

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

Virtualisierung eines Echtzeit-Betriebssystems zur Steuerung eines Roboters mit Schwerpunkt auf die Einhaltung der Echtzeit

By: Halil Pamuk, BSc

Student Number: 51842568

Supervisor: Sebastian Rauh, MSc. BEng

Wien, August 12, 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, August 12, 2024

Signature

Kurzfassung

Erstellung einer Echtzeit-Robotersteuerungsplattform unter Verwendung von Salamander OS, Xenomai, QEMU und PCV-522 in der Yocto-Umgebung. Die Plattform basiert auf Salamander OS und nutzt Xenomai für Echtzeit- Funktionen. Dazu muss im ersten Schritt die Virtualisierungsplattform evaluiert werden. (QEMU, Hyper-V, Virtual Box, etc.) Als weiterer Schritt folgt die Anbindung eines Roboters über eine VARAN-Bus Schnittstelle. Das gesamte System wird in der Yocto-Umgebung erstellt und konfiguriert. Das Hauptziel der Arbeit ist es, herauszufinden, wie die Integration von Echtzeit-Funktionen und effizienten Kommunikationssystemen in eine Robotersteuerungsplattform die Reaktionszeit und Zuverlässigkeit von Roboteranwendungen verbessern kann

Schlagworte: Schlagwort1, Schlagwort2, Schlagwort3, Schlagwort4

Abstract

Sections [4.1](#) and [4.2](#) demonstrate the initial real-time latency values gathered for bare metal and virtualization.

Keywords: Echtzeit, Virtualisierung, Xenomai, VARAN

Contents

1	Introduction	1
1.1	Real-Time Operating Systems	2
1.2	Application Context	3
1.3	State of the Art	4
1.3.1	Virtualization	4
1.3.2	Real-time Kernel	4
1.3.3	Related Work	4
1.4	Problem and Task Definition	15
1.5	Objective	15
2	Theoretical Foundations	16
2.1	Yocto	16
2.2	QEMU	16
2.3	Xenomai	16
2.4	VARAN-Bus	17
2.5	Salamander 4	17
2.5.1	Structure	17
2.5.2	Memory Management	18
3	Methodology	20
4	Initial Real-Time Latency	22
4.1	Salamander 4 Bare Metal	22
4.2	Salamander 4 Virtualization	24
5	Real-Time Performance Tuning	29
5.1	BIOS Configurations	31
5.2	Kernel Configurations	32
5.3	Host OS Configurations	36
5.3.1	CPU affinity and Isolation	37
5.3.2	KVM Entry and KVM Exit	37
5.3.3	Interrupt Affinity	39
5.3.4	RT-Priority	43
5.3.5	Disable RT Throttling	43
5.3.6	Disable Timer Migration	44

5.3.7	Set Device Driver Work Queue	44
5.3.8	Disable RCU CPU Stall Warnings	44
5.3.9	Stop Certain Services	44
5.3.10	Disable Machine Check	45
5.3.11	Boot into text-based environment	45
5.4	QEMU/KVM Configurations	46
5.4.1	Tune LAPIC Timer Advance	46
5.4.2	Set QEMU Options for real-time VM	47
5.5	Guest OS Configurations	49
5.6	Other Configurations	49
6	Real-Time Robotic Application	50
6.1	Setup of Hardware Salamander 4	52
6.2	Setup of QEMU Salamander 4	53
6.3	Robotic Application	56
7	Results	57
8	Discussion	58
9	Summary and Outlook	59
	Bibliography	60
	List of Figures	64
	List of Tables	65
	List of Code	66
	List of Abbreviations	67

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 systems (GPOS) and real-time operating systems (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.

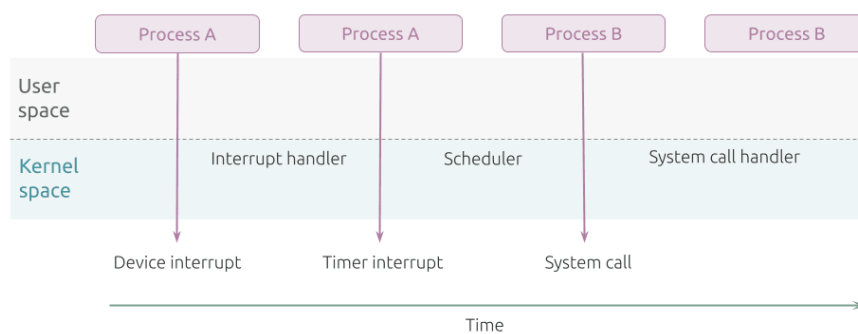


Figure 1: Non-preemptible kernel [2]

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. RTOS are specifically designed to react to events within fixed time limits and prioritise the execution of high-priority processes.

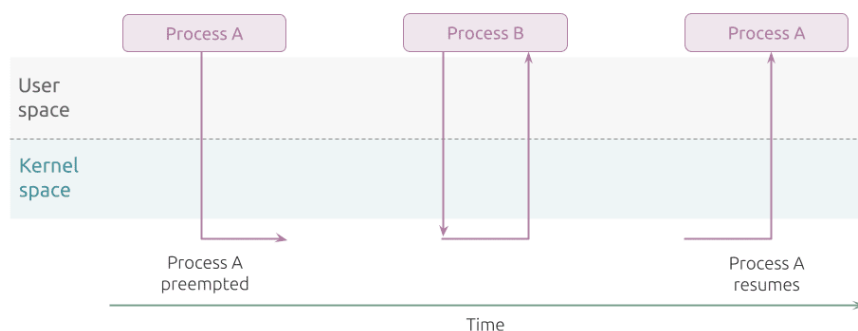


Figure 2: Preemptible kernel [2]

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. 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 [3].



Figure 3: RTOS Structure [3]

In these real-time operating system, task scheduling is based on so-called priority-based preemptive scheduling [4]. 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 [5]. The difference lies exclusively in the consequences of a violation of the time limits. 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.

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 [6].

- 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 chapter 2.5.
- Salamander 4 was created with Yocto [7], an open-source project that helps develop custom Linux-based systems. Yocto is discussed in Section 2.1.
- Salamander 4 is virtualized through Quick emulator (QEMU), an open-source emulator and virtualizer that allows to run different operating systems on a computer QEMU is described in Section 2.2.
- Salamander 4 employs hard real-time with Xenomai 3 [8], a real-time framework for Linux that enables deterministic and low-latency performance. Xenomai 3 is detailed in Section 2.3.
- The goal is to virtualize Salamander 4 and approach the performance of bare metal through real-time performance tunings, focussing on latency requirements set by SIGMATEK. These modifications are subject of chapter 5.
- 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 2.4.
- 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 Section 6.2. This component could then be used to communicate with the Input/Output 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 VI 021 module of SIGMATEK serving as the power supply.
- The robotic application from chapter 6 was written in Lasal Class 2, an object oriented programming tool of SIGMATEK with client-server technology and graphic representation.

1.3 State of the Art

1.3.1 Virtualization

Yocto primarily uses

At the beginning of the work, a detailed analysis of the individual virtualization options of Salamander 4 was carried out.

In particular, the virtualization performance of Ubuntu 22.04, Windows 10 and WSL under QEMU was compared.

1.3.2 Real-time Kernel

1.3.3 Related Work

The paper titled “Demystifying the Real-Time Linux Scheduling Latency” by Daniel Bristot de Oliveira and colleagues explores the intricacies of Linux as a real-time operating system, specifically focusing on the PREEMPT_RT variant. The authors critique the traditional `cyclictst` tool used for measuring scheduling latency, highlighting its limitations due to its black-box approach and lack of theoretical grounding. To address these issues, the paper leverages the Thread Synchronization Model of Linux to derive a set of properties and rules that define the kernel’s behavior from a scheduling perspective¹². This model helps in establishing a sound bound to the scheduling latency by considering all possible sources of delays in synchronization events³⁴. The authors also introduce an efficient tracing method to observe kernel events with minimal overhead, resulting in a practical tool for deriving reliable scheduling latency bounds⁵. The paper concludes with an experimental analysis comparing `cyclictst` and the proposed method, demonstrating that the latter can find sound bounds faster and with acceptable overheads⁶. This work contributes significantly to the understanding and improvement of real-time performance in Linux systems.

The white paper, “Real-Time Performance Tuning Best Practice Guidelines for KVM-Based Virtual Machines,” provides comprehensive guidelines for optimizing real-time performance in KVM-based virtual machines on Intel Architecture (IA) platforms¹. It begins by explaining the importance of real-time systems, which must respond to events within strict time constraints, and the relevance of tuning these systems for applications running in virtual machines (VMs), particularly in the manufacturing sector². The document details the testbed configuration used for performance evaluation, including an industrial PC based on the Intel Whiskey Lake-U platform with specific BIOS, kernel, and host configurations³. Key BIOS settings such as disabling Hyper-Threading and Intel SpeedStep, and enabling Intel VMX and VT-d, are recommended to enhance real-time performance. Kernel configurations include using a real-time kernel, setting specific kernel parameters, and isolating CPUs for real-time workloads. Host configurations

involve setting CPU affinity for system services, cache isolation, and disabling certain services like `irqbalance` and `thermald`. The paper also outlines QEMU-KVM configurations, such as tuning `lapic timer advance` and setting CPU affinity for QEMU threads. For the guest OS in the real-time VM, similar configurations to the host are recommended, with additional tuning for `sched_rt_runtime_us`. The performance evaluation section describes a rigorous testing process, including running heavy workloads alongside the real-time VM, which demonstrated significantly enhanced real-time performance with latency in the range of tens of microseconds⁴. The document concludes that the described tuning methods effectively improve real-time performance for KVM-based VMs, even under heavy workloads⁵.

The paper titled “Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System” explores the integration of real-time operating systems (RTOS) with general-purpose operating systems (GPOS) using Kernel-Based Virtual Machine (KVM) technology. The authors, Ruhui Ma, Fanfu Zhou, Erzhou Zhu, and Haibing Guan, propose an embedded real-time virtualization architecture that combines VxWorks (RTOS) and Linux (GPOS) on a multi-core platform¹. This architecture aims to address issues such as responsiveness, system performance, and CPU resource utilization. The paper details several performance tuning methods, including CPU shielding, prioritization, and a novel co-scheduling algorithm, to achieve sub-millisecond interrupt response latency on the guest VxWorks². Experimental results demonstrate that these tuning methods significantly improve real-time responsiveness and system performance. The study concludes that with careful system setups, such as applying RT patches and real-time tunings on the host Linux, the proposed architecture is suitable for certain embedded real-time scenarios^{3,4}. The work is supported by various Chinese national foundations and highlights the potential of virtualization technology in enhancing the capabilities of embedded systems.

The paper “‘Real Time’ vs. ‘Real Fast’: How to Choose?”¹ by Paul E. McKenney from IBM Linux Technology Center explores the distinction between real-time and real-fast computing in the context of Linux systems. It provides a comprehensive analysis to help developers make informed design choices between achieving quick start times (real-time) and fast completion times (real-fast)². The paper uses two example applications: an embedded fuel-injection system and a Linux kernel build, to illustrate the differences³. The fuel-injection scenario, which requires precise timing, benefits from real-time Linux, while the kernel build, which prioritizes throughput, performs better on non-real-time Linux. The paper discusses factors influencing the choice, such as workload duration, CPU utilization, and memory usage. It also addresses the overheads associated with real-time Linux, including increased locking and interrupt handling costs^{4,5}. The conclusion emphasizes using the right tool for the job and suggests ongoing work to reduce the performance penalty of real-time Linux, potentially making it suitable for a broader range of workloads⁶.

The paper titled “Real-Time Performance Analysis in Linux-Based Robotic Systems” by Hobin Yoon, Jungmoo Song, and Jamee Lee from Samsung Advanced Institute of Technology ex-

plores the feasibility of using real-time flavored Linux for controlling complex humanoid robots¹. The authors emphasize the importance of precise control periods for mobile and humanoid robots, which require timely responses to external stimuli to maintain balance and smooth operation. The study introduces the adoption of EtherCAT technology for real-time communication among distributed devices, highlighting its flexible topology, simple configuration, and cost-effectiveness². The paper details the design and implementation of the Robot Control Kernel Subsystem (RCKS), which integrates EtherCAT master and slave devices, and discusses various real-time performance tunings applied to the Linux kernel to achieve deterministic timing. Experimental results demonstrate that with appropriate real-time tunings, including CPU shielding, memory locking, and spinning nanosleep, Linux can meet the stringent timing requirements of robotic systems, achieving control periods within acceptable limits even under heavy system load. The authors conclude that Linux, with its real-time enhancements, is a viable solution for deploying in advanced robotic systems, offering both real-time capabilities and the benefits of a general-purpose OS³.

The paper titled “Performance Evaluation of Xenomai 3” by Ching-Chun (Jim) Huang, Chan-Hsiang Lin, and Che-Kang Wu, presents an in-depth analysis of Xenomai 3, a real-time framework for Linux. Xenomai 3 operates either as a dual-kernel system alongside Linux or natively over mainline Linux kernels, enhanced by PREEMPT RT for stricter response times¹. The study evaluates Xenomai 3’s performance, particularly in dual-kernel configuration, using ARM Cortex-A series benchmarks. The authors highlight Xenomai 3’s optimizations in RTOS API emulation, thread-to-thread communications, and reduced overhead. The paper compares Xenomai 3 with its predecessor, Xenomai 2, and the PREEMPT RT-enhanced Linux kernel, demonstrating superior maximum response times to external interrupts in dual-kernel setups. The study employs BeagleBone Black hardware and various kernel configurations to measure latencies under idle and CPU-stressed conditions². Results indicate that Xenomai 3 generally outperforms Xenomai 2 and PREEMPT RT, except for a slight degradation in timer-IRQ latency under stress. The paper concludes that Xenomai 3 offers significant improvements in real-time performance and flexibility, making it a robust choice for real-time applications.

The paper titled “Performance Analysis Towards a KVM-Based Embedded Real-Time Virtualization Architecture” explores the challenges and solutions associated with integrating real-time operating systems (RTOS) with general-purpose operating systems (GPOS) in embedded systems using virtualization technology. The authors propose an architecture based on the Kernel-Based Virtual Machine (KVM) that combines VxWorks (an RTOS) and Linux (a GPOS) on a multi-core platform^{1,2}. The primary focus is on evaluating how KVM affects the interrupt-response times of VxWorks when running as a guest OS³. The study highlights the shift from single-core to multi-core processors in embedded systems and the need for legacy RTOS to adapt to multi-core environments⁴. The paper discusses hardware-assisted virtualization, particularly Intel’s VT technology, and the implementation of KVM as a hypervisor within the Linux kernel. Through experimental setups and performance tuning methods like prioritization and

CPU shielding, the authors demonstrate that sub-millisecond interrupt response latency can be achieved, making the proposed architecture suitable for certain embedded real-time applications⁵. The results indicate that with careful system configurations, KVM can effectively support real-time performance requirements in embedded systems.

The paper titled “Performance Analysis Towards a KVM-Based Embedded Real-Time Virtualization Architecture” explores the challenges and solutions associated with integrating real-time operating systems (RTOS) with general-purpose operating systems (GPOS) on multi-core processors in embedded systems. The authors propose an architecture based on the Kernel-Based Virtual Machine (KVM) that combines VxWorks (an RTOS) and Linux (a GPOS) to leverage the benefits of both systems. The study focuses on evaluating the impact of KVM on the interrupt-response times of VxWorks when running as a guest OS. By applying real-time performance tuning methods on the host Linux, the authors demonstrate that sub-millisecond interrupt response latency can be achieved, making the architecture suitable for certain embedded real-time scenarios. The paper also discusses hardware-assisted virtualization, the role of KVM as a hypervisor, and the experimental setup used to measure the performance impacts. The results indicate that with appropriate system configurations, such as prioritization and CPU shielding, the proposed architecture can effectively meet real-time performance requirements.

The paper titled “Real-time capabilities in the standard Linux Kernel: How to enable and use them?” explores the transformation of the Linux operating system into a real-time system using the PREEMPT-RT patches¹. Originally designed as a general-purpose OS, Linux has gained popularity in real-time applications due to its low cost and open-source nature². The authors discuss the importance of real-time systems, which must meet strict temporal requirements to ensure system correctness. They differentiate between hard and soft real-time systems, emphasizing that Linux, with the PREEMPT-RT patch, can achieve soft real-time performance³. The PREEMPT-RT patch, maintained by Ingo Molnar and Thomas Gleixner, allows almost the entire kernel to be preempted, enhancing its real-time capabilities⁴. Key kernel configurations and runtime precautions are highlighted to maintain real-time performance. The paper concludes that while Linux can achieve real-time performance, it is still not as deterministic as commercial real-time operating systems like QNX and Windows CE⁵. However, with proper configuration, Linux can be a viable option for real-time applications.

This paper presents a modular platform for industrial automation that leverages open-source components to run both non-critical activities and real-time control tasks on the same hardware, thereby reducing overall hardware costs. The platform uses a hypervisor to provide isolation, allowing concurrent execution of real-time control code and design tools. The authors detail the software architecture, including the hypervisor and Real-Time Operating System (RTOS), and discuss the necessary extensions for practical application. They address the challenge of interference on shared hardware resources, which can degrade real-time performance. The paper also includes experimental results demonstrating the platform’s effectiveness and performance.

The authors highlight the benefits of using open-source software to avoid expensive commercial solutions and emphasize the importance of multi-core processors and hardware-assisted virtualization in industrial automation. The work is supported by the European Commission through the I-MECH ECSEL project.

Shielded Processors: Guaranteeing Sub-millisecond Response in Standard Linux by Steve Brosky and Steve Rotolo of Concurrent Computer Corporation explores the implementation and benefits of shielded processors in RedHawk Linux for real-time applications. The paper highlights the limitations of standard Linux in providing deterministic performance due to networking and graphics activities. Shielded CPUs, a model developed over a decade of experience, offload operating system overhead to a subset of CPUs, ensuring more deterministic execution and interrupt response times. The authors describe the modifications made to the Linux kernel, including CPU affinity settings and the disabling of local timer interrupts, to create a shielded environment. Benchmark tests demonstrate significant improvements in execution determinism and interrupt response times, with jitter reduced to 1.87% on shielded CPUs compared to 26.17% on standard Linux. The paper concludes that shielded processors can guarantee interrupt response times of less than 30 microseconds, even under heavy system load, making them suitable for applications with hard real-time deadlines. The work underscores the potential of using commodity operating systems like Linux for high-performance real-time applications by leveraging shielded processors.

The paper titled “Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers” by George K. Adam from the CSLab Computer Systems Laboratory, University of Thessaly, investigates the real-time performance of Linux kernels with the PREEMPT_RT patch on embedded devices like the BeagleBoard and Raspberry Pi1. The research focuses on measuring latencies in real-time applications at both user and kernel space using newly developed software modules tailored for these experiments2. The study highlights that the PREEMPT_RT patch significantly improves the real-time performance of Linux kernels, reducing worst-case latencies to around 160 μ s, making these systems more suitable for real-time applications3. The methodology involves response and periodic task models, and the results suggest that the proposed measurement approach can be applied to other Linux-based platforms4. The paper also discusses the challenges and solutions encountered during the experiments, emphasizing the importance of real-time capabilities in embedded systems for applications ranging from consumer electronics to industrial automation.

Asymmetric Scheduling and Load Balancing for Real-Time on Linux SMP Authors: Eric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser12 The paper presents ARTiS, a real-time extension of the GNU/Linux scheduler designed for SMP (Symmetric Multi-Processors) systems3. ARTiS aims to guarantee the preemption of processors for real-time tasks by assigning specific processors to these tasks4. This system ensures low latency by migrating non-preemptible tasks away from real-time processors5. ARTiS also incorporates load-balancing

strategies to fully utilize SMP systems without wasting resources. The implementation of ARTiS involves modifications to the Linux scheduler, resulting in significant improvements in latency compared to the standard Linux scheduler⁶. The paper details the performance evaluation of ARTiS, highlighting its ability to maintain low interrupt latency even under high system load, making it suitable for hard real-time applications. The authors propose further enhancements, including the integration of real-time scheduling policies like EDF (Earliest Deadline First) and RM (Rate Monotonic)⁷.

Title: A Comprehensive Study of Kernel (Issues and Concepts) in Different Operating Systems¹² Authors: Hayfaa Subhi Malallah, Subhi R. M. Zeebaree, Rizgar R. Zebari, Mohammed A. M. Sadeeq, Zainab Salih Ageed, Ibrahim Mahmood Ibrahim, Hajar Maseeh Yasin, Karwan Jameel Merceedi Summary: This paper provides an in-depth comparative analysis of various operating systems (OS) focusing on their kernels, which are the core components managing system resources and providing essential services. The study examines the strengths and limitations of popular OS such as iOS, Android, Mac, Windows, and Linux³. It highlights that Linux, Android, and Windows 10 are particularly noted for their stability, compatibility, and reliability⁴. The paper discusses the evolution of OS kernels, including microkernels, monolithic kernels, and hybrid kernels, and their respective advantages and challenges. It also addresses modern developments in OS technology, such as multicore processors, cloud computing, and the Internet of Things (IoT), and their impact on kernel design and performance. The authors emphasize the importance of security in kernel development, noting that vulnerabilities can lead to significant system compromises. The paper concludes with a discussion on the future directions of OS kernel research and development, advocating for continuous improvement in performance, security, and adaptability to emerging technologies.

Title: Testing the limits of general-purpose hypervisors for real-time control systems Authors: Rui Queiroz, Tiago J. Cruz, Paulo Simões (University of Coimbra) Summary: This paper explores the feasibility of using general-purpose off-the-shelf hypervisors to virtualize real-time control systems within the context of Industry 4.0. The authors address the need for flexibility, security, and resilience in automation infrastructures while maintaining real-time requirements. They discuss the potential benefits of virtualizing servers and control devices in Industrial and Automation Control Systems (IACS), but also highlight the challenges due to the focus of general-purpose hypervisors on maximizing throughput, often at the expense of determinism and increased latency. The paper presents an empirical evaluation of various hypervisors, concluding that while some are capable of handling typical real-time workloads, this capability cannot be generalized to all types of real-time systems. The authors provide a comprehensive analysis of cyber-physical systems (CPS), control devices, and real-time systems, including their latency and jitter requirements. They also delve into the specifics of real-time operating systems (RTOS) and mixed-criticality systems, and review different virtualization techniques, comparing commercial off-the-shelf hypervisors with real-time hypervisors. The study emphasizes the importance of ensuring both spatial and temporal isolation in virtualized environments

to maintain predictable real-time performance. It also discusses the role of scheduling policies in real-time applications and the challenges of I/O sharing between multiple virtual machines. The authors conclude that while general-purpose hypervisors show promise, further research and development are needed to fully realize their potential in real-time control systems.

“Testing the limits of general-purpose hypervisors for real-time control systems” by Rui Queiroz, Tiago J. Cruz, and Paulo Simões from the University of Coimbra: With the advent of Industry 4.0, there is a growing need for flexibility, security, and resilience in industrial automation infrastructures while maintaining real-time requirements. This paper explores the feasibility of using general-purpose off-the-shelf hypervisors to virtualize cyber-physical systems’ servers and control devices. The authors, Rui Queiroz, Tiago J. Cruz, and Paulo Simões, present an empirical evaluation of these hypervisors, focusing on their ability to handle real-time workloads. The study highlights the challenges posed by the need for determinism and low latency, which are often compromised by hypervisors designed to maximize throughput. The paper concludes that while some hypervisors can manage typical real-time workloads, this capability is not universal across all types of real-time systems. The research underscores the importance of understanding the limits and capabilities of conventional hypervisors to ensure they meet the specific requirements of industrial automation control systems.

Title: A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization Authors: Zonghua Gu and Qingling Zhao Summary: This paper provides a comprehensive survey on real-time issues in virtualization for embedded systems, authored by Zonghua Gu and Qingling Zhao. Virtualization, widely accepted in server and cloud computing, has been increasingly applied to embedded systems with stringent timing constraints. The paper discusses various virtualization systems, including KVM, Xen, and L4, and their applications in embedded systems such as avionics, industrial automation, and mobile phones. It highlights the importance of real-time performance, security, and dependability in these systems. The survey covers different types of virtualization, including Type-1 (bare-metal) and Type-2 (hosted) virtualization, as well as full and para-virtualization. It also addresses specific real-time issues like scheduling, partitioning, and the Lock-Holder Preemption problem. The paper concludes with a discussion on the future directions and challenges in the field of real-time virtualization for embedded systems.

Title: How Real (Time) Are Virtual PLCs?1 Authors: Diogenes Javier Perez, Josef Waltl, Laurin Prenzel, Sebastian Steinhorst This paper explores the concept of virtual programmable logic controllers (vPLCs) as a solution to the limitations of traditional hardware-based PLCs in industrial automation. The authors argue that hardware PLCs are costly to maintain and lack the flexibility and scalability needed for modern applications, especially those involving artificial intelligence and machine learning2. The proposed vPLC operates on a commercial off-the-shelf (COTS) server using a real-time hypervisor, which allows it to manage multiple virtual machines and handle resource-intensive tasks efficiently34. The paper details the architecture of the vPLC, which includes a real-time operating system and a hypervisor to ensure low

latency and deterministic behavior. The authors conducted various tests to compare the performance of vPLCs with traditional SoftPLCs, focusing on metrics such as cycle time, jitter, and response time. The results indicate that vPLCs can achieve performance on par with traditional PLCs for applications requiring response times below 10 milliseconds⁵. The study concludes that while further improvements are needed, vPLCs offer significant advantages in terms of flexibility, scalability, and cost-efficiency for future industrial systems⁶.

Impact of Modern Virtualization Methods on Timing Precision and Performance of High-Speed Applications Authors: Veronika Kirova, Kirill Karpov, Eduard Siemens, Irina Zander, Oksana Vasylenko, Dmitry Kachan, Sergii Maksymov This research investigates the precision of timing methods and their efficiency in various virtual environments, focusing on their impact on high-speed network applications. The study examines how timer hardware is shared among CPU- and I/O-bound tasks on both virtualized and bare operating systems. By replacing timing methods within an application for estimating available path bandwidth, the authors analyze the impact on estimation accuracy. The findings highlight that timer overhead and precision are crucial for high-performance network applications, with low-precision timing methods leading to degradation in virtual environments. The paper confirms that using precise timing operations and AvB estimation methods can overcome inefficiencies in standard time-related operations. The research evaluates five different virtual environments to determine the most optimal for high-speed network applications, providing a comprehensive analysis of the impact of virtualization on timing precision and performance.

“Real Time Operating Systems: A Complete Overview” by Mrinal Parikshit Chandane: The paper “Real Time Operating Systems: A Complete Overview” by Mrinal Parikshit Chandane provides an in-depth examination of Real Time Operating Systems (RTOS) and their critical role in embedded systems, particularly in telecommunication applications such as telephony, navigation, and military signaling systems. RTOS is essential for applications that require strict deadline constraints and the ability to handle multiple tasks simultaneously¹. The paper highlights the key features of RTOS, including timing considerations, task prioritization, and communication and synchronization between tasks². The document delves into various design aspects necessary for developing an RTOS, such as task management, multitasking, context switching, task priority, and kernel functions³. It explains how tasks are created, prioritized, and managed within an RTOS, emphasizing the importance of efficient CPU utilization and the prevention of data corruption through mutual exclusion. The paper also discusses the implementation of RTOS, covering topics like programming language selection, task stack management, and the initialization process. Furthermore, the paper presents a practical example of designing and implementing an RTOS for a Philips 8052 microcontroller, demonstrating the application of theoretical concepts in real-world scenarios. The successful implementation of applications such as a column controller, digital calculator, and dancing LEDs showcases the versatility and effectiveness of RTOS in managing complex embedded systems⁴. Overall, the paper provides a comprehensive overview of RTOS, offering valuable insights into its architecture, design, and

implementation, making it a significant contribution to the field of embedded systems⁵.

The paper “Challenges in Real-Time Virtualization and Predictable Cloud Computing” explores the integration of real-time systems with cloud computing and virtualization technologies. The authors, Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu, identify the significant technical challenges in supporting real-time applications within cloud environments¹. They highlight that while cloud computing offers benefits such as reduced operational costs, server consolidation, and elastic resource provisioning, it struggles to meet the demands of soft real-time applications like online video streaming, cloud-based gaming, and telecommunication management². These applications require predictable performance in open, shared, and virtualized environments³. The paper discusses the evolution of cloud computing, emphasizing the shift from local computing resources to distributed cloud-based models. It outlines the benefits of virtualization, including functional execution isolation, easier management, and the coexistence of legacy and new applications⁴. However, the authors note that the multi-tenant nature of cloud computing introduces challenges in providing stable and predictable performance due to resource sharing among multiple virtual machines (VMs). The authors delve into various virtualization techniques, such as full virtualization, hardware-assisted virtualization, para-virtualization, OS-level virtualization, and application-level virtualization, each with its performance characteristics and suitability for real-time applications. They also address the importance of real-time scheduling and resource management in virtualized environments, highlighting the need for hierarchical scheduling and the role of real-time hypervisors. The paper concludes by presenting potential future research directions to enhance the predictability and performance of cloud-based real-time applications, emphasizing the need for improved resource management frameworks and real-time virtualization techniques.

The paper titled “VM-Based Real-Time Services for Automotive Control Applications” is authored by Alejandro Masrur, Sebastian Drossler, Thomas Pfeuffer, and Samarjit Chakraborty. It explores the use of Virtual Machines (VMs) to provide real-time services in automotive control applications¹². The authors utilize the Xen hypervisor to design a real-time control loop on top of a virtualization layer³. They first analyze a typical Xen configuration and identify issues when used for real-time applications⁴. The paper highlights that the SEDF (Simple Earliest Deadline First) scheduler in Xen can be improved with minimal modifications to enhance worst-case performance⁵. To further reduce latency and jitter, the authors propose a new scheduler for Xen that prioritizes real-time VMs using a fixed-priority policy⁶. This approach ensures that timing constraints in automotive control applications are met⁷. The paper includes extensive experiments to illustrate the effectiveness of the proposed solutions.

Title: Hard Real Time Linux* Using Xenomai* on Intel® Multi-Core Processors¹² Author: Amarpreet Singh Ugal, BIOS/Firmware Engineer, Intel Corporation Summary: This document explores the implementation of hard real-time capabilities in Linux using Xenomai on Intel® multi-core processors. Traditional Linux is not a hard real-time operating system because it cannot guarantee that tasks will meet strict deadlines³⁴. To address this, the document discusses

the use of a high-priority real-time kernel that runs between the hardware and standard Linux. This real-time kernel executes real-time tasks and suspends normal Linux processes during this period⁵⁶. The real-time scheduler treats the standard Linux kernel as an idle task, allowing real-time tasks to preempt normal processes⁷⁸. Xenomai, which uses an interrupt pipeline from the ADEOS project, is highlighted as a solution for managing interrupts and ensuring minimal latency and jitter. The document also covers the installation and configuration of Xenomai on CentOS 5.2, including kernel patching and testing with various tools like switchbench, switchtest, cyclicttest, clocktest, and latency. The detailed steps and configurations provided aim to help developers achieve hard real-time performance on Intel® Core™ i7 multi-core processors.

Title: Performance Assessment of Linux Kernels with PREEMPT_RT on ARM-Based Embedded Devices Authors: George K. Adam, Nikos Petrellis, Lambros T. Doulos Summary: This research investigates the real-time performance of Linux kernels and distributions with the PREEMPT_RT real-time patch on ARM-based embedded devices, specifically BeagleBoard and Raspberry Pi. The study aims to provide novel insights into Linux real-time performance by evaluating various metrics such as response latency, periodic task execution, and maximum sustained frequency using heuristic methods and the cyclicttest benchmark tool. The authors introduce a new response task model and develop specific software measurement modules to assess the performance. The results demonstrate that the PREEMPT_RT patch significantly improves real-time performance compared to default Linux kernels, particularly in reducing worst-case latencies. This makes ARM-based devices running Linux with PREEMPT_RT more suitable for time-sensitive embedded control systems. The research also highlights the growing use of Linux in embedded systems for real-time applications, driven by its open-source nature, performance, and adaptability to various hardware systems. The methodology and findings of this study could be applied to other Linux-based real-time platforms, contributing to the broader field of real-time systems research.

Title: Real-Time I/O System for Many-core Embedded Systems 1 Author: Zhe Jiang In this thesis, Zhe Jiang addresses the critical need for time predictability in modern real-time embedded systems, particularly focusing on I/O operations. The proposed solution, BlueIO, is a hardware-implemented real-time I/O virtualization system designed to meet the requirements of predictability, timing accuracy, enhanced performance, scalability, parallel access, and isolation simultaneously². BlueIO integrates key functionalities of I/O virtualization and low-layer I/O drivers through the Virtualized Complicated Device Controller (VCDC) and a clock cycle-level timing-accurate I/O controller, the GPIO Command Processor (GPIOCP)³. This system supports abstract virtualized access to I/O devices from the software domain, providing isolation and concurrent access to I/O operations, thereby improving performance⁴⁵. The thesis also introduces BlueVisor, a scalable real-time hardware hypervisor built upon BlueIO, enabling predictable virtualization on CPU, memory, and I/O, along with fast interrupt handling and inter-virtual machine communication⁶⁷. The work demonstrates significant improvements in I/O performance and low running costs across different operating systems, with a hardware con-

sumption analysis showing linear scalability with the number of CPUs and I/O devices.

Title: Real-Time Operating Systems¹ Author: Jiacun Wang, Monmouth University Summary: This chapter provides an in-depth overview of Real-Time Operating Systems (RTOS), which are crucial for embedded systems requiring timely and predictable responses. The chapter begins by outlining the main functions of general-purpose operating systems, such as process management, memory management, interrupts management, multitasking, file system management, and I/O management. It then delves into the specific characteristics of RTOS kernels, emphasizing the need for predictable timing behavior, priority scheduling, intertask communication, and resource sharing. The chapter also discusses clocks and timers, asynchronous I/O, and memory locking as essential features of RTOS. Finally, it provides examples of widely used RTOS products, including LynxOS, OSE, QNX, VxWorks, and Windows Embedded Compact, highlighting their unique features and applications in various industries such as aerospace, automotive, and telecommunications.

Title: Real-Time Scheduling: From Hard to Soft Real-Time Systems¹ Authors: Giuseppe Lipari (Université de Lille) and Luigi Palopoli (Università di Trento) Summary: This paper explores the evolution of real-time systems, traditionally divided into hard real-time and soft real-time categories. Hard real-time systems are critical, where missing a deadline can lead to catastrophic consequences, such as in automotive brake systems². Soft real-time systems, on the other hand, aim to optimize the quality of service, like multimedia players that can tolerate occasional delays without severe impact. The authors argue that the boundary between these two categories is becoming increasingly blurred. They introduce the concept of time-predictability and criticality to understand the origins of real-time deadlines. The paper presents the resource reservation framework as a method for scheduling both hard and soft real-time tasks. This framework enhances system robustness and resource utilization by isolating the temporal behavior of tasks. The paper also discusses the Constant Bandwidth Server (CBS) algorithm, which integrates with the EDF scheduler to manage execution budgets and periods, ensuring temporal isolation³. The authors highlight the importance of designing systems that remain consistent even when deadlines are missed, emphasizing the need for robust software development practices. In conclusion, the paper demonstrates how resource reservation techniques can be applied to classical control systems, improving performance and robustness⁴. The authors provide a comprehensive overview of existing techniques and tools for soft real-time scheduling and analysis, making a strong case for the integration of these methods in modern real-time systems⁵.

The paper titled “Performance analysis of real-time and general-purpose operating systems for path planning of the multi-robot systems” by Seçkin Canbaz and Gökhan Erdemir explores the performance differences between real-time operating systems (RTOS) and general-purpose operating systems (GPOS) in the context of multi-robot systems. The study focuses on two Linux distributions, Ubuntu and Pardus, and evaluates their effectiveness as both GPOS and

RTOS for path planning tasks¹². The authors highlight the critical distinction between GPOS and RTOS, emphasizing that RTOS can preempt low-priority tasks with high-priority ones, even during kernel calls, which is not possible in GPOS. The research involves using the turtlesim simulation environment to perform path tracking tasks with robot groups of varying sizes. The experiments compare CPU usage and processing times across the different operating systems. Results indicate that Ubuntu GPOS generally outperforms other configurations in terms of lower CPU usage and shorter processing times³. However, RTOS configurations, while slower, offer higher system security and reduced error margins due to their strict time management. The study concludes that while GPOS may be preferable for performance-centric applications, RTOS is essential for tasks with high time dependency. The authors suggest that the choice between GPOS and RTOS should consider factors such as potential failure conditions, process security, and processing time requirements. This research provides valuable insights into the suitability of different operating systems for multi-robot applications, contributing to the broader field of robotics and autonomous systems.

Title: The Real-Time Linux Kernel: A Survey on PREEMPT_RT¹² Authors: Federico Reghenzani, Giuseppe Massari, and William Fornaciari from Politecnico di Milano³ Summary: This survey explores the evolution and current state of the PREEMPT_RT patch for the Linux kernel, which aims to enhance the real-time capabilities of Linux. The authors discuss the increasing demand for real-time applications and the challenges posed by using Commercial-Off-The-Shelf (COTS) hardware in such environments. The PREEMPT_RT patch modifies the existing Linux kernel to improve predictability and reduce latencies, making it suitable for real-time applications. The survey covers various approaches to real-time Linux, including cokernel approaches like RTLinux, Xenomai, and RTAI, and compares them with the PREEMPT_RT patch⁴. The authors highlight the benefits of using Linux in real-time systems, such as the availability of rich hardware support and a well-established programming environment. They also discuss the challenges of integrating real-time capabilities into a general-purpose operating system and the ongoing efforts to address these issues. The survey concludes with an overview of industrial and scientific use cases that have benefited from the PREEMPT_RT patch, emphasizing its significance in fields like aerospace, automotive, and high-performance computing.

1.4 Problem and Task Definition

1.5 Objective

The main objective of this work is to create a real-time robot control platform that integrates Salamander OS, Xenomai, QEMU and PCV-522 in the Yocto environment.

2 Theoretical Foundations

2.1 Yocto

2.2 QEMU

2.3 Xenomai

Xenomai 3 [8] 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 4. 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.

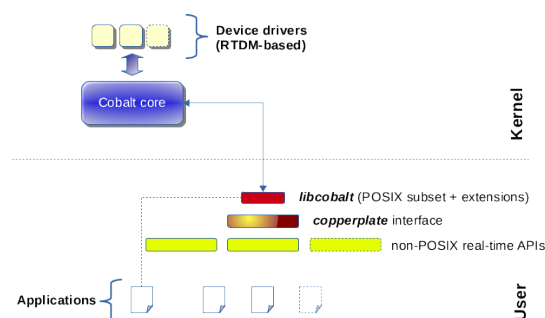


Figure 4: Xenomai Cobalt interfaces [8]

The second approach, called Mercury and shown in Figure 5, relies on the real-time capabilities already present in the native Linux kernel. Often, applications require the PREEMPT-RT extension to augment the mainline kernel's real-time responsiveness and minimize jitter, but this isn't mandatory and depends on the application's specific requirements for responsiveness and tolerance for occasional deadline misses.

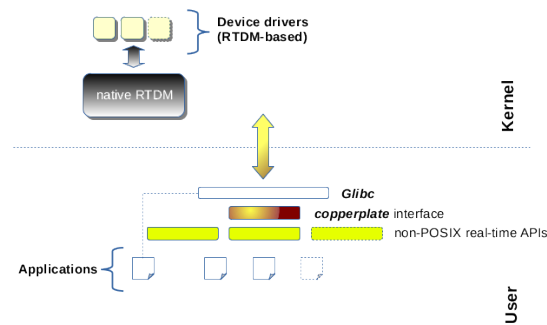


Figure 5: Xenomai Mercury interfaces [8]

Salamander 4 uses the Cobalt real-time core with the Dovetail extension, which allows the kernel to handle real-time tasks with low latency.

2.4 VARAN-Bus

2.5 Salamander 4

This chapter briefly describes the Salamander 4 operating system by SIGMATEK.

2.5.1 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 [8]. 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: System information

Salamander 4 is powered by SIGMATEK's CP 841 [9] 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 executing them.
- **Hardware classes:** Hardware classes in LASAL represent the different types of hardware components that can be controlled by the LASAL CPU. They provide a way to organize and manage the hardware components in a modular and reusable manner. The graphical hardware editor in LASAL allows for a true-to-detail simulation of the actual hardware.
- **Application:** Applications are developed using LASAL CLASS 2 [10], 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.

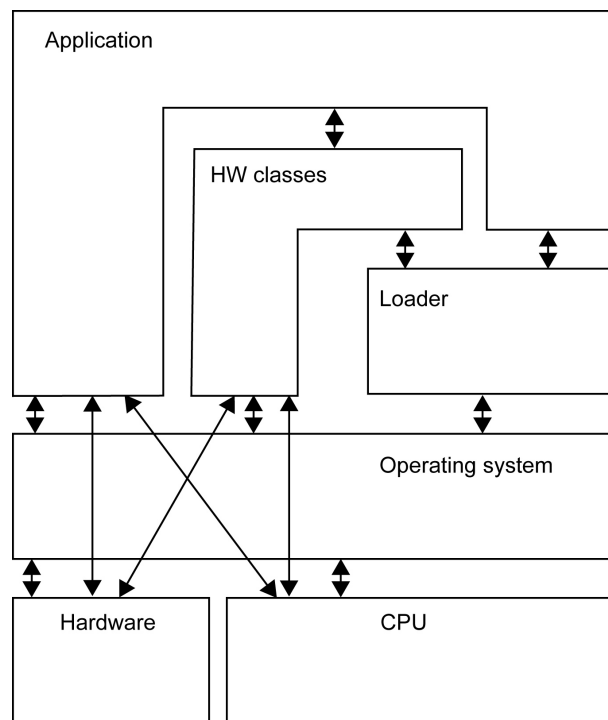


Figure 6: Structure of Salamander 4 CPU

2.5.2 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 the LASAL Class 2 can run, providing defined real-time functions, data types, and other constructs tailored for real-time programming.

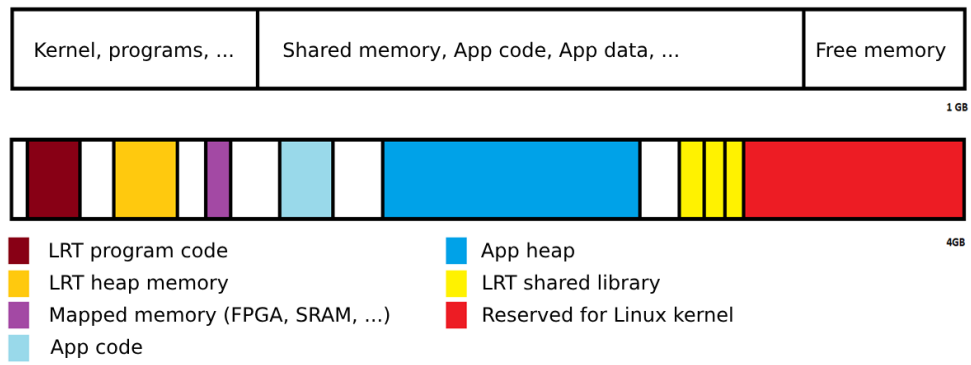


Figure 7: Memory Management

3 Methodology

This section describes in detail all the theoretical concepts and boundary conditions as well as practical methods that contributed to achieving the objectives of this master's thesis.

Trace-cmd can be used for tracing the Linux kernel [11]. It can record various kernel events such as interrupts, scheduler decisions, file system activity, function calls in real time. Trace-cmd helped in getting detailed insights into system behaviour and identify reasons for latency.

The data that was recorded by trace-cmd was then fed into Kernelshark, which is a graphical front-end tool [12]. It visualizes the recorded kernel trace data in a readable way on an interactive timeline, which facilitated the process of identifying patterns and correlations between events. By further filtering the displayed events according to specific criteria such as processes, event types or time ranges, the latency issues were analyzed.

Real-time operating system capabilities were provided by Xenomai, which is real-time development framework that extends the Linux kernel [8]. It enables low-latency and deterministic execution of time-critical tasks. Xenomai 3 introduces a dual-kernel approach with a real-time kernel coexisting alongside Linux. A key utility within the Xenomai suite is the latency tool, which benchmarks the timer latency - 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 latency between expected and actual execution times.

The system configuration is shown in Table 1

Table 1: System configuration

CPU	13 th Gen Intel(R) Core(TM) i7-13800H
Memory	2 × 16GB SO-DIMM DDR5-5600 MT/s, 32GB
GPU	NVIDIA RTX A500 Laptop GPU
BIOS	Dell Version 1.12.0
OS	Ubuntu 22.04.4 LTS

Figure 8 is the output of the `lstopo` command and visualizes the hardware nodes of the system, including CPU cores, caches, memory, and I/O devices.

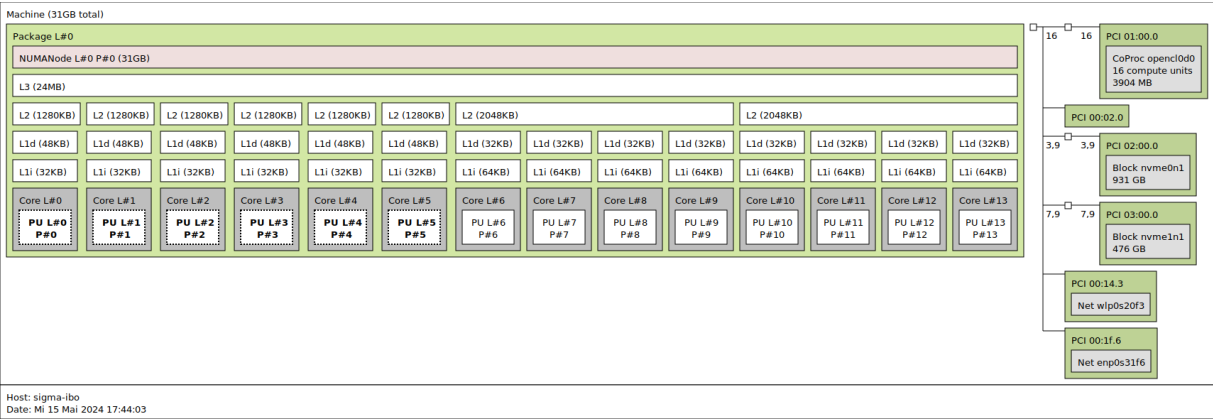


Figure 8: Hardware topology

4 Initial Real-Time Latency

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 proprietary hardware of SIGMATEK used to employ the custom operating system. Salamander 4 virtualization refers to a virtual version of the Salamander 4 hardware platform, achieved through QEMU/KVM. Sections 4.1 and 4.2 specify the details of the measurements for both versions. In the further course, the aim was to bring the latency values of the virtualization closer to those of the hardware and guarantee deterministic and reliable behavior.

4.1 Salamander 4 Bare Metal

The output of the command `uname -a` for Salamander 4 bare metal is shown in Code 2.

```
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 2: Salamander 4 bare metal system information

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 min, avg, max latencies in a Gnuplot-compatible format to the file `gnuplot.txt`. Figure 9 shows the gathered latency values in microseconds.

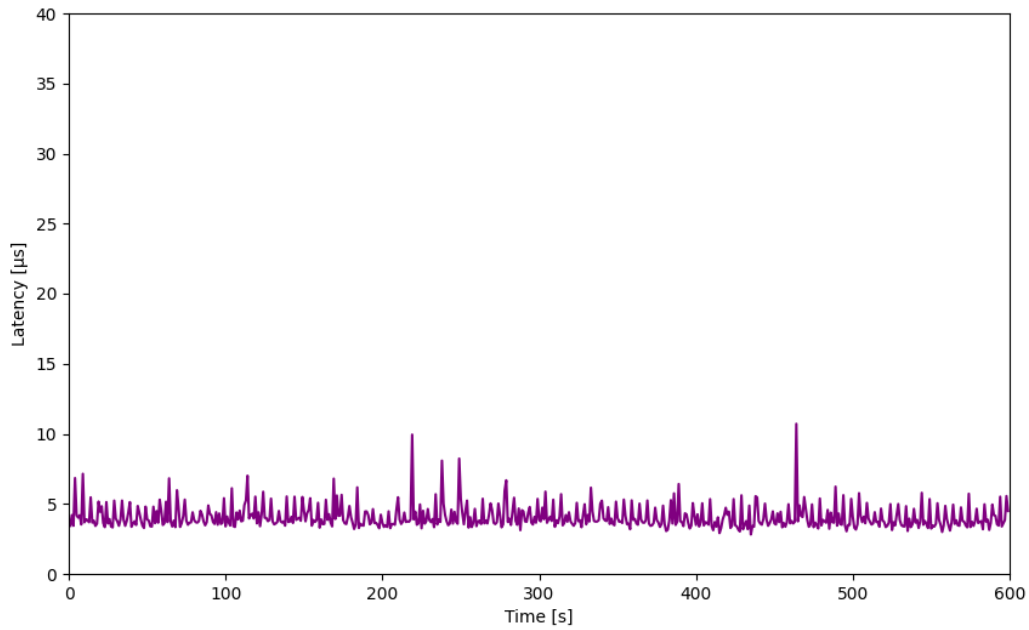


Figure 9: Latency of Salamander 4 bare metal

The statistics obtained from this measurement are provided in Table 2. It gives an overview of the average, maximum, minimum latency, and the standard deviation of the latency values in microseconds.

Table 2: Latency statistics of Salamander 4 bare metal in microseconds

Statistic	Value (μs)
Average Latency	4.06
Maximum Latency	10.71
Minimum Latency	2.82
Standard Deviation	0.85

Figure 10 depicts the variation in latency over the course of said time. Since the data varies strongly, a logarithmic scale was used for both axes.

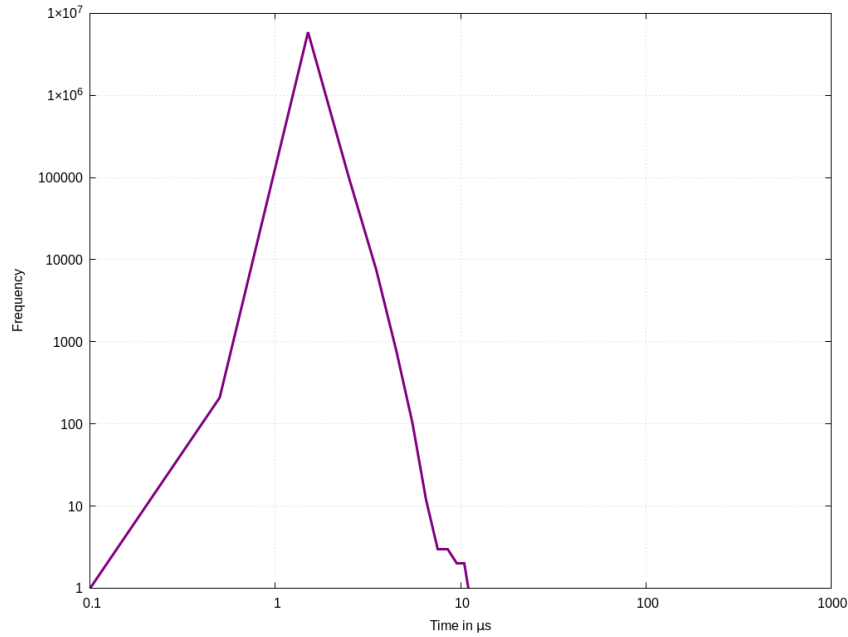


Figure 10: Variation in latency of Salamander 4 bare metal

4.2 Salamander 4 Virtualization

In addition to providing Salamander 4 on its own hardware, SIGMATEK has also developed a virtualised version of this operating system. It was developed using Yocto, an open-source project that allows customised Linux distributions to be created for embedded systems. Upon generating the necessary files, Yocto provides a QEMU folder with the following components shown in Code 3. QEMU is the environment in which the virtualization runs, as it is an open-source tool for hardware virtualization [13].

```

1  sigma_ibo@localhost:~/Desktop/salamander-image$ ls -l
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 3: Contents of QEMU folder for Salamander 4

With the help of the script depicted in Code 4, Salamander 4 is started together with the necessary hardware components in the QEMU environment. This makes it possible to run Salamander 4 on a variety of host systems, regardless of the specific hardware of the host. 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. “bz” stands for big-zipped.
- **drive-c**: Directory serving as C drive for QEMU system, 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 Salamander 4 OS. It is described in Code 4.
- **salamander-image-sigmatek-core2.ext4**: Disk image of the Salamander 4 OS for the Sigmatek Core 2 platform. 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 4. This script is used to start QEMU with required parameters to boot Salamander 4 OS. It will be adjusted in chapter 5 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 4: QEMU script for starting Salamander 4 virtualization

This script is run on a generic Ubuntu 22.04.4 system, as mentioned previously in chapter 3. The kernel version and other details are presented in Code 5, using the `uname -a` command.

```
1 root@sigmatek-core2:~# uname -a
2 Linux sigma-ibo 6.5.0-35-generic #35~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Tue
   May  7 09:00:52 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
```

Code 5: Ubuntu 22.04.4 system information

Measuring the latency of the Salamander 4 virtualization with the default QEMU script in Code 4 and no further adjustments for 10 minutes, the following latency values in Figure 11 were collected.

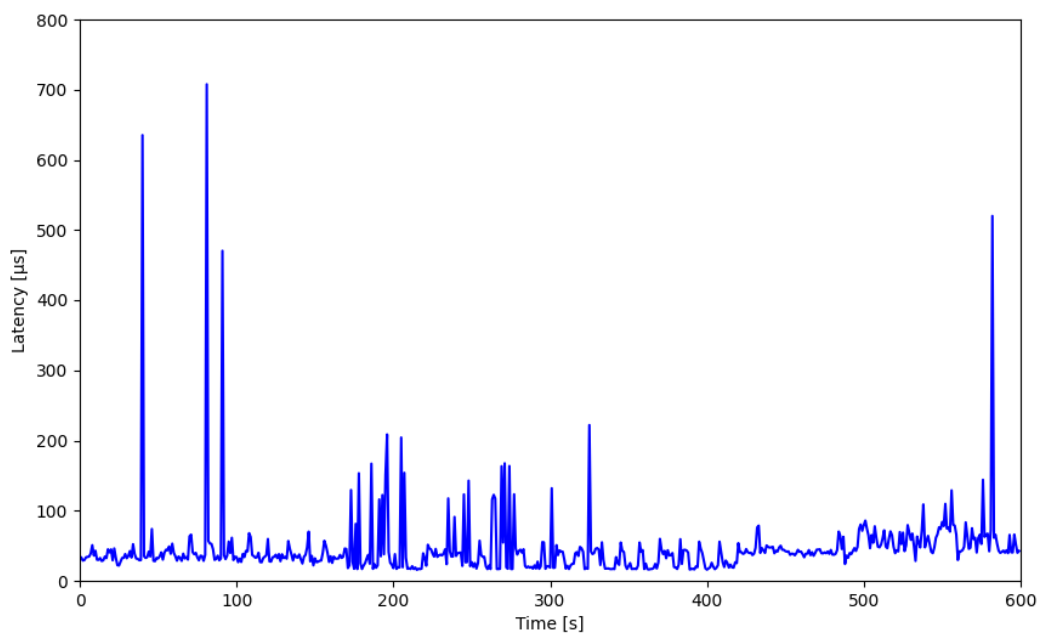


Figure 11: Latency with default settings

The statistics obtained from this measurement are provided in Table 12. It gives an overview of the average, maximum, minimum latency, and the standard deviation of the latency values in microseconds.

Table 3: Latency statistics of default Salamander 4 virtualization in microseconds

Statistic	Value (μs)
Average Latency	46.22
Maximum Latency	707.62
Minimum Latency	15.59
Standard Deviation	52.13

Figure 12 depicts the variation in latency over the course of said time with default settings.

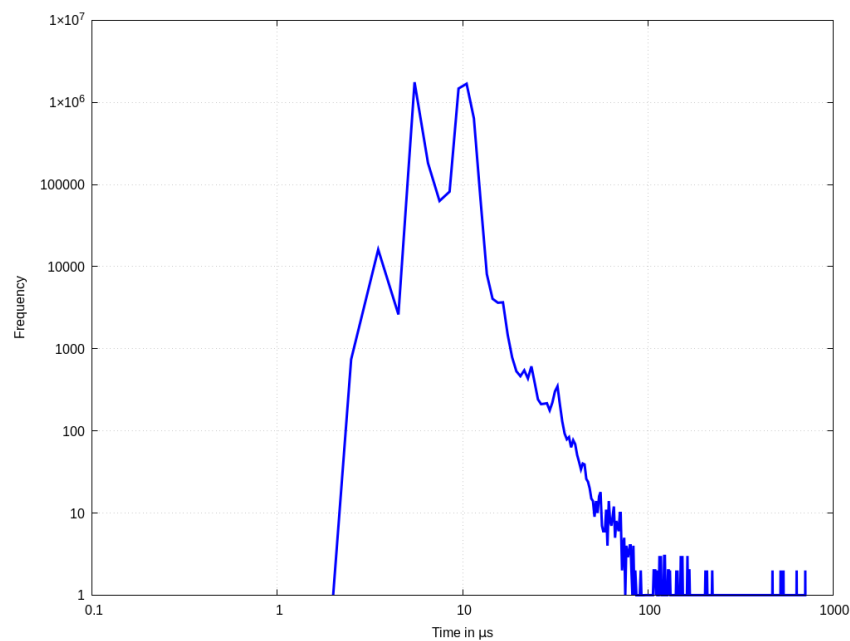


Figure 12: Variation in latency with default settings

Comparing these values to those of bare metal in Figure 13, it is evident that there is a significant initial gap in the statistics. A maximum latency of 707.62 μs is not tolerable for the system and needs to be tuned.

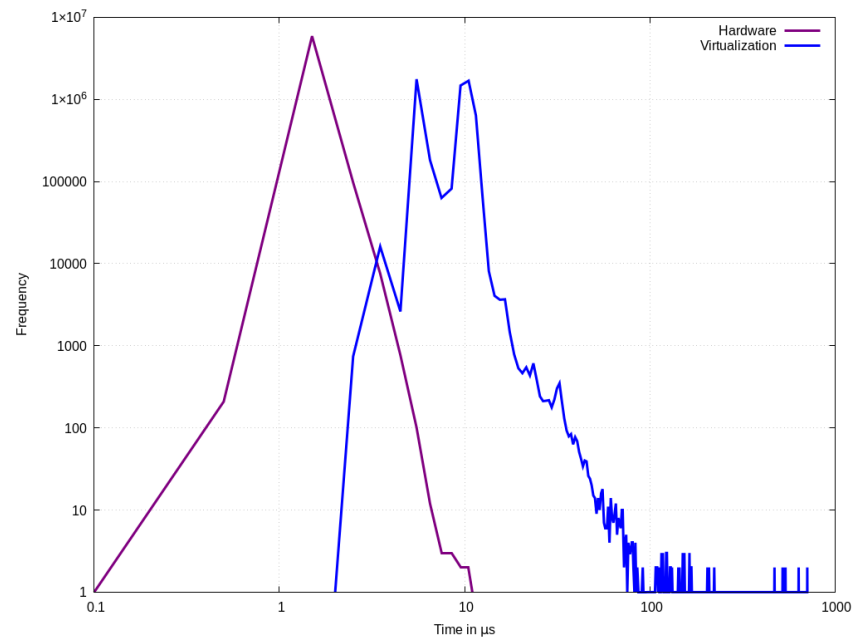


Figure 13: Comparison of variation in latency between hardware and virtualization

5 Real-Time Performance Tuning

In this chapter, the significant initial gap in latency statistics between the virtualized system and the bare metal 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 OS itself. The individual configurations will be discussed in detail and the modifications will be justified with a clear explanation. The goal is to bring the latency of the virtualized system closer to that of the bare metal system, 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 highest possible reliability. The very step of achieving the goal of reducing latency is to apply the preempt-rt patch [14] 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 [15]. The patch also includes support for high-resolution timers and make the system more predictable, which are crucial for real-time applications where the timing of operations must be guaranteed with precise timing and scheduling [16].

Nevertheless, a real-time kernel alone does not make a system truly “real-time” [17]. Additional modifications are required to achieve this. The next sections will deal with these real-time performance tunings. The very first step in this regard is the isolation of a CPU to dedicate it solely to the desired real-time task. CPU isolation and affinity will be covered in detail later in section 5.3.1, but its impact will be briefly demonstrated here.

Figure 14 shows latency of QEMU taskset Salamander4.

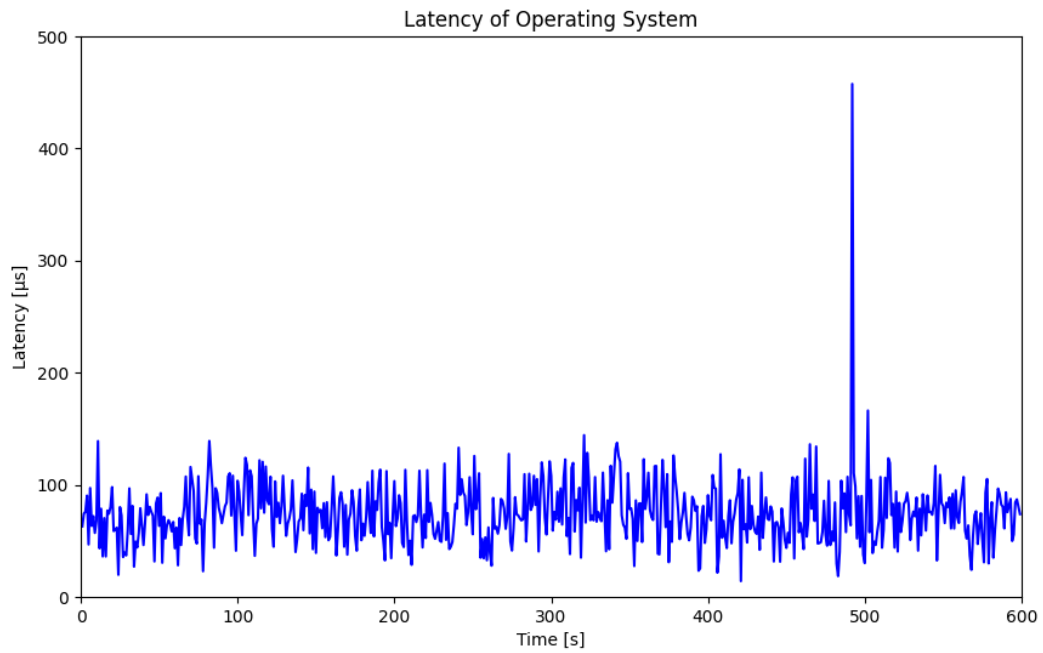


Figure 14: Latency taskset

After each tuning, the latency test of Xenomai will be executed to evaluate the improved latency of the system.

After Article, Figure 15

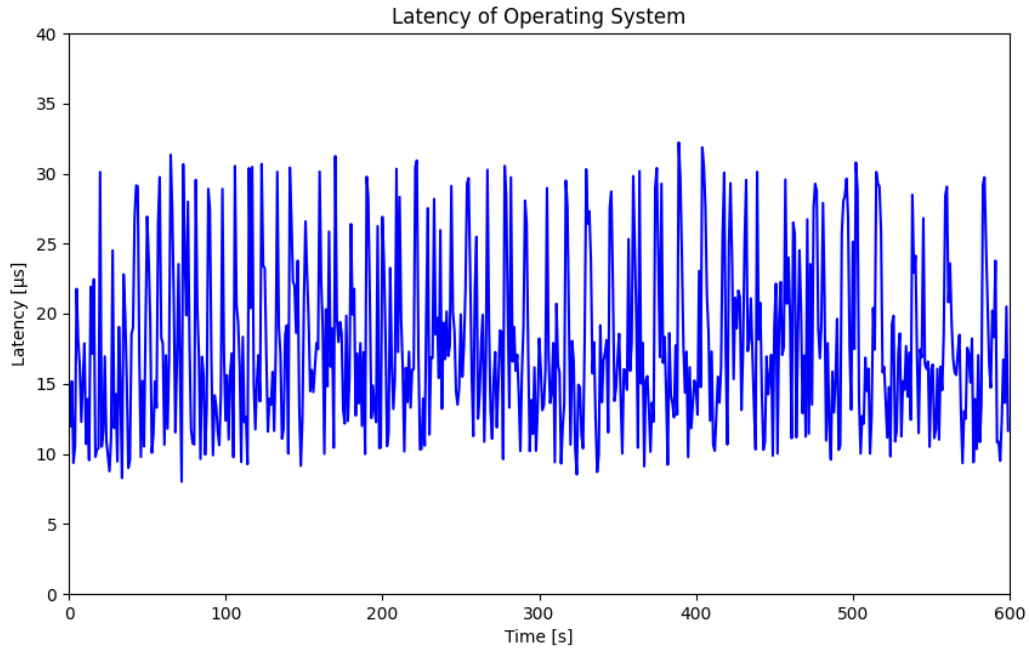


Figure 15: After Article configurations

Table 4: Latency statistics of Salamander 4 virtualization after BIOS configurations in microseconds

Statistic	Value (μs)
Average Latency	17.68
Maximum Latency	32.21
Minimum Latency	8.005
Standard Deviation	6.1

5.1 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 5 illustrates the specific BIOS settings that have been adjusted for the purpose of real-time performance.

Table 5: BIOS Configurations

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:** Hyper-Threading allows CPUs to process two threads simultaneously instead of just one. When it is enabled on the host, this allows the parallelisation 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 technology dynamically adjusts the clock speed of the CPU based on workload. These dynamic changes 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.

5.2 Kernel Configurations

The kernel command-line parameters are shown in Code 6 below.

```

1 GRUB_CMDLINE_LINUX="isolcpus=4 rcu_nocbs=4 rcu_nocb_poll nohz_full=4
    nohz=on default_hugepagesz=1G hugepagesz=1G hugepages=8
    intel_iommu=on rdt=13cat 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 6: Kernel Configuration

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 section 5.3.1.
- **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.
- **nohz=on**: Sets all CPUs to tickless mode system-wide.
- **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, such as graphics cards and network adapters, to the main memory. It can enhance device performance by allowing these devices to directly access and use memory, which is especially helpful when these devices are virtualized.
- **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 is a mechanism that 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 to exit 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 Interrupt Request affinity to none. 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. Section 5.3.3 dives deeper into Interrupt Request affinity.

After the BIOS and Kernel configurations, Figure 16

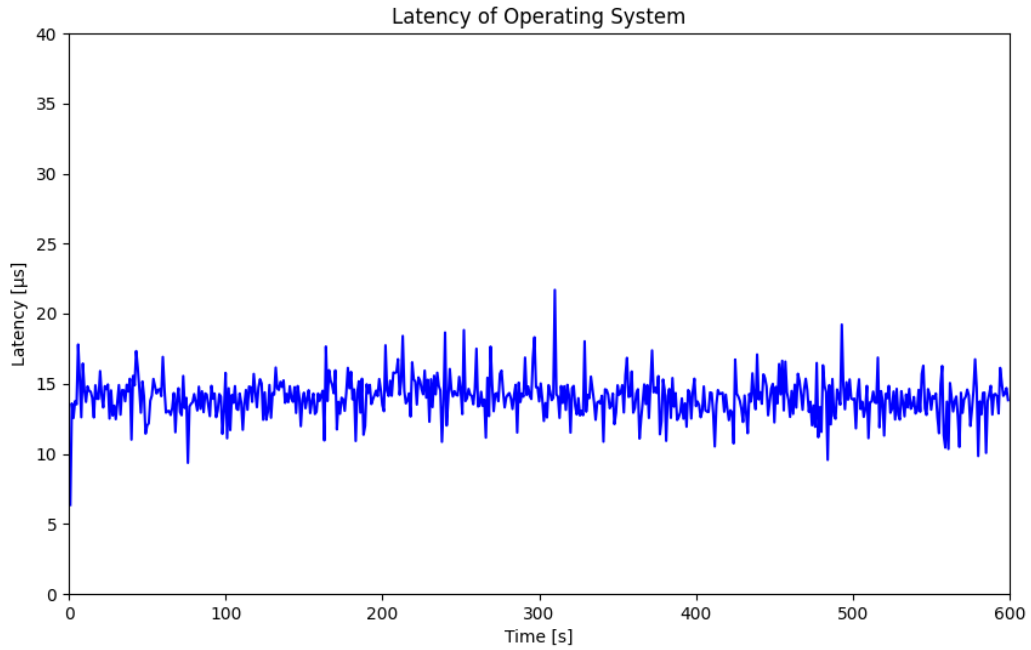


Figure 16: After BIOS and Kernel configurations

Table 6: Latency statistics of Salamander 4 virtualization after Kernel configurations in microseconds

Statistic	Value (μs)
Average Latency	14.00
Maximum Latency	21.694
Minimum Latency	6.351
Standard Deviation	1.44

5.3 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 reduce latency. In the following, a detailed overview of these configurations and their impact on the real-time performance of the guest OS is provided.

5.3.1 CPU affinity and Isolation

Isolating CPUs involves 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 was used to isolate a performance 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 utilize the CPU [18], including `systemd` services. To prevent `systemd` services from running on an isolated CPU, the CPU affinity can be set with line `CPUAffinity=0 1 2 3 5 6 7 8 9 10 11 12 13` 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 7 shows the user and kernel tasks that run on CPU 4. After the isolation, user tasks other than the QEMU process have been removed from running on this CPU. Only 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 7: User and Kernel Tasks

5.3.2 KVM Entry and KVM Exit

Upon isolating a CPU to the QEMU process, it was anticipated that the guest would utilize nearly 100% of the CPU's capacity, with minimal to no intervention from the host. However, the `isolcpus` function only isolates CPUs from the scheduler, meaning that user-space processes won't be scheduled on these CPUs, but kernel tasks and interrupts can still utilize the CPU. This led to the investigation of the causes for the observed high and inconsistent latency. The guest

operates within the `kvm_entry` and `kvm_exit` events of the host. Kernelshark revealed a high frequency of `kvm_exit` events, indicating that the guest frequently relinquishes control of the CPU back to the host. This frequent switching hinders the guest's ability to run continuously, thereby increasing the virtualization latency.

To further understand this, `trace-cmd` was employed to trace various events in the host-guest communication, including the reasons for these events. Specifically, the causes for `kvm_exit` events were analyzed. The command `sudo trace-cmd record -e all -A @3:823 --name Salamander4 -e all` was executed on the host for a duration of 5 seconds. The results in Figure ?? were obtained. Additionally, Table 7 provides a short description of the observed `kvm_exit` events.

Exit Reason	Description
APIC_WRITE	Triggered when the guest writes to its APIC.
EXTERNAL_INTERRUPT	Triggered by external hardware interrupts.
HLT	Triggered when the guest executes the HLT instruction.
EPT_MISCONFIG	Triggered by a misconfiguration in the EPT.
PREEMPTION_TIMER	Triggered when the host's preemption timer expires.
PAUSE_INSTRUCTION	Triggered when the PAUSE instruction is executed.
EPT_VIOLATION	Triggered by a violation of the EPT permission settings.
IO_INSTRUCTION	Triggered when the guest executes an I/O instruction.
EOI_INDUCED	Triggered when an EOI signal is sent to the APIC.
MSR_READ	Triggered when the guest reads from a MSR.
CPUID	Triggered when the guest executes the CPUID instruction.

Table 7: Description of `kvm_exit` reasons

In the process of analyzing the `kvm_exit` events, several reasons for these exits were identified. The most frequent among these were the `APIC_WRITE` and `HLT` events. The former is initiated when the guest writes to its Advanced Programmable Interrupt Controller (APIC), a component of the CPU that manages hardware interrupts. The latter occurs when the guest executes the HLT instruction, effectively halting the CPU until the next external interrupt is fired. Other significant but less frequent events included `EXTERNAL_INTERRUPT` and `IO_INSTRUCTION`. These events are indicative of the guest's interaction with hardware devices and its execution of I/O operations. Events such as `EPT_MISCONFIG` and `PREEMPTION_TIMER` were also noted. These could potentially signal issues with memory management and the host's scheduling of the guest. While events like `PAUSE_INSTRUCTION`, `EPT_VIOLATION`, `EOI_INDUCED`, `MSR_READ`, and `CPUID` were the least frequent, they still provide valuable insights into the guest's behavior and the host-guest interaction.

When the script is started from the host, the QEMU process can be scheduled to run on any available core, as it is not bound to a specific CPU core. This means that the QEMU process may frequently switch between different cores, leading to an increase in latency. As the goal was to reduce latency in the guest, the first step was to isolate a CPU of the host and dedicate it solely to the QEMU process, so that it cannot be used for other tasks on the user level. Without CPU isolation, context switches take place at the operating system level and not at the hypervisor level. This explains why there are fewer `kvm_exit` events. However, this will also, as previously shown, lead to higher latency, as context switches at the operating system level generally take longer than a `kvm_exit` and `kvm_entry`. When the CPU was dedicated to the QEMU process, on the other hand, there was a significant increase in `kvm_exit` events. This is because every context switch takes place at the hypervisor level. Nevertheless, lower latency was achieved thereby, as the qemu process is no longer influenced by the CPU scheduling of the operating system.

5.3.3 Interrupt Affinity

Once the CPUs were isolated, interrupt requests handling was 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. The `/proc/interrupts` file 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 needed 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 (CPU13). When there are 14 CPUs available, the default value for `smp_affinity` would be `3FFF`. Removing CPU 4 out of this bitmask would be setting bit five to zero, resulting in `3FEF` (`11 1111 1110 1111`). The python script in Code 8 was written to show a table of the distribution of interrupt requests across each CPU. By changing the values of the `smp_affinity` files, the assignment of IRQs to CPUs is controlled.

```

1     import os
2     import pandas as pd
3     from tabulate import tabulate
4
5     # Get the number of CPUs
6     num_cpus = os.cpu_count()
7
8     # Initialize a dictionary to store the CPUs for each IRQ
9     irqs = {}
10
11    # Iterate over each IRQ
12    for irq in os.listdir('/proc/irq'):
13        # Check if the smp_affinity file exists for this IRQ
14        if os.path.isfile(f'/proc/irq/{irq}/smp_affinity'):
15            # Read the current smp_affinity
16            with open(f'/proc/irq/{irq}/smp_affinity', 'r') as f:
17                affinity = int(f.read().strip(), 16)
18            # Initialize an empty list to store the CPUs for this IRQ
19            cpus = []
20            # Iterate over each CPU
21            for cpu in range(num_cpus):
22                # Check if the bit for the current CPU is set
23                if ((affinity & (1 << cpu)) != 0):
24                    # Add the CPU to the list for this IRQ
25                    cpus.append(cpu)
26            # Sort the list of CPUs
27            cpus.sort()
28            # Add the list of CPUs to the dictionary for this IRQ
29            irqs[irq] = cpus
30
31    # Create a DataFrame to store the table
32    df = pd.DataFrame(index=sorted(irqs.keys(), key=int),
33                      columns=range(num_cpus))
34
35    # Fill the DataFrame with 'x' where a CPU is assigned to an IRQ
36    for irq, cpus in irqs.items():
37        for cpu in cpus:
38            df.loc[irq, cpu] = 'x'
39
40    # Replace NaN values with empty strings
41    df.fillna('', inplace=True)
42
43    # Print the table in pipe format
44    print(tabulate(df, headers='keys', tablefmt='pipe', showindex=True))
45
46    # Convert the DataFrame to a markdown table
47    markdown_table = df.to_markdown()
48
49    # Write the markdown table to a file
50    with open('table_CPU_IRQ.md', 'w') as f:
51        f.write(markdown_table)

```

Code 8: Check distribution of interrupt requests across each CPU

However, as the proc filesystem resets to its default state after each reboot, manually changing numerous IRQ files is tedious and time-consuming. This process was automated using the shell script in Code 9, which is executed after every reboot. Additionally, the CPU affinity of IRQ threads of NVME (Non-Volatile Memory Express, a type of SSD storage) can be set away from CPU 4 to avoid impacting real-time workloads. This can be done through finding out the respective process IDs through `ps -e | grep irq/*.nvme` and then executing `sudo taskset -a -p -c 0 <PID>`.

```

1      #!/bin/bash
2
3      # Check if a command-line argument is provided
4      if [ -z "$1" ]; then
5          echo "Please provide a CPU number as a command-line argument."
6          exit 1
7      fi
8
9      # Get the CPU number from the command-line argument
10     CPU=$1
11
12     # Define the mask values
13     declare -A mask_values
14     mask_values=( [0]="3ffe" [1]="3ffd" [2]="3ffb" [3]="3ff7" [4]="3fef"
15                   [5]="3fdf" [6]="3fbf" [7]="3f7f" [8]="3eff" [9]="3dff" [10]="3bff"
16                   [11]="37ff" [12]="2fff" [13]="17ff")
17
18     # Run the check_smp_affinity.sh script and get the IRQs
19     IRQs=$(./check_smp_affinity.sh $CPU | grep -o '[0-9]\+')
20
21     # Initialize an empty array to store the IRQs that could not be removed
22     failed_IRQs=()
23
24     # Initialize an empty array to store the IRQs that were successfully
25     # removed
26     succeeded_IRQs=()
27
28     # Loop over the IRQs
29     for IRQ in $IRQs; do
30         # Try to change the smp_affinity
31         echo ${mask_values[$CPU]} | sudo tee /proc/irq/$IRQ/smp_affinity >
32         /dev/null 2>&1
33
34         # If the command failed, add the IRQ to the failed_IRQs array
35         if [ $? -ne 0 ]; then
36             failed_IRQs+=($IRQ)
37         else
38             succeeded_IRQs+=($IRQ)
39         fi
40     done
41
42     # Check if there were any failed IRQs
43     if [ ${#failed_IRQs[@]} -ne 0 ]; then
44         echo "IRQs ${failed_IRQs[@]} could not be removed from CPU $CPU."
45     fi
46
47     # Check if there were any successful IRQs
48     if [ ${#succeeded_IRQs[@]} -ne 0 ]; then
49         # Remove the first entry from the succeeded_IRQs array
50         succeeded_IRQs=("${succeeded_IRQs[@]:1}")
51         echo "IRQs ${succeeded_IRQs[@]} were removed from CPU $CPU."
52     fi

```

Code 9: Change IRQ assignment of a CPU

5.3.4 RT-Priority

Having a real-time kernel itself is a crucial part of achieving deterministic behavior, but not enough to take full advantage of the real-time capabilities. One key aspect of this are real-time priorities, thoroughly explained by Richard Weinberger in [19]. In essence, Table 8 lists minimum and maximum priorities for different scheduling policies in Linux.

Table 8: 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 pre-empted 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.

5.3.5 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.

5.3.6 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.

5.3.7 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`.

5.3.8 Disable RCU CPU Stall Warnings

As already mentioned in section 5.2, 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.

5.3.9 Stop Certain Services

Services like `irqbalance.service`, `thermald.service`, and `wpa_supplicant.service`, as presented in Table 9 can be further sources for random latency and unnecessary overhead. Stopping these services through `sudo systemctl stop <SERVICE>` means they will not be able to interrupt the CPU with their tasks.

Table 9: Description of Services

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

5.3.10 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.

5.3.11 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 the Host configurations, [Figure 17](#)

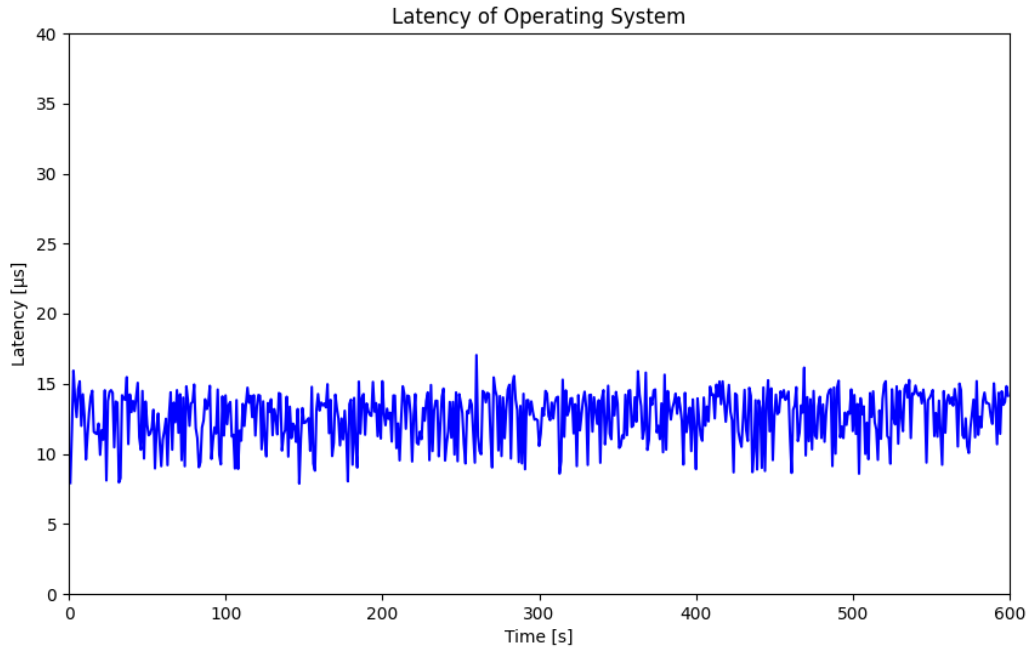


Figure 17: After Host configurations

Table 10: Latency statistics of Salamander 4 virtualization after Host configurations in microseconds

Statistic	Value (μs)
Average Latency	12.61
Maximum Latency	17.041
Minimum Latency	7.872
Standard Deviation	1.8

5.4 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.

5.4.1 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.

5.4.2 Set QEMU Options for real-time VM

QEMU provides several options that can be used to improve the real-time performance of the guest VM. Table 11 briefly explains these options.

Table 11: QEMU options for real-time performance

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

Code 10 shows the final QEMU script used to start the Salamander 4 virtualization, including these options.

```

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

```

Code 10: Tuned QEMU script for starting Salamander 4 virtualization

After the QEMU configurations, Figure 18

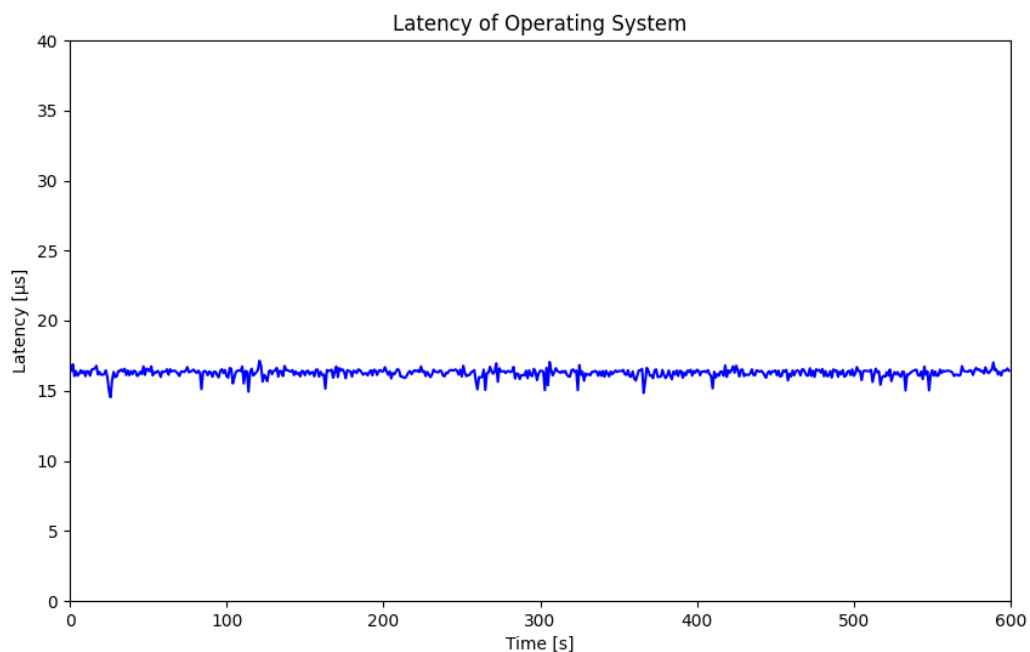


Figure 18: After QEMU configurations

Table 12: Latency statistics of Salamander 4 virtualization after QEMU configurations in microseconds

Statistic	Value (μs)
Average Latency	16.26
Maximum Latency	17.134
Minimum Latency	14.532
Standard Deviation	0.3

5.5 Guest OS Configurations

5.6 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 [20] and gives insight for finding sources of latency on the linux system in [21]. A Checklist for Writing Linux Real-Time Applications is provided by John Ogness in [22]. 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. [23] and [24] 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 mentioned in [25] and [26].

6 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 hardware. 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

The drive system of the robot arm consists of six MG996R digital servo motors [27], 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.



Figure 20: MG996R Servo Motor [27]

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. In this experiment, this PWM signal is generated by the proprietary PW 022 pulse width module of Sigmatek [28]. Its connector layout is illustrated in Figure 21.

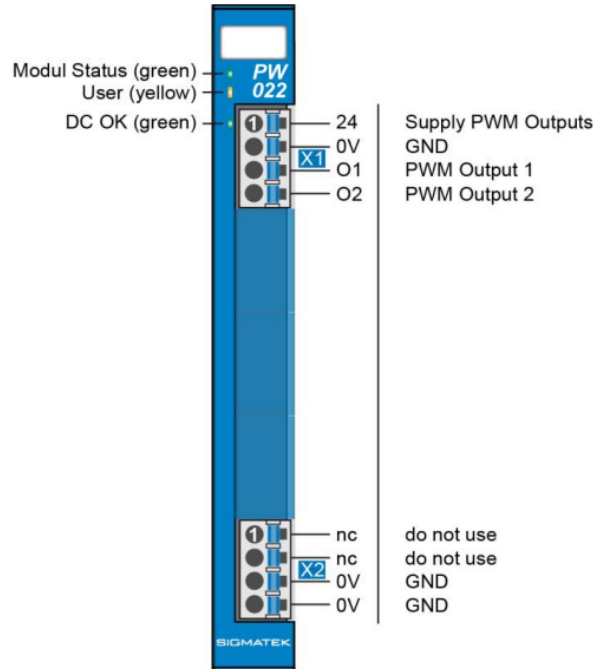


Figure 21: PW 022 pulse width module connector layout [28]

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 [27], this voltage needed to be reduced through resistors before supplying it to a servo motor. For this purpose, two resistors with resistances of 2500 kilohms and 1000 kilohms were connected in series. The voltage across each resistor is calculated using Ohm's law, which is given by equation 1 below.

$$V = I \cdot R \quad (1)$$

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 kilohm resistor is approximately 17.14 volts.
- The voltage across the 1 kilohm resistor is approximately 6.86 volts. This voltage was 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 22.

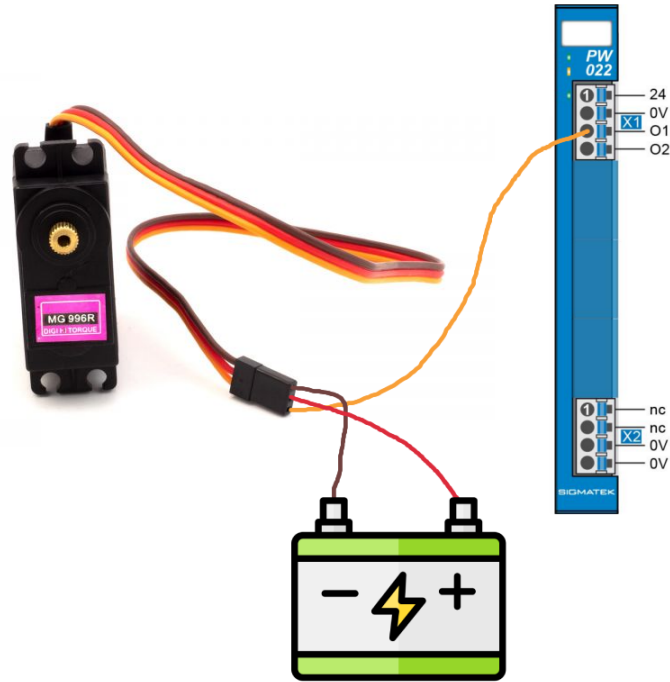


Figure 22: Connection between PW 022 and mini-robot [29]

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

6.1 Setup of Hardware Salamander 4

In this version, Salamander 4 runs on the CP 841 [9] CPU unit, specifically designed for the Salamander 4 operating system. The PW 022 module is directly mounted on the CPU via the S-DIAS bus and communicates over the hard real-time capable Ethernet VARAN with 100 Mbit/s [30]. The setup is visible on Figure 23.

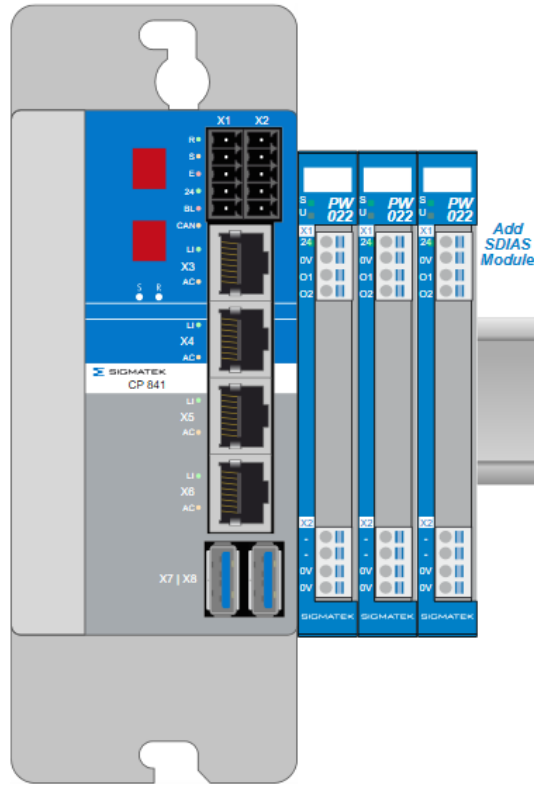


Figure 23: Hardware tree

6.2 Setup of QEMU Salamander 4

In the virtualized setup of the experiment, the VCPU functionality of QEMU is used to connect QEMU with the PWM module. In order to achieve that, the PCV 522 VARAN Manager PCI Insert Card [31], which serves as a bridge between the PC and the rest of the setup, needed 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 was `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 from the previous sections also needed to be modified to include the PCV 522 VARAN Manager PCI Insert Card. 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 11.

```

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 taskset -c 4-5 chrt -f 99 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 schedstats=enable nohlt
    idle=poll quiet xeno_hal.smi=1 xenomai.smi=1 threadirqs" \
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 -object memory-backend-ram,id=ram0,size=4G,prealloc=on \
17 -mem-prealloc -mem-path /dev/hugepages \
18 -device vfio-pci,host=03:00.0 \
19 -no-reboot -nographic

```

Code 11: Include PCI in QEMU script for Salamander 4 virtualization

An additional Varan Connection module VI 021 [32] 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 24.

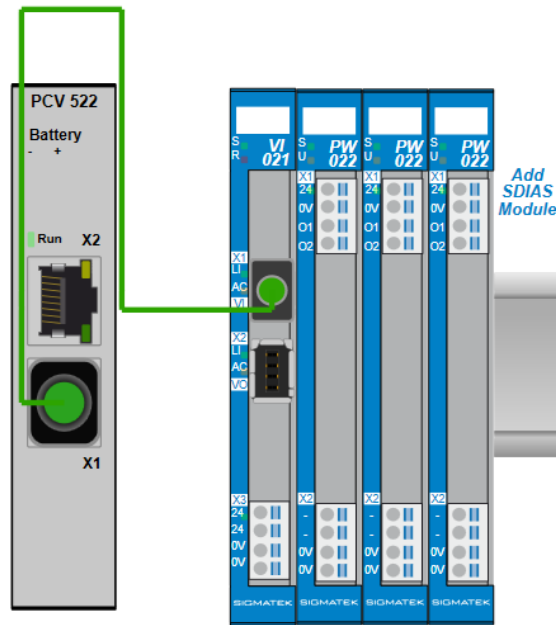


Figure 24: Virtualization tree

Essentially, the CP 841 CPU is being emulated by the VCPU functionality of QEMU, allowing the PCV 522 VARAN Manager PCI Insert Card and the Varan Connection module VI 021 to interact as if they were communicating with a physical CPU unit.

6.3 Robotic Application

7 Results

[6] [11] [12] [8] [9] [10] [7] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25]
[26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44]
[45] [46] [47] [48] [49] [50] [51] [52] [53]

8 Discussion

9 Summary and Outlook

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](https://doi.org/10.11591/ijece.v12i1.pp285-292). (Visited on 08/11/2024).
- [2] *What Is Real-Time Linux? Part II*. <https://ubuntu.com/blog/what-is-real-time-linux-ii>. (Visited on 08/12/2024).
- [3] Jiacun Wang. *Real-Time Embedded Systems*. Hoboken, NJ, USA: Wiley, 2017. ISBN: 978-1-119-42070-5.
- [4] 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](https://doi.org/10.1007/978-3-031-45410-3).
- [5] Giuseppe Lipari, Luigi Palopoli, and Università di Trento. "Real-Time Scheduling: From Hard to Soft Real-Time Systems". In: ().
- [6] pixelart. *SIGMATEK - Komplette Automatisierungssysteme*. <https://www.sigmatek-automation.com/de/>. (Visited on 03/27/2024).
- [7] *Welcome to the Yocto Project Documentation — The Yocto Project @ 4.3.999 Documentation*. <https://docs.yoctoproject.org/>. (Visited on 03/27/2024).
- [8] *Xenomai :: Xenomai*. <https://xenomai.org/>. (Visited on 03/21/2024).
- [9] *CPU-Einheiten* - *SIGMATEK*. <https://www.sigmatek-automation.com/de/produkte/steuerungssysteme/cpu-einheiten/cp-841/>. (Visited on 05/27/2024).
- [10] *Engineering Tool LASAL* - *SIGMATEK*. <https://www.sigmatek-automation.com/de/produkte/engineering-tool-lasal/lasal-class/>. (Visited on 05/27/2024).
- [11] *Trace-Cmd*. <https://trace-cmd.org/>. (Visited on 03/25/2024).
- [12] *KernelShark*. <https://kernelshark.org/>. (Visited on 03/25/2024).
- [13] *QEMU*. <https://www.qemu.org/>. (Visited on 03/27/2024).
- [14] *Realtime:Preempt_rt_versions* [Wiki]. https://wiki.linuxfoundation.org/realtime/preempt_rt_versions. (Visited on 08/05/2024).
- [15] *Realtime Kernel Patchset* - *ArchWiki*. https://wiki.archlinux.org/title/Realtime_kernel_patchset. (Visited on 08/05/2024).

- [16] Steven Rostedt and Darren V Hart. "Internals of the RT Patch". In: ().
- [17] *What Is Real-Time Linux? Part I*. <https://ubuntu.com/blog/what-is-real-time-linux-i>. (Visited on 08/05/2024).
- [18] Ruhui Ma et al. "Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System". In: ().
- [19] *Linux Process Priorities Demystified*. <https://sigma-star.at/blog/2022/02/linux-proc-prios/>. (Visited on 06/11/2024).
- [20] Kernel Recipes. *Kernel Recipes 2016 - Understanding a Real-Time System (More than Just a Kernel) - Steven Rostedt*. Oct. 2016. (Visited on 06/11/2024).
- [21] The Linux Foundation. *Finding Sources of Latency on Your Linux System - Steven Rostedt, VMware*. Sept. 2020. (Visited on 06/11/2024).
- [22] The Linux Foundation. *A Checklist for Writing Linux Real-Time Applications - John Ogness, Linutronix GmbH*. Nov. 2020. (Visited on 06/11/2024).
- [23] *HOWTO: Build an RT-application - RTwiki*. https://archive.kernel.org/oldwiki/rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application.html. (Visited on 06/11/2024).
- [24] *Real-Time Programming with Linux, Part 2: Configuring Linux for Real-Time - Shuhao's Blog*. <https://shuhaowu.com/blog/2022/02-linux-rt-appdev-part2.html#f4>. (Visited on 06/11/2024).
- [25] *KVM/Qemu Virtualization Tuning Guide on Intel® Xeon® Based Systems*. <https://www.intel.com/content/www/us/en/developer/articles/guide/kvm-tuning-guide-on-xeon-based-systems.html>. (Visited on 06/11/2024).
- [26] "Real-Time Performance Tuning Best Practice Guidelines for KVM-Based Virtual Machines". In: (2022).
- [27] *MG996R Servo Motor*. <https://components101.com/motors/mg996r-servo-motor-datasheet>. (Visited on 07/30/2024).
- [28] *Digital Output - SIGMATEK*. <https://www.sigmatek-automation.com/en/products/io-systems/s-dias/digital-output/pw-022/>. (Visited on 07/30/2024).
- [29] *MG996R Digital Servo Motor mit Metall Getriebe*. <https://www.roboterbausatz.de/p/mg996r-digital-servo-motor-mit-metall-getriebe>. (Visited on 07/31/2024).
- [30] *S-DIAS - SIGMATEK*. <https://www.sigmatek-automation.com/en/products/io-systems/s-dias/>. (Visited on 07/30/2024).
- [31] *Controls & HMIs - SIGMATEK*. <https://www.sigmatek-automation.com/en/products/accessories/controls-hmis/pcv-522/>. (Visited on 07/30/2024).

- [32] *Interfaces* & *Split-
ters* - SIGMATEK. <https://www.sigmatek-automation.com/en/products/real-time-ethernet-varan/interfaces-splitters/vi-021/>. (Visited on 07/30/2024).
- [33] 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](https://doi.org/10.3390/electronics10111331). (Visited on 05/08/2024).
- [34] 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](https://doi.org/10.3390/computers10050064). (Visited on 05/16/2024).
- [35] 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](https://doi.org/10.1109/IPDPS.2003.1213237). (Visited on 04/18/2024).
- [36] 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](https://doi.org/10.1016/j.future.2021.12.002). arXiv: [2112.06875](https://arxiv.org/abs/2112.06875) [cs]. (Visited on 03/25/2024).
- [37] Daniel Bristot de Oliveira et al. "Demystifying the Real-Time Linux Scheduling Latency". In: ().
- [38] 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](https://doi.org/10.1016/j.sysarc.2014.07.004). (Visited on 03/25/2024).
- [39] 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](https://doi.org/10.4236/jsea.2012.54033). (Visited on 03/25/2024).
- [40] "Hard Real Time Linux* Using Xenomai* on Intel® Multi-Core Processors". In: ().
- [41] 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](https://doi.org/10.1109/ETFA52439.2022.9921545). (Visited on 03/25/2024).
- [42] 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](https://doi.org/10.3390/fi11080179). (Visited on 03/25/2024).
- [43] Jan Kiszka. "Towards Linux as a Real-Time Hypervisor". In: ().

- [44] 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.
- [45] 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.
- [46] 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](https://doi.org/10.9734/ajrcos/2021/v8i330201). (Visited on 03/25/2024).
- [47] Alejandro Masrur et al. "VM-Based Real-Time Services for Automotive Control Applications". In: *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*. Macau, China: IEEE, Aug. 2010, pp. 218–223. ISBN: 978-1-4244-8480-5. DOI: [10.1109/RTCSA.2010.38](https://doi.org/10.1109/RTCSA.2010.38). (Visited on 03/25/2024).
- [48] Paul E McKenney. "'Real Time' vs. 'Real Fast': How to Choose?" In: ().
- [49] Éric Piel et al. "Asymmetric Scheduling and Load Balancing for Real-Time on Linux SMP". In: *Parallel Processing and Applied Mathematics*. Ed. by David Hutchison et al. Vol. 3911. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 896–903. ISBN: 978-3-540-34141-3 978-3-540-34142-0. DOI: [10.1007/11752578_108](https://doi.org/10.1007/11752578_108). (Visited on 05/06/2024).
- [50] 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](https://doi.org/10.1016/j.micpro.2023.104848). (Visited on 03/25/2024).
- [51] 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](https://doi.org/10.1145/3297714). (Visited on 03/25/2024).
- [52] Hobin Yoon, Jungmoo Song, and Jamee Lee. "Real-Time Performance Analysis in Linux-Based Robotic Systems". In: ().
- [53] *CPU Units - SIGMATEK*. <https://www.sigmatek-automation.com/en/products/control-systems/cpu-units/cp-841/>. (Visited on 07/30/2024).

List of Figures

Figure 1	Non-preemptible kernel	1
Figure 2	Non-preemptible kernel	1
Figure 3	RTOS Structure	2
Figure 4	Xenomai Cobalt interfaces	16
Figure 5	Xenomai Mercury interfaces	17
Figure 6	Structure of Salamander 4 CPU	18
Figure 7	Memory Management	19
Figure 8	Hardware topology	21
Figure 9	Latency of Salamander 4 bare metal	23
Figure 10	Variation in latency of Salamander 4 bare metal	24
Figure 11	Latency with default settings	26
Figure 12	Variation in latency with default settings	27
Figure 13	Comparison of variation in latency between hardware and virtualization	28
Figure 14	Latency taskset	30
Figure 15	After Article configurations	31
Figure 16	After BIOS and Kernel configurations	36
Figure 17	After Host configurations	46
Figure 18	After QEMU configurations	48
Figure 19	Mini-robot of the experiment	50
Figure 20	MG996R Servo Motor	50
Figure 21	PW 022 pulse width module connector layout	51
Figure 22	Connection between PW 022 and mini-robot	52
Figure 23	Hardware tree	53
Figure 24	Virtualization tree	55

List of Tables

Table 1	System configuration	20
Table 2	Latency statistics of Salamander 4 bare metal in microseconds	23
Table 3	Latency statistics of default Salamander 4 virtualization in microseconds	27
Table 4	Latency statistics of Salamander 4 after BIOS configurations	31
Table 5	BIOS Configurations	32
Table 6	Latency statistics of Salamander 4 after Kernel configurations	36
Table 7	Description of kvm_exit reasons	38
Table 8	Minimum and maximum priorities for different scheduling policies	43
Table 9	Description of Services	45
Table 10	Latency statistics of Salamander 4 after Host configurations	46
Table 11	QEMU options for real-time performance	47
Table 12	Latency statistics of Salamander 4 after QEMU configurations	49

List of Code

Code 1	System information	17
Code 2	Salamander 4 bare metal system information	22
Code 3	Contents of QEMU folder for Salamander 4	24
Code 4	QEMU script for starting Salamander 4 virtualization	25
Code 5	Ubuntu 22.04.4 system information	26
Code 6	Kernel Configuration	33
Code 7	User and Kernel Tasks	37
Code 8	Check distribution of interrupt requests across each CPU	40
Code 9	Change IRQ assignment of a CPU	42
Code 10	Tuned QEMU script for starting Salamander 4 virtualization	48
Code 11	Include PCI in QEMU script for Salamander 4 virtualization	54

List of Abbreviations

CPU Central Processing Unit

QEMU Quick Emulator

IRQ Interrupt Request