FH University of Applied Sciences
TECHNIKUM WIEN

MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Mechatronics/Robotics

Virtualization of a Real-Time Operating System for Robot Control with a Focus on Real-Time Compliance

By: Halil Pamuk, BSc

Student Number: 51842568

Supervisor: Sebastian Rauh, MSc. BEng

Wien, September 2, 2024

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Wien, September 2, 2024 Signature

Kurzfassung

Virtualisierung von Echtzeitbetriebssystemen für die Robotersteuerung bietet viele Vorteile im Vergleich zu herkömmlichen hardwarebasierten Ansätzen. Lösungen, die auf reiner Hardware basieren, sind oft für einen bestimmten Zweck gebaut und daher in ihrer Flexibilität eingeschränkt. Es ist schwierig, die Hardware-Topologie an neue Anforderungen anzupassen, und es kann kostspielig sein, insbesondere bei der Skalierung von Operationen. Der physische Zugang für Updates und Wartung ist herausfordernd, was zu Ausfallzeiten und Produktivitätsverlusten führt. Während die Virtualisierung diese Probleme adressiert, führt sie zu erhöhtem Overhead und Latenz.

Diese Arbeit untersucht die Virtualisierung des proprietären Salamander 4-Betriebssystems unter Verwendung von QEMU/KVM. Salamander 4 wird mit Yocto gebaut und verwendet harte Echtzeit mit Xenomai 3. Das Hauptziel besteht darin, die Latenzlücke zwischen der Bare-Metal und der Virtualisierung zu überbrücken, um ein deterministisches und zuverlässiges Verhalten zu gewährleisten, das für Echtzeit-Roboteranwendungen entscheidend ist.

Erste Latenzmessungen zeigten eine signifikante Leistungslücke zwischen der Bare-Metal und dem virtualisierten Setup. Daher wird ein umfangreicher Tuning-Prozess durchgeführt, um Echtzeit-Performance und Determinismus zu erreichen. Diese Modifikationen umfassen Konfigurationen, die das BIOS, den Kernel, das Host-Betriebssystem, die QEMU/KVM-Virtualisierungsschicht und das Salamander 4 Gast-Betriebssystem selbst betreffen. Die Worst-Case-Latenz wurde von 707.622 µs auf 17.134 µs reduziert, was der Bare-Metal-Performance von 10.709 µs nahekommt. Darüber hinaus wird die Verbesserung der Echtzeit-Performance und des Determinismus anhand einer Roboteranwendung validiert, bei der die abgestimmte Virtualisierung mit der nicht abgestimmten und der Bare-Metal-Version verglichen wird.

Insgesamt bietet diese Arbeit eine umfassende Methodik, um ein virtualisiertes Gastsystem in einem Hostsystem mit deterministischem Verhalten echtzeitfähig zu machen.

Schlagworte: Virtualisierung, Echtzeitsysteme, Latenzreduktion, Robotersteuerung

Abstract

Virtualization of Real-Time Operating Systems (RTOS) for robotic control has many advantages in comparison to traditional hardware-based approaches. Solutions based on pure hardware are often built for a distinct purpose and are hence limited in flexibility. It is difficult to adapt the hardware topology to new requirements and can be costly, especially when scaling operations. Physical access for updates and maintenance is challenging, leading to downtime and lost productivity. While virtualization addresses these issues, it introduces increased overhead and latency.

This thesis investigates the virtualization of the proprietary Salamander 4 operating system, using QEMU/KVM. Salamander 4 is built with Yocto and employs hard real-time with Xenomai 3. The primary objective is to bridge the latency gap between the virtualized and bare metal versions in order to ensure deterministic and reliable behavior, which is crucial for real-time robotic applications.

Initial latency measurements revealed a significant performance gap between the bare metal and virtualized setup. Thus, an extensive tuning process is carried out to achieve real-time performance and determinism. These modifications involve configurations spanning the BIOS, kernel, host operating system, QEMU/KVM virtualization layer, and the Salamander 4 guest operating system itself. The worst-case latency was brought down from 707.622 µs to 17.134 µs, closely aligning with the bare metal performance of 10.709 µs. In addition, the improvement in real-time performance and determinism is validated using a robotic application, where the tuned virtualization is compared with the untuned and the bare metal version.

Altogether, this thesis provides a comprehensive blueprint for making a virtualized guest system real-time capable in a host system with deterministic behavior.

Keywords: Virtualization, Real-Time Systems, Latency Reduction, Robot Control

Acknowledgements

First, I would like to thank my supervisor, Sebastian Felix Rauh MSc. BEng., for his expert guidance and assistance throughout this master's thesis. His insightful answers to all my questions and his calming presence helped me stay focused and composed.

I want to thank my wife, Nour Pamuk, for her unwavering support, love, and patience. She gave me the strength to overcome this challenge. This work is as much hers as it is mine.

Lastly, I am deeply grateful to my parents, Semra Pamuk and Mehmet Pamuk, and to my close family for their understanding and encouragement during my studies. Without their unconditional love and support, this work would not have been possible.

Contents

1 Introduction

In today's industrial production and automation, robot systems are well established and of crucial importance. Robots must react to their environment and perform time-critical tasks within strict time constraints. Delays or errors can have catastrophic consequences in some cases. Modern operating systems nowadays are categorized into two types: General-Purpose Operating System (GPOS) and Real-Time Operating System (RTOS). The key difference is whether the system is time-critical or not [1]. In a traditional GPOS, such as Windows or Linux-based distributions, a high-priority task cannot interrupt a kernel operation, as illustrated in Figure 1. This makes them not suitable for real-time requirements, as they cannot guarantee deterministic execution times.

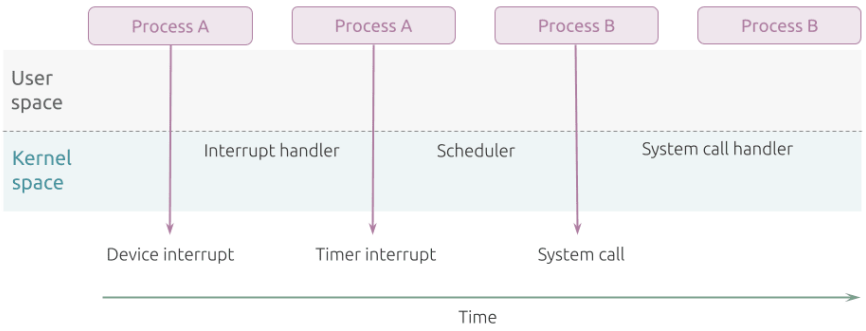

Figure 1: Non-preemptible Kernel [2]

However, in an RTOS, a high-priority process can interrupt a lower-priority task, even if it is in the middle of a kernel operation, as shown in Figure 2. An RTOS is specifically designed to react to events within fixed time limits and prioritize the execution of high-priority processes.

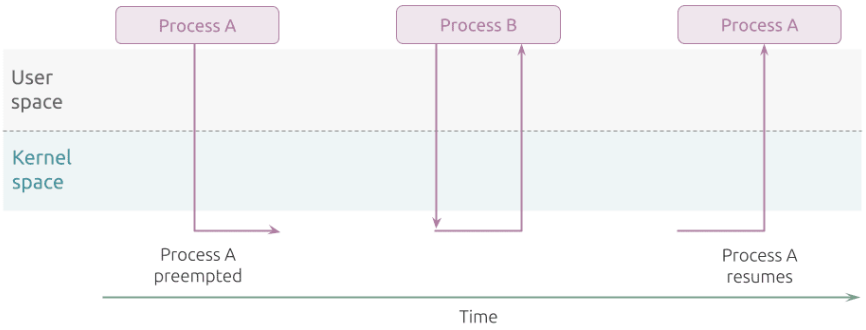

Figure 2: Preemptible Kernel [2]

Many Linux distributions can function as both GPOS and RTOS with kernel modifications.

1.1 Real-Time Operating Systems

The RTOS structure is visualized in Figure 3. The core component of an RTOS that enables real-time capabilities is the kernel. It is responsible for managing system resources, scheduling tasks, and ensuring deterministic behavior [3]. It employs preemptive scheduling mechanisms to allow high-priority tasks to preempt lower-priority tasks, ensuring that time-critical tasks are not delayed. Additionally, RTOS kernels are designed to allocate and manage memory resources efficiently and minimize interrupt latency, which is crucial for real-time applications that require immediate response to external events [4].

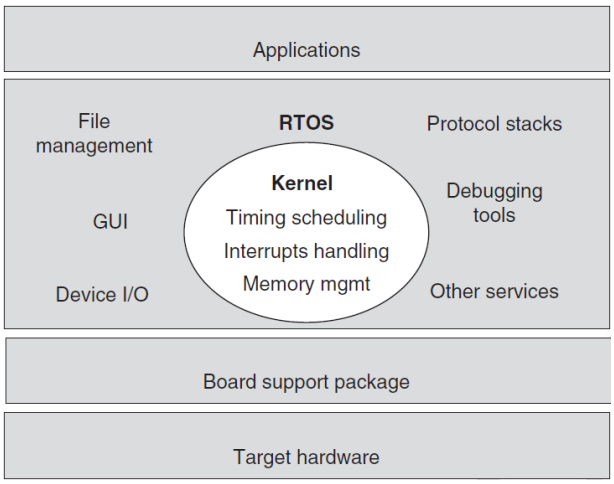

Figure 3: RTOS Structure [4]

In these real-time operating systems, task scheduling is based on so-called priority-based preemptive scheduling [5]. Each task in a software application is assigned a priority. A higher priority means that a faster response is required. Preemptive task scheduling ensures a very fast response. Preemptive means that the scheduler can stop a currently running task at any point if it recognizes that another task needs to be executed immediately. The basic rule on which priority-based preemptive scheduling is based is that the task with the highest priority that is ready to run is always the task that must be executed. So, if both a task with a lower priority and a task with a higher priority are ready to run, the scheduler ensures that the task with the higher priority runs first. The lower-priority task is only executed once the higher priority task has been processed.

Real-time systems are usually classified as either soft or hard real-time systems [6]. The difference lies exclusively in the consequences of a violation of the time limits [7]. Hard real-time is when the system stops operating if a deadline is missed, which can have catastrophic consequences. Soft real-time exists when a system continues to function even if it cannot perform the tasks within a specified time. If the system has missed the deadline, this has no critical consequences. The system continues to run, although it does so with undesirably lower output quality [8].

1.2 Application Context

This section briefly defines the context and scope within which this master's thesis was composed, highlighting the specific contributions and requirements set by SIGMATEK GmbH & Co KG [9].

- This work was written at SIGMATEK GmbH & Co KG, a company that researches, develops, produces, and distributes automation technology for industrial machinery and plant engineering. SIGMATEK uses its own custom Linux-based operating system, to be run on their self-manufactured CPUs. This operating system will be referred to as "Salamander 4" in this work. The details of Salamander 4 are explained in Section 2.2.
- Salamander 4 is virtualized through Quick Emulator (QEMU), an open-source emulator and virtualizer that allows running different operating systems on a computer. QEMU is described in Section 2.3.
- Salamander 4 was created with Yocto, an open-source project that helps develop custom Linux-based systems. Yocto is discussed in Section 2.4.
- Salamander 4 employs hard real-time with Xenomai 3, a real-time framework for Linux that enables deterministic and low-latency performance. Xenomai 3 is detailed in Section 2.5.
- The goal is to virtualize Salamander 4 and approach the performance of bare metal (hardware) through real-time performance tunings, focusing on latency requirements set by SIGMATEK. These modifications are the subject of Section 3.2.
- The `latency` tool of the Xenomai suite to measure the scheduling latency of the real-time OS was suggested to be used by SIGMATEK. The goal was set to achieve latency values below 50 µs in the duration of the measurement.
- SIGMATEK uses the VARAN bus for communication between machines and systems, a real-time Ethernet network that was specially developed for industrial automation. Its operating principle is outlined in Section 2.8.
- For the purpose of replacing the physical CPU, SIGMATEK provided a PCI Insert Card module PCV 522, that was plugged into the PC, as explained in Subsection 3.3.2. This component could then be used to communicate with the Input/Output (I/O) peripherals, just like the physical CPU could when the I/O modules were attached to it. Subsequently, the PCV 522 module of SIGMATEK was connected with the Pulse Width Module PW 022 of SIGMATEK over the VARAN bus and the Varan Connection Module VI 021 of SIGMATEK.
- The robotic application from Section 3.3 was written in Lasal Class 2, an object-oriented programming tool of SIGMATEK and part of SIGMATEK's Engineering Platform LASAL, with client-server technology and graphic representation.

1.3 Related Work and State of the Art

This master's thesis builds upon a number of previous studies in the field of virtualization of real-time systems and the inherent latency. In this section, the most influential studies are presented and their relevance to the work is addressed.

Perneel and Auricle Technologies Pvt. Ltd. [10] mention in their work that Linux was initially developed as a general-purpose operating system without real-time applications in mind. They talk about the evolution of real-time behavior in the standard Linux kernel and state that it has recently gained popularity among the real-time community, largely due to its open-source nature and stability. The authors explain how to transform Linux into a real-time operating system by implementing the PREEMPT-RT patch. The paper also emphasizes some kernel configurations that ensure that the system's real-time behavior is preserved during runtime. Even though these modifications to the kernel enable soft real-time performance, the authors underline the fundamental rule in real-time software, that latency improvements have a negative impact on throughput and kernel performance. This is due to the added overhead of the `CONFIG_PREEMPT_RT` option. They concluded with their Linux build and a testing application that Linux can be a viable option for real-time applications, if it is correctly configured. Reghenzani, Massari, and Fornaciari [11] provide an extensive overview of the real-time Linux kernel research, specifically on the PREEMPT RT patch. Adam [12] applied the PREEMPT_RT patch on both Raspberry Pi 3 and BeagleBone Black and achieved significantly lower latencies compared to standard Linux kernels.

Cinque et al. [13] survey different virtualization approaches, including separation kernels, microkernels, enhanced general-purpose hypervisors, and lightweight virtualization solutions like containers and unikernels. Similarly, Sandstrom et al. [14], and Taccari et al. [15] discuss and compare the current state-of-the-art in virtualization technologies with a focus on embedded real-time platforms. The latter emphasizes that research on real-time systems is focusing on virtualization and multicore scheduling and underlines that flexibility and cost are key metrics. Both works acknowledge the great benefits of virtualization, that is, reducing overall hardware costs, since all non-critical activities and real-time control tasks can run on the same hardware. In addition, Javier Perez et al. [16] state that hardware-based Programmable Logic Controllers (PLCs) are costly to maintain and lack the flexibility needed for modern, resource-intensive applications like Machine Learning (ML) or Artificial Intelligence (AI).

Despite the promising advantages of virtualization, one must be aware of the potential drawbacks of it, too. Gu and Zhao [17] highlight issues like lock-holder preemption and task-grain scheduling. García-Valls, Cucinotta, and Lu [18] identify challenges in supporting real-time applications in the cloud, including resource management, quality of service guarantees, temporal and spatial isolation, and network performance. Scordino et al. [19] address the challenge of interference on shared hardware resources, which can degrade real-time performance.

Ma et al. [20] [21] describe in their work how KVM impacts real-time performance, which has helped this work gain insight into the base overhead of virtualization. When a physical interrupt occurs while the guest RTOS is running, there are at least six stages that need to be traversed before the interrupt can be delivered to the guest RTOS. Once the interrupt is delivered to the guest RTOS, it incurs additional latency, called Interrupt Routine Time (IRT), which is the time between when an interrupt is recognized and the first instruction of the corresponding Interrupt Service Routine (ISR) of the guest OS is started. These stages are the primary sources of latencies caused by KVM, they are briefly described in Figure 4.

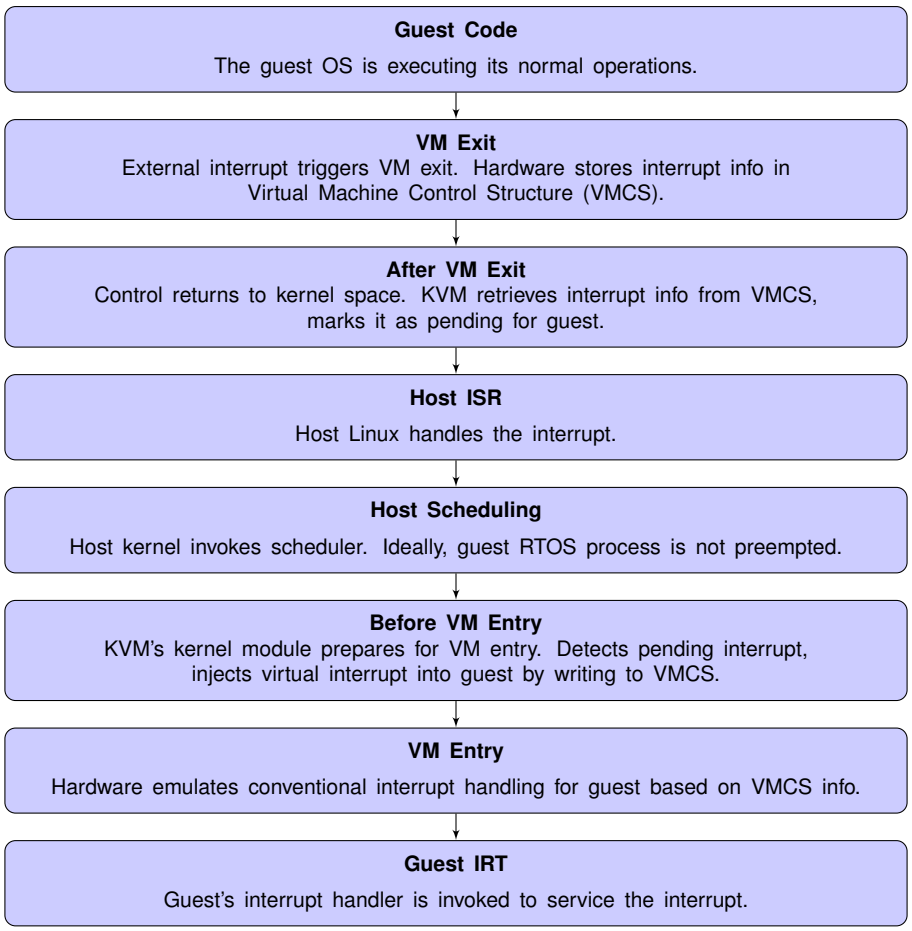

Figure 4: Flowchart of Operations in a virtualized Environment (Own Figure, inspired by Ma et al.)

The paper also details several performance tuning methods, including CPU shielding and prioritization, which have been applied in this work.

Brosky and Rotolo [22] report in their work about the shielded CPU model for enhancing real-time performance in symmetric multiprocessor systems. By dedicating one or more CPUs to high-priority processes and interrupts, the goal is to provide deterministic execution of real-time applications and interrupt responses. The authors mention that there is no user interface for setting process CPU affinity in the standard Linux, while there is one for setting interrupt CPU affinity via the `/proc/irq/*/smp_affinity` files. These files were used in this master's thesis. According to the authors, the shielded CPU model is especially useful for tasks that require guaranteed interrupt response times, very fast interrupt responses, and high-frequency, deterministic execution.

In the course of writing this thesis, the white paper "Real-Time Performance Tuning Best Practice Guidelines for KVM-Based Virtual Machines" [23] by Intel has been instrumental. Given that the working computer used for this work is equipped with an Intel processor, the guidelines presented in this document were particularly relevant and valuable for optimizing real-time performance in KVM-based virtual machines. Section 3.2 contains an extensive tuning process, which involves configurations spanning the BIOS, kernel, host OS, QEMU/KVM virtualization layer, and the Salamander 4 OS itself. The document concludes that the described tuning methods effectively improve real-time performance for KVM-based VMs, even under heavy workloads.

Yoon, Song, and Lee [24] use some of these real-time tunings in their research, including CPU shielding, memory locking, and spinning nanosleep, for controlling humanoid robots equipped with around 60 servo motors and sensors. They implement EtherCAT for real-time communication of distributed devices, whereas this thesis uses Varan. The authors highlight the importance of timely data processing in robots that collect data from their environment and respond accordingly through their sensors and actuators. The authors conclude with an experiment that their robotic system could achieve deterministic control of actuators and sensors even under heavy system load through the use of Linux with the real-time preemption patch. Their research has similarities to this master's thesis in terms of real-time tunings for robot control. However, this thesis also includes the virtualization aspect that is central.

Kiszka [25] specifically focuses on Linux as a hypervisor and analyzes its real-time capabilities when using KVM and QEMU. Most importantly for this master's thesis, the author warns of starvation when the priorities of QEMU's threads are raised and lifted into a real-time scheduling class. Starvation is when a process does not get the resources it needs for a long time because the resources are being allocated to other processes. A way to reduce this risk is by not using more virtual CPUs (vCPUs) than there are actual processor cores. Moreover, Linux has a feature that can limit how much CPU time all real-time tasks can use in the host system, which was important information for the real-time tunings of this thesis later.

McKenney [26] addresses the overheads associated with real-time Linux when using KVM and QEMU. The author lists several sources of overhead, including memory locking to avoid page-fault latencies, increased overhead of locking and interrupts due to more-aggressive preemption, threaded interrupts that permit long-running interrupt handlers to be preempted by high-priority real-time processes, real-time task scheduling that requires global scheduling, high-resolution timers for tens-of-microseconds accuracy and precision, and preemptible RCU which has slightly higher read-side overhead than classic RCU. This information helped understand the trade-offs involved in real-time Linux systems.

De Oliveira et al. [27] critique the traditional `cyclictest` tool used for measuring scheduling latency, highlighting its limitations due to its black-box approach and lack of theoretical grounding. This contributed to the decision not to use this tool to measure the scheduling latency in this work. Instead, the `latency` tool was used, as was determined in Section 1.2.

1.4 Problem and Task Definition

As stated previously, virtualization of real-time systems has many advantages in comparison to traditional hardware-based approaches. Solutions based on pure hardware are often built for a distinct purpose and are hence limited in flexibility. It is difficult to adapt the hardware topology to new requirements or new applications. Moreover, during periods when the demand is lower, expensive hardware may sit idle, leading to inefficient use of resources and a poor return on investment [28]. Another crucial factor is, of course, the high cost factor associated with procuring and maintaining separate hardware for each real-time system [29]. Especially when organizations scale their operations, they have to add more physical machines, and the expenses rise rapidly [30]. Besides, one needs physical access to the hardware for any updates or maintenance, which is hard to achieve, especially for systems in remote or difficult-to-access locations. This can lead to system downtime and may not only disrupt the operations but can also result in lost productivity and potential revenue [31].

Virtualization promises to overcome these challenges [29] [32] [33]. However, virtualization also has its own challenges and pitfalls. The additional layers of abstraction can lead to increased overhead and latency [34] [35]. This is not desired and can be particularly problematic for real-time systems that require strict timing constraints. Making sure that tasks run predictably and on time in a virtualized environment can be challenging. The hypervisor and other virtual components can cause variations in how long tasks take to execute [18]. When there are multiple virtual machines (VMs) sharing the same physical resources, the VMs can affect each other's performance and consequently violate real-time constraints [32].

In this master's thesis, the task is the virtualization of a real-time operating system to control a robot, with a focus on compliance with real-time determinism. Chapter 2 explains in detail the procedure, how this was tackled. As already established in Section 1.2, Salamander 4 is built with Yocto, employs hard real-time with Xenomai 3, and is virtualized through QEMU/KVM. The latency values are measured with the `latency` program of the Xenomai tool suite. The initial situation is described in Section 3.1. In a nutshell, there is a significant initial gap in latency between the virtualized Salamander 4 and the Salamander 4 on bare metal. For this reason, an extensive tuning process is carried out to achieve real-time performance and determinism. These modifications involve configurations spanning the BIOS, kernel, host OS, QEMU/KVM virtualization layer, and the Salamander 4 guest OS itself. The individual configurations are discussed in detail in Section 3.2 and the modifications are justified with a clear explanation.

1.5 Objective

The goal is to bring the latency of the virtualized Salamander 4 closer to that of the hardware (bare metal), thereby providing a comprehensive blueprint for making a virtualized guest system in a host system real-time capable with deterministic behavior. Section 3.1 outlines the initial situation of this master's thesis and serves as the foundation for the real-time performance tunings discussed in Section 3.2. The objective is to achieve latency values below 50 µs during the measurement period through real-time tunings, with no outliers whatsoever. To determine and validate the performance of the tunings, the `latency` tool is used, as stated in Section 1.2. On top of that, the aim is to use a robotic application in a practical scenario to demonstrate and validate that the tunings significantly reduce latency and improve determinism compared to the unmodified virtualization.

2 Methodology

This chapter describes in detail all the theoretical concepts and practical methods that contributed to achieving the objectives of this master's thesis.

2.1 Host Operating System

The host operating system is the basis for the virtualization of the guest operating system. In the course of this work, it is continuously tuned to achieve real-time behavior and determinism. Table 1 illustrates the specifications of the host.

Table 1: Host Operating System Configuration

CPU	Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 6 cores
Memory	4 × 8GB DDR4-2666/2400 MHz, 32GB
GPU	Intel UHD Graphics 630 GPU
Storage	500GB NVMe SSD
BIOS	Dell Version 1.29.0
OS	Ubuntu 22.04.4 LTS
Kernel Version	6.8.0-40-generic

Figure 5 is the output of the `lstopo` command and visualizes the hardware nodes of the system, including CPU cores, caches, memory, and I/O devices.

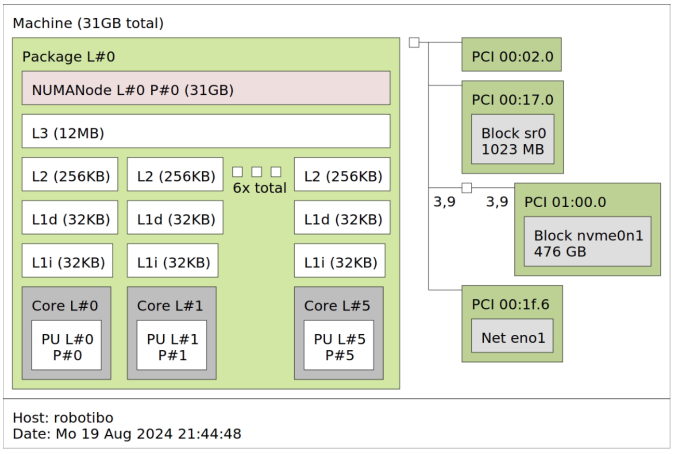

Figure 5: Host Operating System Hardware Topology

2.2 Guest Operating System

This section briefly describes the guest operating system, Salamander 4.

2.2.1 Structure

Salamander 4 is the proprietary operating system of SIGMATEK. It is based on Linux version 5.15.94 and integrates Xenomai 3.2, a real-time development environment. Salamander 4 is a 64-bit system, which refers to the x86_64 architecture. The real-time behaviour is achieved through the use of Symmetric Multi-Processing (SMP) and Preemptive Scheduling (PREEMPT). In addition, it uses IRQPIPE to process interrupts in a way that meets the real-time requirements of the system. The output of the command `uname -a` can be observed in Code 1.

```
1 root@sigmatek-core2:~# uname -a
2 Linux sigmatek-core2 5.15.94 #1 SMP PREEMPT IRQPIPE Tue Feb 14 18:18:05 UTC 2023
      x86_64 GNU/Linux
```

Code 1: Guest Operating System Information

Salamander 4 is powered by SIGMATEK's CP 841 [36] and is comprised of the following software modules:

- **Operating system**: The operating system in a LASAL CPU manages the hardware and software resources of the system. It is provided in a completely PC-compatible manner, working with a standard PC BIOS.
- **Loader**: The loader is a part of the operating system that is responsible for loading programs from executables into memory, preparing them for execution and then running them.
- **Hardware classes**: Hardware classes in LASAL represent the different types of hardware components that can be controlled by the LASAL CPU. They help organize and manage the hardware components in a modular manner. The graphical hardware editor in LASAL allows for a accurate simulation of the actual hardware.
- **Applications**: Applications are developed using LASAL CLASS 2 [37], a solution for automation tasks that supports object-oriented programming and design in compliance with IEC 61131-3.

These modules and the interfaces (indicated by an arrow) between them are shown in Figure 6.

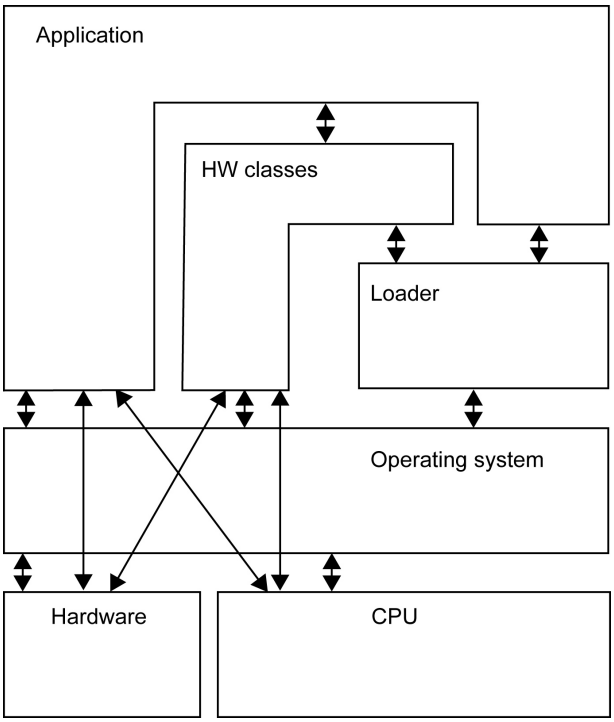

Figure 6: Structure of Salamander 4 CPU [38]

2.2.2 Memory Management

For the sake of completeness, Figure 7 displays the memory management of Salamander 4. LRT stands for Lasal Runtime and creates an execution environment where applications developed using LASAL Class 2 can run. It provides real-time functions, data types, and other constructs for real-time programming.

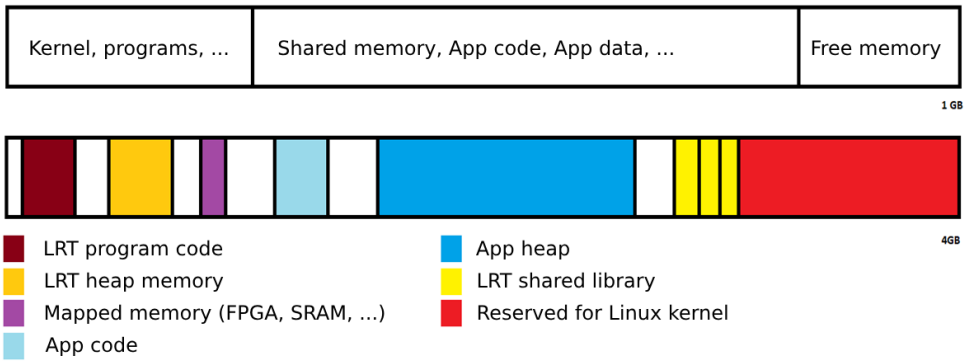

Figure 7: Memory Management of Salamander 4 [38]

2.3 QEMU

Quick Emulator (QEMU) [39] is an open-source emulator and virtualizer that allows running operating systems and programs for one machine on a different machine. It can emulate different hardware architectures and is used in this thesis to emulate Salamander 4. When QEMU is used together with Kernel-based Virtual Machine (KVM), it can provide near-native performance by running code directly on the host CPU. The QEMU script used to emulate Salamander 4 and the meaning of the options are described in detail in Subsection 3.1.2.

2.4 Yocto

Salamander 4 is built with Yocto [40], an open-source project that helps develop custom Linux-based operating systems for embedded devices. It is not a Linux version itself but a framework to make custom versions. From the kernel settings to the software packages included, almost everything in the Linux distribution can be adjusted with Yocto, hence it fits custom needs. Yocto organizes different components of the system into layers, making code management and updates easier. It builds the operating system with BitBake and automates the compilation and assembly of software. Upon generating the necessary files, Yocto provides a QEMU folder with the following components shown in Code 2.

```
1  sigma_ibo@localhost:~/build/tmp/deploy/qemu/salamander-image$ ls -1
2  bzImage
3  drive-c
4  ovmf.code.qcow2
5  qemu_def.sh
6  salamander-image-sigmatek-core2.ext4
7  stek-drive-c-image-sigmatek-core2.tar.gz
8  vmlinux
```

Code 2: Contents of QEMU Folder for Salamander 4

The following is a description of the components used for the virtualization of Salamander 4.

- **bzImage**: Compressed Linux kernel image that is loaded by QEMU at system start. The short form “bz“ stands for big-zipped.
- **drive-c**: Directory serving as C drive for QEMU, created and filled by `qemu_def.sh` script.
- **ovmf.code.qcow2**: Firmware file for QEMU that enables UEFI boot process. `OVMF` stands for Open Virtual Machine Firmware, `qcow2` is a format for disk image files used by QEMU, it stands for “QEMU Copy On Write version 2”.

- **qemu_def.sh**: Shell script that starts QEMU with required parameters to boot the Salamamder 4 OS. It is described later in more detail.
- **salamander-image-sigmatek-core2.ext4**: Disk image of the Salamander 4 OS. It uses the `ext4` file system and serves as the root file system in the QEMU virtual machine, acting as the virtual hard drive.
- **stek-drive-c-image-sigmatek-core2.tar.gz**: Compressed tarball containing a pre-configured environment for the Salamander 4 OS. It is unpacked and sets up the `drive-c/` directory with system and log files in the `qemu_def.sh` script.
- **vmlinux**: Uncompressed Linux kernel image, typically used for debugging.

The initial QEMU script after the custom Yocto build and the starting point for this work is shown in Code 3. This script is used to start QEMU with required parameters to boot the Salamander 4 OS. It will be adjusted in Section 3.2 in order to accompany real-time performance tunings.

```
1  #!/bin/sh
2
3  if  [ ! -d drive-c/ ]; then
4        echo "Filling drive-c/"
5        mkdir drive-c/
6        tar -C drive-c/ -xf stek-drive-c-image-sigmatek-core2.tar.gz
7  fi
8
9  exec qemu-system-x86_64 -M pc,accel=kvm -kernel ./bzImage \
10 -m 2048 -drive
       file=salamander-image-sigmatek-core2.ext4,format=raw,media=disk \
11 -append "console=ttyS0 console=tty1 root=/dev/sda rw panic=1
       sigmatek_lrt.QEMU=1 ip=dhcp rootfstype=ext4" \
12 -net nic,model=e1000,netdev=e1000 -netdev bridge,id=e1000,br=nm-bridge \
13 -fsdev local,security_model=none,id=fsdev0,path=drive-c -device
       virtio-9p-pci,id=fs0,fsdev=fsdev0,mount_tag=/mnt/drive-C \
14 -device vhost-vsock-pci,guest-cid=3,id=vsock0 \
15 -drive if=pflash,format=qcow2,file=ovmf.code.qcow2 \
16 -no-reboot -nographic
```

Code 3: QEMU Script for starting Salamander 4 Virtualization

This script is run on a generic Ubuntu 22.04.4 system, as mentioned previously in Section 2.1.

2.5 Xenomai

Xenomai 3 [41] is an open-source real-time framework that offers two paths to real-time performance. The first approach supplements the Linux kernel with a compact real-time core dubbed Cobalt, demonstrated in Figure 8. Cobalt runs side-by-side with Linux, but it handles all time-critical activities like interrupt processing and real-time thread scheduling with higher priority than the regular kernel activities.

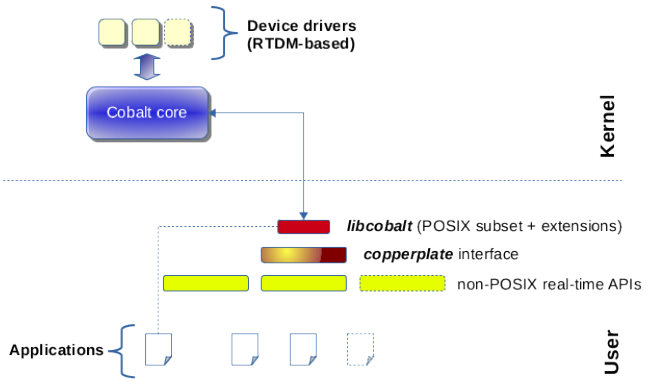

Figure 8: Xenomai 3 Cobalt Interfaces [42]

The second approach, called Mercury and shown in Figure 9, relies on the real-time capabilities already present in the native Linux kernel. Often, applications require the PREEMPT-RT patch [43] to augment the mainline kernel's real-time responsiveness and minimize jitter, but this is not mandatory and depends on the application's specific requirements for responsiveness and tolerance for occasional deadline misses.

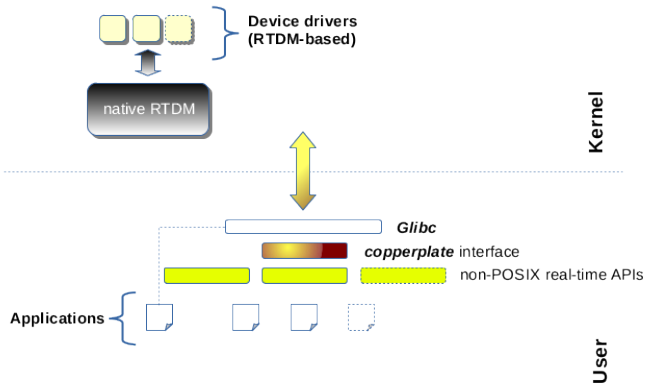

Figure 9: Xenomai 3 Mercury Interfaces [42]

Salamander 4 uses the Cobalt real-time core with the Dovetail extension, which allows the kernel to handle real-time tasks with low latency. A key tool of the Xenomai suite that is used in this work is the `latency` tool [44]. It benchmarks the timer latency, which is the time it takes for the kernel to respond to timer interrupts or task activations. The tool creates real-time tasks

or interrupt handlers and measures the difference between the expected and actual execution times.

2.6 Trace-cmd

`Trace-cmd` [45] is the front-end tool of the `ftrace` [46] tool that is built inside the Linux kernel and used for tracing the Linux kernel. `Trace-cmd` can record detailed reports of kernel events such as interrupts, scheduler decisions, function calls, and custom events in real time. It is easier with this tool to enable and disable tracing, set filters, and view results. `Trace-cmd` helped debugging and identifying the reasons for latency in Salamander 4 to continuously improve its real-time performance and determinism. A very practical aspect of `trace-cmd` is that the guest can be traced from the host. For this purpose, the guest needs to be configured with `vsocks`, which are a virtual socket that enable direct communication between the host and guest. To enable vsockets and tracing, the guest kernel requires the following configurations, according to Linux developer and maintainer Steve Rostedt [47].

```
# Vsockets settings
CONFIG_VSOCKETS=m
CONFIG_VHOST_VSOCK=m
CONFIG_VIRTIO_VSOCKETS=m
CONFIG_VIRTIO_VSOCKETS_COMMON=m
CONFIG_VSOCKETS_DIAG=m
CONFIG_VSOCKETS_LOOPBACK=m
# Tracing settings
CONFIG_TRACING=y
CONFIG_FTRACE=y
CONFIG_FUNCTION_TRACER=y
CONFIG_FUNCTION_GRAPH_TRACER=y
CONFIG_DYNAMIC_FTRACE=y
CONFIG_DYNAMIC_FTRACE_WITH_REGS=y
CONFIG_DYNAMIC_FTRACE_WITH_DIRECT_CALLS=y
CONFIG_DYNAMIC_FTRACE_WITH_ARGS=y
CONFIG_SCHED_TRACER=y
CONFIG_FTRACE_SYSCALLS=y
CONFIG_TRACER_SNAPSHOT=y
CONFIG_KPROBE_EVENTS=y
CONFIG_UPROBE_EVENTS=y
CONFIG_BPF_EVENTS=y
CONFIG_DYNAMIC_EVENTS=y
CONFIG_PROBE_EVENTS=y
CONFIG_SYNTH_EVENTS=y
CONFIG_HIST_TRIGGERS=y
```

Code 4: Kernel Flags for Vsocks and Tracing

After compiling the kernel with the additional settings above, the guest needs access to the vsocket. In QEMU, the line `-device vhost-vsock-pci,guest-cid=3,id=vsock0` was added. This command configures a virtual socket (`vsock`) device in QEMU, assigning the guest a unique Context Identifier (`cid`) of 3 for direct communication with the host. The CID can be chosen as desired. Once the guest is booted with `vhost` enabled, the `sudo trace-cmd agent` command can be run. The agent now listens to connections on port 823 by default with the specified CID, 3 in this case. The host can trace itself and the guest from this moment going forward. For example, `sudo trace-cmd record -e all -A @3:823 -name Salamander4 -e all` records all events from the host and guest. `sudo trace-cmd record -e kvm:kvm_entry -e kvm:kvm_exit -A @3:823 -name Salamander4 -e all` traces only the `kvm_entry` and `kvm_exit` events of the host along with the exit reasons and all guest events. As a last example, `sudo trace-cmd record -e kvm -e sched -e irq -e -A @3:823 -name Salamander4 -e all` enables tracing of KVM, scheduling, and interrupt events on the host, for capturing the KVM events that manage the guest, scheduling activities, and interrupt handling respectively.

2.7 Kernelshark

Viewing the tracing data that was recorded with `trace-cmd` on a text-based file can be laborious. A better way of analyzing the data is done through `KernelShark` [48], which is a graphical front-end tool. It visualizes the recorded kernel trace data in a readable way on an interactive timeline, which facilitates the process of identifying patterns and correlations between events. The latency issues can be analyzed by further filtering the displayed events according to processes, event types, or time ranges. After the recording, `trace-cmd` generates a host `trace.dat` file and a guest `trace-Salamander4.dat` file, which in this case can be loaded into `KernelShark` with the command `kernelshark trace.dat -a trace-Salamander4.dat`. Once both files are loaded, the trace data can be visually inspected. All the current CPUs for both the host and the guest can be turned off, and the KVM Combo plots menu can be selected to view the guest vCPUs plotted on top of the host threads that run them, as depicted in Figure 10.

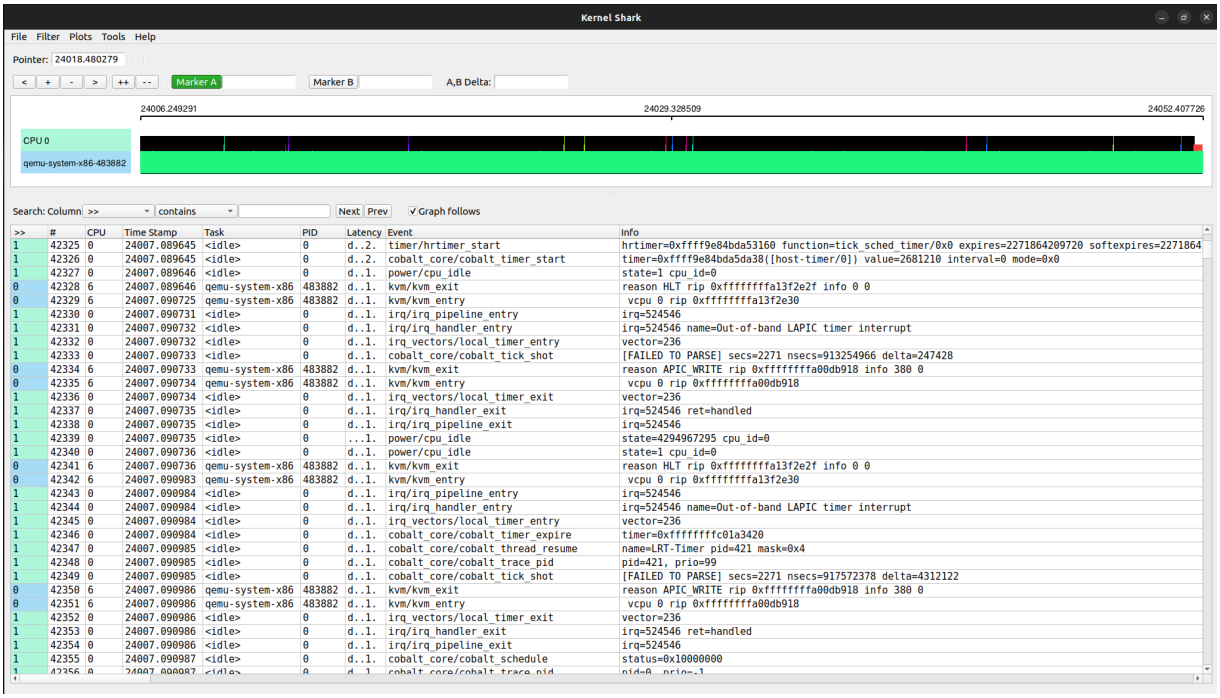

Figure 10: Guest vCPUs plotted on top of Host Threads in KernelShark

2.8 VARAN-Bus

VARAN is a real-time Ethernet bus designed for industrial automation, capable of connecting larger systems and machines with smaller components and sensors in hard real-time. This bus operates on the manager/client principle, where each network includes a manager and one or more subordinate clients. Multiple VARAN networks can be linked together into a single synchronized network using a higher-level VARAN manager. The jitter between these systems is kept below 100 ns. This synchronization between systems and their data exchange is illustrated in Figure 11.

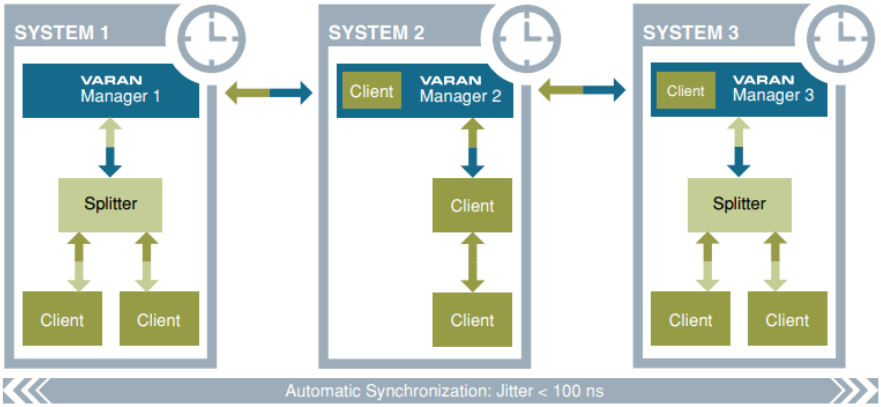

Figure 11: Synchronization between Systems and their Data Exchange

2.9 Approach

The goal is to virtualize the Salamander 4 OS and bring its real-time performance closer to that of the bare metal version and guarantee deterministic and reliable behavior. For that purpose, the first step is to apply the PREEMPT-RT patch to the host system and subsequently tune the real-time performance through BIOS, kernel, host, QEMU/KVM, and guest configurations. These modifications are added sequentially and tested together. First, the BIOS is configured, followed by the kernel. The BIOS settings are not reverted when moving to the kernel configurations. Next, the host is configured, but the BIOS and kernel settings remain unchanged. The guest OS is already based on Xenomai and therefore, no additional modifications are necessary for the guest OS itself. Finally, QEMU/KVM settings are applied. This means that the configurations are not isolated but are applied and then tested as a whole. The key metric here is the latency of the system, and the real-time performance is evaluated using the `latency` tool of the Xenomai suite after each configuration step. At the end, the results are demonstrated altogether and a holistic comparison and discussion are carried out. In addition, the improvement in real-time performance and determinism is demonstrated using a robotic application, where the tuned virtualization is compared with the untuned and the bare metal version.

3 Implementation

3.1 Initial Situation

As a starting point, initial latency values of both the bare metal and virtualization versions were measured with the `latency` tool of the Xenomai tool suite. Salamander 4 bare metal refers to the custom operating system developed by SIGMATEK, running on their proprietary hardware. Salamander 4 virtualization refers to a virtual version of Salamander 4, achieved through QEMU/KVM. Subsections 3.1.1 and 3.1.2 specify the details of the measurements for both setups. In the further course of this master's thesis, the aim is to bring the latency values of the virtualization closer to those of bare metal and guarantee deterministic and reliable behavior.

3.1.1 Salamander 4 Bare Metal

As a reference point, the `latency` program was executed on Salamander 4 bare metal for a duration of 10 minutes. The complete command used was `latency -h -g gnuplot.txt -T 600`, which runs the latency measurement tool for 600 seconds and prints histograms of minimum, average, and maximum latencies in a Gnuplot-compatible format to the file `gnuplot.txt`. Figure 12 depicts the distribution of latency over the course of said time.

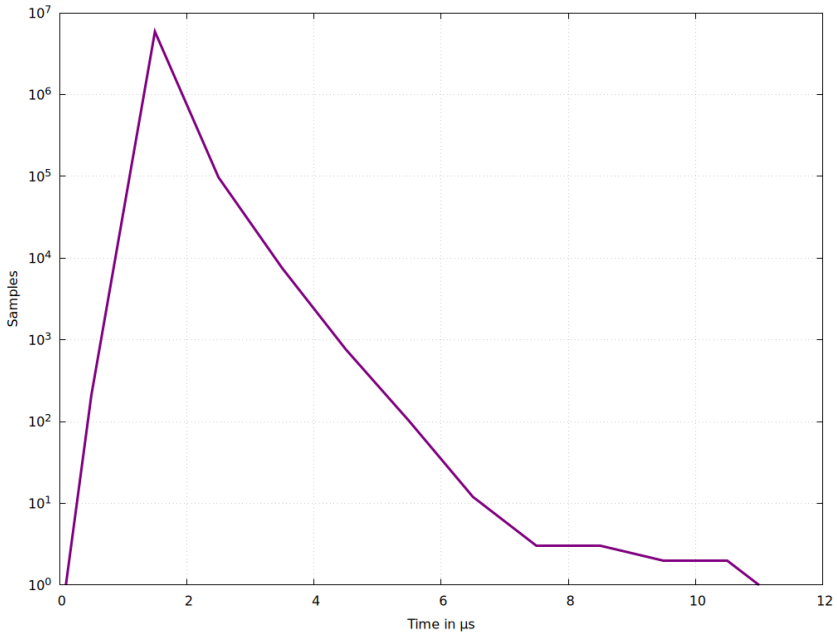

Figure 12: Latency Distribution of Salamander 4 Bare Metal

The statistics obtained from this measurement are provided in Table 2 and Table 3. The test was conducted with a sampling period of 100 µs, using a periodic user-mode task, and was assigned a priority of 99. If any sample's latency value is greater than 100 µs, it would be considered an overrun.

Table 2: Latency Parameters of Bare Metal

Param	Samples	Average (µs)	Std Dev (µs)
min	599	0.711	0.454
avg	5,999,988	1.019	0.150
max	599	3.528	0.895

Table 3: Minimum, Average, and Maximum Latency with Overrun Counts of Bare Metal

Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Overruns
0.613	1.380	10.709	0

3.1.2 Salamander 4 Virtualization

In addition to providing Salamander 4 on its own hardware, SIGMATEK has also developed a virtualized version of this operating system. Figure 13 shows the distribution of latency measured over 10 minutes using the default QEMU script in Code 3 without any further adjustments.

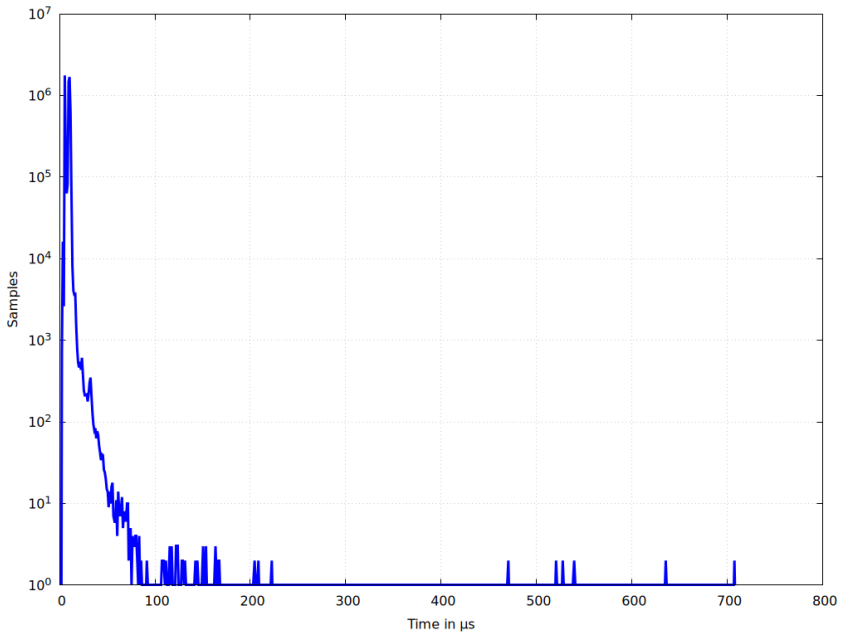

Figure 13: Latency Distribution of Salamander 4 Untuned Virtualization

The statistics obtained from this measurement are provided in Table 4 and Table 5. The test was again conducted with a sampling period of 100 µs, using a periodic user-mode task, and was assigned a priority of 99.

Table 4: Latency Parameters of Untuned Virtualization

Param	Samples	Average (µs)	Std Dev (µs)
min	599	3.713	1.355
avg	5,999,922	8.247	2.521
max	599	45.705	52.196

Table 5: Minimum, Average, and Maximum Latency with Overrun Counts of Untuned Virtualization

Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Overruns
2.536	8.940	707.622	43

3.1.3 Initial Comparison

Comparing the latency values of the virtualization to those of bare metal in Figure 14, it is evident that there is a significant initial gap in latency. A maximum latency of 707.62 µs is not tolerable for a real-time virtualization and needs to be tuned.

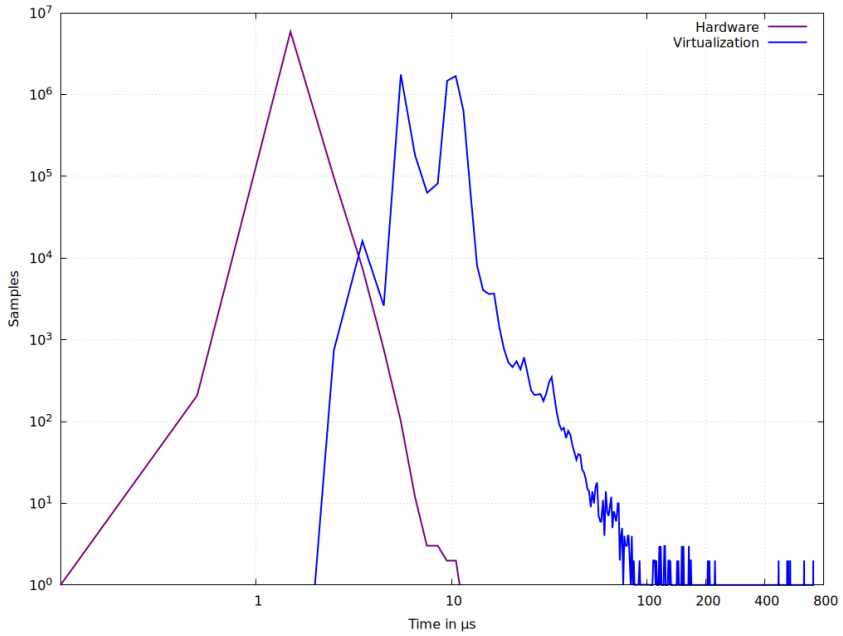

Figure 14: Initial Comparison of Latency Distribution between Bare Metal and Virtualization

3.2 Real-Time Performance Tunings

In this section, the initial large gap in latency between the bare metal and the virtualized system is tackled. For this reason, an extensive tuning process is carried out. This involves configurations spanning the BIOS, kernel, host OS, QEMU/KVM virtualization layer, and the Salamander 4 guest OS itself. The individual configurations are discussed in detail, and the modifications are justified with a clear explanation. The goal is to bring the latency of the virtualization closer to that of the bare metal, thereby ensuring deterministic behavior under real-time constraints.

It is important to mention that a real-time system implies determinism at every component of the system, ranging from hardware over the kernel to the software and the application on it. As already established earlier, the guest OS Salamander 4 runs Xenomai and is hence real-time capable. The host system also needs to be aware of the real-time determinism in order to achieve the highest possible reliability. The very first step of achieving the goal of reducing latency is to apply the PREEMPT-RT patch to the host system. This patch is a set of modifications to the Linux kernel with the goal of making it fully preemptible. This means almost all parts of the kernel can be preempted, and higher-priority tasks can interrupt lower-priority ones. This significantly reduces the time it takes for high-priority tasks to start executing after an event [49]. The patch also includes support for high-resolution timers and makes the system more predictable [50]. Determinism and predictability are key for real-time applications where the timing of operations must be guaranteed with precise timing and scheduling [51].

Nevertheless, a real-time kernel alone does not make a system truly “real-time” [52]. Additional modifications are required to achieve this. The next subsections will deal with these real-time performance tunings. After each tuning, the `latency` test of Xenomai is executed to evaluate the improved latency. The most important metric of a latency test is, without a doubt, the worst latency value, because the system is only as reliable as its worst measurement. Since the period of the tests is set to 100 µs, there are approximately 10,000 samples per second. Therefore, 10 minutes or 600 seconds correspond to approximately 6,000,000 samples.

3.2.1 BIOS Configurations

BIOS stands for Basic Input/Output System. It abstracts the hardware and enables basic functions of a computer during the booting process, such as starting the operating system and loading other software. Since the BIOS is embedded very deep, its configuration can significantly influence the real-time performance of the system. Table 6 illustrates the specific BIOS settings that have been adjusted for the purpose of real-time performance. As an important note, this is not a definitive or complete list. Other devices may have additional settings that can be modified to achieve even better latency.

Table 6: BIOS Configurations for Real-Time Performance

Option	**Status**
Hyper Threading	Disabled
Intel SpeedStep®	Disabled
Intel® Speed Shift Technology	Disabled
C-States	Disabled
VT-d	Enabled

In the following, these settings along with their impact on system latency are briefly described.

- **Hyper Threading**: When hyper-threading is enabled in the BIOS settings, CPUs can work on two threads simultaneously instead of just one. This allows the parallelization of tasks and increases performance. However, in a real-time system like the guest Salamander 4, this can lead to increased latencies due to contention between threads. In order to ensure more deterministic behavior in the guest, it is disabled on the host.
- **Intel SpeedStep**: This dynamically adjusts the clock speed of the CPU based on workload. These dynamic adjustments of the speed can lead to unpredictable latencies in a real-time system. It is also disabled on the host to maintain a constant CPU speed.
- **Intel® Speed Shift Technology**: Similar to SpeedStep, Speed Shift allows the processor to directly control its frequency and voltage. This can lead to unpredictable latencies, too. Hence, it is also disabled on the host.
- **C States**: These are low-power idle states where the clock frequency and voltage of the CPU are reduced. Transitioning between C-states can cause variable latencies. To prevent this from happening, C-states are disabled on the host.
- **VT-d**: Direct access to physical devices from within virtual machines is possible when VT-d is enabled on the host. This can help reduce latencies associated with I/O operations in the virtual machine. It is therefore enabled on the host.

After applying the PREEMPT-RT patch and first real-time tunings in the BIOS settings, the `latency` test was run to see whether this had any impact on the determinism of the system. Figure 15 shows the latency values after tuning BIOS configurations.

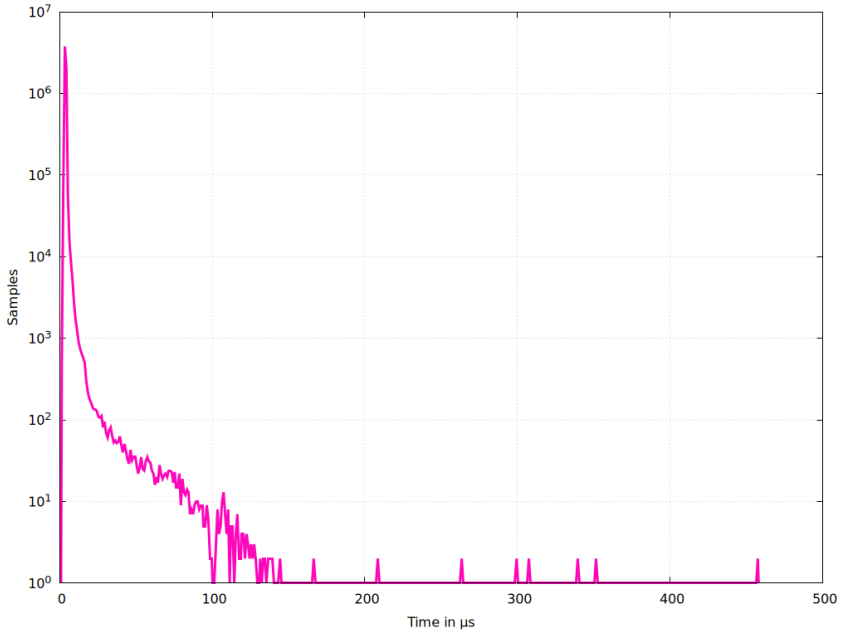

Figure 15: Latency Distribution of Salamander 4 Virtualization after tuning BIOS Configurations

The statistics obtained from this measurement are provided in Table 7 and Table 8.

Table 7: Latency Parameters after BIOS Configurations

Param	Samples	Average (µs)	Std Dev (µs)
min	599	1.419	0.507
avg	5,999,885	3.398	1.251
max	599	74.015	26.590

Table 8: Minimum, Average, and Maximum Latency with Overrun Counts after BIOS Configurations

Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Overruns
0.969	3.948	457.545	22

The maximum latency value recorded was 457.545 µs. When compared to the initial worst latency of 707.62 µs, this is a step for the better, but certainly still not acceptable within this work's definition of hard real-time determinism. Besides, the number of overruns decreased from 43 to 22.

3.2.2 Kernel Configurations

The kernel command-line parameters are shown in Code 5 below. To access and modify them, the file is located in `/etc/default/grub`.

```
1       GRUB_CMDLINE_LINUX="isolcpus=4 rcu_nocbs=4 rcu_nocb_poll nohz_full=4
            default_hugepagesz=1G hugepagesz=1G hugepages=8 intel_iommu=on
            rdt=l3cat nmi_watchdog=0 idle=poll clocksource=tsc tsc=reliable audit=0
            skew_tick=1 intel_pstate=disable intel.max_cstate=0
            intel_idle.max_cstate=0 processor.max_cstate=0
            processor_idle.max_cstate=0 nosoftlockup no_timer_check nospectre_v2
            spectre_v2_user=off kvm.kvmclock_periodic_sync=N kvm_intel.ple_gap=0
            irqaffinity=0"
```

Code 5: Kernel Configurations for Real–Time Performance

In the following, these settings along with their impact on system latency are briefly described.

- **isolcpus=4**: Isolates CPU 4 from the general scheduler, meaning no process will be scheduled to run on this CPU unless it is explicitly assigned. CPU isolation is explained in detail in Subsubsection 3.2.3.1.
- **rcu_nocbs=4**: The Linux kernel uses a synchronization mechanism called RCU, or Read-Copy-Update. It lets writers update the data in a way that guarantees readers will always see the same version while enabling multiple readers to access shared data without locks. The RCU subsystem uses callback functions that need to be invoked once readers are done with the data they accessed. By default, these callbacks are handled by the CPUs that executed the read-side critical sections. This parameter offloads RCU callback handling from CPU 4 to other CPUs. CPU 4 remains dedicated to high-priority tasks, which helps in reducing latency.
- **rcu_nocb_poll**: This is used together with `rcu_nocbs` and causes the system to actively poll for RCU callbacks to invoke, instead of waiting for the next RCU grace period. This reduces latency.
- **nohz_full=4**: Makes CPU core 4 "tickless", meaning the kernel tries to avoid sending periodic scheduling-clock interrupts to the CPU when there are no runnable tasks. This lowers latency by reducing unnecessary wake-ups but may increase power consumption because the CPU is not able to enter a low-power state when idle. Additionally, timer interrupts cannot be fully eliminated because certain events, such as incoming interrupts or task activations, can still cause the kernel to send timer interrupts to the tickless CPU.
- **default_hugepagesz=1G, hugepagesz=1G, hugepages=8**: Huge pages are large contiguous areas of memory that can be used by applications and the kernel, instead of the traditional 4KB small pages. The default huge page size is set to 1GB, and 8 huge

pages of 1GB size are reserved at boot. This pre-allocation makes sure that these large memory regions are available to be used by the kernel or applications without having to dynamically allocate and potentially fail.

- **intel_iommu=on**: Enables Intel's IOMMU (Input/Output Memory Management Unit), which connects a DMA (Direct Memory Access)-capable I/O bus to the main memory. It allows these devices to directly access and use memory, which is especially helpful for virtualization and device passthrough.

- **rdt=l3cat**: Activates the L3 CAT (L3 Cache Allocation Technology) feature of Intel's RDT (Resource Director Technology). Unlike L1 and L2 caches, where each core has its fixed capacity, L3 cache is a shared pool among multiple cores. L3 CAT controls the amount of L3 cache that a process can use. By controlling cache allocation, it can prevent a single process from monopolizing the L3 cache, which is particularly beneficial in virtualized environments where multiple virtual machines share the same physical host.

- **nmi_watchdog=0**: Disables the NMI (Non-Maskable Interrupt) watchdog, which is a debugging feature of the Linux kernel. It works by periodically generating non-maskable interrupts. If the system does not respond to these interrupts within a certain timeframe, the NMI watchdog concludes that the system has hung and generates a system dump for debugging. This constant monitoring consumes CPU cycles and can introduce undesirable latency in real-time systems.

- **idle=poll**: Changes the CPU's idle loop behavior to active polling. Instead of entering a low-power state when idle, the CPU continuously polls for new tasks. This can reduce task start latency in real-time systems, but it increases power consumption.

- **clocksource=tsc, tsc=reliable**: TSC (Time Stamp Counter) is a high-resolution timer provided by most x86 processors that counts the number of CPU cycles since it was last reset. Accurate timekeeping is crucial, particularly for real-time systems. These parameters set the clocksource to TSC and mark it as a reliable source of timekeeping, meaning it increments at a consistent rate and does not stop when the processor is idle.

- **audit=0**: Disables the Linux audit system. When it is enabled, it generates log entries for security-relevant events, which is a slow operation since they are written to disk. If there are a large number of such events, the audit system can consume significant CPU time and I/O bandwidth, which could lead to higher latency.

- **skew_tick=1**: Enables a mode in the Linux kernel that reduces timer interrupt overhead. Normally, timer interrupts happen simultaneously on all CPUs, resulting in all CPUs exiting their low-power states at once. This can lead to increased contention for system resources. When enabled, the kernel offsets the timer interrupts on different CPUs, spreading them out over time.

- **intel_pstate=disable**: Disables the Intel P-state driver, which is a part of the Linux kernel that handles power management for Intel CPUs. It controls the frequency of the CPU by scaling it up when demand is high and scaling it down to save power when demand is low. This dynamic frequency scaling is disabled because it leads to increased latencies for real-time systems.
- **intel.max_cstate=0, intel_idle.max_cstate=0, processor.max_cstate=0, processor_idle.max_cstate=0**: These parameters disable deeper C-states (CPU power saving states). Normally, when a CPU is idle, it can enter various C-states, with higher-numbered states representing deeper sleep states that save more power but take longer to wake up from. In real-time systems, these wake-up delays can be problematic. Disabling them keeps the CPUs ready to respond quickly to new tasks and helps in reducing latency.
- **nosoftlockup**: Disables the soft lockup detector in the Linux kernel. A soft lockup is when a CPU is busy executing kernel code for a long period of time without giving other tasks a chance to run. Especially threads with SCHED_FIFO policy occupy the CPU for an extensive duration. This is detected and reported by the soft lockup detector, hence it is disabled to prevent these unnecessary warnings.
- **no_timer_check**: Disables the check for broken timer interrupt sources. Broken timer interrupt sources are problems with hardware or software that prevent timer interrupts from working as intended. Such a timer may not generate interrupts at the expected rate or at all. The kernel skips the checks for these broken timer interrupt sources, which can cause unnecessary overhead in a real-time system where every CPU cycle counts.
- **nospectre_v2, spectre_v2_user=off**: These parameters disable mitigations for the Spectre v2 vulnerability. Spectre v2 is a hardware vulnerability that affects many modern microprocessors and can allow malicious programs to access sensitive data they are not supposed to. While this is necessary for security, it has an impact on performance and should be turned off in controlled environments where the risk of exploitation is low.
- **kvm.kvmclock_periodic_sync=N, kvm_intel.ple_gap=0**: These are KVM (Kernel-based Virtual Machine) related parameters. They disable the periodic synchronization of the kvmclock and set the gap between PLE (Pause Loop Exiting) events to 0. The kvmclock is a paravirtualized clock source provided by KVM to its guest OS and disabling this reduces latency introduced by clock synchronization. By setting the gap to 0, the virtual machine exits the pause loop immediately, which can reduce latency in spinlock-intensive workloads.
- **irqaffinity=0**: Sets the default IRQ affinity mask so that interrupts are handled by non-isolated CPUs. This means that no CPU core is preferred over another for handling IRQs. Instead, it lets the operating system decide how to distribute these IRQs across all the CPUs. Subsubsection 3.2.3.2 dives deeper into interrupt request affinity.

After the kernel modifications on top of the BIOS settings from the last subsection, Figure 16 demonstrates the latency values gathered in 10 minutes.

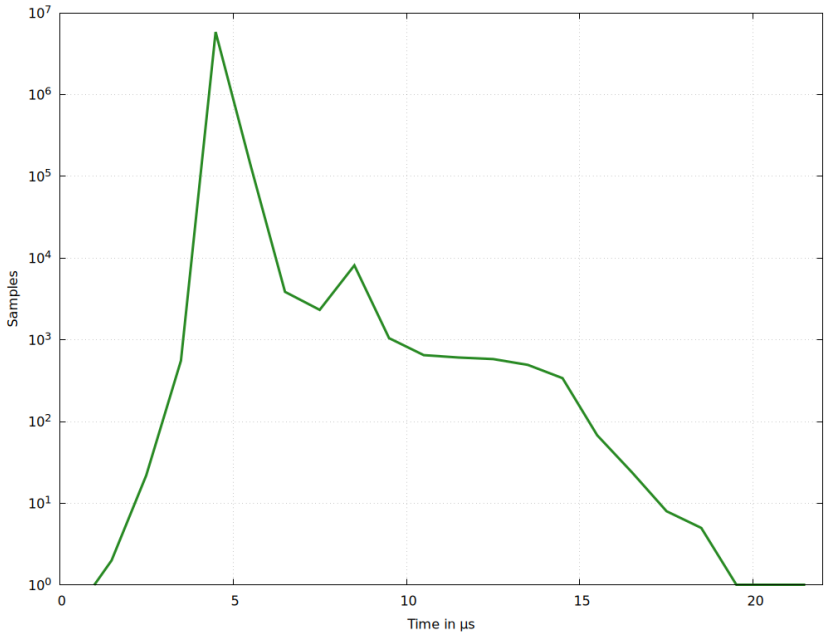

Figure 16: Latency Distribution of Salamander 4 Virtualization after tuning Kernel Configurations

The statistics obtained from this measurement are provided in Table 9 and Table 10.

Table 9: Latency Parameters after Kernel Configurations

Param	Samples	Average (µs)	Std Dev (µs)
min	599	3.356	0.551
avg	5,999,972	4.036	0.290
max	599	13.484	1.454

Table 10: Minimum, Average, and Maximum Latency with Overrun Counts after Kernel Configurations

Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Overruns
2.545	4.811	21.694	0

There is a significant improvement in the latency of the virtualized Salamander 4 OS. With a worst latency value of 21.694 µs, the real-time tunings appear to be working as desired. Additionally, no overruns occurred during the latency test. This means that the system consistently stayed within the required time constraints

3.2.3 Host OS Configurations

The host OS needs to provide an environment where the guest OS can operate in real-time. This entails a number of host-side adjustments to reduce interruptions and latency. In the following, a detailed overview of these configurations and their impact on the real-time performance of the Salamander 4 guest OS is provided.

3.2.3.1 CPU Affinity and Isolation

Isolating CPUs means removing all user-space threads and unbound kernel threads, since bound kernel threads are tied to specific CPUs and hence cannot be moved. For CPU isolation, the `isolcpus` function is used to isolate a CPU from the general scheduling algorithms of the operating system. This means that the isolated CPUs will not be used for regular task scheduling, allowing them to be dedicated for the real-time specific tasks. However, the `isolcpus` function only isolates at the user level and does not affect kernel tasks. Consequently, these kernel tasks and interrupts can still use the CPU [20], including systemd services. To prevent systemd services from running on an isolated CPU, the CPU affinity can be set with the line `CPUAffinity=0 1 2 3 5` in the `/etc/systemd/system.conf` file to indicate the CPUs that systemd services are allowed to run on. Every CPU other than the isolated CPU 4 is allowed. Code 6 shows the user and kernel tasks that run on CPU 4. An isolated CPU can be bound to a task by using `taskset -c <CPU>`, as shown later in Subsection 3.2.4. After the isolation, user tasks other than the QEMU process have been removed from running on this CPU. Only a few per-CPU kernel threads that are tied to this CPU still take CPU time.

```
1  sigma_ibo@sigma-ibo:~$ cat /sys/devices/system/cpu/isolated
2  4
3  sigma_ibo@sigma-ibo:~$ ps axHo psr,pid,lwp,args,policy,nice,rtprio | awk
       '$1 == 4'
4  4      38      38 [cpuhp/4]                    TS    0      -
5  4      39      39 [idle_inject/4]              FF    -     50
6  4      40      40 [migration/4]                FF    -     99
7  4      41      41 [ksoftirqd/4]                TS    0      -
8  4      42      42 [kworker/4:0-events]         TS    0      -
9  4      43      43 [kworker/4:0H-kblockd]       TS  -20      -
10 4     153     153 [kworker/4:1-events]         TS    0      -
11 4   81649   81649 qemu-system-x86_64 -M pc,ac  TS    0      -
12 4   81649   81654 qemu-system-x86_64 -M pc,ac  TS    0      -
13 4   81649   81676 qemu-system-x86_64 -M pc,ac  TS    0      -
14 4   81649   81702 qemu-system-x86_64 -M pc,ac  TS    0      -
15 4   81649   82185 qemu-system-x86_64 -M pc,ac  TS    0      -
16 4   81649   82187 qemu-system-x86_64 -M pc,ac  TS    0      -
17 4   82134   82134 [kworker/4:1H-kblockd]       TS  -20      -
```

Code 6: User and Kernel Tasks on the isolated CPU

3.2.3.2 Interrupt Affinity

Once the CPUs are isolated, interrupt request handling is the next step. The purpose of interrupt requests is to inform the CPU to stop working on a certain job and start working on another. This allows hardware devices to communicate with the CPU through frequent context switches, which can introduce latency in real-time systems. All the existing interrupts can be monitored using the command `watch -d -n 1 cat /proc/interrupts` to observe changes in the interrupt requests handled by each CPU in real-time. The IRQs need to be removed from the isolated CPU by manipulating the `/proc/irq/<IRQ>/smp_affinity` files. The value in the `smp_affinity` file is a bitmask in hexadecimal format where each bit corresponds to a CPU. The Least Significant Bit (LSB) on the right corresponds to the first CPU (CPU0), and the Most Significant Bit (MSB) on the left corresponds to the last CPU (CPU5). When there are 6 CPUs available, the default value for `smp_affinity` would be 3F (11 1111). Removing CPU 4 out of this bitmask would be setting bit five to zero, resulting in 2F (10 1111). Looking individually in the `/proc/irq/<IRQ>/smp_affinity` files for every IRQ would be time-consuming. The Python script in Code 7 was written to show a table of the distribution of interrupt requests across each CPU, since the `/proc/interrupts` file only shows if an interrupt has occurred and does not indicate which CPUs are capable of serving which IRQs. The program first reads the `smp_affinity` files for each IRQ from the `/proc/irq` directory and stores the CPUs assigned to handle each IRQ in a dictionary. It then creates a table showing the distribution of interrupt requests across all CPUs.

As the `proc` filesystem resets to its default state after each reboot, manually changing numerous IRQ files through the said bitmasks is tedious and time-consuming. This process can be automated using the shell script in Code 8, which can automatically be executed after every reboot.

```
import os
import pandas as pd
from tabulate import tabulate

# Get the number of CPUs
num_cpus = os.cpu_count()

# Initialize a dictionary to store the CPUs for each IRQ
irqs = {}

# Iterate over each IRQ
for irq in os.listdir('/proc/irq'):
    # Check if the smp_affinity file exists for this IRQ
    if os.path.isfile(f'/proc/irq/{irq}/smp_affinity'):
        # Read the current smp_affinity
        with open(f'/proc/irq/{irq}/smp_affinity', 'r') as f:
            affinity = int(f.read().strip(), 16)
        # Initialize an empty list to store the CPUs for this IRQ
        cpus = []
        # Iterate over each CPU
        for cpu in range(num_cpus):
            # Check if the bit for the current CPU is set
            if ((affinity & (1 << cpu)) != 0):
                # Add the CPU to the list for this IRQ
                cpus.append(cpu)
        # Sort the list of CPUs
        cpus.sort()
        # Add the list of CPUs to the dictionary for this IRQ
        irqs[irq] = cpus

# Create a DataFrame to store the table
df = pd.DataFrame(index=sorted(irqs.keys(), key=int),
    columns=range(num_cpus))

# Fill the DataFrame with 'x' where a CPU is assigned to an IRQ
for irq, cpus in irqs.items():
    for cpu in cpus:
        df.loc[irq, cpu] = 'x'

# Replace NaN values with empty strings
df.fillna('', inplace=True)

# Print the table in pipe format
print(tabulate(df, headers='keys', tablefmt='pipe', showindex=True))

# Convert the DataFrame to a markdown table
markdown_table = df.to_markdown()

# Write the markdown table to a file
with open('table_CPU_IRQ.md', 'w') as f:
    f.write(markdown_table)
```

Code 7: Code to show IRQ distribution across each CPU

```
#!/bin/bash

# Check if a command-line argument is provided
if [ -z "$1" ]; then
   echo "Please provide a CPU number as a command-line argument."
   exit 1
fi

# Get the CPU number from the command-line argument
CPU=$1

# Define the mask values
declare -A mask_values
mask_values=( [0]="3ffe" [1]="3ffd" [2]="3ffb" [3]="3ff7" [4]="3fef"
    [5]="3fdf" [6]="3fbf" [7]="3f7f" [8]="3eff" [9]="3dff" [10]="3bff"
    [11]="37ff" [12]="2fff" [13]="17ff")

# Run the check_smp_affinity.sh script and get the IRQs
IRQs=$(./check_smp_affinity.sh $CPU | grep -o '[0-9]\+')

# Initialize an empty array to store the IRQs that could not be removed
failed_IRQs=()
# Initialize an empty array to store the IRQs that were successfully
    removed
succeeded_IRQs=()

# Loop over the IRQs
for IRQ in $IRQs; do
   # Try to change the smp_affinity
   echo ${mask_values[$CPU]} | sudo tee /proc/irq/$IRQ/smp_affinity >
       /dev/null 2>&1
   # If the command failed, add the IRQ to the failed_IRQs array
   if [ $? -ne 0 ]; then
      failed_IRQs+=($IRQ)
   else
      succeeded_IRQs+=($IRQ)
   fi
done

# Check if there were any failed IRQs
if [ ${#failed_IRQs[@]} -ne 0 ]; then
   echo "IRQs ${failed_IRQs[@]} could not be removed from CPU $CPU."
fi

# Check if there were any successful IRQs
if [ ${#succeeded_IRQs[@]} -ne 0 ]; then
   # Remove the first entry from the succeeded_IRQs array
   succeeded_IRQs=("${succeeded_IRQs[@]:1}")
   echo "IRQs ${succeeded_IRQs[@]} were removed from CPU $CPU."
fi
```

Code 8: Script to change IRQ Assignment of a CPU

3.2.3.3 RT-Priority

Having a real-time kernel itself is an indispensable step of achieving deterministic behavior, but not enough by itself to take full advantage of the real-time capabilities. One key aspect of this is real-time priorities, thoroughly explained by Richard Weinberger [53]. In essence, Table 11 lists minimum and maximum priorities for different scheduling policies in Linux.

Table 11: Minimum and Maximum Priorities for Different Scheduling Policies

Scheduling Policy	Min Priority	Max Priority
SCHED_OTHER	0	0
SCHED_FIFO	1	99
SCHED_RR	1	99
SCHED_BATCH	0	0
SCHED_IDLE	0	0
SCHED_DEADLINE	0	0

`SCHED_FIFO` allows deterministic, high-priority execution of critical tasks without being preempted by lower-priority processes. The virtual machine can either be started with `chrt -f <PRIO>` or adjusted at a later point with `chrt -f <PRIO> <PID>`. It is also important that there are no other unexpected real-time processes running on the system concurrently.

3.2.3.4 Disable RT Throttling

If a real-time task consumes 100% of the CPU time, the system may become unresponsive as a whole. This happens because an RT process is constantly using the CPU and the Linux scheduler will not schedule other non-RT processes in the meantime. To prevent complete system lockups, the kernel has a function to throttle RT processes if they consume 0.95 seconds out of every 1 second of CPU time. It does this by pausing the process for the remaining 0.05 seconds, which is not desired because this could result in missed deadlines. RT throttling can be disabled by writing the value "-1" to the `/proc/sys/kernel/sched_rt_runtime_us` file. This change also needs to be made permanent because the `proc` filesystem resets to its default state after each reboot. For this purpose, the line `kernel.sched_rt_runtime_us = -1` can be appended to the end of the `/etc/sysctl.conf` file, which is read at boot time and used to configure kernel parameters. This reduces the potential for missed deadlines.

3.2.3.5 Disable Timer Migration

Timer migration allows timers to be moved from one CPU to another, which means the kernel can balance load across multiple CPUs. In a real-time system, this can introduce latency and jitter. To disable timer migration, the value "0" needs to be written to the `/proc/sys/kernel/timer_migration` file. This change also needs to be made permanent by writing the line `kernel.timer_migration = 0` to the `/etc/sysctl.conf` file. This reduces the amount of context switches and interrupts.

3.2.3.6 Set Device Driver Work Queue

The device driver work queue allows time-consuming tasks to be offloaded to be processed later in a separate kernel thread. By setting the work queue away from CPU 4 to another CPU, it is free to handle real-time tasks without being interrupted by these work queue tasks. This is done by specifying a bitmask to exclude CPU 4 in the files `/sys/devices/virtual/workqueue/cpumask` and `/sys/bus/workqueue/devices/writeback/cpumask`. In this case, that is 2F.

3.2.3.7 Disable RCU CPU Stall Warnings

As already mentioned in Section 3.2.2, the Linux kernel uses RCU as a synchronization mechanism for reading from and writing to shared data. An RCU CPU stall in the Linux kernel can occur due to several reasons. These include a CPU looping in an RCU read-side critical section, a CPU looping with interrupts disabled, a CPU looping with preemption disabled, or a CPU not getting around to less urgent tasks, known as "bottom halves". The number of seconds the kernel should wait before checking for stalled CPUs and reporting a stall warning can be set via the `/sys/module/rcupdate/parameters/rcu_cpu_stall_timeout` file. These warnings can be suppressed altogether to reduce the potential for increased latency by writing "1" to the `/sys/module/rcupdate/parameters/rcu_cpu_stall_suppress` file. This setting is also not persistent across reboots, so the command needs to be added to a startup script.

3.2.3.8 Stop Services

Table 12 presents services that can be further sources for random latency and unnecessary overhead. Stopping these services through `sudo systemctl stop <SERVICE>` prevents them from interrupting the CPU with their tasks.

Table 12: Services that are stopped to reduce random Latency and Overhead

Service	**Description**
irqbalance.service	Distributes hardware interrupts across CPUs
thermald.service	A daemon that prevents overheating
wpa_supplicant.service	A service for wireless network devices

3.2.3.9 Disable Machine Check

Machine checks report hardware errors and these checks can cause interruptions and increase latency. Hence, it is best to disable them in real-time scenarios by writing a “0” to the `/sys/devices/system/machinecheck/machinecheck0/check_interval` file.

3.2.3.10 Boot into Text-Based Environment

The Graphical User Interface (GUI) consumes great amounts of system resources. It often runs different background processes that are not essential for real-time systems and hence increase latency. These resources can be freed up by switching to a text-based environment. This way, processing power and memory can be allocated to critical real-time tasks which in turn can lead to lower latency and deterministic behavior. The command `systemctl set-default multi-user.target` and a following reboot allow for booting into a text-based environment. For the sake of completeness, the graphical interface can be brought back through the command `systemctl set-default graphical.target` and a following reboot.

After these comprehensive host configurations, the result of the `latency` test is depicted in Figure 17.

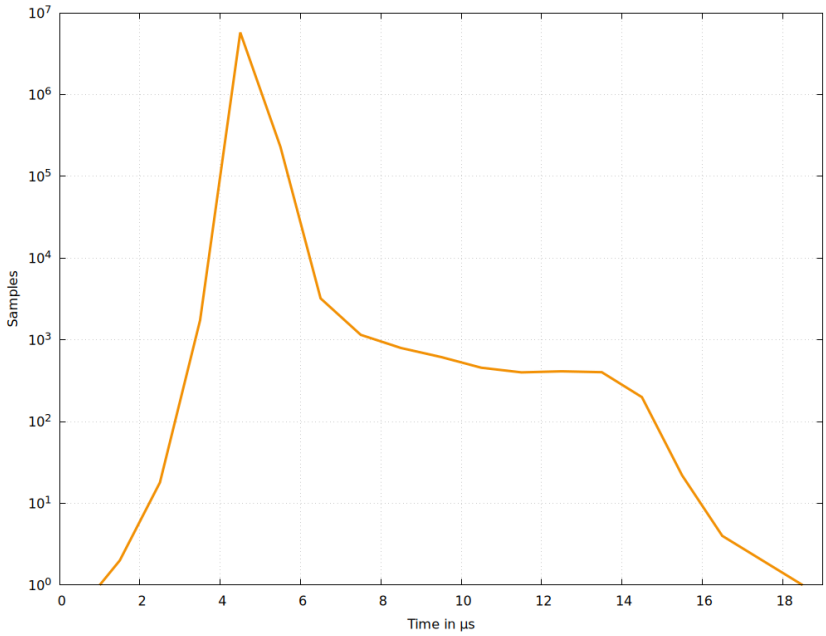

Figure 17: Latency Distribution of Salamander 4 Virtualization after tuning Host Configurations

The statistics obtained from this measurement are provided in Table 13 and Table 14.

Table 13: Latency Parameters for Host Configurations

Param	Samples	Average (µs)	Std Dev (µs)
min	599	2.998	0.242
avg	5,999,972	4.043	0.255
max	599	12.124	1.828

Table 14: Minimum, Average, and Maximum Latency with Overrun Counts after Host Configurations

Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Overruns
2.591	4.834	18.441	0

A worst latency value of 18.441 µs shows that the virtualized Salamander 4 OS got even more reliable with the host configurations.

3.2.4 QEMU/KVM Configurations

KVM allows the guest OS to run directly on the hardware, bypassing the need for traditional emulation which can introduce delays. QEMU, when used with KVM, provides hardware-assisted virtualization, which also lowers latency in the guest OS.

3.2.4.1 Tune LAPIC Timer Advance

The Local Advanced Programmable Interrupt Controller (LAPIC) is a built-in timer that handles the delivery of interrupts to the CPU. It generates interrupts at a rate. This rate can be tuned in the `/sys/module/kvm/parameters/lapic_timer_advance_ns` file to reduce the frequency of interrupts and therefore decrease the latency of the guest VM. The default value is "-1", which means that the kernel will automatically calculate an appropriate advance for the timer. Here, it is set to the value "7500". Hence, the timer interrupt will be delivered 7500 nanoseconds earlier than it is actually due. This gives the VM more time to handle it.

3.2.4.2 Set QEMU Options for Real-Time VM

QEMU provides options that can be used to improve the real-time performance of the guest VM. Table 15 briefly explains these options.

Table 15: QEMU Options for Real-Time Performance

QEMU Option	**Description**
`-object memory-backend-ram, id=ram0,size=4G,prealloc=on`	Locks the memory of the VM to 4GB and prevents it from being swapped out to disk
`-mem-prealloc -mem-path /dev/hugepages/`	Enables the use of hugepages and improves memory access

Code 9 shows the final QEMU script used to start the Salamander 4 virtualization, including these options.

```
#!/bin/sh

if  [ ! -d drive-c/ ]; then
      echo "Filling drive-c/"
      mkdir drive-c/
      tar -C drive-c/ -xf stek-drive-c-image-sigmatek-core2.tar.gz
fi

exec taskset -c 4 chrt -f 99 qemu-system-x86_64 -M pc,accel=kvm -kernel
    ./bzImage \
-m 2048 -drive
    file=salamander-image-sigmatek-core2.ext4,format=raw,media=disk \
-append "console=ttyS0 console=tty1 root=/dev/sda rw panic=1
    sigmatek_lrt.QEMU=1 ip=dhcp rootfstype=ext4 schedstats=enable nohlt
    idle=poll quiet xeno_hal.smi=1 xenomai.smi=1 threadirqs" \
-net nic,model=e1000,netdev=e1000 -netdev bridge,id=e1000,br=nm-bridge \
-fsdev local,security_model=none,id=fsdev0,path=drive-c -device
    virtio-9p-pci,id=fs0,fsdev=fsdev0,mount_tag=/mnt/drive-C \
-device vhost-vsock-pci,guest-cid=3,id=vsock0 \
-drive if=pflash,format=qcow2,file=ovmf.code.qcow2 \
-object memory-backend-ram,id=ram0,size=4G,prealloc=on \
-mem-prealloc -mem-path /dev/hugepages \
-no-reboot -nographic
```

Code 9: Tuned QEMU Script for starting Salamander 4 Virtualization

After QEMU configurations, the result of the `latency` test is depicted in Figure 18.

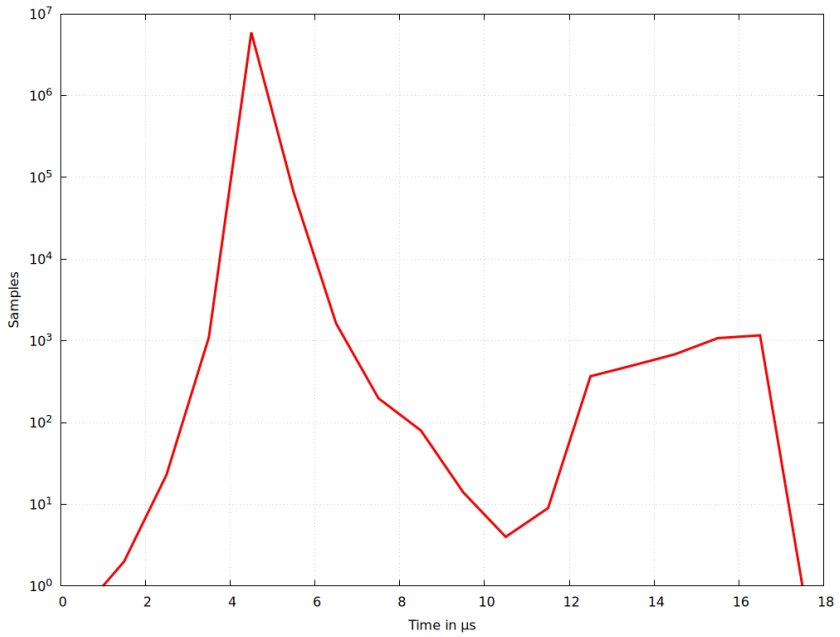

Figure 18: Latency Distribution of Salamander 4 Virtualization after tuning QEMU Configurations

The statistics obtained from this measurement are provided in Table 16 and Table 17.

Table 16: Latency Parameters after QEMU Configurations

Param	Samples	Average (µs)	Std Dev (µs)
min	599	3.125	0.432
avg	5,999,973	4.018	0.291
max	599	15.883	0.351

Table 17: Minimum, Average, and Maximum Latency with Overrun Counts after QEMU Configurations

Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Overruns
2.614	4.779	17.134	0

The maximum latency value decreased slightly. More notably, the standard deviation of the maximum latency samples, with the test lasting 600 seconds and thus having 600 worst latencies for each second, also decreased from 1.828 µs to 0.351 µs. Although the difference is not much, when the latency values are this small, it becomes noticeable. This means the system is more stable and predictable, thanks to the real-time tunings performed up to this point.

3.2.5 Guest OS Configurations

The guest OS, Salamander 4, is already based on Xenomai and uses the Cobalt real-time core with the Dovetail extension. Therefore, no additional modifications were necessary for the guest OS itself. The focus was primarily on optimizing the host and virtualization layer to achieve the desired real-time performance.

3.2.6 Other Configurations

There are other configurations that may be relevant depending on the case. Linux Kernel Developer Steven Rostedt explains all relevant aspects of a real-time system that must be considered in [54] and gives insights for finding sources of latency on the Linux system in [55]. A checklist for writing Linux real-time applications is provided by John Ogness in [56]. Every layer of the system stack must be deterministic to ensure predictable and reliable latency, including hardware, operating system, middleware and drivers, and the application software. [57] and [58] describe the process of writing hard real-time Linux programs using the real-time preemption patch in great detail. Various hardware and software tunings are listed in [23] and [59].

3.3 Real-Time Robotic Application

This chapter compares the latency of the Salamander 4 operating system before and after the real-time performance tunings with the latency of Salamander 4 running on bare metal. This comparison is done with the aid of a program that will be explained shortly. The goal is to understand how closely the performance of the virtualization can match that of the bare metal. The experimental setup includes a six-axis mini-robot, illustrated in Figure 19.

Figure 19: Mini-Robot of the Experiment [60]

The robot arm consists of six MG996R digital servo motors [61], equipped with a metal gearbox. The motor is able to rotate in a range of approximately 180 degrees and its position can be controlled with a high degree of accuracy. Each servo motor has three wires that need to be connected as shown in Figure 20.

Wire Color	Description
Brown	Ground wire connected to the ground of system
Red	Powers the motor typically +5V is used
Orange	PWM signal is given in through this wire to drive the motor

Figure 20: MG996R Servo Motor [62]

To drive the motor, it has to be powered using the red and brown wires and can be controlled by sending PWM signals to the orange wire. A 1,500 µs pulse every 20 milliseconds centers the servo. A 1,000 µs pulse turns it 90 degrees left, and a 2,000 µs pulse turns it 90 degrees right. This is depicted in Figure 21.

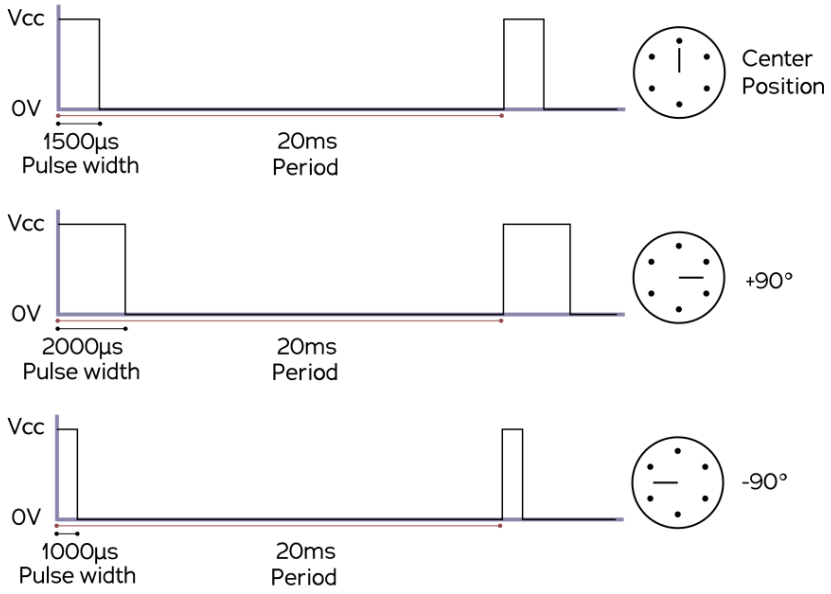

Figure 21: Controlling the MG996R Servo Motor [63]

In this experiment, this PWM signal is generated by the proprietary PW 022 pulse width module of Sigmatek [64]. Its connector layout is illustrated in Figure 22.

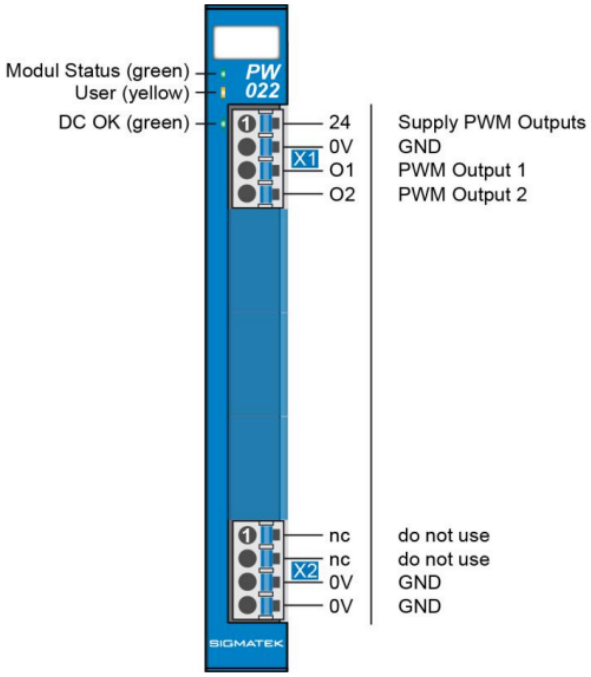

Figure 22: PW 022 Pulse Width Module Connector Layout [64]

The module has two +24 V switching PWM outputs with an adjustable frequency for controlling inductive loads. Since the mentioned servo motors operate between 4.8 volts and 7.2 volts [61], this voltage needs to be reduced through resistors before supplying it to a servo motor. For this purpose, two resistors with resistances of 2500 ohms and 1000 ohms are connected in series. The voltage across each resistor is calculated using Ohm's law, which is given by equation 1 below.

$$V = I \cdot R \tag{1}$$

In this equation, V represents the voltage across the resistor, I is the current flowing through the resistor, and R is the resistance of the resistor. The resulting voltages across the resistors are as follows:

- The voltage across the 2.5 kiloohm resistor is approximately 17.14 volts.
- The voltage across the 1 kiloohm resistor is approximately 6.86 volts. This voltage is then supplied to the control wire of servo motor 1 of the mini-robot.

Connecting a second servo motor of the mini-robot means repeating the process of reducing the voltage through resistors for the second PWM output of the PW 022 module. The connection between the PW 022 module and the motor of the mini-robot is demonstrated in Figure 23.

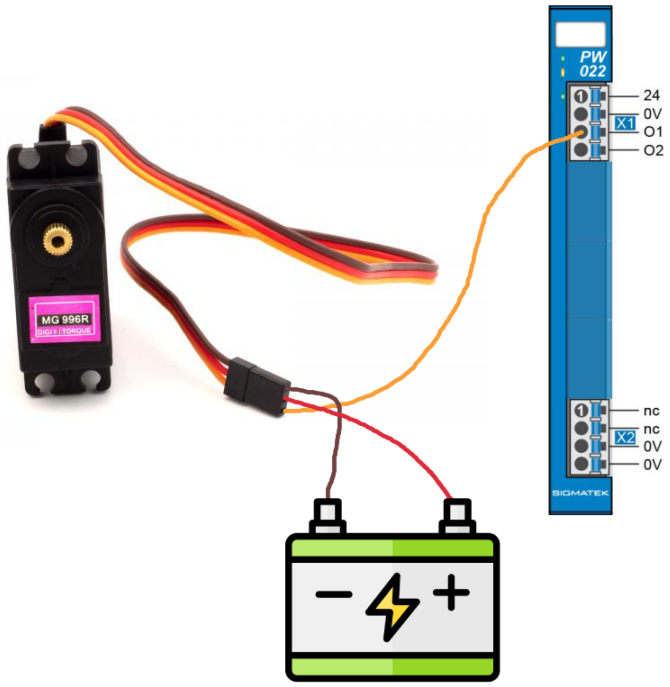

Figure 23: Connection between PW 022 and Mini-Robot [65]

The program is written in Lasal Class 2 and is applied to all three mentioned versions of Salamander 4 to measure the reaction time of the robot to the commands. Prior to examining the software program, the next two subsections briefly explain the setup of each version of the experiment.

3.3.1 Setup of Salamander 4 Bare Metal

In this version, Salamander 4 runs on the CP 841 CPU unit, specifically designed for the Salamander 4 operating system. The PW 022 modules are directly mounted on the CPU via the S-DIAS bus [66]. There are three PW 022 modules because each can control two motors simultaneously, and the mini-robot has six motors. The setup is visible on Figure 24.

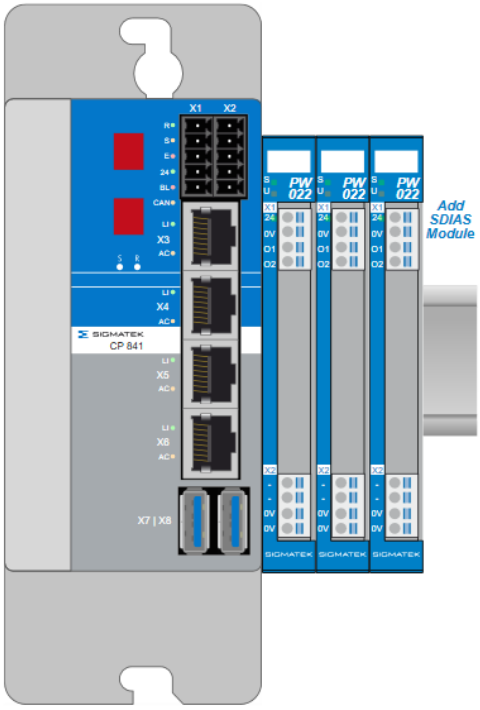

Figure 24: Setup of Salamander 4 Bare Metal

3.3.2 Setup of Salamander 4 Virtualization

In the virtualized setup of the experiment, the vCPU functionality of QEMU is used to connect QEMU with the PWM modules. In order to achieve that, the PCV 522 VARAN Manager PCI Insert Card [67], which serves as a bridge between the PC and the rest of the setup, needs to be plugged into the PC. The command `lspci -nn` lists all PCI devices along with their vendor and device IDs. In this case, the command for binding the PCV 522 module to the VFIO-PCI driver is `sudo sh -c 'echo "5112 2200" > /sys/bus/pci/drivers/vfio-pci/new_id'` with vendor ID `5112` and device ID `2200`. To verify that the device has been successfully bound to the VFIO-PCI driver, `lspci -v` can be used. On top of the binding process, the QEMU script also needs to be modified to include the PCV 522 VARAN Manager PCI Insert Card in the virtualization. This is done by adding `-device vfio-pci,host=03:00.0` to the QEMU script, where `03:00.0` refers to the PCI address of the device, with `03` being the bus number, `00` the device number, and `0` the function number. The final script can be seen in Code 10.

```
#!/bin/sh

if  [ ! -d drive-c/ ]; then
      echo "Filling drive-c/"
      mkdir drive-c/
      tar -C drive-c/ -xf stek-drive-c-image-sigmatek-core2.tar.gz
fi

exec taskset -c 4 chrt -f 99 qemu-system-x86_64 -M pc,accel=kvm -kernel
    ./bzImage \
-m 2048 -drive
    file=salamander-image-sigmatek-core2.ext4,format=raw,media=disk \
-append "console=ttyS0 console=tty1 root=/dev/sda rw panic=1
    sigmatek_lrt.QEMU=1 ip=dhcp rootfstype=ext4 schedstats=enable nohlt
    idle=poll quiet xeno_hal.smi=1 xenomai.smi=1 threadirqs" \
-net nic,model=e1000,netdev=e1000 -netdev bridge,id=e1000,br=nm-bridge \
-fsdev local,security_model=none,id=fsdev0,path=drive-c -device
    virtio-9p-pci,id=fs0,fsdev=fsdev0,mount_tag=/mnt/drive-C \
-device vhost-vsock-pci,guest-cid=3,id=vsock0 \
-drive if=pflash,format=qcow2,file=ovmf.code.qcow2 \
-object memory-backend-ram,id=ram0,size=4G,prealloc=on \
-mem-prealloc -mem-path /dev/hugepages \
-device vfio-pci,host=03:00.0 \
-no-reboot -nographic
```

Code 10: Final QEMU Script for Salamander 4 Virtualization with PCI Configuration

An additional Varan Connection module VI 021 [68] was required to enable the connection between the PCV 522 module and the PW 022 module that generates the signal to move the mini-robot. This setup is depicted in Figure 25.

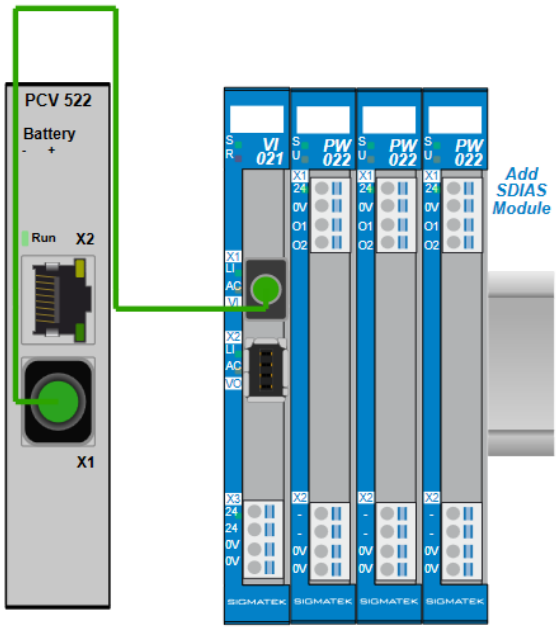

Figure 25: Setup of Salamander 4 Virtualization

Essentially, the CPU is being emulated by the vCPU functionality of QEMU, allowing the PW 022 modules to interact with the vCPU through the PCV 522 Insert Card and the VI 021 module as if they were communicating directly with a physical CPU unit.

3.3.3 Robotic Application

Robots operate in real-time, meaning commands must be executed within a set time frame. As explained earlier, there are hard real-time systems, which must always meet their deadlines, and soft real-time systems, which can occasionally miss them. If a robot misses these deadlines, it can cause unwanted movements and jumpy behavior. The most crucial part of latency distribution for determinism are the outliers. Even if a system's latency is as expected 99% of the time, the remaining 1% can be worse than all the other measurements combined. Looking only at the mean and standard deviation misses these systemic issues.

The impacts of real-time performance tunings on the determinism and latency of Salamander 4 virtualization were already shown in the previous subsection. Here, these improvements are demonstrated using a robotic application. This application is written in Lasal Class 2, as mentioned in Section 1.2. The goal is to determine the time from command issuance to the signal reaching the PWM, excluding the 20ms signal period required by the servomotor to process the signal and initiate movement. The program sequence is shown in Figure 26.

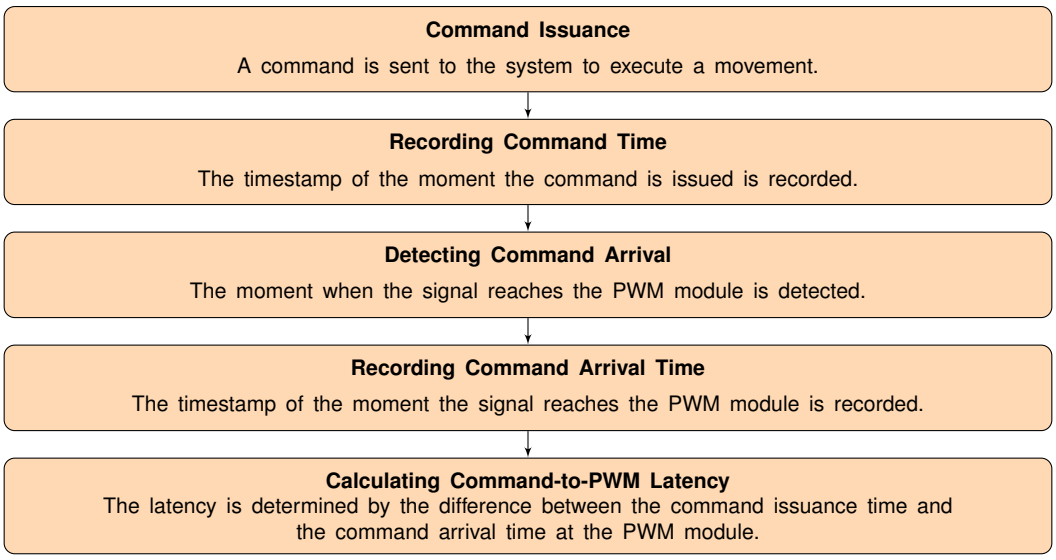

Figure 26: Flowchart of Robotic Application

This latency measurement above is performed 1,000 times to get reliable results. These results are presented in the next subsubsections.

3.3.3.1 Latency in Salamander 4 Bare Metal

The latency values for bare metal Salamander 4 OS acquired from the program above are demonstrated in Figure 27 and detailed in Table 18.

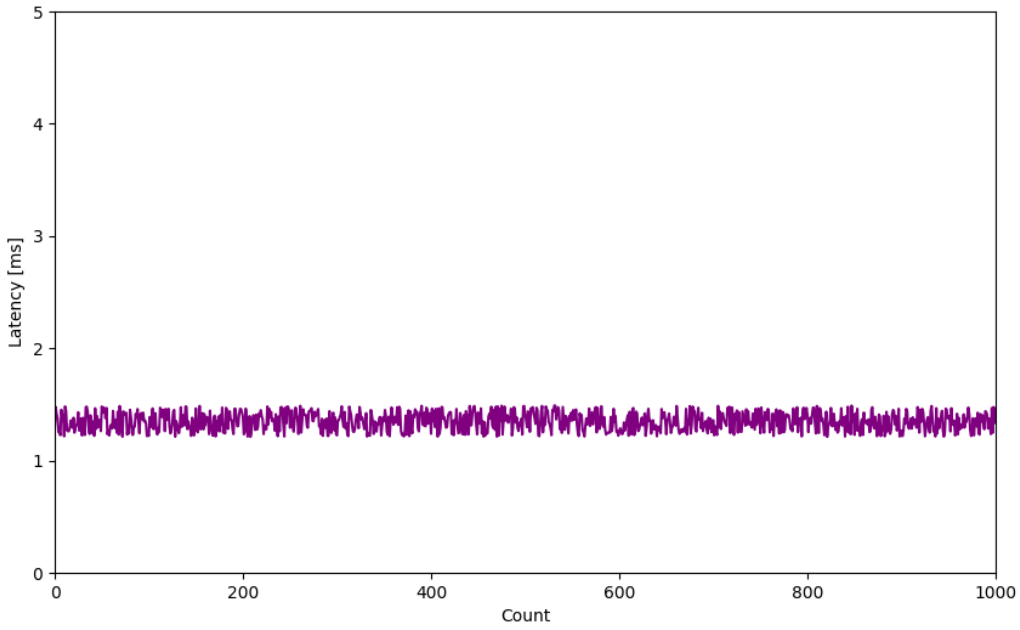

Figure 27: Robotic Application Latency Values of Salamander 4 Bare Metal

Table 18: Robotic Application Latency Statistics Bare Metal

Samples	Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Std Dev (µs)
1000	1.211	1.347	1.49	0.082

3.3.3.2 Latency in Untuned Salamander 4 OS Virtualization

The latency values acquired for the untuned virtualization of Salamander 4 OS are demonstrated in Figure 28 and detailed in Table 19.

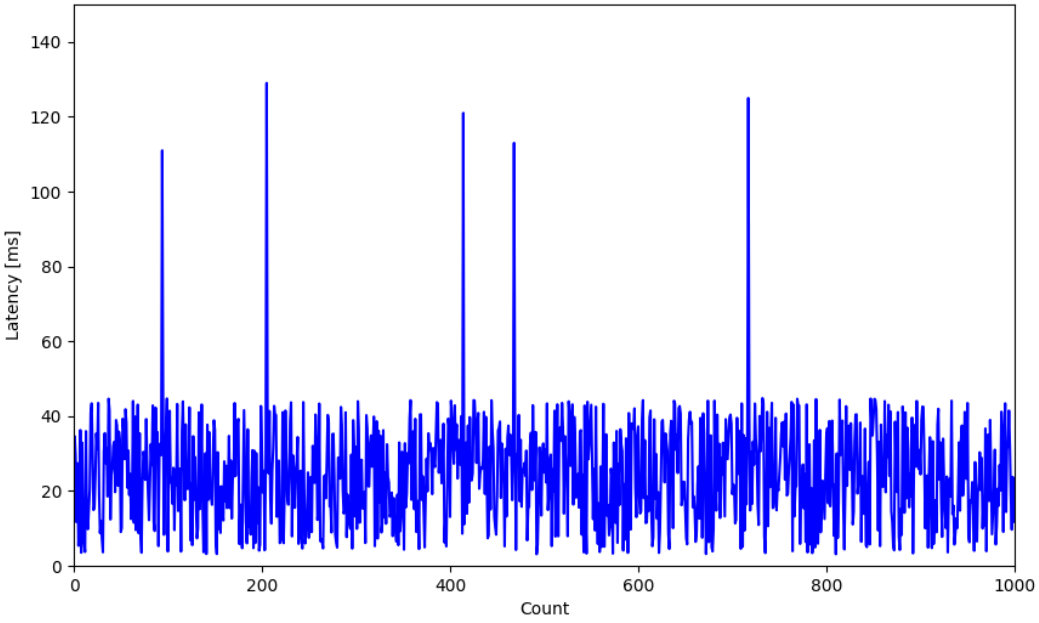

Figure 28: Robotic Application Latency Values of Salamander 4 Untuned Virtualization

Table 19: Robotic Application Latency Statistics Untuned Virtualization

Samples	Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Std Dev (µs)
1000	3.1	24.603	129.46	13.876

3.3.3.3 Latency in Tuned Salamander 4 OS Virtualization

The latency values acquired for the tuned virtualization of Salamander 4 OS are demonstrated in Figure 29 and detailed in Table 20.

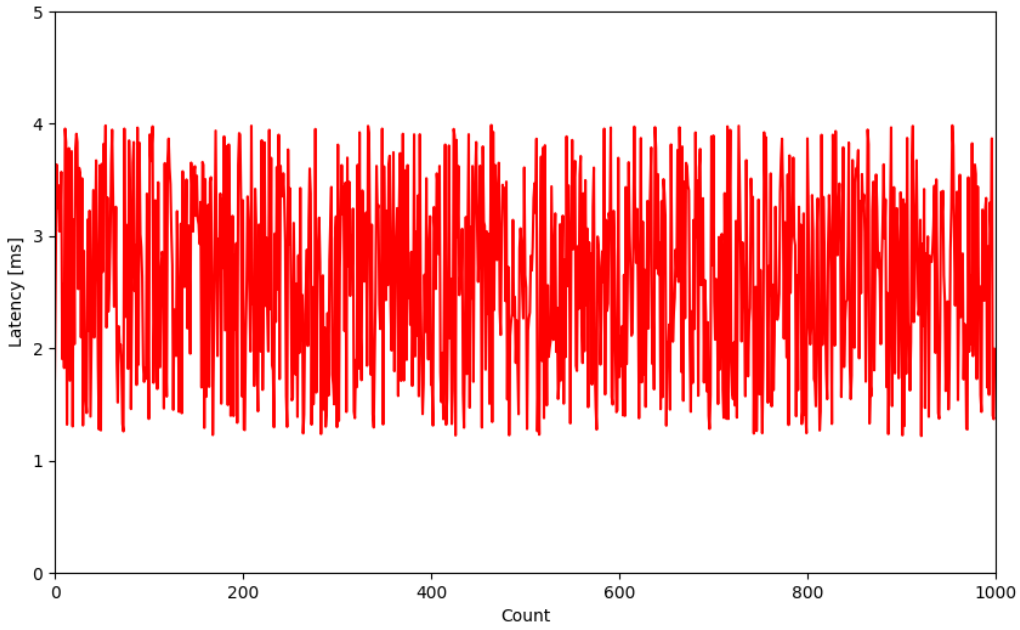

Figure 29: Robotic Application Latency Values of Salamander 4 Tuned Virtualization

Table 20: Robotic Application Latency Statistics Tuned Virtualization

Samples	Lat Min (µs)	Lat Avg (µs)	Lat Max (µs)	Std Dev (µs)
1000	1.219	2.62	3.988	0.182

4 Results

The goal was to virtualize the Salamander 4 OS and bring its real-time performance closer to that of the bare metal version and guarantee deterministic and reliable behavior. This chapter summarizes the results after the real-time performance tunings, both in terms of the `latency` program and the robotic application. Figure 30 displays the change in latency after each real-time performance tuning, detailed in Section 3.2.

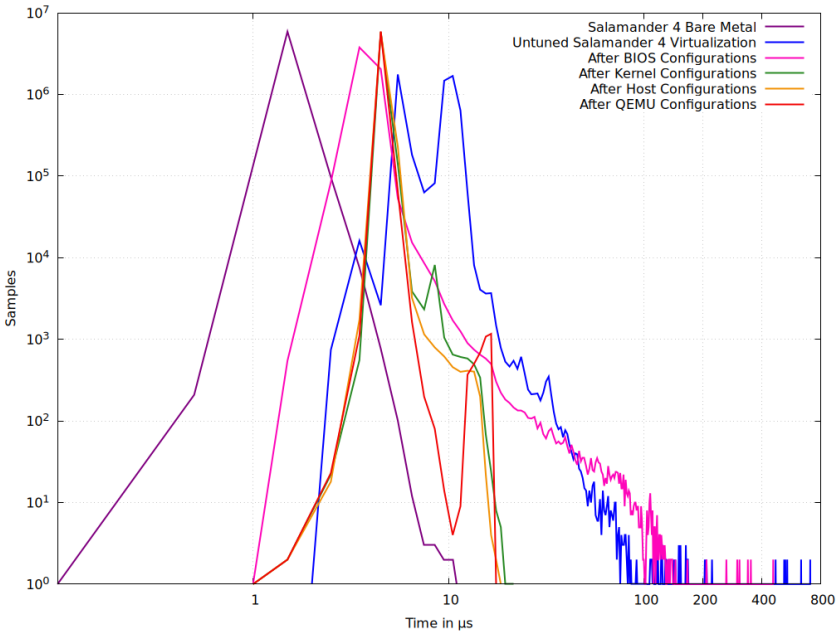

Figure 30: Comparison of Latency Distribution of Salamander 4 after Configurations

As mentioned earlier in Section 2.9, these modifications were added sequentially and tested together. Previous tunings were not reverted when moving to the next tunings. Table 21 gives an overview of the most important metrics of the measurements after each tuning.

Table 21: Comparison of Latency Statistics in Salamander 4 after Configurations

Salamander 4 Version	**Lat Min (µs)**	**Lat Avg (µs)**	**Lat Max (µs)**	**Overruns**
Salamander 4 Bare Metal	0.613	1.380	10.709	0
Untuned Salamander 4 Virtualization	2.536	8.940	707.622	43
After BIOS Configurations	0.969	3.948	457.545	22
After Kernel Configurations	2.545	4.811	21.694	0
After Host Configurations	2.591	4.834	18.441	0
After QEMU Configurations	2.614	4.779	17.134	0

The improved latency was also tested with the robotic application in Subsection 3.3. The difference between the command time and the time the signal reaches the PWM module was measured 1,000 times to get reliable results for the bare metal, untuned, and tuned versions of Salamander 4. The results are visualized in Figure 31 and detailed in Table 22.

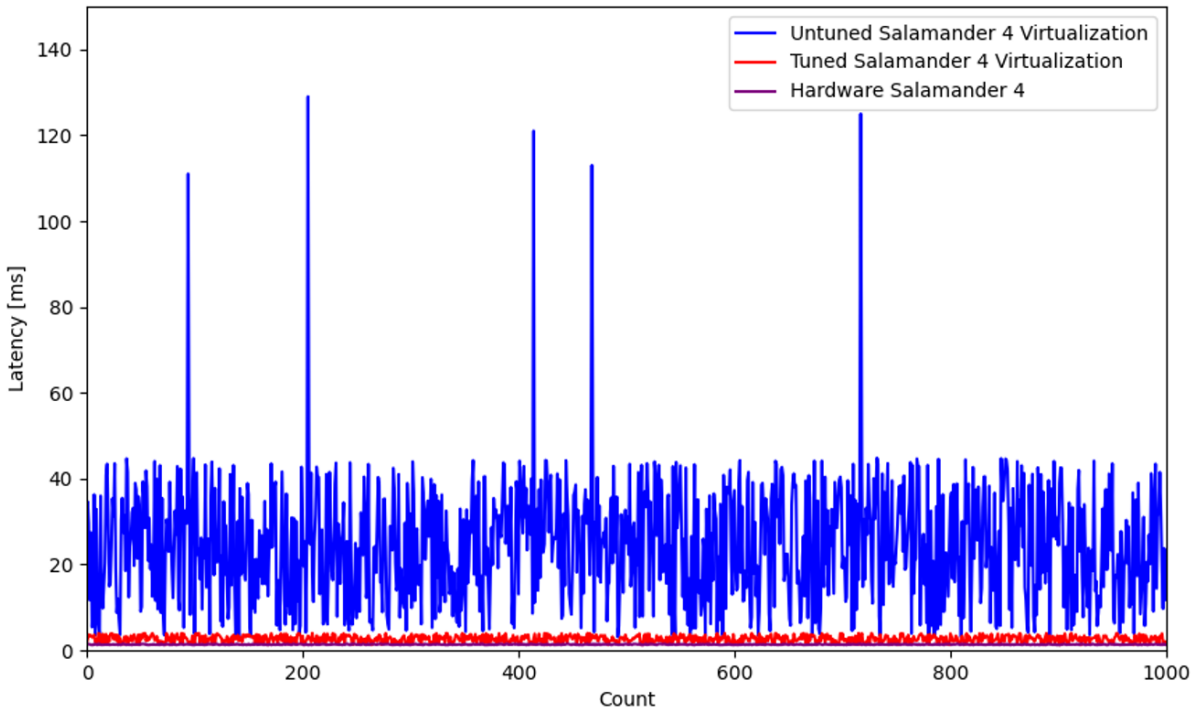

Figure 31: Comparison of Robotic Application Latency after Configurations

Table 22: Comparison of Robotic Application Latency Statistics

Version	Lat Min (ms)	Lat Avg (ms)	Lat Max (ms)	Std Dev (ms)
Salamander 4 Bare Metal	1.211	1.347	1.49	0.082
Untuned Salamander 4 Virtualization	3.1	24.603	129.46	13.876
Tuned Salamander 4 Virtualization	1.219	2.62	3.988	0.812

5 Discussion

This master's thesis shows that the PREEMPT-RT patch along with the applied real-time performance tunings have a positive impact on reducing latency and improving determinism. The maximum latency in the final tuned version of Salamander 4 virtualization is significantly lower than in the untuned setup. The worst latency value in a 10-minute measurement span decreased from 707.622 µs to 17.134 µs after tuning BIOS, kernel, host, QEMU/KVM, and guest configurations. Especially the BIOS and kernel configurations played a crucial role in reducing the maximum latency, with a major fall to 21.694 µs. From this moment on, there were also no overruns in the test any longer. The host configurations, including CPU and interrupt affinity, and real-time prioritization of QEMU, further reduced latency down to 18.441 µs. The guest OS, Salamander 4, already had real-time capabilities through Xenomai. The focus was primarily on optimizing the host and virtualization layer to achieve the desired real-time performance. Finally, QEMU/KVM configurations, such as tuning the LAPIC timer advance and using hugepages, brought the latency down to the final worst latency value of 17.134 µs in 10 minutes. This value is very close to the bare metal value of 10.709 µs. The goal was set to achieving latency values below 50 µs in the duration of the measurement. This goal is achieved.

As a next step, the robotic application in a practical scenario also demonstrates and validates the improvement of the tuned virtualization. The difference between the command time and the time the signal reaches the PWM module was measured 1,000 times for the untuned, tuned, and bare metal versions of Salamander 4. The most important metric here is also the worst latency of the signal reaching the PWM. It dropped from 129 ms to 3.988 ms in the developed application. Comparing this to the worst latency value of the bare metal version, which is 1.49 ms, it is apparent that the tuned virtualization through real-time performance configurations came very close to the determinism and reliability of the hardware.

It is important to underline one more time the fact that these modifications were added sequentially and tested together. After applying the PREEMPT-RT patch, the BIOS was configured, followed by the kernel. The BIOS settings were not reverted when moving to the kernel configurations. Next, the host was configured, but the BIOS and kernel settings remained unchanged. The guest configurations were not changed since Xenomai already provides real-time capabilities. Finally, QEMU/KVM settings were applied. This means that the configurations are not isolated but are applied and then tested as a whole.

6 Summary and Outlook

Hardware-based systems are limited in adaptability and can be costly, especially when scaling operations. Physical access for updates and maintenance is challenging, leading to downtime and lost productivity. While virtualization addresses these issues, it introduces increased overhead and latency.

In this master's thesis, the goal was the virtualization of a real-time operating system to control a robot, with a focus on compliance with real-time determinism. Specifically, the aim was to bring the latency of the virtualized Salamander 4 closer to that of the bare metal version, thereby providing a comprehensive blueprint for making a virtualized guest system in a host system real-time capable with deterministic behavior. Salamander 4 is built with Yocto, employs hard real-time with Xenomai 3, and is virtualized through QEMU/KVM. The testing robot was connected to the OS via a VARAN bus interface.

The initial latency values were first measured with the `latency` program of the Xenomai tool suite. The tests were conducted for 10 minutes with a sampling period of 100 µs, using a periodic user-mode task, and were assigned a priority of 99. If any sample's latency value was greater than 100 µs, this was considered an overrun. There was a significant initial gap in latency statistics between the virtualized Salamander 4 and the Salamander 4 on bare metal. To address this gap, an extensive tuning process was carried out to achieve real-time performance and determinism. These modifications were added sequentially and tested together. After applying the PREEMPT-RT patch, the BIOS was configured, followed by the kernel. The BIOS settings were not reverted when moving to the kernel configurations. After these two steps, the worst latency decreased from an initial value of 707.62 µs to 21.694 µs. From this moment on, there were also no overruns in the tests any longer. The host configurations, including CPU and interrupt affinity, and real-time prioritization of QEMU, further reduced latency down to 18.441 µs. The guest OS, Salamander 4, already had real-time capabilities through Xenomai. The focus was primarily on optimizing the host and virtualization layer to achieve the desired real-time performance. Finally, QEMU/KVM configurations, such as tuning the LAPIC timer advance and using hugepages, brought the latency down a little more to the final worst latency value of 17.134 µs in 10 minutes. This value is very close to the bare metal value of 10.709 µs. The goal was set to achieving latency values below 50 µs in the duration of the measurement. This goal was achieved.

On top of that, a robotic application further confirmed the improvements. The difference between the command time and the time the signal reaches the PWM module was measured

1,000 times for the untuned, tuned, and bare metal versions of Salamander 4. The worst latency of the signal reaching the PWM dropped from an initial value of 129 ms to 3.988 ms. Comparing this to the worst latency value of the bare metal version, which was 1.49 ms, it is apparent that the tuned virtualization through real-time performance configurations came very close to the determinism and reliability of the hardware.

While this thesis provides a comprehensive blueprint for making a virtualized guest system in a host system real-time capable with deterministic behavior, future work can be done to extend this knowledge. Additional configurations and optimizations of the virtualization layer can be done and tested to further reduce latency and improve determinism. Other than that, the use of other hypervisors and virtualization technologies can be investigated to compare their performance for real-time applications. Moreover, extensive testing of the virtualized system under various workloads can be conducted to evaluate its performance and reliability in different conditions and stressed situations, as demonstrated by Huang, Lin, and Wu [69], Kirova et al. [70], or Adam, Petrellis, and Doulos [71].

Bibliography

[1] Seçkin Canbaz and Gökhan Erdemir. “Performance Analysis of Real-Time and General-Purpose Operating Systems for Path Planning of the Multi-Robot Systems”. In: *International Journal of Electrical and Computer Engineering (IJECE)* 12.1 (Feb. 2022), p. 285. ISSN: 2722-2578, 2088-8708. DOI: 10.11591/ijece.v12i1.pp285-292.

[2] *What Is Real-Time Linux? Part II*. URL: https://ubuntu.com/blog/what-is-real-time-linux-ii (visited on 08/12/2024).

[3] HayfaaSubhi Malallah et al. “A Comprehensive Study of Kernel (Issues and Concepts) in Different Operating Systems”. In: *Asian Journal of Research in Computer Science* (May 2021), pp. 16–31. ISSN: 2581-8260. DOI: 10.9734/ajrcos/2021/v8i330201.

[4] Jiacun Wang. *Real-Time Embedded Systems*. Hoboken, NJ, USA: Wiley, 2017. ISBN: 978-1-119-42070-5.

[5] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Fourth edition. Cham, Switzerland: Springer, 2024. ISBN: 978-3-031-45409-7. DOI: 10.1007/978-3-031-45410-3.

[6] Giuseppe Lipari and Luigi Palopoli. *Real-Time Scheduling: From Hard to Soft Real-Time Systems*. 2015. DOI: 10.48550/ARXIV.1512.01978. (Visited on 08/28/2024).

[7] Singh Ugal Amarpreet. *Hard Real Time Linux* Using Xenomai* on Intel® Multi-Core Processors*. Tech. rep. Intel, Oct. 2009.

[8] Rui Queiroz, Tiago Cruz, and Paulo Simões. “Testing the Limits of General-Purpose Hypervisors for Real-Time Control Systems”. In: *Microprocessors and Microsystems* 99 (June 2023), p. 104848. ISSN: 01419331. DOI: 10.1016/j.micpro.2023.104848.

[9] pixelart. *SIGMATEK - Komplette Automatisierungssysteme*. URL: https://www.sigmatek-automation.com/de/ (visited on 03/27/2024).

[10] Luc Perneel and Auricle Technologies Pvt. Ltd. “Real-Time Capabilities in the Standard Linux Kernel: How to Enable and Use Them?” In: *International Journal on Recent and Innovation Trends in Computing and Communication* 3.1 (2015), pp. 131–135. ISSN: 2321-8169. DOI: 10.17762/ijritcc2321-8169.150127.

[11] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. “The Real-Time Linux Kernel: A Survey on PREEMPT_RT”. In: *ACM Computing Surveys* 52.1 (Jan. 2020), pp. 1–36. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3297714.

[12] George K. Adam. “Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers”. In: *Computers* 10.5 (May 2021), p. 64. ISSN: 2073-431X. DOI: 10.3390/computers10050064.

[13] Marcello Cinque et al. “Virtualizing Mixed-Criticality Systems: A Survey on Industrial Trends and Issues”. In: *Future Generation Computer Systems* 129 (Apr. 2022), pp. 315–330. ISSN: 0167739X. DOI: 10.1016/j.future.2021.12.002. arXiv: 2112.06875 [cs].

[14] Kristian Sandstrom et al. “Virtualization Technologies in Embedded Real-Time Systems”. In: *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*. Cagliari, Italy: IEEE, Sept. 2013, pp. 1–8. ISBN: 978-1-4799-0864-6 978-1-4799-0862-2. DOI: 10.1109/ETFA.2013.6648012.

[15] Gilberto Taccari et al. “Embedded Real-Time Virtualization: State of the Art and Research Challenges”. In: Oct. 2014.

[16] Diogenes Javier Perez et al. “How Real (Time) Are Virtual PLCs?” In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. Stuttgart, Germany: IEEE, Sept. 2022, pp. 1–8. ISBN: 978-1-66549-996-5. DOI: 10.1109/ETFA52439.2022.9921545.

[17] Zonghua Gu and Qingling Zhao. “A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization”. In: *Journal of Software Engineering and Applications* 05.04 (2012), pp. 277–290. ISSN: 1945-3116, 1945-3124. DOI: 10.4236/jsea.2012.54033.

[18] Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. “Challenges in Real-Time Virtualization and Predictable Cloud Computing”. In: *Journal of Systems Architecture* 60.9 (Oct. 2014), pp. 726–740. ISSN: 13837621. DOI: 10.1016/j.sysarc.2014.07.004.

[19] Claudio Scordino et al. “Real-Time Virtualization For Industrial Automation”. In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vienna, Austria: IEEE, Sept. 2020, pp. 353–360. ISBN: 978-1-72818-956-7. DOI: 10.1109/ETFA46521.2020.9211890.

[20] Ruhui Ma et al. “Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System”. In: *Journal of Information Science and Engineering* (Sept. 2013). DOI: 10.6688/JISE.2013.29.5.13.

[21] Jun Zhang et al. “Performance Analysis towards a KVM-Based Embedded Real-Time Virtualization Architecture”. In: *5th International Conference on Computer Sciences and Convergence Information Technology*. Seoul, Korea (South): IEEE, Nov. 2010, pp. 421–426. ISBN: 978-1-4244-8567-3. DOI: 10.1109/ICCIT.2010.5711095.

[22] S. Brosky and S. Rotolo. “Shielded Processors: Guaranteeing Sub-Millisecond Response in Standard Linux”. In: *Proceedings International Parallel and Distributed Processing Symposium*. Nice, France: IEEE Comput. Soc, 2003, p. 9. ISBN: 978-0-7695-1926-5. DOI: 10.1109/IPDPS.2003.1213237.

[23] *Real-Time Performance Tuning Best Practice Guidelines for KVM-Based Virtual Machines*. Tech. rep. Intel, Jan. 2022.

[24] Hobin Yoon, Jungmoo Song, and Jamee Lee. “Real-Time Performance Analysis in Linux-Based Robotic Systems”. In: (Jan. 2009).

[25] Jan Kiszka. “Towards Linux as a Real-Time Hypervisor”. In: (Jan. 2009).

[26] Paul E McKenney. “‘Real Time’ vs. ‘Real Fast’: How to Choose?” In: (2008).

[27] Daniel Bristot De Oliveira et al. “Demystifying the Real-Time Linux Scheduling Latency”. In: *LIPIcs, Volume 165, ECRTS 2020* 165 (2020), 9:1–9:23. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ECRTS.2020.9.

[28] Dirk Gabriel, Walter Stechele, and Stefan Wildermann. “Resource-Aware Parameter Tuning for Real-Time Applications”. In: *Architecture of Computing Systems – ARCS 2019*. Ed. by Martin Schoeberl et al. Vol. 11479. Cham: Springer International Publishing, 2019, pp. 45–55. ISBN: 978-3-030-18655-5 978-3-030-18656-2. DOI: 10.1007/978-3-030-18656-2_4.

[29] Aditya Bhardwaj and C. Rama Krishna. “Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey”. In: *Arabian Journal for Science and Engineering* 46.9 (Sept. 2021), pp. 8585–8601. ISSN: 2193-567X, 2191-4281. DOI: 10.1007/s13369-021-05553-3.

[30] Maryam Abbasi et al. “Exploring OpenStack for Scalable and Cost-Effective Virtualization in Education”. In: *New Trends in Disruptive Technologies, Tech Ethics and Artificial Intelligence*. Ed. by Daniel H. De La Iglesia, Juan F. De Paz Santana, and Alfonso J. López Rivero. Vol. 1452. Cham: Springer Nature Switzerland, 2023, pp. 135–146. ISBN: 978-3-031-38343-4 978-3-031-38344-1. DOI: 10.1007/978-3-031-38344-1_13.

[31] Hassana Mahfoud et al. “Real-Time Predictive Maintenance-Based Process Parameters: Towards an Industrial Sustainability Improvement”. In: *International Conference on Advanced Intelligent Systems for Sustainable Development (AI2SD’2023)*. Ed. by Mostafa Ezziyyani, Janusz Kacprzyk, and Valentina Emilia Balas. Vol. 931. Cham: Springer Nature Switzerland, 2024, pp. 18–34. ISBN: 978-3-031-54287-9 978-3-031-54288-6. DOI: 10.1007/978-3-031-54288-6_3.

[32] Rui Queiroz et al. “Container-Based Virtualization for Real-time Industrial Systems—A Systematic Review”. In: *ACM Computing Surveys* 56.3 (Mar. 2024), pp. 1–38. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3617591.

[33] Marcello Cinque et al. “Evaluating Virtualization for Fog Monitoring of Real-Time Applications in Mixed-Criticality Systems”. In: *Real-Time Systems* 59.4 (Dec. 2023), pp. 534–567. ISSN: 0922-6443, 1573-1383. DOI: 10.1007/s11241-023-09410-4.

[34] Daniel Casini et al. “Latency Analysis of I/O Virtualization Techniques in Hypervisor-Based Real-Time Systems”. In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Nashville, TN, USA: IEEE, May 2021, pp. 306–319. ISBN: 978-1-66540-386-3. DOI: 10.1109/RTAS52030.2021.00032.

[35] Binbin Zhang et al. “Evaluating and Optimizing I/O Virtualization in Kernel-based Virtual Machine (KVM)”. In: *Network and Parallel Computing*. Ed. by Chen Ding, Zhiyuan Shao, and Ran Zheng. Vol. 6289. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 220–231. ISBN: 978-3-642-15671-7 978-3-642-15672-4. DOI: 10.1007/978-3-642-15672-4_20.

[36] *CPU Units - SIGMATEK*. URL: https://www.sigmatek-automation.com/en/products/control-systems/cpu-units/cp-841/ (visited on 07/30/2024).

[37] *Engineering Tool LASAL - SIGMATEK*. URL: https://www.sigmatek-automation.com/de/produkte/engineering-tool-lasal/lasal-class/ (visited on 05/27/2024).

[38] *LASAL OS - SIGMATEK*. (Visited on 07/30/2024).

[39] *QEMU*. URL: https://www.qemu.org/ (visited on 03/27/2024).

[40] *Welcome to the Yocto Project Documentation — The Yocto Project ® 4.3.999 Documentation*. URL: https://docs.yoctoproject.org/ (visited on 03/27/2024).

[41] *Xenomai :: Xenomai*. URL: https://xenomai.org/ (visited on 03/21/2024).

[42] *Overview :: Xenomai 3*. URL: https://v3.xenomai.org/overview/ (visited on 08/28/2024).

[43] *Realtime:Preempt_rt_versions [Wiki]*. URL: https://wiki.linuxfoundation.org/realtime/preempt_rt_versions (visited on 08/05/2024).

[44] *LATENCY(1)*. URL: https://doc.xenomai.org/v3/html/man1/latency/index.html (visited on 08/19/2024).

[45] *Trace-Cmd*. URL: https://trace-cmd.org/ (visited on 03/25/2024).

[46] *Ftrace - Function Tracer — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/trace/ftrace.html (visited on 09/01/2024).

[47] Steven Rostedt. *Trace-Cmd Host Guest Tracing Howto*. URL: https://rostedt.org/host-guest-tutorial/ (visited on 08/20/2024).

[48] *KernelShark*. URL: https://kernelshark.org/ (visited on 03/25/2024).

[49] *Realtime Kernel Patchset - ArchWiki*. URL: https://wiki.archlinux.org/title/Realtime_kernel_patchset (visited on 08/05/2024).

[50] Petro Lutsyk, Jonas Oberhauser, and Wolfgang J. Paul. *A Pipelined Multi-Core Machine with Operating System Support: Hardware Implementation and Correctness Proof*. Lecture Notes in Computer Science Theoretical Computer Science and General Issues 9999. Cham: Springer, 2020. ISBN: 978-3-030-43242-3.

[51] Steven Rostedt and Darren V Hart. “Internals of the RT Patch”. In: (Jan. 2007).

[52] *What Is Real-Time Linux? Part I*. URL: https://ubuntu.com/blog/what-is-real-time-linux-i (visited on 08/05/2024).

[53] *Linux Process Priorities Demystified*. URL: https://sigma-star.at/blog/2022/02/linux-proc-prios/ (visited on 06/11/2024).

[54] Kernel Recipes. *Kernel Recipes 2016 - Understanding a Real-Time System (More than Just a Kernel) - Steven Rostedt*. Oct. 2016. URL: https://www.youtube.com/watch?v=w3yT8zJe0Uw (visited on 06/11/2024).

[55] The Linux Foundation. *Finding Sources of Latency on Your Linux System - Steven Rostedt, VMware*. Sept. 2020. URL: https://www.youtube.com/watch?v=Tkra8g0gXAU (visited on 06/11/2024).

[56] The Linux Foundation. *A Checklist for Writing Linux Real-Time Applications - John Ogness, Linutronix GmbH*. Nov. 2020. URL: https://www.youtube.com/watch?v=NrjXEaTSyrw (visited on 06/11/2024).

[57] *HOWTO: Build an RT-application - RTwiki*. URL: https://archive.kernel.org/oldwiki/rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application.html (visited on 06/11/2024).

[58] *Real-Time Programming with Linux, Part 2: Configuring Linux for Real-Time - Shuhao's Blog*. URL: https://shuhaowu.com/blog/2022/02-linux-rt-appdev-part2.html#f4 (visited on 06/11/2024).

[59] *KVM/Qemu Virtualization Tuning Guide on Intel® Xeon® Based Systems*. URL: https://www.intel.com/content/www/us/en/developer/articles/guide/kvm-tuning-guide-on-xeon-based-systems.html (visited on 06/11/2024).

[60] *6DF Robotic Arm*. URL: https://yfrobot.com/products/yfrobot-4wd-chassis-steered-by-a-servo-motor-%e7%9a%84%e5%89%af%e6%9c%ac (visited on 08/23/2024).

[61] *MG996R Servo Motor*. URL: https://components101.com/motors/mg996r-servo-motor-datasheet (visited on 07/30/2024).

[62] *6 Pack Mg996r Metal Gear Digital Servo Motor - Compatible With Ardu...* URL: https://www.fruugo.co.il/6-pack-mg996r-metal-gear-digital-servo-motor-compatible-with-arduino-rc-models-horn-arm/p-163197201-346945567?language=en (visited on 08/23/2024).

[63] twiereng. *Emily, a Quadruped Dog Robot*. URL: https://www.instructables.com/Emily-a-Quadruped-Dog-Robot/ (visited on 08/23/2024).

[64] *Digital Output - SIGMATEK*. URL: https://www.sigmatek-automation.com/en/products/io-systems/s-dias/digital-output/pw-022/ (visited on 07/30/2024).

[65] *MG996R Digital Servo Motor mit Metall Getriebe*. URL: https://www.roboter-bausatz.de/p/mg996r-digital-servo-motor-mit-metall-getriebe (visited on 07/31/2024).

[66] *S-DIAS - SIGMATEK*. URL: https://www.sigmatek-automation.com/en/products/io-systems/s-dias/ (visited on 07/30/2024).

[67] *Controls & HMIs - SIGMATEK*. URL: https://www.sigmatek-automation.com/en/products/accessories/controls-hmis/pcv-522/ (visited on 07/30/2024).

[68] *Interfaces & Splitters - SIGMATEK*. URL: https://www.sigmatek-automation.com/en/products/real-time-ethernet-varan/interfaces-splitters/vi-021/ (visited on 07/30/2024).

[69] CC Huang, Chan-Hsiang Lin, and Che-Kang Wu. "Performance Evaluation of Xenomai 3". In: *Proceedings of the 17th Real-Time Linux Workshop (RTLWS)*. 2015, pp. 21–22.

[70] Veronika Kirova et al. "Impact of Modern Virtualization Methods on Timing Precision and Performance of High-Speed Applications". In: *Future Internet* 11.8 (Aug. 2019), p. 179. ISSN: 1999-5903. DOI: 10.3390/fi11080179.

[71] George K. Adam, Nikos Petrellis, and Lambros T. Doulos. "Performance Assessment of Linux Kernels with PREEMPT_RT on ARM-Based Embedded Devices". In: *Electronics* 10.11 (June 2021), p. 1331. ISSN: 2079-9292. DOI: 10.3390/electronics10111331.

List of Figures

List of Tables

List of Codes

List of Abbreviations

AI Artificial Intelligence

CID Context Identifier

CPU Central Processing Unit

GPOS General-Purpose Operating System

GUI Graphical User Interface

IRQ Interrupt Request

IRT Interrupt Response Time

ISR Interrupt Service Routine

KVM Kernel-based Virtual Machine

LAPIC Local Advanced Programmable Interrupt Controller

LSB Least Significant Bit

ML Machine Learning

MSB Most Significant Bit

PLC Programmable Logic Controller

PWM Pulse Width Modulation

QEMU Quick Emulator

RTOS Real-Time Operating System

SMP Symmetric Multiprocessing

vCPU Virtual Central Processing Unit

VM Virtual Machine

VMCS Virtual Machine Control Structure

vsock Virtual Socket