execution in parallel of the `rt_monitor` application inside the two Rootful Containers

The last step was to safely remove the containers and the GO file to prepare the system for a new execution:

```
1: sudo podman rm -f rt_rootful_A rt_rootful_B
2: rm -f /home/petalinux/sync/GO
```

3.6.2 Concurrent execution of two *Rootless* Containers

In the same fashion, it was also possible to prove that two (or more) real-time *Rootless* Containers could run at the same time, also executing the same real-time application.

In this case, some additional precautions were needed in order to let the *Rootless* Container run flawlessly, as had already happened in the case of the single *Rootless* Container.

In more detail, the steps followed were the same as in the *Rootful-Rootful* case (see Par. 3.6.1), while the differences are described below:

- Before launching a *Rootless* Container, in order to make the EVL devices usable from it, it was necessary to check that the user running the containers belonged to the same group of the owner of the EVL devices themselves. Hence, outside the container, on the board, it was mandatory to run the commands:

  ```
  1: ls -l /dev/evl/control /dev/evl/thread/clone
  ```

  And, if the owner of the devices was `evl`, then:

  ```
  1: sudo chmod -R g+rwX /dev/evl
  2: sudo chgrp -R petalinux /dev/evl
  ```

  where `petalinux` was the user that had to run the *Rootless* containers.

- Obviously, it was necessary to adapt the names of the debug and output files respect to what reported in the case of *Rootful-Rootful*, specifying that the intended-to-be-run containers were two *Rootless*.

- Every command had to be run without `sudo`.

- The shell script previously shown to run the containers needed to be updated with the new names of the files and specifying the name of the containers as *Rootless*, for clarity reasons.

The output of such execution is shown in Figs. 3.27, 3.28, 3.29 and 3.30.

```
xilinx-zcu102-20221:~/rt_app$ cat /home/petalinux/sync/start_times.txt
rootless_B start 1769899640.138420557
rootless_A  start 1769899640.184116510
rootless_B whoami=0:0
rootless_A whoami=0:0
```

Figure 3.27: The two Rootless Containers start at ~4.5ms apart (they truly run overlapping in time) through the timing barrier mechanism

```
xilinx-zcu102-20221:~/rt_app$ wc -l /home/petalinux/logs/rootless_dir_A/ss.csv /
home/petalinux/logs/rootless_dir_B/ss.csv
  5000 /home/petalinux/logs/rootless_dir_A/ss.csv
  5000 /home/petalinux/logs/rootless_dir_B/ss.csv
 10000 total
```

Figure 3.29: Word Count of words written in the log files by both Rootless Containers to check if the execution of the rt_monitor application was flawless

```
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/rootless_A.de
bug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/rootless_B.de
bug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
xilinx-zcu102-20221:~/rt_app$ podman inspect rt_rootless_A --format '{{.State.Ex
itCode}} {{.State.Error}}'

0
xilinx-zcu102-20221:~/rt_app$ podman inspect rt_rootless_B --format '{{.State.Ex
itCode}} {{.State.Error}}'
0
```

Figure 3.28: Clean state and execution of the rt_monitor application inside two concurrent Rootless Containers

3.6.3 Concurrent execution of Mixed (one *Rootless* and one *Rootful*) Containers

In the *mixed* case, the steps followed were the same as in the two previous cases (see Par. 3.6.1 and 3.6.2) but some additive precautions had to be taken:

- `sudo` had to be used to run only the *Rootful* container.
- before running the *Rootless* Container it was mandatory to test if the EVL devices already belonged to the same user that had to run the containers, otherwise it had to be changed

```
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/rootless_dir_A/ss.c
sv
Iteration 4995 - Execution Time: 0.274 ms
Iteration 4996 - Execution Time: 0.351 ms
Iteration 4997 - Execution Time: 0.278 ms
Iteration 4998 - Execution Time: 0.616 ms
Iteration 4999 - Execution Time: 0.281 ms
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/rootless_dir_B/ss.c
sv
Iteration 4995 - Execution Time: 0.923 ms
Iteration 4996 - Execution Time: 0.383 ms
Iteration 4997 - Execution Time: 0.543 ms
Iteration 4998 - Execution Time: 5.490 ms
Iteration 4999 - Execution Time: 0.295 ms
```

Figure 3.30: Output timing values of the execution overlapping in time of the rt_monitor application inside the two Rootless Containers

as shown in the Par. 3.6.2.

The output of this case is shown in Figs. 3.31, 3.32, 3.33 and 3.34.

```
xilinx-zcu102-20221:~/rt_app$ cat /home/petalinux/sync/start_times.txt
P_rootful start 1769904109.826103358
P_rootless  start 1769904109.848378865
P_rootful whoami=0:0
P_rootless whoami=0:0
```

Figure 3.31: The Rootful and Rootless Containers start at ~2.2ms apart through the timing barrier mechanism

```
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/P_rootless_dir/ss.c
sv
Iteration 4995 - Execution Time: 0.557 ms
Iteration 4996 - Execution Time: 0.691 ms
Iteration 4997 - Execution Time: 0.273 ms
Iteration 4998 - Execution Time: 0.931 ms
Iteration 4999 - Execution Time: 0.850 ms
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/P_rootful_dir/ss.cs
v
Iteration 4995 - Execution Time: 0.569 ms
Iteration 4996 - Execution Time: 0.683 ms
Iteration 4997 - Execution Time: 0.542 ms
Iteration 4998 - Execution Time: 0.353 ms
Iteration 4999 - Execution Time: 0.647 ms
```

Figure 3.34: Output timing values of the execution in parallel of the `rt_monitor` application inside the Rootful and Rootless Containers

```
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/P_rootless.de
bug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
Hello from Real-Time Evaluation Monitor
P_rootless rc=0
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/P_rootful.deb
ug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
Hello from Real-Time Evaluation Monitor
P_rootful rc=0
xilinx-zcu102-20221:~/rt_app$
xilinx-zcu102-20221:~/rt_app$ podman inspect rt_P_rootless --format '{{.State.Ex
itCode}} {{.State.Error}}'
0
xilinx-zcu102-20221:~/rt_app$ sudo podman inspect rt_P_rootful --format '{{.Stat
e.ExitCode}} {{.State.Error}}'
0
```

Figure 3.32: Clean state and execution of the `rt_monitor` application inside a Rootful and a Rootless Container running with overlapping times

```
xilinx-zcu102-20221:~/rt_app$ wc -l /home/petalinux/logs/P_rootless_dir/ss.csv /
home/petalinux/logs/P_rootful_dir/ss.csv
  5000 /home/petalinux/logs/P_rootless_dir/ss.csv
  5000 /home/petalinux/logs/P_rootful_dir/ss.csv
 10000 total
```

Figure 3.33: Word Count of words written in the log files by the two Containers to check if the execution of the `rt_monitor` application was flawless

```
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/P_rootless_1.
debug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/P_rootless_2.
debug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
xilinx-zcu102-20221:~/rt_app$ sed -n '1,120p' /home/petalinux/sync/P_rootful.deb
ug
Starting real-time task on CPU core 1...
Real-time periodic task started on CPU core 1
xilinx-zcu102-20221:~/rt_app$
xilinx-zcu102-20221:~/rt_app$ podman inspect rt_P_rootless_1 --format '{{.State.
ExitCode}} {{.State.Error}}'
0
xilinx-zcu102-20221:~/rt_app$ podman inspect rt_P_rootless_2 --format '{{.State.
ExitCode}} {{.State.Error}}'
0
xilinx-zcu102-20221:~/rt_app$ sudo podman inspect rt_P_rootful --format '{{.Stat
e.ExitCode}} {{.State.Error}}'
0
```

Figure 3.36: Clean state and execution of the `rt_monitor` application inside the three concurrent Containers

```
xilinx-zcu102-20221:~/rt_app$ cat /home/petalinux/sync/start_times.txt
P_rootful start 1770033863.896586910
P_rootless_1  start 1770033863.901132648
P_rootless_2  start 1770033863.902882325
P_rootful whoami=0:0
P_rootless_1 whoami=0:0
P_rootless_2 whoami=0:0
```

Figure 3.35: The three Containers start, neatly, at ~4.5ms and ~1.7ms apart thanks to the timing barrier mechanism, guaranteeing the constraint of concurrent execution

3.6.4 Concurrent execution of two *Rootless* and one *Rootful* Container

Finally, one last experiment for this section was executed to prove that multiple mixed real-time containers were able to run concurrently with the provided set-up. In particular, two *Rootless* Containers and a *Rootful* were created. This required, as in the previous cases, that each container accessed its own EVL objects with unique identifiers, while using the same EVL kernel services, in order to avoid that different processes (different execution of the same `rt_monitor` application) tried to create EVL objects with the same name. For this reason, inside each shell script to launch the containers, it was necessary to specify different identifiers, such as:

```
1: EVL_INSTANCE=A #first rootless container
2: EVL_INSTANCE=B #second rootless container
3: EVL_INSTANCE=C #rootful container
```

The steps followed then were the same as before, and the result of this execution is shown in Figs. 3.35, 3.36, 3.37 and 3.38.

```
xilinx-zcu102-20221:~/rt_app$ wc -l /home/petalinux/logs/P_rootless_dir_1/ss.csv
 /home/petalinux/logs/P_rootless_dir_2/ss.csv /home/petalinux/logs/P_rootful_dir
/ss.csv
  5000 /home/petalinux/logs/P_rootless_dir_1/ss.csv
  5000 /home/petalinux/logs/P_rootless_dir_2/ss.csv
  5000 /home/petalinux/logs/P_rootful_dir/ss.csv
 15000 total
```

Figure 3.37: Word Count of words written in the log files by the three Containers to check if the execution of the `rt_monitor` application was flawless

```
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/P_rootless_dir_1/ss
.csv
Iteration 4995 - Execution Time: 1.923 ms
Iteration 4996 - Execution Time: 0.621 ms
Iteration 4997 - Execution Time: 0.681 ms
Iteration 4998 - Execution Time: 0.275 ms
Iteration 4999 - Execution Time: 0.344 ms
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/P_rootless_dir_2/ss
.csv
Iteration 4995 - Execution Time: 0.275 ms
Iteration 4996 - Execution Time: 0.625 ms
Iteration 4997 - Execution Time: 0.800 ms
Iteration 4998 - Execution Time: 0.416 ms
Iteration 4999 - Execution Time: 0.579 ms
xilinx-zcu102-20221:~/rt_app$ tail -n 5 /home/petalinux/logs/P_rootful_dir/ss.cs
v
Iteration 4995 - Execution Time: 2.092 ms
Iteration 4996 - Execution Time: 0.359 ms
Iteration 4997 - Execution Time: 0.611 ms
Iteration 4998 - Execution Time: 0.366 ms
Iteration 4999 - Execution Time: 0.274 ms
```

Figure 3.38: Output timing values of the concurrent execution of the `rt_monitor` application inside the three Containers

3.7 Testing the scalability of the system

The last matter addressed by this thesis was related to the *scalability* of the system. In particular, it was interesting to understand up to how many concurrent real-time containers the emulated-system could still properly work.

To test this, different experiments were conducted, each time increasing the number of concurrent running containers.

The scripts and the precautions taken to run them were the same as in the previous case with two-three containers (see Par. 3.6).

In more detail, the shell script was slightly modified respect to the previous shell commands used to launch Podman containers, to easily launch a variable big number of containers at the same time and trying to automatize the whole process, for practical reasons.

This script is available at Appendix B.

The evaluation of the results obtained is reported in Par. 4.2.

Chapter 4

Benchmarking and Evaluation

The following chapter evaluates the results obtained in the second part of the project implemented in chapter 3, where the feasibility of running *Rootful* and *Rootless* Podman containers with real-time capabilities (due to the EVL integration) was demonstrated on an embedded platform (even if it was an emulated one). It was also demonstrated that it is possible to run multiple containers that, at the same time, can access the EVL devices to run real-time applications. However, all of this has been implemented, but has not been evaluated yet.

In order to provide a *latency measurement* of the execution times of a real-time application within this kind of system and to analyze the *overhead* introduced by the utilization of containers, the distributions of the execution times provided by the experiments implemented were analyzed in detail.

First, a more detailed overview of the real-time application used to evaluate the system and the steps followed to collect the data produced by each execution of it in different contexts are provided in Par. 4.1.

Then, a systematic evaluation of the different experiments is provided. Specifically, the results of the following experiments were deeply analyzed from the point of view of the main statistical metrics and eventually compared among them, as follows:

- comparison of results obtained from the execution of the application on *Bare-metal*, inside a single *Rootful* container and then inside a single *Rootless* one (see Par. 4.1.2);
- comparison of results obtained from the concurrent execution of two *Rootful* containers and the single *Rootful* container-case (see Par. 4.1.3);
- comparison of results obtained from the concurrent execution of two *Rootless* containers and the single *Rootless* container-case (see Par. 4.1.4);

- comparison of results obtained from the concurrent execution of two *Rootless* and one *Rootless* containers and the single *Rootful* and *Rootless*container-cases (see Par. 4.1.5);

Subsequently, the evaluation of the results related to the system scalability tests is provided. In particular, Par. 4.2 explains how the collection of the big amount of data coming from these experiments was performed. Then, a deep analysis of the results of a few test cases with a variable number of concurrent *Rootless* containers running (16 containers Par. 4.2.2, 32 containers Par. 4.2.2 and 64 containers Par. 4.2.2) is reported. All of this culminates in the *overall analysis* of all the cases of multiple concurrent containers (from 2 to 64) presented in Par. 4.2.3.

Finally, a further test with 80 and 100 concurrent running containers is reported and analyzed in Par. 4.2.4, demonstrating that it was not possible to scale up to such a large number of containers and that 64 was the last experiment that was executed successfully.

4.1 A real-time application with Dummy Workload to evaluate the functioning of real-time containers

The term *Benchmark* refers to a set of software tests designed to provide a measure of a computer's performance with regard to various operations. However, the benchmark can also be defined as the determination of the ability of that software to perform a particular task for which it was designed more or less quickly, precisely, or accurately [44].

In the present thesis, in order to evaluate the performance aspects of the proposed system and the impact of the Podman Containers on the execution of real-time applications on an embedded system with EVL kernel, a *benchmark real-time application* was executed in three different contexts.

The application *Real_Time_Execution_monitor.c* was provided by the *DLR* and it measured the execution time of a dummy workload, including any *preemptions* that happen while it runs, in a WCET-like goal.

The execution of such workload is repeated for 5000 iterations and in more detail, at each iteration of the periodic task, the program records the current time immediately before executing the workload. The workload is a CPU-bound type of task and, as soon as the computation completes, the program records the current time again. This way, computing the difference between the two timestamps obtained, it is possible to get the elapsed time required to complete one execution of the task itself. The interesting aspect is that obviously such elapsed time also includes the eventual delays caused by operating system scheduling, preemptions, virtualization and container-related overhead that may occur while the task is running.

This means that by repeating this procedure for 5000 iterations, the experiment collects a distribution of execution times that characterizes both the average cost and the variability (i.e. *jitter*) of executing the same workload under different deployment environments.

The integral code is reported in App. A, but the effective measurements is performed as shown in the snippet below:

```
1: // some cpu load (dummy loop)
2: volatile int product = 0;
3: for (int j = 0; j < 100000; j++) product += j*j;
4: clock_gettime(CLOCK_MONOTONIC, &end_time); // end timestamp
5: // calc duration in ms
6: exec_times[i] = (end_time.tv_sec - start_time.tv_sec) * 1e3 + (end_time.tv_nsec -
       start_time.tv_nsec) / 1e6;
```

To truly evaluate the overhead induced by the containers, a comparison was set between the data collected by running the same experiment in three different contexts:

- on bare metal;
- inside a Rootful Podman Container;
- inside a Rootless Podman Container.

Obviously, all these experiments were carried out on the QEMU-emulated board ZCU102 and so an overhead due to the virtualization layer of the system must be taken into account when it comes to evaluating the output timings.

4.1.1 Collection of data from the execution of the benchmark application in different experiments

First, the case considered was the execution of the application on the QEMU-emulated ZCU102 board *bare metal*, i.e., outside any container (see. Par. 3.13). This execution provided the output *ss_timing_log_with_load_evl_bash.csv*, which contained 5000 iterations of the benchmark application and the time required to run the dummy workload inside.

The last 10 iterations are shown in Fig. 4.1.

The data collected inside a CSV file created by the same application, were then exported outside the container - through the *netcat* command, as shown in Fig. 4.2 - to be properly

```
xilinx-zcu102-20221:/home/petalinux/rt_app# tail /tmp/ss_timing_log_with_load_evl_bash.csv
Iteration 4990 - Execution Time: 0.272 ms
Iteration 4991 - Execution Time: 0.643 ms
Iteration 4992 - Execution Time: 0.273 ms
Iteration 4993 - Execution Time: 0.273 ms
Iteration 4994 - Execution Time: 0.734 ms
Iteration 4995 - Execution Time: 0.428 ms
Iteration 4996 - Execution Time: 0.329 ms
Iteration 4997 - Execution Time: 0.699 ms
Iteration 4998 - Execution Time: 0.677 ms
Iteration 4999 - Execution Time: 0.467 ms
xilinx-zcu102-20221:/home/petalinux/rt_app#
```

Figure 4.1: Benchmark rt-application iterations and timing log *bare metal* (outside any container)

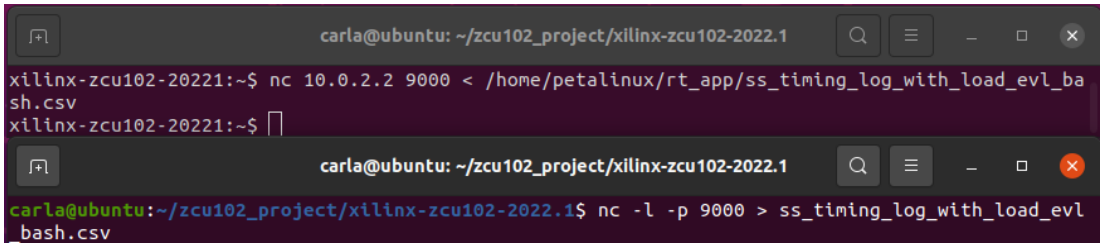


Figure 4.2: Netcat command to transfer the csv file from the container inside the QEMU-emulated board to the host machine (represented by the virtual machine on which all of the software stack was layered

analyzed. The same procedure was also applied in all the other cases described in the next paragraphs.

Then, the same application was executed inside a Rootful and a Rootless Podman Container. In these cases, a folder was created to store the output logs of the two containers, as well two files - called *rootless.csv* and *rootful.csv* were created inside the same folder.

```
1: mkdir -p /home/petalinux/logs
2: : > /home/petalinux/logs/rootless.csv
3: sudo sh -c ': > /home/petalinux/logs/rootful.csv'
```

These files were attached to the log file produced by the execution of the Benchmark application inside each container to distinguish between the output of the two containers. In fact, such an attachment was provided by the bind-mounting of the new file to the container, through the following command, in the case of *Rootful* Container:

```
1: -v /home/petalinux/logs/rootful.csv: \
2: /tmp/ss_timing_log_with_load_evl_bash.csv:rw
```

and through the following one, in the case of the Rootless Container:

```
1: -v /home/petalinux/logs/rootless.csv: \
2: /tmp/ss_timing_log_with_load_evl_bash.csv:rw
```

The last 10 iterations produced by the execution of the Benchmark application inside the Rootful Container are shown in Fig. 4.3

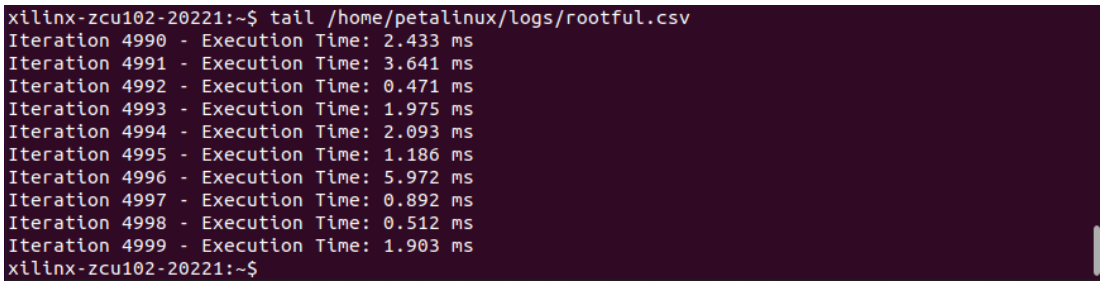


Figure 4.3: Benchmark rt-application iterations and timing log inside Rootful Podman Container

While, the last 10 iterations produced by the execution of the Benchmark application inside the Rootless Container are shown in Fig. 4.4

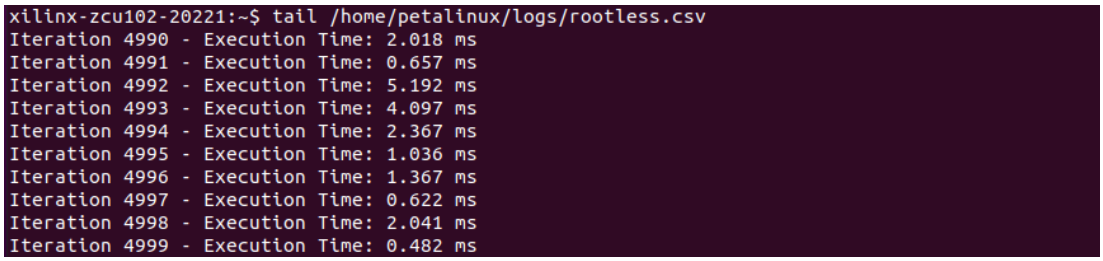


Figure 4.4: Benchmark rt-application iterations and timing log inside Rootless Podman Container

The evaluation of the data shown in Figs. 4.1, 4.3, 4.4 is reported in Par. 4.1.2 as a comparison between the three cases.

4.1.2 Test with *Bare-Metal*, *Rootful* and *Rootless* Containers

After having collected the data, it was possible to apply some pre-processing steps to gather them inside an unique csv file and to feed it to the *JMP* Software to perform some metrics evaluations and graphics drawing.

In more detail, on JMP, the function of *Distribution analysis* was used to analyze the following distributions (as reported in Fig. 4.5):

- Distribution of 5000 Execution Times coming from the *Bare-Metal execution* of the real-time benchmark application;
- Distribution of 5000 Execution Times coming from the *Rootful execution* of the real-time benchmark application;
- Distribution of 5000 Execution Times coming from the *Rootless execution* of the real-time benchmark application;

The metrics collected and analyzed were instead:

- Mean value;
- Minimum value;
- Maximum value;
- Standard Deviation;
- Quantile plot;
- Box-plot graphs

Hence, as it was possible to observe in the statistics reported into Fig. 4.5, it was evident that containers added *latency* and more *jitter*, in comparison with the bare-metal execution. This was evident from the bigger tails that the execution in both containers showed and also in the maximum values of execution times that from a mean value of ~0.543 for the Bare-Metal spiked to ~1.823 for the Rootful and to ~1.474 for the Rootless. This meant that containerization introduced an overhead of ~2.7× (*Rootful*) and ~2.4× (*Rootless*) compared to the *bare-metal* execution.

Furthermore, the outliers introduced by the containers execution were also higher than 100 ms, respect to the bare-metal execution that it was only up to ~5 ms. Such a behavior could demonstrate loss of temporal determinism inside containers.

Realistically, this was the expected output since adding a virtualization layer, even if it was a light one as the one provided by containers. It still introduces delays with respect to the *bare-metal* case, even if sporadic ones, as evinced by the statistics. Anyway, such behavior could be considered slightly dangerous for real-time constraints.

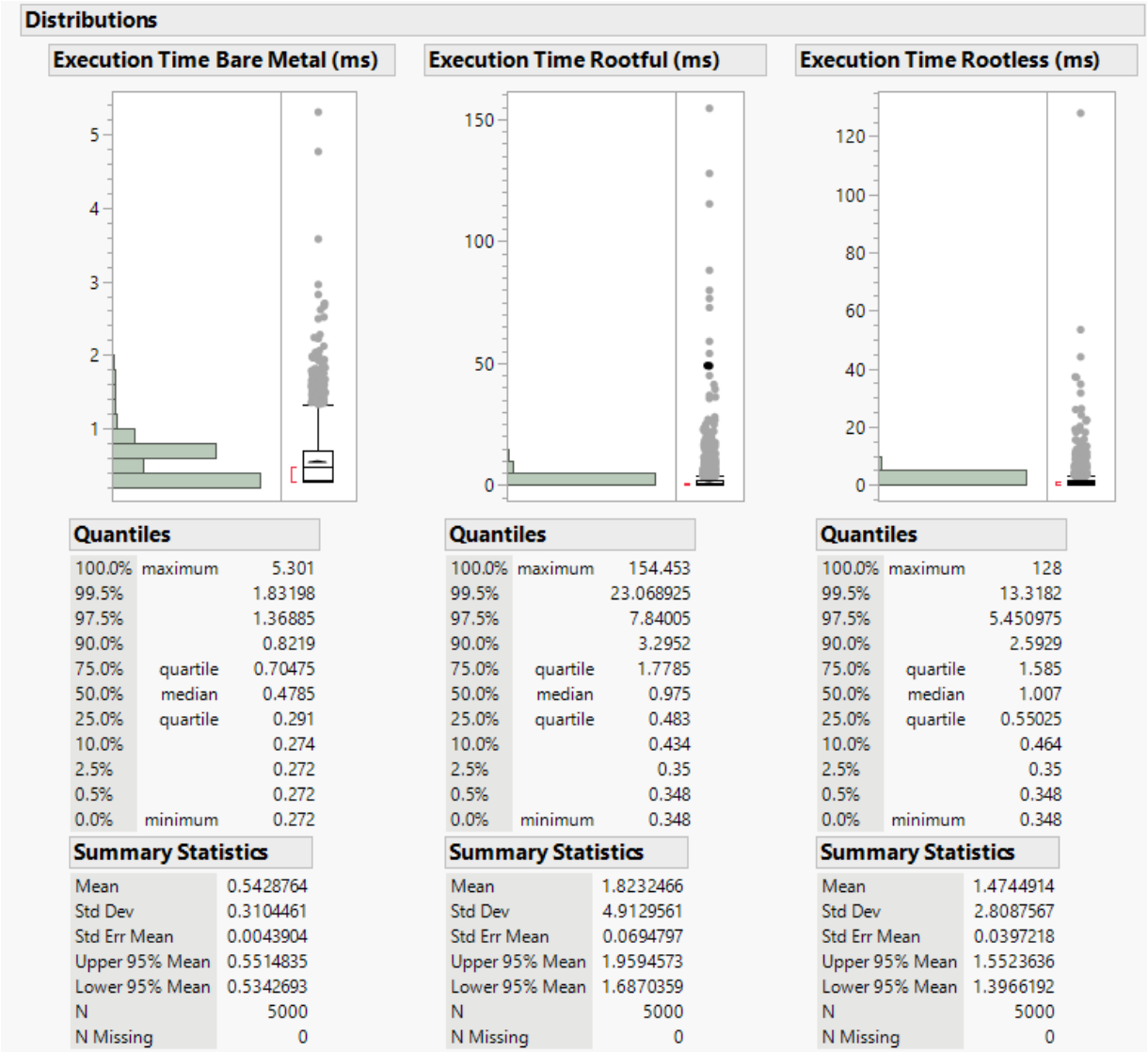


Figure 4.5: Distribution of execution times of a real-time benchmark application repeated 5000 times in three different contexts: bare-metal, inside a Podman Rootful container and inside a Podman Rootless container, all of them on a QEMU-emulated ZCU102 board. The statistics were elaborated with JMP software.

However, the boxplots reported in Fig. 4.5 were not clearly readable because of the big difference in the order of values between the *bare-metal* case and the containers ones. For this reason, a new plot with a *Logaritmic scale* was realized using a *Python* script and it was reported in Fig. 4.6.

It was possible to observe that the *bare metal* execution showed a smaller IQR, that means that the *variance* of data was low, hence that the execution times of application were quite predictable and deterministic, perfect condition for a real-time application. The container's

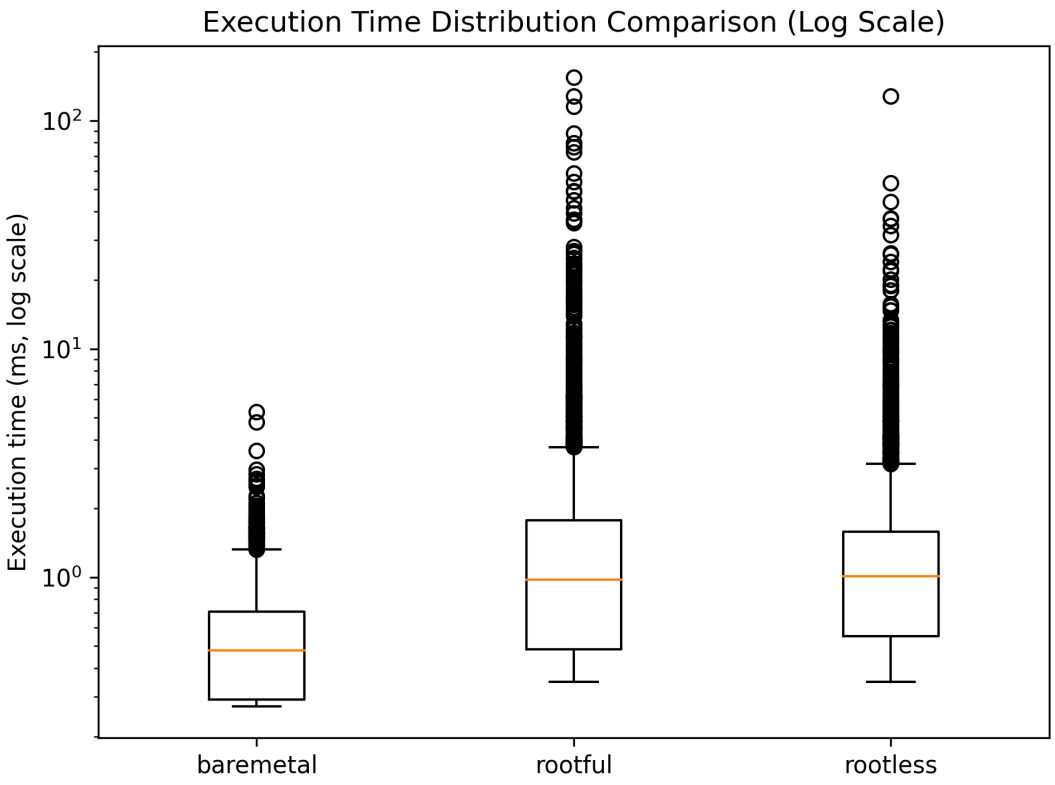


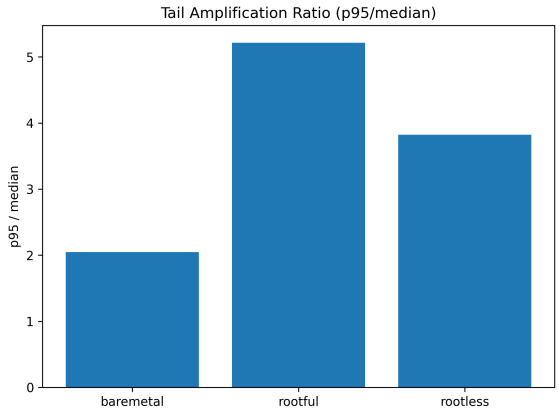
Figure 4.6: Box plot relatives to the experiments reported in the Fig. 4.5. To properly compare the distributions of execution times between the three cases considered, a log-scale was used.

executions instead, showed fatter boxplots, especially the *Rootful* case, highlighting a bigger variance and unpredictability caused by the virtualization layer.

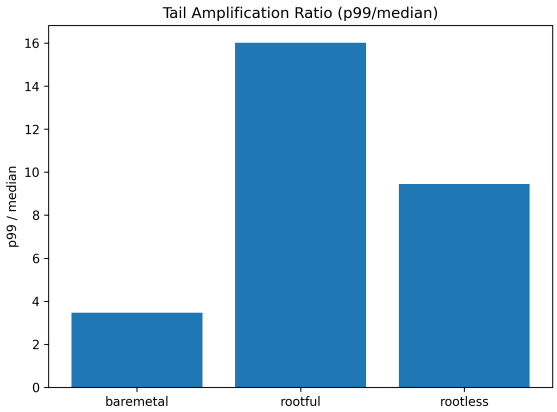
Another interesting metric to keep in consideration was also the *Standard deviation* that also represented a factor of degradation of temporal determinism. In more detail, it appeared to spike from ~0.31 ms for *bare-metal* execution to ~4.92 ms for *Rootful* and to ~2.81 ms for *Rootless* execution.

Still, another interesting aspect to evaluate was the *tail* of the latency-distribution computed on the 5000 execution times collected from each of the three experiments ran.

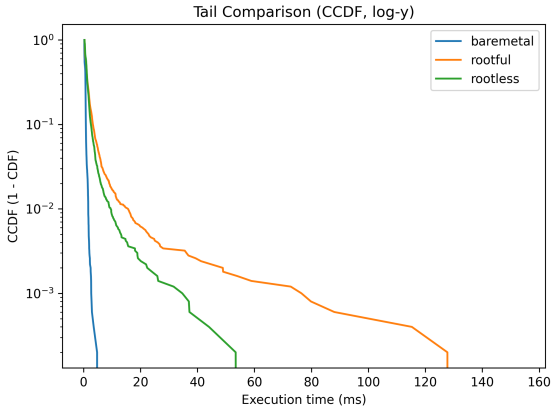
In a latency-distribution, the tail represents the probability mass of high-latency events (i.e. extreme values), while it includes how many extreme events occur, how large they are and how fast the probability decays as latency increases. This means that evaluating the tails of a distribution can be helpful in understanding not only how often the slow events occur but also how fast the probability decays, and in a real-time system this can be translated into *predictability* and *determinism* that influence the *reliability* of the system.



(a) *95th-percentile/median*



(b) *99th-percentile/median*



(c) *CCDF tail comparison: P(latency ≥ x)*

Figure 4.7: Analysis of *tails* in the *latency-distribution* of 2 concurrent Rootless containers in comparison to a single Rootless one

Hence, in the plots in Fig. 4.7, it was possible to observe that the ratio between the 95th-percentile and the median was much greater than 1, especially in the case of the containers and that it became even greater when it came to the 99th-percentile. This meant that latency spikes dominated. In particular, from the 99th-percentile it was possible to affirm that:

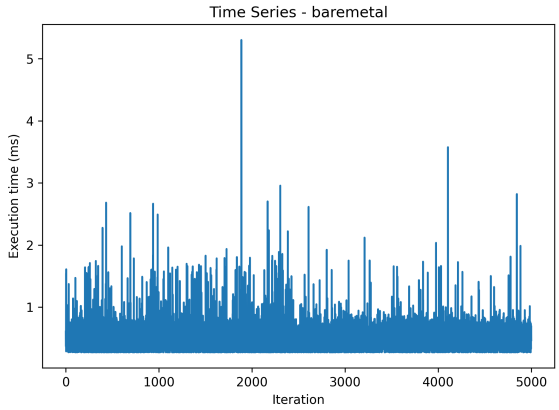
- *Baremetal* case: worst 1% requests were ~3.5× slower than typical;
- *Rootful* case: worst 1% requests were ~9.5x slower than typical;
- *Rootless* case: worst 1% requests were ~16x slower than typical;

and this represented a *massive* tail explosion with consequently instability of the system. Anyway, if the 99th-percentile addressed the behavior of 1% of the events, the 95th-percentile appeared to be less extreme, indicating just sporadic system-level pauses.

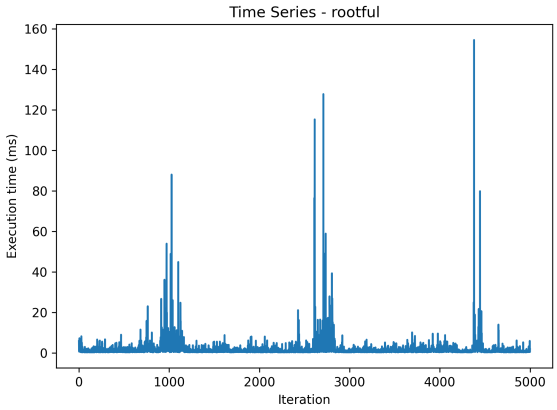
All of this was also visible on the *Complementary Cumulative Distribution Function* (see Fig. 4.7c), where on a `logarithmic-y` scale the `probability of latency` $\geq x$ was shown.

In particular:

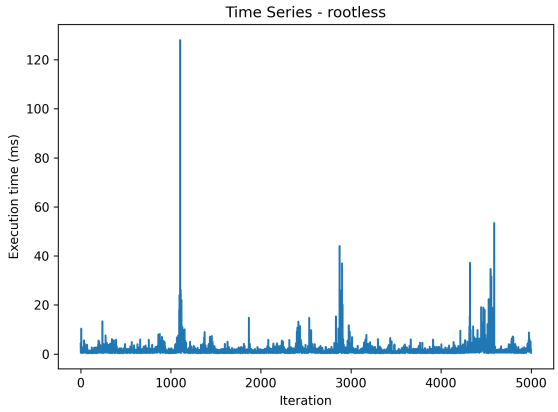
- *Baremetal* case: the tail probability decays very fast (almost vertically) and this indicated that there was no heavy tail behavior;
- *Rootful* case: there was an extremely slow decay evident in the long horizontal stretch;
- *Rootless* case: the decay becomes slower than the *bare-metal* case but it appeared still controlled.



(a) *Time series of Bare-metal execution*



(b) *Time series of Rootful execution*



(c) *Time series of Rootless execution*

Figure 4.8: Analysis of *time series* generated from executing the same real-time application 5000 times on *Bare-metal, Rootful* and *Rootless* containers

As Fig. 4.8 shows, in the case of the *Rootful* execution, there were more high-spikes than in

the case of the *Rootless* one. Furthermore, in both cases, they seemed to quite be periodical and to happen cyclically, differently from the *Bare-metal* execution.

Finding a valid reason to explain all of these behaviors of the system was out the scope of this thesis but the unexpected behavior of the *Rootful* case in comparison to the *Rootless* case could probably be related to the fact that Rootful containers typically involve full cgroup hierarchy, daemon interaction, root privileges interacting with host kernel subsystem, while *Rootless* containers run without a privileged daemon. So, even if counterintuitive, a possible conclusion could be that *Rootless* containers could have slightly higher average overhead but they presented fewer extreme delays.

For these reasons, it was possible to conclude from these results that, even if the average execution time barely changed among the experiments, the worst-case latency got severely worse in containers. Moreover, *Rootless* execution often showed slightly more jitter (maybe due to the extra user namespaces and helper processes), though this depended heavily on the configuration.

4.1.3 Test with 2 Rootful concurrent Containers

In order to prove the capability of the QEMU-emulated system to run multiple Podman containers concurrently and to be indeed useful for the final goal of the *DLR's Space App* project, a test with 2 *Rootful* containers running at the same time was executed (see Par. 3.6.1).

The output of the experiment was compared to the execution times gotten from the execution of a single *Rootful* container, to compare how multiple running containers could affect each other and the system.

In more detail, the *Real_Time_Execution_monitor.c* application was run inside a single *Rootful* container and then inside two different *Rootful* containers running concurrently. The same statistics provided in the experiment in Par. 4.1.2 were collected and analyzed. A preliminary analysis was conducted through the software *JMP*, to compare the main statistics between the three containers (see Fig.4.9 and then also a comparison of the boxplots based on the distribution of execution times (ms) (see Fig. 4.10) was realized.

In particular, it was possible to observe that, counterintuitively, when it comes to the 2 *Rootful* containers running concurrently, the *median latency* slightly decreases (see Tab. 4.1). This suggested that, since the workload tested was not heavily-CPU-bound, it was the container startup or the scheduling effect to dominate. However, as it was also possible to observe from the boxplots (see Fig. 4.10), the maximum latency instead started to increase, meaning that the *tail latency* (i.e. the probability of extreme delays) became worse when running multiple

containers concurrently.

Figure 4.9: Distribution of 5000 execution times of the real-time application executed inside 2 Rootful Containers running concurrently on the QEMU-emulated ZCU102 board. The comparison was made with the execution of a Single Rootful Container.

Another interesting point of view could also be observed in the *CCDF* plot (see Fig. 4.11a), that represented the probability that latency was greater than or equal to a certain x ($P(X \geq x)$). It was worth to notice that one of the two *Rootful* concurrent container did not behave very differently from the single *Rootful* container, while the second one showed a slightly heavier tail behavior.

Container	Median latency (ms)	Mean latency (ms)	Maximum latency (ms)
Single_Rootful	0.975	1.82	154
Concurrent_Rootful_1	0.623	1.27	193
Concurrent_Rootful_2	0.626	1.37	226

Table 4.1: Comparison of how main statistics change from the single *Rootful* to the two-*Rootful* execution

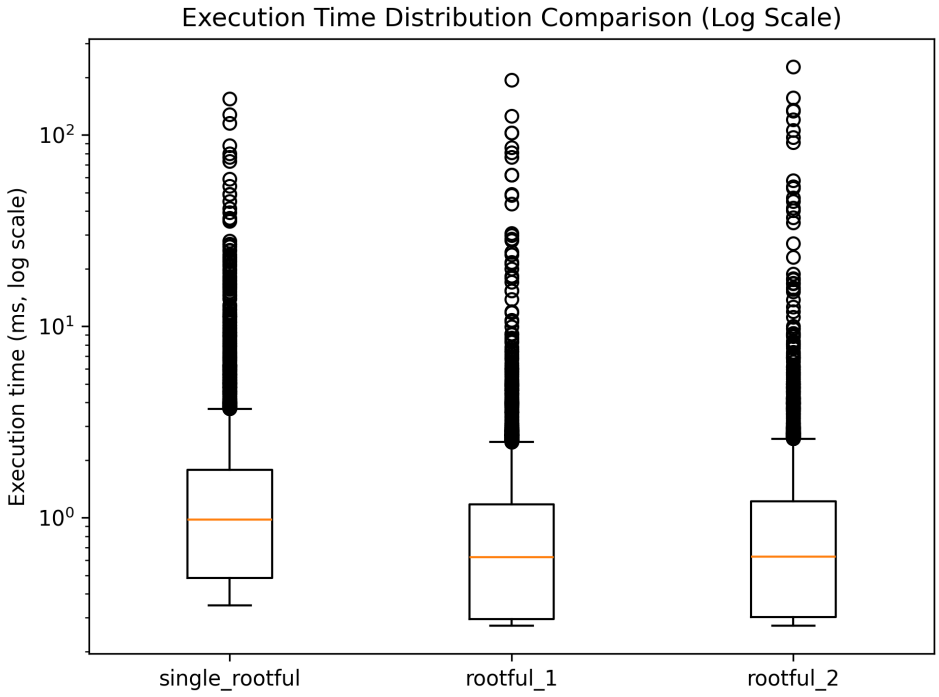
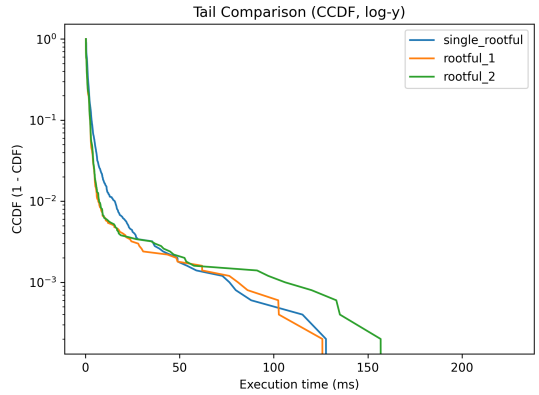


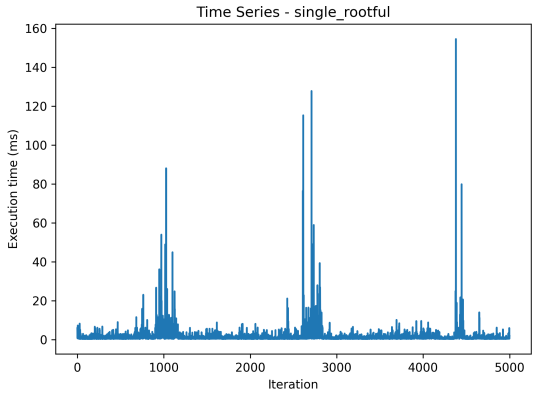
Figure 4.10: Box plot elaborated with a Logaritmic-Scale relatives to the experiments reported in Fig.4.9

Also, from the *time series analysis* (see Fig. 4.11) it was possible to observe that in the case of multiple containers, the spikes were higher - as expected - and they tended to occur in bursts, suggesting the presence of system-level disturbances such as scheduler stalls or kernel write-back activity.

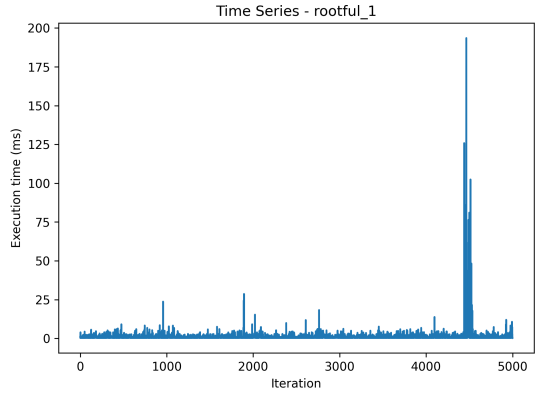
This could be considered an interesting result, but the causes must be further investigated, since the latter was just a hypothesis and finding them was out-of-scope of this thesis.



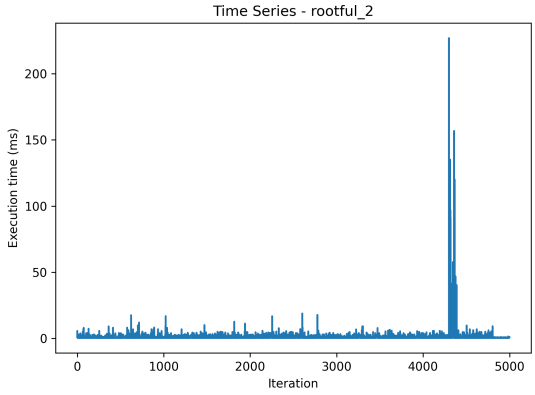
(a) *Complementary Cumulative Distribution Function*



(b) *Time series single Rootful container*



(c) *Time series Concurrent Rootful_1*



(d) *Time series concurrent Rootful_2*

Figure 4.11: Analysis of tails behavior and time series from the distribution of execution times printed by the execution of the application in the case of a single *Rootful* container and in the case of two concurrent *Rootful* containers

4.1.4 Test with 2 Rootless Containers

The real goal of this thesis though, was to test the capability of such embedded system (the *Xilinx AMD ZCU102 board* even if only emulated through *QEMU* and integrated with the *EVL kernel* (of *Xenomai 4* to get real-time capabilities) to run *Rootless* containers. Since the capability of such thing was already proved in Par. 4.1.2 where also the benchmark real-time application was executed successfully inside a *single Rootless* container, then the next step to test was to check if the system could correctly execute multiple *Rootless* containers running concurrently, with the same real-time benchmark application running inside of them if starting at the same time (the synchronization mechanism was provided through a *timing barrier*, see Par. 3.6.2).

Executing multiple *concurrent Rootless* containers indeed, was fundamental for the ultimate

goal of the DLR's project *Stellar Apps* which this thesis is part of. The idea in that case was indeed to enable a system made of multiple nodes (where each node was represented by a board like the one emulated in this thesis) where multiple experiments of different nature could execute concurrently, eventually also with real-time guarantees, without interfering with each other, being directly pulled from a sort of *application store* and so coming also from third-parties.

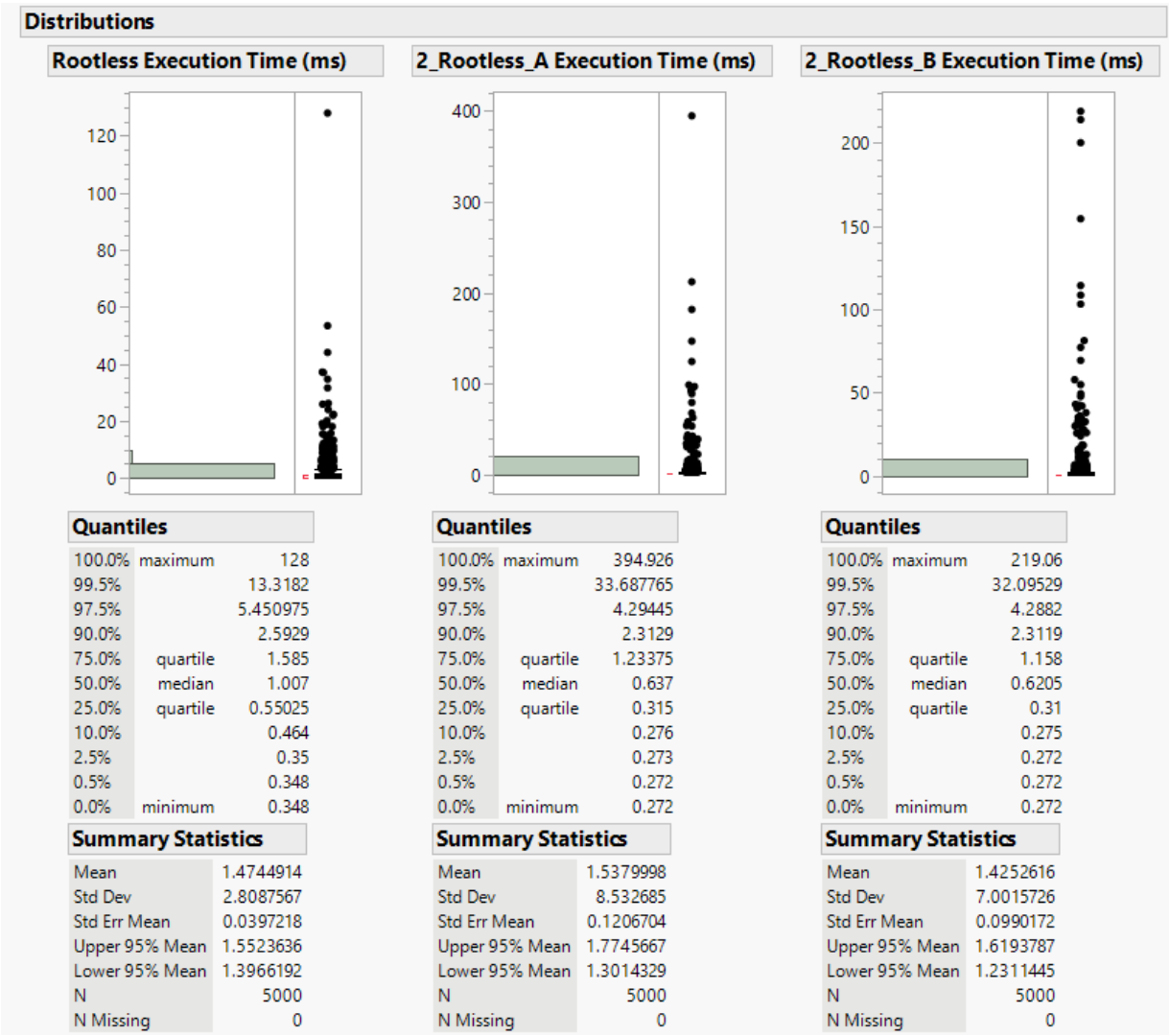


Figure 4.12: A comparison between the distribution of 5000 execution times of the real-time application executed first in a Single Rootless Container and then inside 2 concurrent Rootless Containers on the QEMU-emulated board.

In order to ensure safety and security, containers (like Podman or also Docker) can be used to run the applications. Obviously, having containers that require super-user privileges is not advisable: this is why this thesis wanted to prove the possibility to run real-time applications

also in *Rootless* containers and to run them concurrently. Indeed, they do not require super-user privileges and can so guarantee more isolation and security also in case of malicious applications or cyber-attacks.

Hence, in order to analyze the distributions of execution times written inside some *CSV* files, the test in Par. 3.6.2 was realized. The steps followed were again the same as the previous tests and were thoroughly reported in Par. 4.1.1.

As always, the data from the CSV files were first analyzed through *JMP software*, in order to compute some of the main metrics and statistics. Such values are reported in Fig. 4.12.

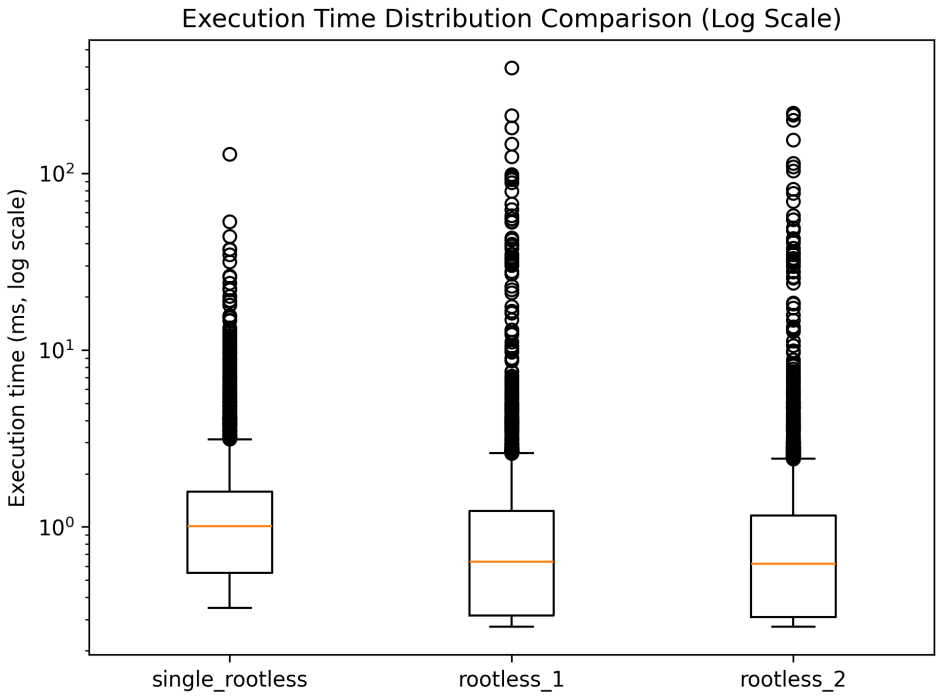


Figure 4.13: Box plot elaborated with a Logaritmic-Scale relatives to experiments in Fig.4.12

It was possible to observe that, as it already happened in the case of multiple concurrent *Rootful* containers, when it came to the 2 *Rootless* containers running concurrently, the *median latency* slightly decreased (see Tab. 4.2). A possible explanation of such behavior could be found in better CPU scheduling or in improved parallelism. Anyway, regardless of the reasons (investigating which was out the scope of this thesis), the interesting aspect to consider was that concurrency did not hurt the average performance. This could be also seen in the boxplots graphs (see Fig. 4.13) where the *Interquartile Range (IQR)* were quite tight, suggesting that the central distribution was more stable than the case of *Rootful* containers. However, instead, as it was also possible to observe from the boxplots (see Fig. 4.10) and as it was also reported in Tab.

4.2, the maximum latency started to increase, meaning that the *tail latency* (i.e. the probability of extreme delays) became worse when running multiple containers concurrently.

In particular, from the table it was also possible to make a comparison between the case with *Rootless* containers and *Rootful* ones. The interesting aspect appeared to be that even if the average behavior of the *Rootless* containers seemed to be quite comparable to the *Rootful* case, suggesting that using or not super-user privileges did not heavily impact the outcome, but at the same time, it was possible to observe a notable spike in the maximum latency that, in the case of the first of the two concurrent *Rootless* containers almost doubled.

Container	Median latency (ms)	Mean latency (ms)	Maximum latency (ms)
Single_Rootless	1.007	1.474	128
Concurrent_Rootless_1	0.637	1.538	394.926
Concurrent_Rootless_2	0.6205	1.425	219.06
Single_Rootful	0.975	1.82	154
Concurrent_Rootful_1	0.623	1.27	193
Concurrent_Rootful_2	0.626	1.37	226

Table 4.2: Comparison of how statistics change from the single *Rootless* to the two-*Rootless* execution, comparing them also with the case of *Rootful* containers

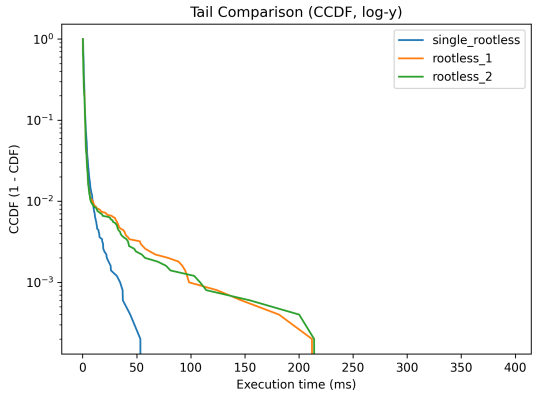
All of this was also evident in the boxplot (see Fig. 4.13) where there were many high-latency outliers that reach ~150-200 ms. This meant that the distribution was *heavy-tailed*, even if its body was quite stable.

To further analyze the tails behavior, the *Complementary Cumulative Distribution Function plot* was provided (see Fig. 4.14a) and it indicated the probability that latency was $\geq x$. Furthermore, since the y-axis was *logarithmic*, tail behavior became very clear. In particular, it was possible to observe that the *Single Rootless* curve was the one that dropped fastest, meaning that extremely latency events were less frequent than in the case of the 2 concurrent *Rootless* containers. Something that was quite interesting though, was the fact that the two curves related to the concurrency were very similar, indicating that, even if concurrent containers introduced variability, the effect was consistent across runs.

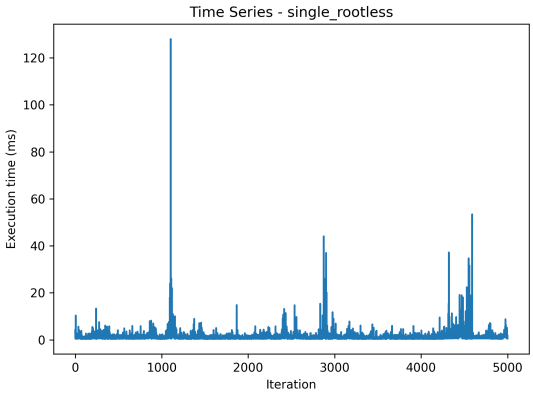
Still, the *time series analysis* provided by the Fig. 4.14, was also useful to find eventual temporal patterns. As it was also observed in the case of the experiment with *Rootless* containers, even here the spikes in the concurrent execution appeared to be higher than the single running container case, but they also appeared to happen in bursts. This could suggest that there could be an event capable of affecting multiple consecutive iterations. Furthermore, it was possible to discern two regimes the system alternated between:

- Normal regime: execution time ~0.5-2 ms;
- Stall regime: execution time ~20-400 ms.

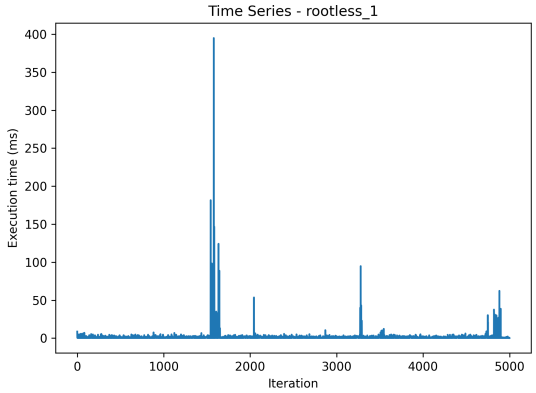
Some possible causes could be a kernel scheduling delay, filesystem write-back or runtime daemon activity. Anyway, as already reported in previous cases, finding these causes was beyond the scope of this thesis, so they may be further investigated as part of future work.



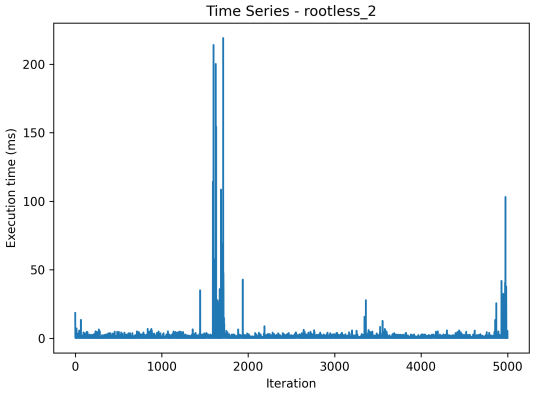
(a) *Complementary Cumulative Distribution Function*



(b) *Time series single Rootless container*



(c) *Time series Concurrent Rootless_1*



(d) *Time series concurrent Rootless_2*

Figure 4.14: Analysis of tails behavior and time series from the distribution of execution times printed by the execution of the application in the case of a single *Rootless* container and in the case of two concurrent *Rootless* containers

4.1.5 Test with 3 mixed (2 Rootless and 1 Rootful) Containers

To further test the system, a mixed execution with 2 *Rootless* containers and *Rootful* was realized. The outputs of such experiment were then compared to the cases of a single *Rootful* and a single

Rootless container running at a time. The metrics and statistics collected are integrally reported in Fig. 4.15, and a summary of the most important ones is reported in Tab. 4.3 instead.

Metrics	Rootful	Rootless_1	Rootless_2	Single_Rootful	Single_Rootless
Mean	2.0148588	3.3425592	3.57897	1.8232466	1.4744914
Min	0.272	0.272	0.272	0.348	0.348
Max	263.037	376.387	431.436	154.453	128
Std Dev	10.118880515	12.03354157	15.271874623	4.9129560931	2.8087567421
Variance	102.39174289	144.80612271	233.23015451	24.1371137573	7.889115536
Quantile_99	21.74424	36.72182	33.6647	15.7368	9.6863

Table 4.3: Metrics to compare the concurrent execution of 3 mixed (2 Rootless and 1 Rootful) Containers

What can be really interesting to observe was the fact that, as it can be observed in Fig. 4.16, the *Mixed Rootful* container had the lowest median latency (~0.35 ms) even lower than the Single-container cases (~1 ms), and the median latency of the *mixed Rootless* containers (~0.8 ms) appeared to be lower than the single-container cases as well (even if higher than the *Rootful* case).

Counterintuitively, this could possibly mean that running multiple containers concurrently could allow a better CPU pipeline utilization, better cache warm-up and also an improved parallel scheduling. Hence, this could be a case of *latency hiding through concurrency*.

However, the difference between the single-cases and the mixed-multiple-case was visible in the *Interquartile Range* that represented the stability of the distribution. Indeed, it was possible to observe from the boxplots that the single-containers (especially the *Single Rootless*) had relatively tight distribution (small boxplot), while the concurrent-case showed wider boxplots, suggesting a larger spread (i.e. *variance*) among execution times, introducing a higher variability and, hence, less predictability in the execution of the real-time application. The outliers appeared to be many in both cases, even if in the concurrent-mixed-case they were slightly higher (there are more high-latency spikes up to ~400 ms). This indicated the presence of *heavy-tailed latency distributions*.

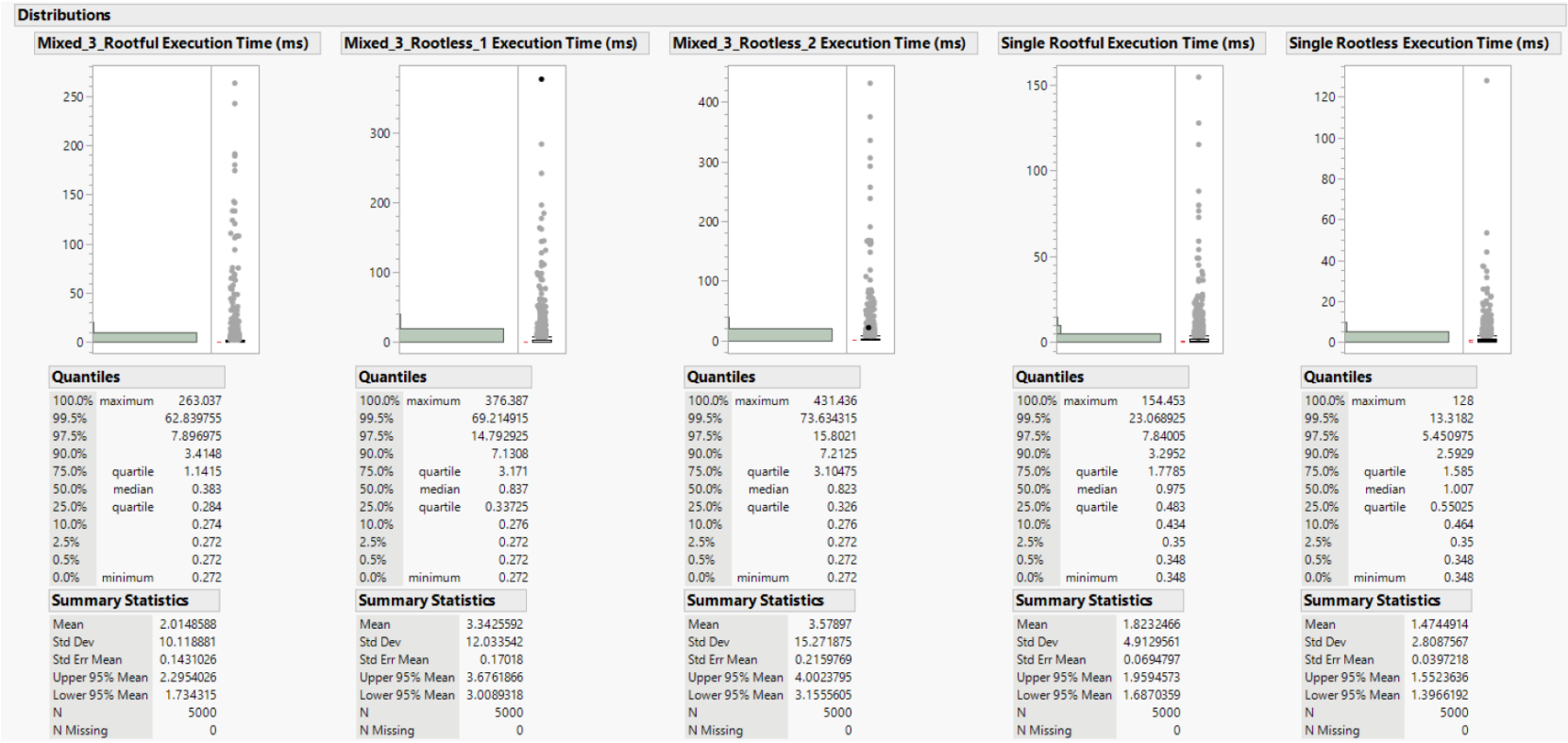


Figure 4.15: Comparison of the distribution of 5000 execution times of the RT-application executed first in the case of a Single Rootful and a Single Rootless Containers running separately and then inside 3 mixed Containers running concurrently on the QEMU-emulated board.

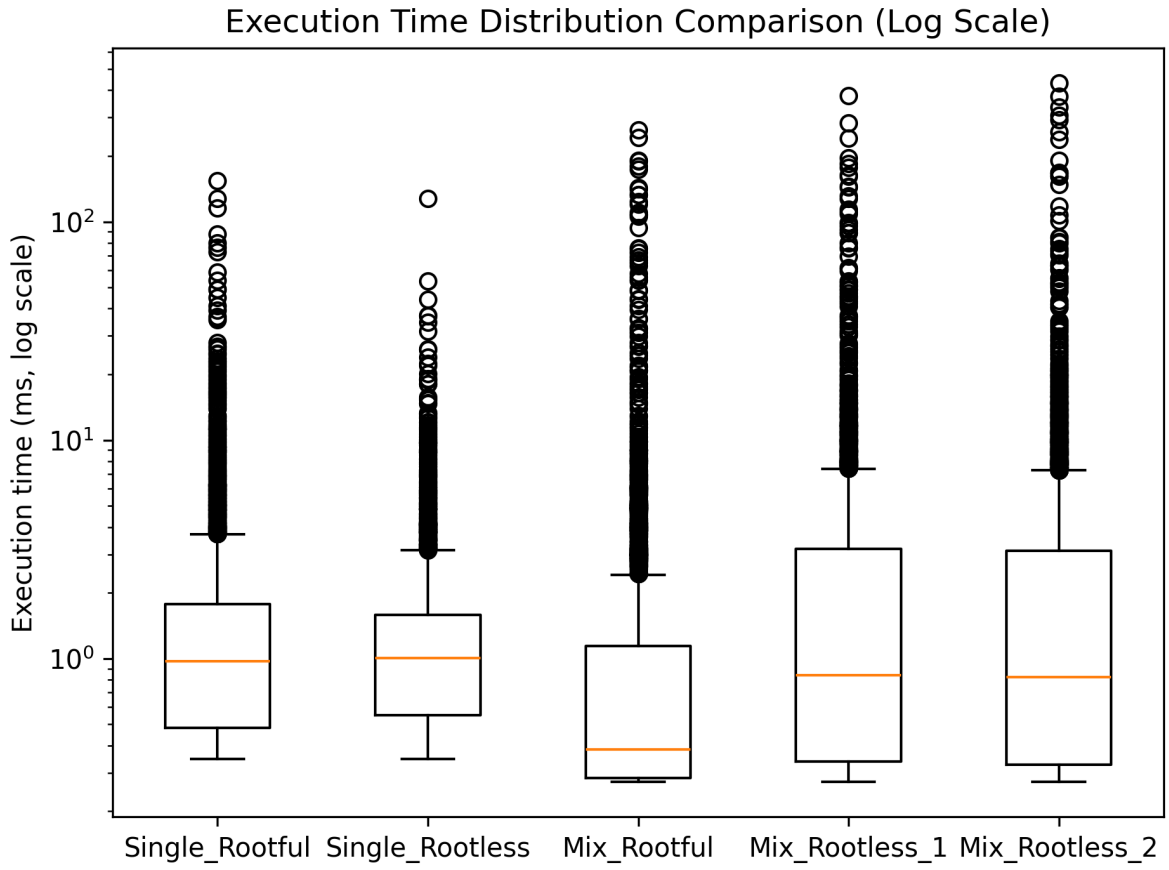


Figure 4.16: Box plot elaborated with a Logaritmic-Scale relatives to the experiments reported in the Fig.4.15

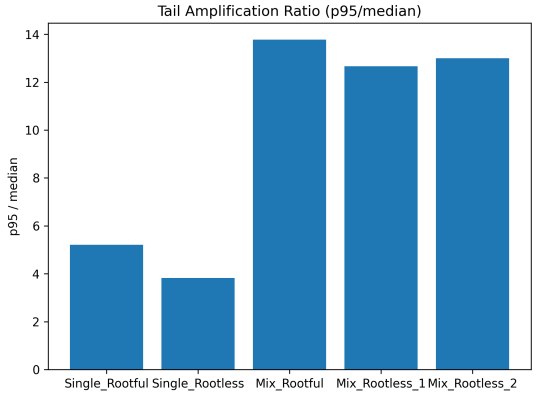
The *tail behavior* can be further analyzed in the *Complementary Cumulative Distribution Function* (see Fig. 4.17c) where it is clear that *Rootless* execution appears to be more stable in the tail when running alone.

Ultimately, it was also useful to investigate the *tail amplification ratios* as in Fig. 4.17. In more detail, the histogram that reported the ratio between 95th-percentile and median latency for each of the containers considered, showed that, under concurrency, the 5% of latency spikes became ~13 larger than the median.

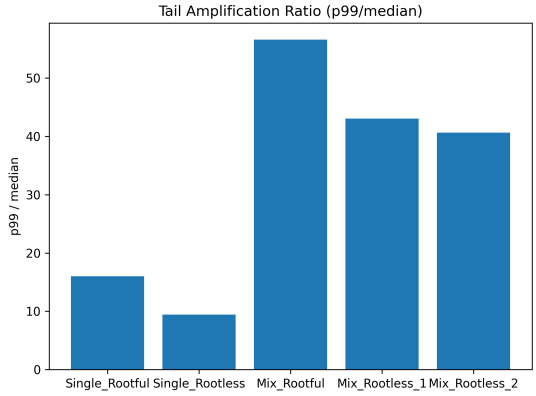
While this became still more dramatic if considering the ratio between the 99th-percentile and median latency, since the worst case appeared to be the *Mixed Rootful* container, where $p99 \simeq 56 x median$, that meant that the slowest 1% of requests were over 50x slower than normal execution. This was a very heavy tail that suggested that the system became much less predictable.

What became evident, not only from this last experiment but also from the previous ones at the previous paragraphs, was that, when containers run concurrently, *Rootful* tails became extremely heavy, while *Rootless* also degraded but less dramatically.

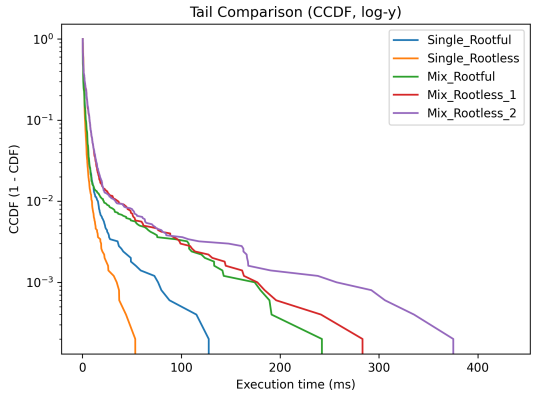
This was a good result for the goal of this thesis, since the ultimate goal was to run multiple *Rootless* containers and not *Rootful* ones.



(a) *95th-percentile/median*



(b) *99th-percentile/median*



(c) *CCDF tail comparison: P(latency ≥ x)*

Figure 4.17: Analysis of *tails* in the *latency-distribution* of 3 mixed concurrent containers in comparison to a single Rootful and a single Rootless one

4.2 Test Scalability of the system with Dummy workload

4.2.1 Collection of big-amount of data

In order to test the scalability of the system and to evaluate how many real-time containers could properly run concurrently using EVL devices and guaranteeing low-latency levels, more configurations with an increased number of containers were tried. In each configuration, all the *Rootless* containers considered executed the *Real_Time_Execution_monitor.c* application.

To launch containers more conveniently, a shell script was prepared, where it is possible to change the number of containers to launch using the variable `NUM_CONTAINERS`. It is mandatory to specify that in order to run the Rootless container with such script, it is necessary to first check if EVL devices belong to the same Linux-group of the user that is running them, otherwise said containers will not get access to the devices and they will not run even if they will still be created.

Such shell script - reported in Appendix B - required some minutes to complete.

In order to analyze the results of the experiment, it was necessary to collect the output .csv files from the QEMU-emulated board and to pass them on the host (virtual machine), from where they could easily be transferred on a PC running *JMP Software*. To automate the transfer of the files, the `scp` (secure copy) command was used, with a *Dropbear RSA key* to use the SSH service available on the board.

In more detail, a Dropbear private key was generated on the QEMU board through the commands:

```
1: mkdir -p /home/petalinux/.ssh
2: chmod 700 /home/petalinux/.ssh
3: dropbearkey -t rsa -s 2048 -f /home/petalinux/.ssh/id_dropbear_rsa
4: chmod 600 /home/petalinux/.ssh/id_dropbear_rsa
```

Then, the public associated key was printed in an OpenSSH-compatible format, for the host (virtual machine) sshd to accept it:

```
1: dropbearkey -y -f /home/petalinux/.ssh/id_dropbear_rsa | sed -n
      's/^ssh-rsa/ssh-rsa/p'
```

On the Virtual Machine, the public key was added to the list of already-accepted keys:

```
1: mkdir -p ~/.ssh
2: chmod 700 ~/.ssh
3: nano ~/.ssh/authorized_keys
4: chmod 600 ~/.ssh/authorized_keys
```

After having tested the success of the workflow on the board with the passwordless login dbclient as shown below

```
1: dbclient -i /home/petalinux/.ssh/id_dropbear_rsa carla@10.0.2.2 'echo OK'
```

it was possible not only to secure-copy all the .csv files from the QEMU-board on the virtual machine using the command below, but also to do it without inserting the host password for every new file transmitted:

```
1: for i in $(seq 1 10); do
2:     scp -i /home/petalinux/.ssh/id_dropbear_rsa \
3:           /home/petalinux/logs/ss_10_rootless_${i}.csv \
4:           carla@10.0.2.2:/home/carla/zcu102_project/xilinx-zcu102-2022.1 \
                 /recv_10_rootless/ss_10_Rootless_${i}.csv
5: done
```

This trick guaranteed the possibility to also easily transfer 100 files in one-go, making possible to further moving towards the test with 20, 50 or 100 containers.

The files thus obtained were then analyzed with a *Python script*, to properly collect the main metrics and draw explicative plots.

4.2.2 From 10 to 64 concurrent Rootless containers

To understand until which point the system could behave like a real-time one, various tests were realized with a different number of Rootless containers. In more detail, different numbers of containers were executed concurrently on the QEMU-emulated board, each one of them running the real-time benchmark application inside. The following configurations were tried:

- 10 concurrent Rootless containers executing the real-time benchmark application;
- 16 Rootless containers;
- 20 Rootless containers;

- 32 Rootless containers;
- 50 Rootless containers;
- 64 Rootless containers.

Then, it is interesting to show how the *fairness of the system* and the behavior of the containers changed accordingly to the increasing number of containers running concurrently. Hence, to do a *variability intra-experiment analysis*, some statistics (like mean, standard deviation, percentile 95...) were collected for every experiment and the cases of 16, 32 and 64 containers were detailed showed in Figs.4.18, 4.19, 4.20.

In particular, the *Coefficient of variation CV* between the containers running in the same experiments (e.g. the CoV valued between the 16 containers running concurrently in the experiment with 16 total containers) was computed as

$$CV = \frac{\text{std(mean\_container)}}{\text{mean(mean\_container)}}$$

and it changed among the experiments as showed in Tab.4.4.

Experiment	Coefficient of Variation
10 containers	0.0149985661
16 containers	0.0213171823
20 containers	0.0198131649
32 containers	0.0282240115
50 containers	0.0324014397
64 containers	0.0315654185

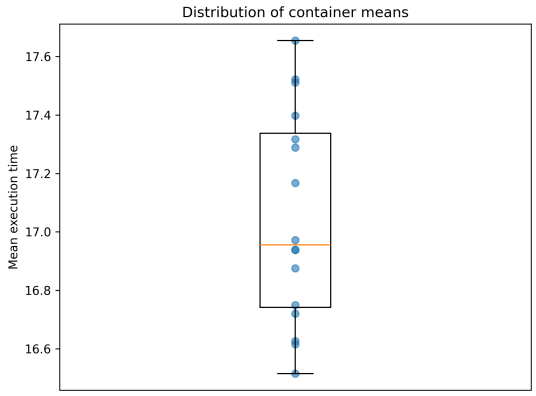
Table 4.4: *Coefficient of Variation* between the containers inside each experiment

In order to further analyze the behavior of the containers among them per experiment, it was necessary to analyze a distribution of distributions, since each container runs 5000 times and, for example, 32 containers were run in one-go for the experiment with *32 Rootless containers*. For this reasons, a Python script was used to produce for each experiment:

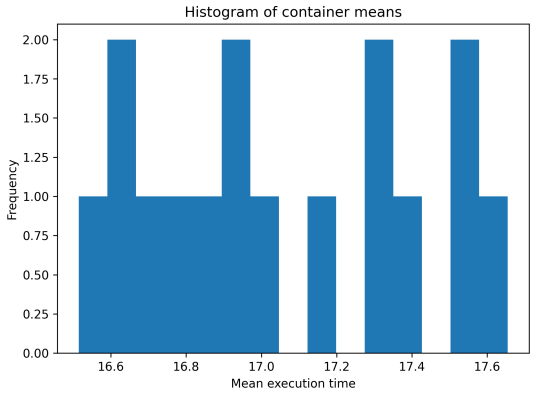
- Distribution of means (single boxplot) to analyze the dispersion between containers;
- Histogram of the means to show the frequency of the means among containers;
- *Lorenz curve* graph to show fairness and inequality of the system behavior;
- Heatmap of main statistics to show some systematical patterns among concurrent running containers.

For these outputs in the cases of 16, 32 and 64-containers experiments, see Figs. 4.18, 4.19, 4.20.

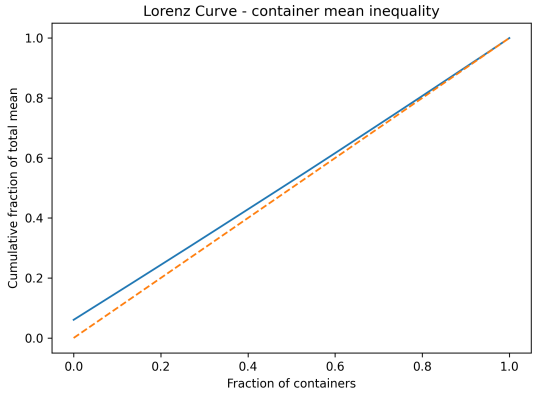
- 16 *Rootless* concurrent containers



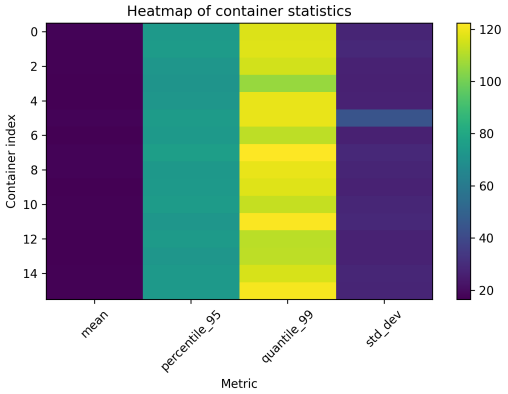
(a) *Distribution of means*



(b) *Histogram of the means*



(c) *Lorenz curve*



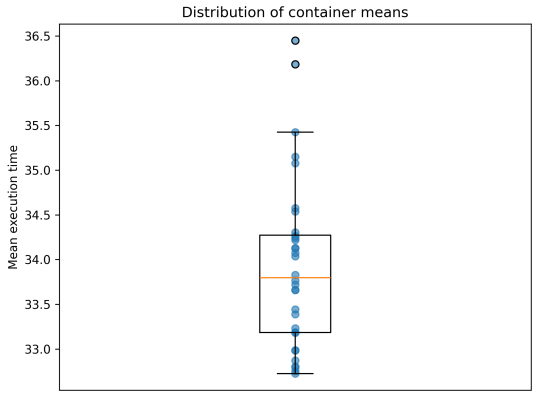
(d) *Heatmap*

Figure 4.18: Analysis of fairness of the systems among containers in the experiment running 16 concurrent Rootless containers

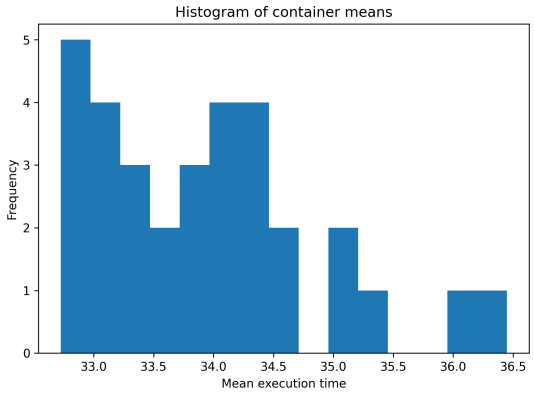
In particular, in the experiment with 16 containers (see Fig.4.18), from the boxplot it was possible to observe a limited dispersion and absence of extreme outliers, confirming homogeneous performance across containers. Furthermore, the narrow *interquartile range* (16,7 - 17,3) suggested that the concurrent execution of 16 containers did not significantly skew the average performance among containers themselves. Such *uniformly distribution* was also confirmed by the histogram that showed a distribution without clear clustering, suggesting that the container performance was not divided into distinct performance classes. Still, the *Lorenz curve* almost perfectly followed the bisector of the graph `Fraction of containers` –

`Cumulative fraction of total mean`, highlighting minimal inter-container inequality in mean execution time. This meant that resource allocation and scheduling were fairly balanced across concurrently running containers.

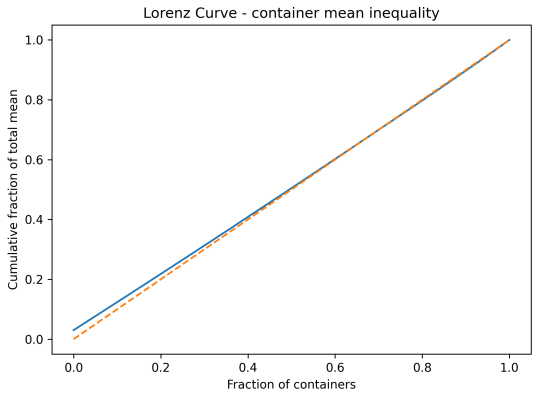
In the end, the heatmap also revealed a relatively uniform distribution of average execution times across containers, indicating balanced mean performance. However, an increased variability was observed in the *99th percentile* and *standard deviation*, suggesting that the *tail latency behavior* was not perfectly homogeneous across containers. In conclusion, this analysis suggested that, even if minor variability was observed in higher percentiles and standard deviation, no significant outliers or multimodal behavior emerged. Hence, the system could be considered fair since it was capable of keeping balanced resource allocation even under concurrent container execution.



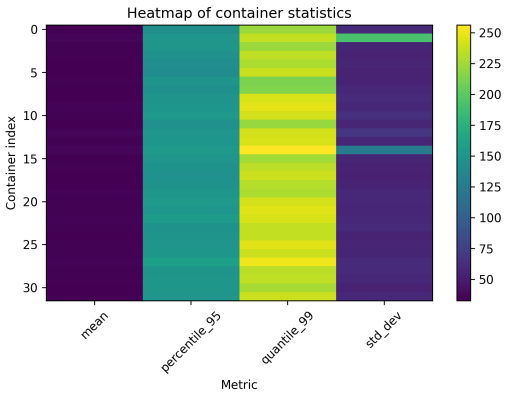
(a) *Distribution of means*



(b) *Histogram of the means*



(c) *Lorenz curve*



(d) *Heatmap*

Figure 4.19: Analysis of fairness of the systems among containers in the experiment running 32 concurrent Rootless containers

- 32 *Rootless* concurrent containers

In the case of 32 concurrent containers instead (see Fig.4.19), it was possible to observe that the increased number of concurrent containers led to higher overall execution times and increased inter-container variability.

Even if mean execution time remained evenly distributed across containers (as suggested by *heatmap* and *Lorenz curve*), the *95th percentile column* showed moderate variation across containers, the *99th percentile* (that indicated tail latency) also showed a stronger variability and the *standard deviation* exhibited visible heterogeneity suggesting that some containers were more unstable than others.

Furthermore, the minor deviation exhibited by the *Lorenz curve* respect to the bisector suggested a gradual increase in inter-container variability compared to the configuration with 16 containers.

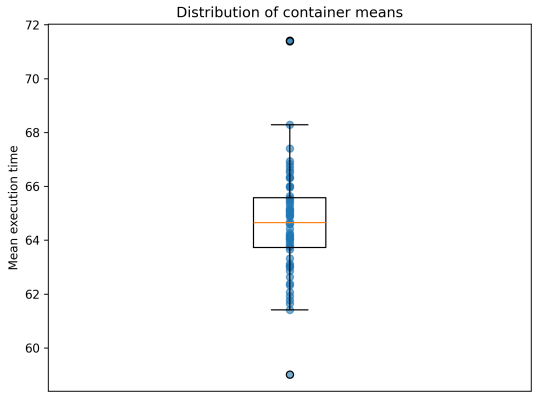
From the *boxplot* it was possible to observe not only an increased median (~33,8-34 ms) as expected, but also a wider *interquartile range* than in the 16-containers case. Also, there were more outliers than the previous case and a slight right skew.

All of this suggested that the performance degradation of the system scaled with concurrency and that the resource contention among concurrent containers primarily impacted tail latency behavior rather than average performance.

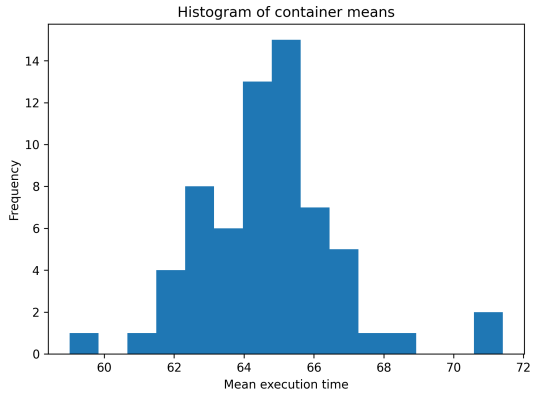
- 64 *Rootless* concurrent containers

The last case analyzed in depth was the experiment with 64 concurrent containers (see Fig.4.20), where some interesting outputs were observed.

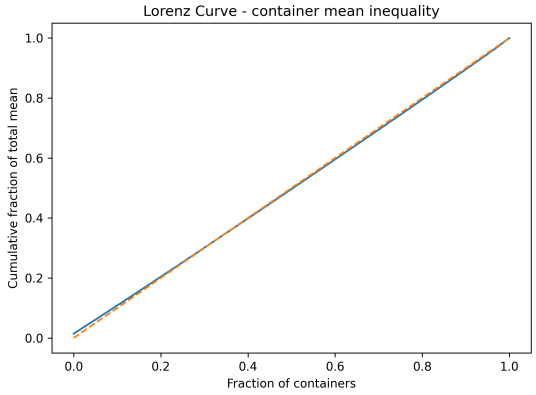
In particular, the *Lorenz curve*, even with an increased number of containers, still seemed to be very close to the bisector of the graph, suggesting that the system remained fair in terms of average execution time and that the degradation of its performances was global and not concentrated. This meant that all containers slowed down together hence, the slowdown is systemic rather than localized. Obviously, the median execution time was significantly increased (up to ~64-65 ms) and also the *interquartile range* was noticeably wider than the previous cases but these were expected conditions in a going-to-degrade system. Also, there were more extreme outliers that indicates resource saturation.



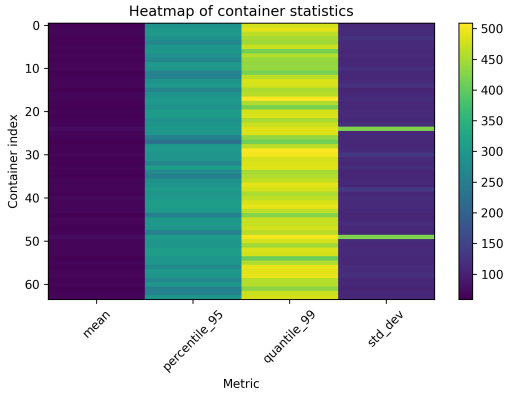
(a) *Distribution of means*



(b) *Histogram of the means*



(c) *Lorenz curve*



(d) *Heatmap*

Figure 4.20: Analysis of fairness of the systems among containers in the experiment running 64 concurrent Rootless containers

4.2.3 Overall comparison between all the experiments

Finally, in order to perform an overall comparisons of system performance and how container - level metrics evolved as the number of concurrent containers increased from 2 to 64, four boxplots reporting between-container distributions across different concurrency level were reported in Figs. 4.21, 4.22, 4.23, 4.24.

In more detail, Fig. 4.21 showed the boxplots representing the distribution per-container mean execution times at a given concurrency-level. The median (horizontal line in the boxplot - 50th percentile) increased almost linearly:

- 2 –> ~1-2 ms
- 10 –> ~10 ms

- 16 –> ~17 ms
- 20 –> ~20 ms
- 32 –> ~34 ms
- 50 –> ~51 ms
- 64 –> ~64-65 ms

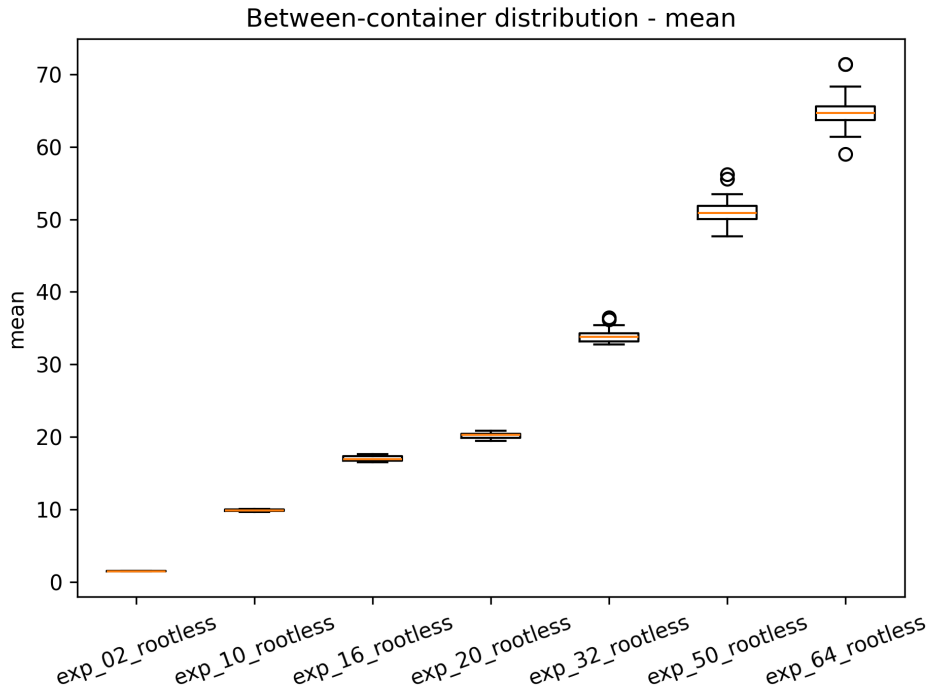


Figure 4.21: Comparison of Boxplots of *means* among various experiments with a different number of concurrent containers running

This meant that the *mean latency* scaled approximately proportionally to the number of containers and this was the typical behavior of a CPU or shared-resource contention, where the latter increased dispersion across containers, as it was visible through the widening of the boxplots at the increasing of the number of containers.

The graph of *95th-percentile* instead, showed that the medians (e.g. the *tail latency*) grew faster than the mean (e.g.*average latency*), because the increasing was around:

- 32 containers –> ~150 ms
- 50 containers –> ~220 ms

- 64 containers –> ~295-300 ms

This meant that as concurrency increased, the system became more sensitive to contention bursts.

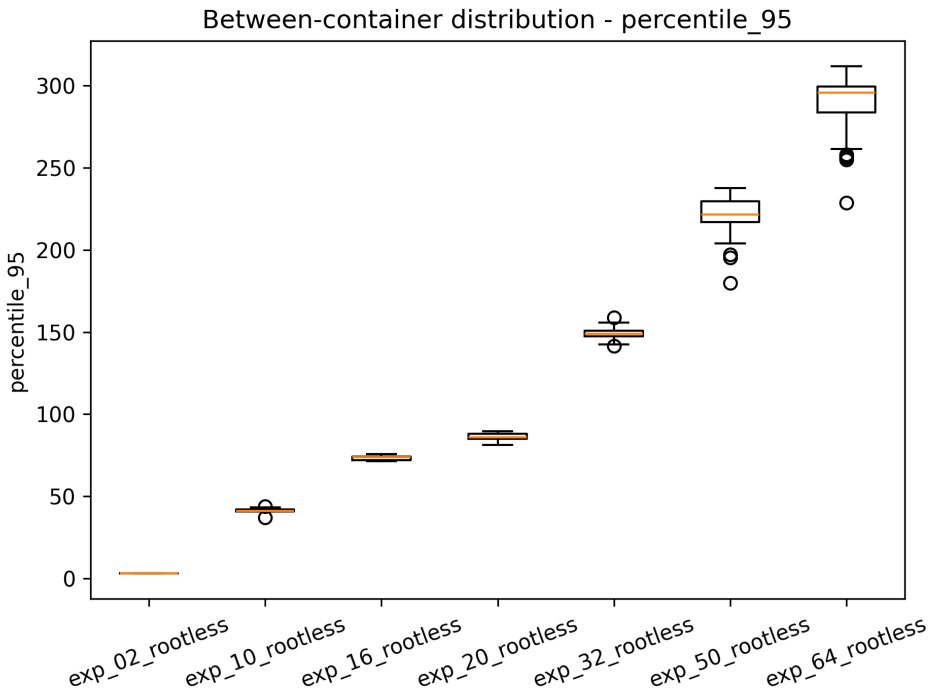


Figure 4.22: Comparison of Boxplots of *95-th percentile* among various experiments with a different number of concurrent containers running

Moreover, the *99th-percentile* instead, represented the *extreme tail* and highlights the real scalability limit of the system. In particular, the linear growth visible as

- 32 containers –> ~235 ms
- 50 containers –> ~350 ms
- 64 containers –> ~470 ms

suggested that the extreme tail latency deteriorated faster than the central tendency, with a much larger *interquartile-range* at 50 and 64 containers and clear extreme outliers. All of this indicated that the increasing concurrency level was starting to affect the *queueing mechanism, scheduling delays* and *cache/memory contention.*

Finally, the *Standard Deviation* between-container distribution represented the *intra-container Variability* that aimed to address how unstable each container was internally. It was possible

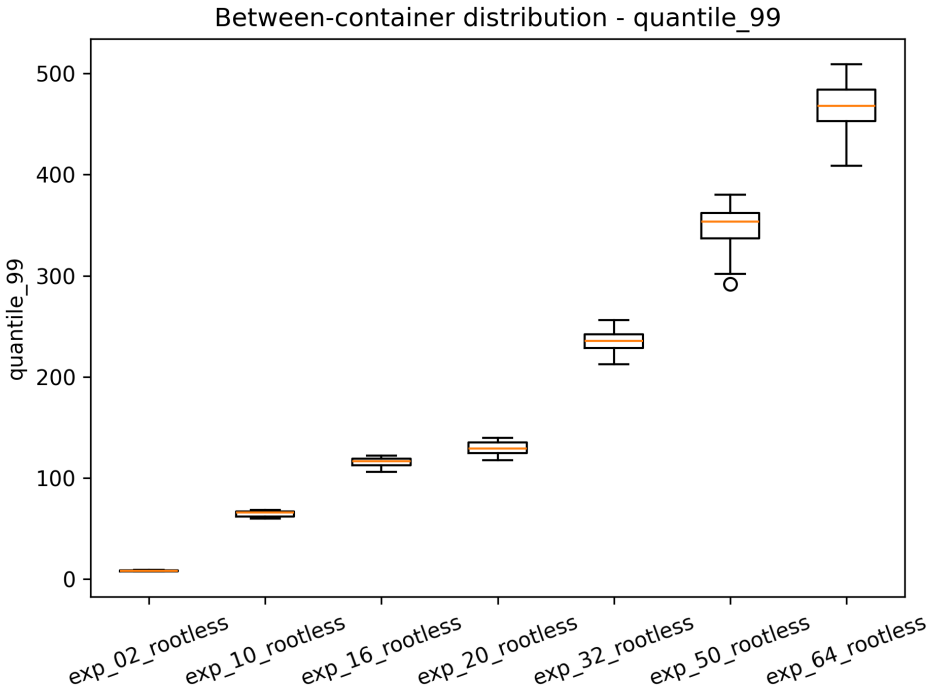


Figure 4.23: Comparison of Boxplots of *99-th percentile (0.99 quantile)* among various experiments with a different number of concurrent containers running

to observe from the plot that such variability appeared as very small at low concurrency but then it started to exhibit a strong increase with 32 containers and almost exploded at 50 and 64 containers. Indeed, with 64 containers, the median standard deviation was ~110 and some extreme outliers (>400) were also present. Then, this showed that not only do containers slowed down but that they also became internally unstable after a certain threshold. This was an expected behavior though when the CPU becomes saturated, when the number of context switches increased, the run queue length grew and the cache thrashing also increased.

Subsequently, it was possible to affirm that the system appeared as stable at low concurrency, but it started to enter a saturation regime between 32 and 50 containers, operating in a stressed state at 64 containers. Despite that, the fairness was maintained.

Hence, in conclusion, it was possible to affirm that as the number of concurrently running containers increased from 2 to 64, three major trends emerged:

1. Linear growth in mean execution time, suggesting increasing resource contention and reduced per-container throughput;

2. Progressive increase in inter-container variability, as evidenced by widening interquartile

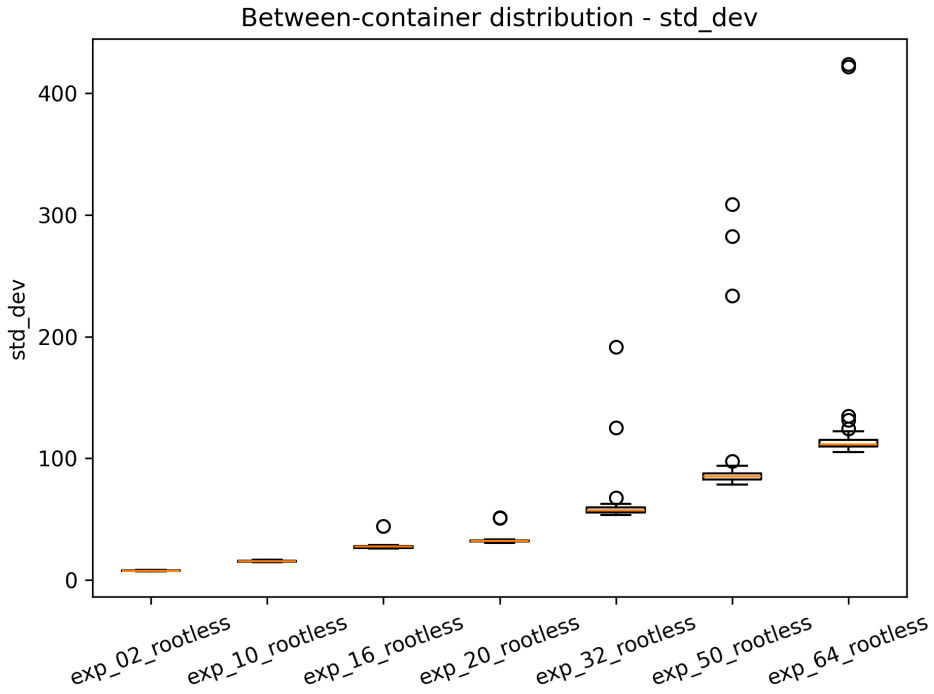


Figure 4.24: Comparison of Boxplots of *standard deviation* among various experiments with a different number of concurrent containers running

ranges and more pronounced dispersion in boxplots;

3. Amplification of tail latency variability, particularly visible in the 99th percentile and standard deviation heatmap columns.

All of this meant that the system showed good fairness, predictable scaling degradation, increasing tail instability but no catastrophic imbalance, features that suggested that:

- Scheduler fairness was maintained;
- Bottleneck was likely CPU or shared memory bandwidth;
- Scalability limit was resource-based, not scheduling-based.

4.2.4 Towards 80-100 concurrent containers

Moreover, in order to really stress the system, also the configuration with 100 concurrent Rootless containers was tested, but it failed with an error (see Fig.4.25).

Due to the high-concurrency rootless container instantiation required to the system, as shown in the error, the system exhibited a RCU (Read-Copy-Update) grace-period stall due to scheduler starvation of the `rcu_sched` kthread.

RCU is a lockless synchronization mechanism widely used in the Linux kernel for protecting read-mostly data structures.

The `rcu_sched` kthread was responsible for managing grace periods instead. This meant that the stall of the latter caused a cascade of errors, like:

- Memory that should be freed remained pinned;
- Slab growth increased;
- Reclaim became ineffective;
- Eventually, the system entered Out-Of-Memory conditions.

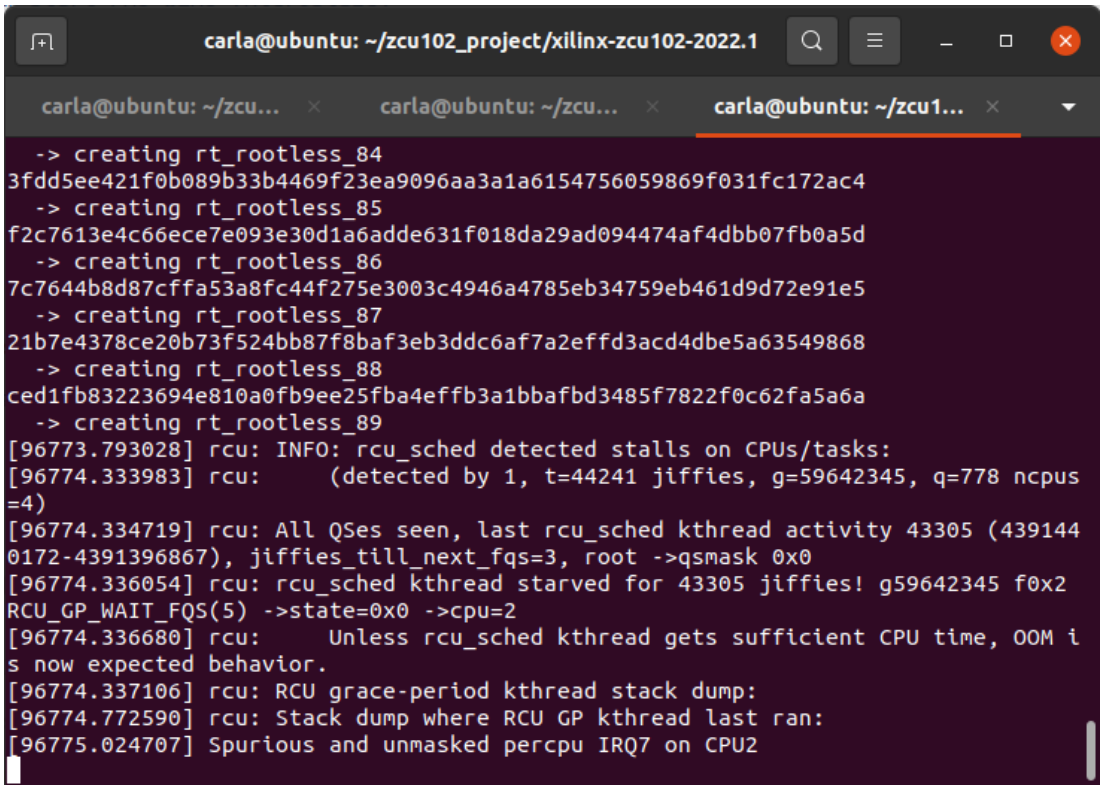


Figure 4.25: Error when trying to create and start 100 concurrent Rootless containers

Such conditions were mostly caused by the fact that launching 100 rootless containers simultaneously caused a massive namespace instantiation (PID, mount, user, network namespaces and cgroup structure for each of them), filesystem and overlayfs pressure, and, finally, an

extreme - even if very short - scheduling spike due to the synchronus waking up of all of the containers. In the end, the error was due to a scheduler saturation that caused synchronization subsystem starvation under extreme concurrency and it indicated that the platform had exceeded its concurrency scalability threshold under the tested workload. This was related to the low power that the emulated system exhibits: 4 cores, limited memory bandwidth, SD-backed storage.

After this experiment failed, a further test with *only* 80 containers was conducted, but it failed with a similar error (see Fig.4.26. This error was completely related to the previous one and it showed that the container runtime failed to complete namespace and cgroup initialization within the libpod lifecycle timeout window. This resulted in a container creation timeout error that highlighted the impossibility to start so many containers at the same time in this type of (emulated) system.

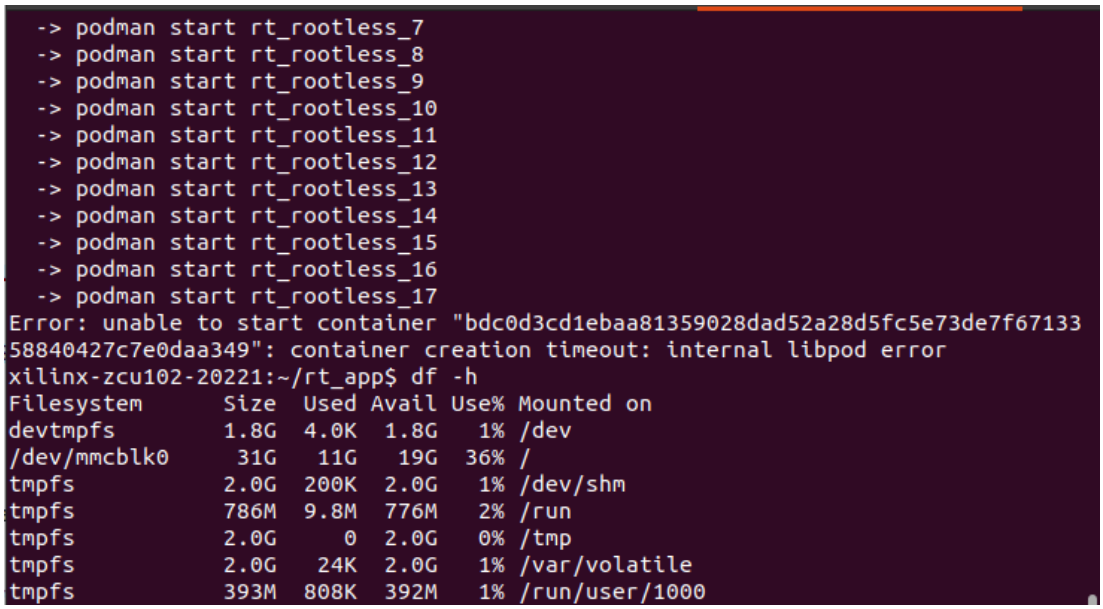


Figure 4.26: Error when try starting 80 concurrent Rootless containers

This proved that 64 concurrent containers (last numbers of containers correctly executed) could be considered the limit for this kind of system.

It is mandatory though to specify that the limits of the QEMU-emulated board were found with this number of concurrent containers also because of the synchronization barrier provided, based on the *polling loop* executed by each containers after its creation in order to check if the barrier file (*GO file*) for them to start has been created. This mechanism woke up the containers every 0.001 s and, on a system with 4 ARM A53 cores, when the number of containers increased,

it became too CPU-heavy and consuming, as it could be seen in the error previously shown (Fig.4.25. Probably, if a different synchronization mechanism had been used, the limit at the number of concurrent containers running would have been higher.

Still, it is important to keep in consideration the type of real-time application running inside each container: the application provided could be considered a *dummy* one, very low-consuming on CPU and memory. The *scalability test* conducted in this thesis aimed to only measure the feasibility and the impact of real-time containers running concurrently. However, in the case of the DLR's project *Stellar Apps*, it is highly probable that the real-time applications running inside the containers will be more memory and CPU consuming but, at the same time, the number of containers running concurrently in the system will be less than 64. This means that further investigation of a different synchronization mechanism is needed.

Chapter 5

From emulation to the real board

The following chapter describes how the system developed in the previous chapters was in the end deployed on a real *AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Platform* available to use in the laboratory of *University Federico II of Naples*.

In particular, first an overview of the board itself and how the connection to it was realized is proposed in Par. 5.1. Then, among the files produced by the previous steps, some were selected and collected to be copied on an SD Card - which required a formatting and a previous preparation - in order to deploy the system on the real board, as described in Par. 5.3. Subsequently, some execution examples of the implemented system, like the real-time application running inside containers and on bare-metal, are presented in Par. 5.4. Finally, Par. 5.5 provides an initial evaluation of the results obtained on the real board and compares them with those observed in the emulated setup.

5.1 Real board target and connection

After having proved that it was feasible to implement a working system based on the Linux EVL kernel and with real-time Podman containers enabled on an emulated embedded platform (via QEMU), the final step was to deploy all of this framework on the actual target hardware.

The *AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Platform* used in this thesis was accessible through a server machine named *hisa-origami*, which acted as a gateway to the board. For this reason, the connection was established via SSH to the gateway machine, using a username and public key authentication.

From the gateway, the access to the ZCU102 board —located within the same local network — was obtained through the *screen* utility, which provided access to the *board's serial console*.

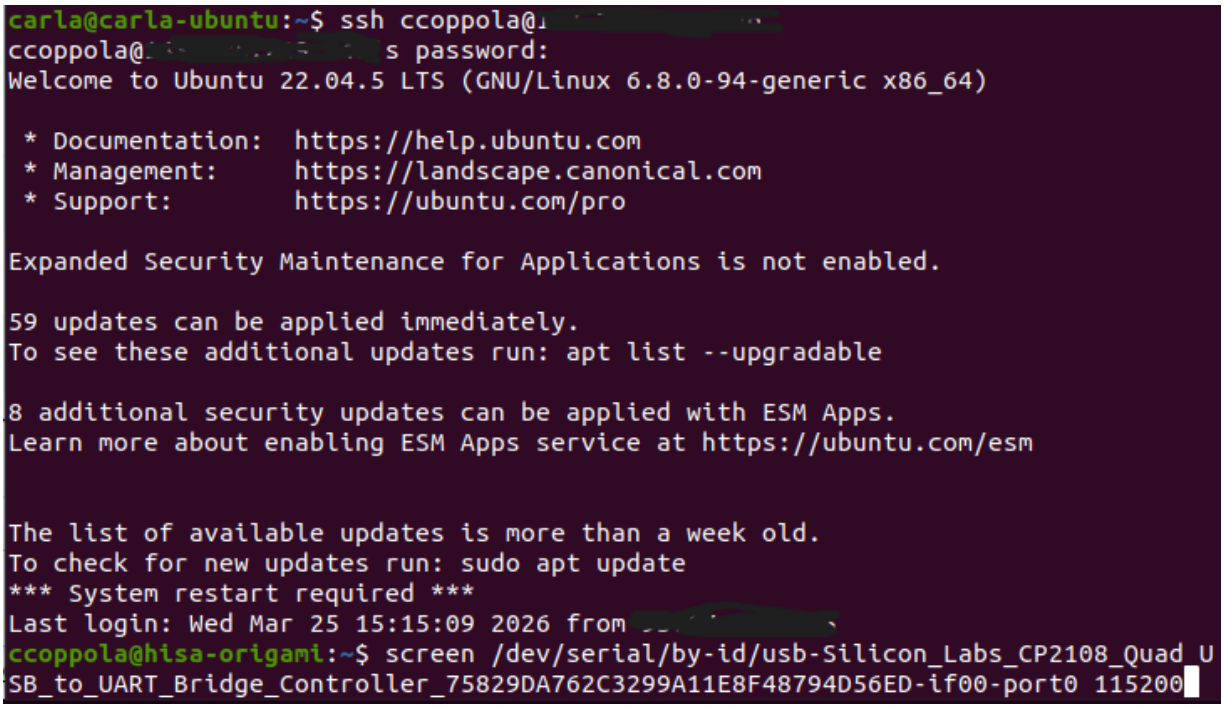


Figure 5.1: Connection to the gateway (server *hisa-origami*) to access the *real* ZCU102 board through the *serial* connection

As illustrated in Fig. 5.1, it was necessary to identify the correct *serial port* and configure the appropriate *baud rate* in order to properly interact with the board.

At that point, being the connection bi-directional, it was possible to send commands to the board through the serial access and to interact with it that way as better explained in Par. 5.4.

5.2 Collection of files needed to import the project on the real board

To deploy the project on the physical ZCU102 board, it was necessary to prepare a minimal set of files to be placed on an SD Card to reproduce the boot architecture of the platform.

In more detail, unlike the QEMU-based setup, where each firmware component was explicitly loaded via command-line arguments, the real board relies on a predefined boot sequence driven by the on-chip BootROM and U-Boot. For this reason, it was only necessary to ensure that the SD Card contained, at the end of the process, four essential files: *BOOT.BIN, Image, system.dtb*, and *boot.scr*.

In particular, the *BOOT.BIN* file provided by the PetaLinux BSP (located at the path `~/zcu102`

`_project/xilinx-zcu102-2022.1/pre-built/linux/images/BOOT.BIN`), was a composite binary that bundled the PMU firmware, ARM Trusted Firmware (BL31), and U-Boot; it was needed because it replaced the manual loading performed in QEMU.

Then, the *Image* file (at the path `~/zcu102_project/linux-evl/arch/arm64/boot/Image`) represented the actual EVL-enabled Linux kernel compiled during the project and was therefore required to run real-time workloads.

Still, the *system.dtb* file (located at `~/zcu102_project/pre-built/linux/images/system.dtb`) was also needed not only because it described the actual hardware configuration of the ZCU102, but also because it was previously patched to include the necessary kernel *boot arguments* (e.g., enabling cgroup v2), required to properly run Podman containers on the real board.

Finally, the *boot.scr* file (at the path `~/zcu102_project/pre-built/linux/images/boot.scr`) ensured that the loading of the kernel and device tree from the SD card into memory happened correctly, triggering the boot process.

All the steps followed to first prepare the SD card and then to put these files on it are precisely described in Par. 5.3.

In addition to these boot files, also the root filesystem image (rootfs.ext4) (located at path `~/zcu102_project/images/linux/rootfs.ext4`) had to be put on the SD card. Indeed, it included all user-space components such as the EVL libraries, kernel modules, Podman and the RT-application code.

Together, these files provided a complete and self-contained system image: the bootloader initialized the platform, U-Boot loaded the kernel and hardware description, while the kernel mounted the root filesystem, enabling the execution of real-time containerized applications on the target hardware.

5.3 Preparation of SD Card

Initially, a vanilla-Linux kernel was loaded on the SD Card of the board and so, in order to make the Linux EVL work, it was necessary to format the SD Card and put the customized system on it.

First, the provided SD Card of 16 GB was inserted into the host system (VM) where, launching the shell command `lsblk`, it was possible to see that it was recognized by the system as `/dev/sdb` with 2 partitions. In particular though, to install the provided system on it, it was necessary for the SD Card to get 2 partitions of a specific type and dimension, as follows:

1. `sdb1` FAT32 type of ~512 MB to host *boot files* (BOOT.BIN, Image, DTB, boot.scr);

2. `sdb2` ext4 type with the rest of GB (~15 GB) to host the *root filesystem*.

For this reason, some cleaning and repartition steps had to be executed on it, using the following commands:

```
1: sudo umount /dev/sdb1
2: sudo umount /dev/sdb2
3: sudo fdisk /dev/sdb
```

Inside *fdisk utility* that was opened by the last previous command, some operations had to be run in order to first delete the old partitions and then to recreate them with the customize dimension allocated. So, inside fdisk:

```
1: d # delete partition 2
2: d # delete partition 1
3: n # new partition
4: p # primary
5: 1 # partition 1
6: <enter> # default start
7: +512M # size
8: n
9: p
10: 2
11: <enter>
12: <enter> # use remaining space
13: t # change type
14: 1 # select partition 1
15: b # W95 FAT32
16: w # write changes
```

After this repartition, the output of `lsblk` changed as Fig. 5.2 shows.

At that point, the partitions could be formatted as follows:

```
1: sudo mkfs.vfat -F 32 -n BOOT /dev/sdb1
2: sudo mkfs.ext4 -L rootfs /dev/sdb2
```

And then they could be mounted in order to copy the files previously selected on the SD Card:

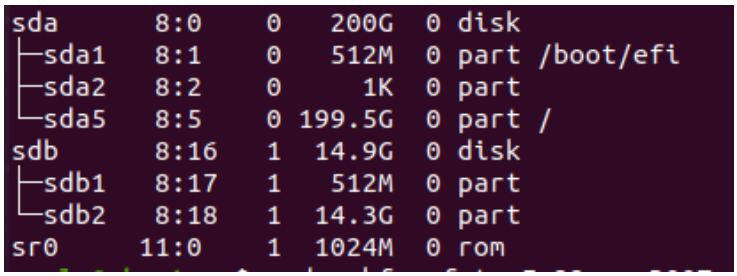


Figure 5.2: SD Card partitions after repartition

```
1: sudo mkdir -p /mnt/sdboot
2: sudo mount /dev/sdb1 /mnt/sdboot
3:
4: PROJ=~/zcu102_project/xilinx-zcu102-2022.1
5:
6: sudo cp $PROJ/pre-built/linux/images/BOOT.BIN /mnt/sdboot/
7: sudo cp ~/zcu102_project/linux-evl/arch/arm64/boot/Image /mnt/sdboot/
8: sudo cp $PROJ/pre-built/linux/images/system.dtb /mnt/sdboot/
9: sudo cp $PROJ/pre-built/linux/images/boot.scr /mnt/sdboot/
10:
11: sync
12: sudo umount /mnt/sdboot
```

Therefore, the *filesystem* could be written, but the first attempt ended in an error due to the fact that the filesystem taken from the final system implementation had a dimension equal to 32 GB, because it was resized twice (the first to 4 GB and the second to 32 GB) to host the multiple containers on it to test the scalability of the system. Hence, the problem arose from the fact that the dimension of the physical SD Card was *only* 16 GB and it could not host 32 GB of filesystem (moreover, on the sdb2 partition which was *only* 14,3 GB).

For this reason, it was first necessary to shrink the filesystem. Indeed, not all 32 GB were actually used and, in more detail, using the command

```
1: e2fsck -f $PROJ/images/linux/rootfs.ext4
```

it was found out that *only* 2669361/8388608 blocks were used by the system on the filesystem, as shown in Fig. 5.3. Since it was an `ext4` kind of filesystem, each block is 4 KB, therefore the space really used was:

```
carla@ubuntu:~$ e2fsck -f $PROJ/images/linux/rootfs.ext4
e2fsck 1.45.5 (07-Jan-2020)
/home/carla/zcu102_project/xilinx-zcu102-2022.1/images/linux/rootfs.ext4: recovering journal
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
/home/carla/zcu102_project/xilinx-zcu102-2022.1/images/linux/rootfs.ext4: 261940/7868416 files (0.1% non
-contiguous), 2669361/8388608 blocks
carla@ubuntu:~$ resize2fs $PROJ/images/linux/rootfs.ext4 12G
resize2fs 1.45.5 (07-Jan-2020)
Resizing the filesystem on /home/carla/zcu102_project/xilinx-zcu102-2022.1/images/linux/rootfs.ext4 to 3
145728 (4k) blocks.
The filesystem on /home/carla/zcu102_project/xilinx-zcu102-2022.1/images/linux/rootfs.ext4 is now 314572
8 (4k) blocks long.

carla@ubuntu:~$ truncate -s 12G $PROJ/images/linux/rootfs.ext4
carla@ubuntu:~$ e2fsck -f $PROJ/images/linux/rootfs.ext4
e2fsck 1.45.5 (07-Jan-2020)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
/home/carla/zcu102_project/xilinx-zcu102-2022.1/images/linux/rootfs.ext4: 261940/2950656 files (0.1% non
-contiguous), 2361369/3145728 blocks
```

Figure 5.3: Check of blocks used by the filesystem and resize of it to be flashed on an SD Card of 16 GB

$$2{,}669{,}361 \cdot 4\,\text{KB} \approx 10.2\,\text{GB}$$

This meant that the filesystem only needed $\sim$ 10 GB and that shrinking its dimension to 12 GB could be considered totally safe. As shown in Fig. 5.3, the resize was operated as:

```
1: resize2fs $PROJ/images/linux/rootfs.ext4 12G
2: truncate -s 12G $PROJ/images/linux/rootfs.ext4
3: e2fsck -f $PROJ/images/linux/rootfs.ext4
```

In the end, having a filesystem which dimension was finally smaller than the sdb2 partition, it was possible to actually flash it on the SD Card, as follows:

```
1: sudo dd if=$PROJ/images/linux/rootfs.ext4 of=/dev/sdb2 bs=4M status=progress
2: sudo sync
3: sudo e2fsck -f /dev/sdb2
4: sudo resize2fs /dev/sdb2
```

The SD Card was then ready to be inserted into the board.

5.4 Execution of the implemented system on the real-board

Before truly executing the system on the real board, it was appropriate to check that all and only the files required with all the proper bootargs were correctly present on the SD Card.

```
1: sudo mount /dev/sdb1 /mnt/sdboot
2: ls -lh /mnt/sdboot
```

The output showed, as expected, only the 4 files BOOT.BIN, Image, system.dtb, boot.scr.

Subsequently it was necessary to check for *bootargs* in order to ensure the proper working of Podman Containers.

```
1: grep -A3 -B3 "bootargs" /tmp/system.dts
```

The previous command, as also reported in Fig. 5.4, showed that, even if the part about the cgroup was already set, a critical part was missing. In particular, in order to make the kernel boot and know where the root filesystem was on the SD card, it was mandatory to also specify the bootarg `root=/dev/mmcblk0p2 rw rootwait`. Indeed, the first attempt to launch the system simply by inserting the SD card in the board and starting it resulted in a *kernel panic*.

For this reason, a reboot was needed and, the autoboot was stopped, giving the possibility to launch the command:

```
1: setenv bootargs "earlycon console=ttyPS0,115200 clk_ignore_unused init_fatal_sh=1
      root=/dev/mmcblk0p2 rw rootwait systemd.unified_cgroup_hierarchy=1
      cgroup_no_v1=all"
2: boot
```

This step allowed to properly start the system and to use the same credentials already set for the emulated system to enter, finding, as expected, the same files and folders already present.

At that point, it was possible to interact with the system in the same exact way as for the QEMu-emulated board.

```
carla@ubuntu:~$ grep -A3 -B3 "bootargs" /tmp/system.dts

        chosen {
                nvmem0 = "/axi/i2c@ff030000/i2c-mux@74/i2c@0/eeprom@54";
                bootargs = " earlycon console=ttyPS0,115200 clk_ignore_unused init_fatal_sh=1 systemd.un
ified_cgroup_hierarchy=1 cgroup_no_v1=all";
                stdout-path = "serial0:115200n8";
        };
```

Figure 5.4: Bootargs to boot the pure cgroup v2 system to enable Podman containers

```
xilinx-zcu102-20221:/home/petalinux/logs# head -10 ss_timing_log_with_load_evl_b
ash.csv
Iteration 0 - Execution Time: 0.702 ms
Iteration 1 - Execution Time: 0.514 ms
Iteration 2 - Execution Time: 0.500 ms
Iteration 3 - Execution Time: 0.512 ms
Iteration 4 - Execution Time: 0.500 ms
Iteration 5 - Execution Time: 0.510 ms
Iteration 6 - Execution Time: 0.500 ms
Iteration 7 - Execution Time: 0.513 ms
Iteration 8 - Execution Time: 0.500 ms
Iteration 9 - Execution Time: 0.511 ms
```

Figure 5.5: Execution times of RT-application on bare-metal system on the real board

5.5 Evaluation of the results

In the end, in order to test the difference between the execution times, latency and overhead of the real-time application executed on the emulated system and on the real board, the experiment already conducted on the QEMU-emulated board was repeated also on the real ZCU102 and it resulted in an interesting output.

In more detail, following the exact same steps described in Par. 3.5.1, the real-time application was executed on the *bare metal* (Fig. 5.5) system first, then inside a Podman *Rootful* container (Fig. 5.6) and finally inside a Podman *Rootless* one (Fig. 5.7).

The interesting thing observed was that, comparing the results obtained on the real ZCU102

```
root@71283c794998:/tmp# head -10 ss_timing_log_with_load_evl_bash.csv
Iteration 0 - Execution Time: 0.517 ms
Iteration 1 - Execution Time: 0.513 ms
Iteration 2 - Execution Time: 0.500 ms
Iteration 3 - Execution Time: 0.524 ms
Iteration 4 - Execution Time: 0.500 ms
Iteration 5 - Execution Time: 0.512 ms
Iteration 6 - Execution Time: 0.500 ms
Iteration 7 - Execution Time: 0.518 ms
Iteration 8 - Execution Time: 0.500 ms
Iteration 9 - Execution Time: 0.517 ms
```

Figure 5.6: Execution times of RT-application inside *Rootful* container on the real board

```
root@8187a9a501d0:/bench# head -10 /tmp/ss_timing_log_with_load_evl_bash.csv
Iteration 0 - Execution Time: 0.517 ms
Iteration 1 - Execution Time: 0.500 ms
Iteration 2 - Execution Time: 0.514 ms
Iteration 3 - Execution Time: 0.500 ms
Iteration 4 - Execution Time: 0.528 ms
Iteration 5 - Execution Time: 0.500 ms
Iteration 6 - Execution Time: 0.520 ms
Iteration 7 - Execution Time: 0.500 ms
Iteration 8 - Execution Time: 0.512 ms
Iteration 9 - Execution Time: 0.500 ms
```

Figure 5.7: Execution times of RT-application inside *Rootless* container on the real board

with those obtained on the QEMU-emulated system, there was a significant difference in temporal behavior. In particular, in the emulated environment, execution times appeared to be very different from each other, showing a great *variability* ranging approximately from 0.27 ms up to 1.6 ms, indicating a large dispersion around the mean.

This resulted in a *high standard deviation*, also indicating a lack of temporal determinism. Such behavior though, was mainly attributable to the intrinsic limitations of QEMU, where an overhead was introduced by instruction emulation and virtualized timing mechanisms. Consequently, execution times appeared affected by external factors such as host load and context switching, leading to unpredictable latency fluctuations.

In contrast, the measurements obtained on the real hardware platform showed a *much tighter distribution*, with execution times consistently clustered around approximately 0.5–0.7 ms. The significantly *lower standard deviation* observed in this case indicates a *high level of temporal stability and predictability*, highlighting the fact that the goal of achieving real-time capabilities also in containers on an embedded (real) system was successfully achieved. Indeed, this behavior is essential for real-time systems, where guaranteeing bounded and repeatable execution times is more critical than achieving low average latency.

```
xilinx-zcu102-20221:/home/petalinux/rt_app# ./analyze_times.py /tmp/ss_timing
g_with_load_evl_bash.csv

File: /tmp/ss_timing_log_with_load_evl_bash.csv
Count  : 5000
Mean   : 0.505785 ms
Median : 0.510000 ms
Min    : 0.500000 ms
Max    : 0.551000 ms
xilinx-zcu102-20221:/home/petalinux/rt_app#
```

Figure 5.8: Mean, median, min, max values computed on execution times provided by the execution of the RT-application on the *bare-metal* system on the *real-board*

```
xilinx-zcu102-20221:/home/petalinux/rt_app# ./analyze_times.py /home/petalinu
ogs/rootful_real.csv

File: /home/petalinux/logs/rootful_real.csv
Count  : 5000
Mean   : 0.506927 ms
Median : 0.511000 ms
Min    : 0.500000 ms
Max    : 0.718000 ms
```

Figure 5.9: Mean, median, min, max values computed on execution times provided by the execution of the RT-application in the *Rootful container* on the *real-board*

```
xilinx-zcu102-20221:/home/petalinux/rt_app# ./analyze_times.py /home/petalinu
ogs/rootless_real.csv

File: /home/petalinux/logs/rootless_real.csv
Count  : 5000
Mean   : 0.507385 ms
Median : 0.511000 ms
Min    : 0.500000 ms
Max    : 0.668000 ms
```

Figure 5.10: Mean, median, min, max values computed on execution times provided by the execution of the RT-application in the *Rootless container* on the *real-board*

In reality, since only 10 out of 5000 iterations did not represent a significant sample to formulate assumptions and even if an extensive analysis as in the case of QEMU-emulated board was not conducted, some tests were performed on the whole provided CSV files containing the execution times in order to truly prove that the real-system behavior was indeed better than the emulated one.

In particular, some simple *Python scripts* were used to evaluate the *min, max, mean* and *median* values on the CSV files provided by the execution of the RT-application on the *real hardware* and, specifically, on the bare-metal system, in the *Rootful* container and in the *Rootless* one. The results obtained are shown in Figs. 5.8, 5.9 and 5.10.

As it appeared, the *min value* was equal to 0,5 ms in the three cases, while in the same experiments performed on the QEMU-emulated hardware, the WCET appeared to be up to just 0.348 ms. Even if this result was slightly counterintuitive, since a better behavior was expected on the real-hardware, in reality the expectations were then confirmed by the *mean* and *median* values, which appeared to be respectively equal to ~0,507 ms (WCET in the Rootful case) and 0,511 ms in both Rootful and Rootless cases. Furthermore, the most significant value for the *real-time behavior* that is represented by the *maximum value* also appeared to be only equal to 0,718 ms in the WCET (*Rootful* case) on the real-hardware, while in the same conditions but on the emulated board it appeared to arrive up to 154,453 ms, with a WCET *mean* value of ~1,82

ms.

Moreover, it the *real-board* execution, also the *overhead* provided by the use of containers appeared to be really contained and not impactful on the real-time behavior, against the bigger one introduced by the containers in the execution on the emulated system.

All of this proved that the use of EVL real-time scheduling and direct access to hardware resources ensured minimal interference and reduced jitter, allowing the system to operate in a deterministic manner.

Overall, these results demonstrated that while QEMU was suitable for functional testing and development, it was not adequate for evaluating real-time performance. Only execution on real hardware could provide meaningful insights into timing behavior and for this reason it is suggested, for future work, to repeat the experiments conducted in chapter 4 on the real board for a more-realistic (and also better) analysis.

Chapter 6

Conclusions

The rapid transformation of the space sector driven by the increasing of small satellites like CubeSats, as well as the increasing involvement of universities, research institutions, and private companies introduced new requirements for spacecraft software architectures. Indeed, as discussed in chapter 1, modern *On-Board Computers (OBCs)* are no longer expected to execute a single monolithic application, but rather to host multiple independent and potentially untrusted third-party applications concurrently, which may also require real-time guarantees.

Therefore, in this context, ensuring *isolation*, *security* and *real-time guarantees* became a critical challenge. For these same reasons, *container technologies* are emerging as a promising solution to provide lightweight isolation compared to traditional virtualization approaches. However, their compatibility with real-time execution models, especially on ARM embedded platforms such as the ones mostly used in the space field, has not been extensively investigated prior to this work.

Hence, the primary goal of this thesis was to explore the feasibility of combining *real-time execution* and *container-based isolation* on an embedded platform representative of those used in space systems. In particular, the target hardware considered was represented by an *AMD Xilinx Zynq UltraScale+ MPSoC (ZCU102)* platform adopted in real-world projects such as DLR's *Stellar Apps* which this thesis belongs to. In particular, following the classic development process, the complete software stack was designed and implemented before on the QEMU-emulated board (which emulation was conducted and described thoroughly in chapter 3), and in the end it was also successfully deployed on the real ZCU102 board available in the laboratory of *University Federico II of Naples*, as described in chapter 5.

To achieve *real-time capabilities* on the Linux-based system, the *Xenomai 4 EVL co-kernel* was integrated. This was necessary because the standard Linux kernel is a general-purpose

system that cannot guarantee deterministic execution required by time-critical applications.

By introducing the EVL core, real-time threads could be executed in an OOB scheduling domain, ensuring bounded latency and predictable behavior. However, this integration posed several challenges, including compatibility with the PetaLinux/Yocto build system used and the correct configuration of user-space libraries and dependencies. The last part of the emulation of the board consisted into enabling *Podman containers* that could access EVL devices for getting proper real-time capabilities, and in overcoming all the challenges that emerged from access permissions and namespaces isolation.

The second part of the thesis, on the other hand, focused on *evaluating* the interaction between EVL core and Podman containers, in order to analyze the latency of executing a real-time application inside and outside the containers in different experiments and to estimate the overhead introduced by containers. In more detail, three were the fundamental research questions addressed.

The first question concerned the *feasibility* of executing real-time applications inside both *Rootful* and *Rootless* containers. The results demonstrated that this was indeed possible, provided that the container configuration was carefully aligned with the underlying system constraints, and that *Rootless* containers tended to behave better than *Rootful*, providing less latency and overhead. This result was especially relevant in the context of project like DLR's *Stellar Apps*, where *Rootless* execution is preferred for security reasons.

The second question investigated the feasibility of *running multiple concurrent real-time containers*. The experimental results confirmed that multiple container instances could effectively and simultaneously attach their threads to the EVL devices and execute real-time workloads without interfering with each other at the co-kernel level. A proper analysis of the results obtained from different experiments was provided in chapter 4.

Finally, the third question addressed the *scalability of the system*. The evaluation showed stable behavior up to 64 *Rootless* concurrent real-time containers. Beyond this point, system performance degraded or encountered stalls, primarily due to limitations of the emulated environment and, probably, of the synchronization mechanism used to ensure the simultaneous execution of the containers.

6.1 Open questions and future steps

Despite these promising results, several limitations and *open challenges* were identified. In particular, the concurrent execution of *Rootful* and *Rootless* containers revealed asymmetric behaviors

that require further investigation, likely related to the interaction between user namespaces and EVL's attachment mechanisms. Moreover, the current system did not implement any form of global resource management or admission control across containers, leading to potential contention and unbounded latency under high load. For this reason, future work should focus on the introduction of *real-time orchestration mechanisms* at the container level. In particular, the adoption of a feasibility-based resource allocator, such as one based on the *Constant Bandwidth Server (CBS)* model, could enable controlled admission of containers, bounded resource allocation, and improved fairness among concurrent applications. Additionally, extending the evaluation to real hardware platforms would provide valuable insights for practical deployment.

In conclusion, this thesis demonstrated that the integration of *Xenomai 4 EVL* with *Podman containers* on an embedded ARM Linux-based platform is not only feasible but also architecturally sound. Real-time applications can be successfully executed inside containers — both *Rootful* and *Rootless* — while preserving deterministic behavior, provided that system configuration is carefully managed. These results contributed to bridging the gap between real-time requirements and modern containerized software architectures, and represented a step toward enabling flexible, secure, and scalable onboard computing platforms for future space missions.

Appendix A

Real-Time Benchmarking application

The Real Time application used to test the correct functioning of the Podman real-time containers in chapter 3 (see Par. 3.5) was provided by the *DLR* and it represented a benchmark evaluation monitor that launched some threads that needed to use EVL devices because of real-time requirements.

A *dummy-workload* - represented by a simple ADD-MUL operation is performed and the application is run 5000 times, in order to collect the jitter and variability of the time required to run it and to evaluate the responsiveness of the system (with the additive layer of containers) in a real-time environment.

The C code of the *Real_Time_Evaluation_monitor.c* benchmark real-time application is the following:

```
1: #define _GNU_SOURCE
2: #include <string.h>
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <evl/thread.h>
6: #include <evl/clock.h>
7: #include <evl/timer.h>
8: #include <evl/sched.h>
9: #include <evl/mutex.h>
10: #include <evl/proxy.h>
11: #include <time.h>
12: #include <unistd.h>
13: #include <fcntl.h>
```

```
14: #include <sched.h>
15: #include <pthread.h>
16:
17: #define PERIOD_NS 1000000 // 1 ms (1,000,000 nanoseconds)
18: #define RUN_TIME 5 // run time = 5s
19: #define CPU_CORE 1 // thread pinned to core 1
20: #define FILE_NAME "/tmp/ss_timing_log_with_load_evl_bash.csv"
21: static struct evl_mutex rt_mutex; // Creating a struct for mutex
22:
23: // buffer to hold exec time logs
24: static double exec_times[RUN_TIME * 1000];
25:
26: // global name buffers for multiple instances
27: static char evl_main_name[32];
28: static char evl_task_name[32];
29:
30: void *real_time_task(void *arg) {
31:      struct timespec next_time, start_time, end_time; //for recoding timing
32:
33:      clock_gettime(CLOCK_MONOTONIC, &next_time); // grab current time
34:
35:     // attach thread to evl
36:       if (evl_attach_thread(EVL_CLONE_PRIVATE, evl_task_name) < 0) {
37:            perror("Failed to attach real-time thread");
38:                          return NULL;
39:       }
40:
41:       evl_printf("Real-time periodic task started on CPU core %d\n", CPU_CORE);
42:       //printing to know when the loop starts
43:
44:       for (int i = 0; i < (RUN_TIME * 1000); i++) {
45:            clock_gettime(CLOCK_MONOTONIC, &start_time); // start timestamp
46:
47:          // some cpu load (dummy loop)
48:          volatile int product = 0;
49:          for (int j = 0; j < 100000; j++) product += j*j;
50:          clock_gettime(CLOCK_MONOTONIC, &end_time); // end timestamp
51:          // calc duration in ms
52:          exec_times[i] = (end_time.tv_sec - start_time.tv_sec) * 1e3 +
53:                         (end_time.tv_nsec - start_time.tv_nsec) / 1e6;
54:
55:          // add period to next_time
```

```
56:           next_time.tv_nsec += PERIOD_NS;
57:           while (next_time.tv_nsec >= 1000000000) {
58:                 next_time.tv_sec++;
59:                 next_time.tv_nsec -= 1000000000;
60:           }
61:           evl_usleep(PERIOD_NS / 1000); // sleep 1ms using evl
62:         }
63:         return NULL;
64: }
65:
66: int main() {
67:     int ret;
68:     pthread_t rt_thread;
69:     struct evl_sched_attrs attr;
70:     cpu_set_t cpu_set;
71:     printf("Hello from Real-Time Evaluation Monitor\n");
72:
73:     // To generate unique names once (for multiple instances)
74:     const char *inst = getenv("EVL_INSTANCE");
75:     if (!inst || !*inst)
76:         inst = "0";
77:     snprintf(evl_main_name, sizeof(evl_main_name), "rt-main-%s", inst);
78:     snprintf(evl_task_name, sizeof(evl_task_name), "rt-task-%s", inst);
79:
80:     // set scheduler config
81:     attr.sched_policy = SCHED_FIFO;
82:     attr.sched_priority = 80; // fairly high prio
83:
84:     // try attaching main thread to evl
85:     ret = evl_attach_thread(EVL_CLONE_PRIVATE, evl_main_name);
86:     if (ret < 0) {
87:         fprintf(stderr, "evl_attach_thread failed: %s (%d)\n", strerror(-ret), ret);
88:         return -1;
89:     }
90:
91:     // limit to specific cpu core
92:     CPU_ZERO(&cpu_set);
93:     CPU_SET(CPU_CORE, &cpu_set);
94:
95:     pthread_attr_t attr_thread;
96:     pthread_attr_init(&attr_thread);
97:     pthread_attr_setaffinity_np(&attr_thread, sizeof(cpu_set_t), &cpu_set);
```

```
 98:
 99:     evl_printf("Starting real-time task on CPU core %d...\n", CPU_CORE);
100:
101:     // real time thread
102:     pthread_create(&rt_thread, &attr_thread, real_time_task, NULL);
103:     pthread_join(rt_thread, NULL);
104:
105:     // open output file
106:     int log_fd = open(FILE_NAME, O_WRONLY | O_CREAT | O_TRUNC, 0644);
107:     if (log_fd < 0) {
108:         perror("couldn't open log file");
109:         return -1;
110:     }
111:
112:     // log all data to csv
113:     char log_buffer[256];
114:     for (int i = 0; i < (RUN_TIME * 1000); i++) {
115:         snprintf(log_buffer, sizeof(log_buffer),
116:                 "Iteration %d - Execution Time: %.3f ms\n", i, exec_times[i]);
117:         write(log_fd, log_buffer, strlen(log_buffer));
118:     }
119:
120:     close(log_fd);
121:     return 0;
122: }
```

Appendix B

Shell script to run multiple concurrent Rootless containers

The following shell script was used to test the *scalability of the system*. In particular, modifying the number of containers to run (`NUM_CONTAINERS="$1:-10"`), it was possible to execute various experiments with a variable number of concurrent running *Rootless* containers (from 2 up to 100, finding 64 containers as a limit for the proposed system).

```
#!/bin/bash
set -euo pipefail

# --------- CONFIG ----------
NUM_CONTAINERS="${1:-10}" # usage: ./run_rootless_experiment.sh 10
IMAGE="docker.io/library/debian:stable-slim"

SYNC_DIR="/home/petalinux/sync"
BENCH_DIR="/home/petalinux/rt_app"
LOGS_BASE="/home/petalinux/logs"

# How long to wait after GO before checking results (seconds)
# (Best effort; you can tweak or set to 0)
POST_GO_WAIT="${POST_GO_WAIT:-1}"

# Devices + libs you already pass
EVL_DEVICES=(
  "--device" "/dev/evl/control"
```

```
  "--device" "/dev/evl/thread/clone"
  "--device" "/dev/evl/clock/monotonic"
  "--device" "/dev/evl/clock/realtime"
)

EVL_LIBS=(
  "-v" "/usr/lib/libevl.so.6:/usr/lib/libevl.so.6:ro"
  "-v" "/usr/lib/libbpf.so.1:/usr/lib/libbpf.so.1:ro"
  "-v" "/usr/lib/libelf.so.1:/usr/lib/libelf.so.1:ro"
)

# --------- HELPERS ----------
wait_for_containers_exit() {
  # Wait until all containers are not running anymore.
  # This avoids racing the log/inspect/wc/tail steps.
  for i in $(seq 1 "${NUM_CONTAINERS}"); do
    local name="rt_rootless_${i}"
    while true; do
      local running
      running="$(podman inspect "${name}" --format '{{.State.Running}}'
          2>/dev/null || echo "false")"
      if [[ "${running}" != "true" ]]; then
        break
      fi
      sleep 0.1
    done
  done
}

# --------- PREP SYNC ----------
echo "[prep] sync dir: ${SYNC_DIR}"
mkdir -p "${SYNC_DIR}"

rm -f "${SYNC_DIR}/GO" "${SYNC_DIR}/start_times.txt"
touch "${SYNC_DIR}/start_times.txt"
chmod 666 "${SYNC_DIR}/start_times.txt"

# Remove old per-container debug files
for i in $(seq 1 "${NUM_CONTAINERS}"); do
  rm -f "${SYNC_DIR}/rootless_${i}.debug"
done

```

```
# --------- PREP LOG DIRS ----------
echo "[prep] logs base: ${LOGS_BASE}"
for i in $(seq 1 "${NUM_CONTAINERS}"); do
  LOGDIR="${LOGS_BASE}/rootless_dir_${i}"
  mkdir -p "${LOGDIR}"
  chmod 777 "${LOGDIR}"
  rm -f "${LOGDIR}/ss.csv"
done

# --------- CLEAN UP OLD CONTAINERS ----------
echo "[cleanup] removing old containers named rt_rootless_1..${NUM_CONTAINERS} (if
    any)"
for i in $(seq 1 "${NUM_CONTAINERS}"); do
  podman rm -f "rt_rootless_${i}" >/dev/null 2>&1 || true
done

# --------- LAUNCH CONTAINERS ----------
echo "[run] launching ${NUM_CONTAINERS} rootless containers..."

for i in $(seq 1 "${NUM_CONTAINERS}"); do
  NAME="rt_rootless_${i}"
  HOSTNAME="rt_less_${i}"
  LOGDIR="${LOGS_BASE}/rootless_dir_${i}"
  DEBUGFILE="/sync/rootless_${i}.debug"

  echo " -> creating ${NAME}"

  podman run -d --name "${NAME}" \
    --log-driver=k8s-file \
    --network=none \
    --hostname "${HOSTNAME}" \
    -v "${BENCH_DIR}:/bench:rw" \
    -v "${SYNC_DIR}:/sync:rw" \
    -v "${LOGDIR}:/logs:rw" \
    "${EVL_DEVICES[@]}" \
    "${EVL_LIBS[@]}" \
    "${IMAGE}" \
    bash -lc "{
      while [ ! -e /sync/GO ]; do sleep 0.001; done
      echo \"rootless_${i} start \$(date +%s.%N)\" >> /sync/start_times.txt
      echo \"rootless_${i} whoami=\$(id -u):\$(id -g)\" >> /sync/start_times.txt
      ln -sf /logs/ss.csv /tmp/ss_timing_log_with_load_evl_bash.csv
```

```
101:       EVL_INSTANCE=${i} /bench/rt_monitor_EVL
102:       echo \"rootless_${i} rc=\$?\"
103:     } >> ${DEBUGFILE} 2>&1"
104: done
105:
106: echo "[run] all containers launched."
107:
108: # --------- TRIGGER START ----------
109: echo "[run] triggering simultaneous start: touch ${SYNC_DIR}/GO"
110: touch "${SYNC_DIR}/GO"
111:
112: # Optional short wait
113: if [[ "${POST_GO_WAIT}" != "0" ]]; then
114:   sleep "${POST_GO_WAIT}"
115: fi
116:
117: # Prefer waiting until all containers exit (safer than fixed sleep)
118: echo "[run] waiting for containers to finish..."
119: wait_for_containers_exit
120:
121: # --------- POST-RUN REPORT ----------
122: echo
123: echo "==================== start_times.txt ===================="
124: cat "${SYNC_DIR}/start_times.txt" || true
125:
126: echo
127: echo "==================== debug (first 120 lines each) ===================="
128: for i in $(seq 1 "${NUM_CONTAINERS}"); do
129:   DBG="${SYNC_DIR}/rootless_${i}.debug"
130:   echo
131:   echo "----- ${DBG} (head -n 120) -----"
132:   if [[ -f "${DBG}" ]]; then
133:     sed -n '1,120p' "${DBG}"
134:   else
135:     echo "[missing] ${DBG}"
136:   fi
137: done
138:
139: echo
140: echo "==================== podman inspect exit/error ===================="
141: for i in $(seq 1 "${NUM_CONTAINERS}"); do
142:   NAME="rt_rootless_${i}"
```

```
143:   echo -n "${NAME}: "
144:   podman inspect "${NAME}" --format '{{.State.ExitCode}} {{.State.Error}}'
           2>/dev/null || echo "[inspect failed]"
145: done
146:
147: echo
148: echo "==================== ss.csv line counts ===================="
149: for i in $(seq 1 "${NUM_CONTAINERS}"); do
150:   CSV="${LOGS_BASE}/rootless_dir_${i}/ss.csv"
151:   if [[ -f "${CSV}" ]]; then
152:     wc -l "${CSV}"
153:   else
154:     echo "0 ${CSV} (missing)"
155:   fi
156: done
157:
158: echo
159: echo "==================== ss.csv tails (last 5 lines each) ===================="
160: for i in $(seq 1 "${NUM_CONTAINERS}"); do
161:   CSV="${LOGS_BASE}/rootless_dir_${i}/ss.csv"
162:   echo
163:   echo "----- ${CSV} (tail -n 5) -----"
164:   if [[ -f "${CSV}" ]]; then
165:     tail -n 5 "${CSV}"
166:   else
167:     echo "[missing] ${CSV}"
168:   fi
169: done
170:
171: echo
172: echo "[done] experiment completed."
```

Bibliography

[1] United Nations Office for Outer Space Affairs. *Online Index of Objects Launched into Outer Space (OSOIndex)*. URL: https://www.unoosa.org/oosa/osoindex/search-ng.jspx (visited on 08/01/2026).

[2] J. Vander Hook et al. "Nebulae: A Proposed Concept of Operation for Deep Space Computing Clouds". In: *Proceedings of the IEEE Aerospace Conference*. 2020, pp. 1–14.

[3] H. Otte et al. "Preliminary Design of the Stellar Apps Software Platform for Developing and Executing On-board Applications". In: *Proceedings of the EDHPC Conference 2025*. German Aerospace Center (DLR), Institute of Software Technology. Germany, 2025.

[4] Exodus Orbitals. *How Exodus Orbitals Simplifies Satellite and Space Prototyping with Docker*. Tech. rep. Case Study. Exodus Orbitals Inc., 2023.

[5] Erik Kulu. "CubeSats Nanosatellites - 2024 Statistics, Forecast and Reliability". In: Oct. 2024. DOI: 10.52202/078365-0069.

[6] Dominik Marszk et al. "OPS-SAT in Orbit: A Technical Rundown of this Open Experimentation Platform". In: *International Astronautical Congress (IAC)*. International Astronautical Federation, 2019.

[7] François-Xavier Molina et al. "Cubedate: Securing Software Updates in Orbit for Low-Power Payloads Hosted on CubeSats". In: *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2022.

[8] Raphael Rohrmueller and Bjoern Annighoefer. "Hypervisors and Plug & Fly in a New Space Launcher: A Scalable Approach to Enhance Space Launcher Development". In: *2024 AIAA/IEEE 43rd Digital Avionics Systems Conference (DASC)*. 2024. DOI: 10.1109/DASC62030.2024.10749521.

[9] Gabriele Marra, Ulysse Planta, Philipp Wüstenberg, and Ali Abbasi. "On the Feasibility of CubeSats Application Sandboxing for Space Missions". In: *Workshop on Security of Space and Satellite Systems (SpaceSec)*. NDSS Symposium, 2024.

[10] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V. Papadopoulos. "Real-Time Containers: A Survey". In: *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. 2020. DOI: 10.4230/OASIcs.Fog-IoT.2020.7.

[11] Marcello Cinque and Domenico Cotroneo. "Towards Lightweight Temporal and Fault Isolation in Mixed-Criticality Systems with Real-Time Containers". In: *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2018. DOI: 10.1109/DSN-W.2018.00029.

[12] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte. "Achieving Isolation in Mixed-Criticality Industrial Edge Systems with Real-Time Containers". In: *ACM Transactions on Programming Languages and Systems* (2022).

[13] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. "Container-based real-time scheduling in the Linux kernel". In: *SIGBED Rev.* (2019).

[14] Linux man-pages project. *cgroups(7) — Linux manual page*. 2026. URL: https://www.man7.org/linux/man-pages/man7/cgroups.7.html.

[15] Václav Struhár et al. "Hierarchical Resource Orchestration Framework for Real-time Containers". In: *ACM Transactions on Embedded Computing Systems* 23.1 (2024). DOI: 10.1145/3592856.

[16] Xenomai Project. *Xenomai 4 Overview*. 2025. URL: https://v4.xenomai.org/overview/index.html.

[17] Dina Sboui. "Real-time Guarantees for High-Performance Safety-Critical Applications". MA thesis. University of Carthage, National Institute of Applied Sciences and Technology (hosted by DLR), 2025.

[18] A. (Bram) Meijer. "REAL-TIME ROBOT SOFTWARE FRAMEWORK ON RASPBERRY PI USING XENOMAI AND ROS2". MA thesis. University of Twente, The Netherlands, 2021.

[19] Xenomai Project. *Dovetail Interface*. 2025. URL: https://v4.xenomai.org/dovetail/index.html.

[20] Xenomai Project. *The EVL Core*. 2025. URL: https://v4.xenomai.org/core/index.html.

[21] Xenomai Project. *Xenomai 4 Core Documentation*. URL: https://v4.xenomai.org/core/index.html.

[22] EVL Project. *Dovetail Kernel API*. 2025. URL: https://evlproject.org/dovetail/kernel-api/.

[23] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. "RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets". In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Vol. 133. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 5:1–5:22. DOI: 10.4230/LIPIcs.ECRTS.2019.5.

[24] AMD. *AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. EK-U1-ZCU102-G evaluation board and development kit description. URL: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/ek-u1-zcu102-g.html (visited on 01/06/2025).

[25] Yocto Project. *Yocto Project Overview and Concepts Manual*. Linux Foundation. 2025. URL: https://docs.yoctoproject.org/overview-manual/.

[26] Xilinx Inc. *PetaLinux Tools Documentation Reference Guide (UG1144)*. Version 2022.2. AMD Xilinx. Oct. 2022. URL: https://www.xilinx.com.

[27] Wikipedia contributors. *QEMU*. 2026. URL: https://it.wikipedia.org/wiki/QEMU.

[28] Xilinx Wiki. *What is QEMU?* Accessed: 2026-03-17. 2025. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/822247454/What+is+QEMU.

[29] Various Authors. "Virtualization Technologies and Their Applications in Embedded Systems". In: *Electronics* 10.6 (2021), p. 759. DOI: 10.3390/electronics10060759.

[30] AMD. *QEMU and Virtual Platform Documentation*. AMD Xilinx. 2025. URL: https://docs.amd.com/api/khub/documents/VOM6aBLyjscyBTKoyz7J5Q/content.

[31] ICTP. *QEMU and Embedded System Emulation*. 2017. URL: https://indico.ictp.it/event/7987/session/39/contribution/147/material/slides/0.pdf.

[32] Xenomai Project. *Xenomai 3 Overview*. 2025. URL: https://v3.xenomai.org/overview/index.html.

[33] Xenomai Project. *Xenomai 4 EVL Linux Kernel Repository*. 2025. URL: https://source.denx.de/Xenomai/xenomai4/linux-evl.git.

[34] Podman Project. *Podman Documentation*. 2025. URL: https://docs.podman.io/en/latest/.

[35] Wikipedia contributors. *Podman*. 2026. URL: https://en.wikipedia.org/wiki/Podman.

[36] Red Hat. *Finding, Running, and Building Containers with Podman, Skopeo, and Buildah*. 2025. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_atomic_host/7/html/managing_containers/finding_running_and_building_containers_with_podman_skopeo_and_buildah.

[37] Podman Developers. *Podman: A Tool for Managing OCI Containers and Pods*. 2025. URL: https://github.com/containers/podman.

[38] Red Hat. *Building, Running, and Managing Containers in RHEL 8*. 2025. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/building_running_and_managing_containers/assembly_working-with-containers_building-running-and-managing-containers.

[39] Wikipedia contributors. *OS-level Virtualization*. 2026. URL: https://en.wikipedia.org/wiki/OS-level_virtualization.

[40] AMD Xilinx. *2022.1 Release*. Accessed: 2026-03-17. 2022. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2347204609/2022.1+Release.

[41] AMD Xilinx. *2022.1 Release*. 2022. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2347204609/2022.1+Release.

[42] Linux Kernel Organization. *The Linux Kernel Archives*. Linux Foundation, 2025. URL: https://www.kernel.org/.

[43] Meson Project. *The Meson Build System*. 2025. URL: https://mesonbuild.com/.

[44] Wikipedia contributors. *Benchmark (informatica)*. URL: https://it.wikipedia.org/wiki/Benchmark_(informatica).