FH University of Applied Sciences TECHNIKUM WIEN

MASTER THESIS

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

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, June 10, 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, June 10, 2024 Signature

Kurzfassung

Erstellung einer Echtzeit-Robotersteuerungsplattform unter Verwendung von Salamander OS, Xenomai, QEMU und PCV-521 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 inital real-time latency values gathered for bare metal and virtualization.

Keywords: Echtzeit, Virtualisierung, Xenomai, VARAN

Contents

1 Introduction

In today's industrial production and automation, robot systems are well established and of crucial importance. Robots must react to their environment and perform time-critical tasks within strict time constraints. Delays or errors can have catastrophic consequences in some cases. Traditional operating systems, such as Windows or Linux, are often not suitable for these types of real-time requirements as they cannot guarantee deterministic execution times. Therefore, real-time operating systems are required that are specifically designed to react to events within fixed time limits and prioritise the execution of high-priority processes.

The core component of an RTOS that enables real-time capabilities is the kernel. The kernel 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. The kernel also implements priority-based scheduling algorithms, such as Rate Monotonic Scheduling (RMS) or Earliest Deadline First (EDF), to schedule tasks based on their priorities and timing constraints. Additionally, RTOS kernels are designed to minimize interrupt latency, which is crucial for real-time applications that require immediate response to external events.

In these RTOS systems, task scheduling is based on so-called priority-based preemptive scheduling. 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 categorized as either soft or hard real-time systems. 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.

1.1 Application Context

This master's thesis was written at SIGMATEK GmbH & Co KG [1]. SIGMATEK uses its own customized Linux distribution, namely Salamander 4, to be run on their self-manufactured CPUs. Salamander 4 system employs hard real-time with Xenomai 3 and requires a worst latency value between 20 and 50 µs. The goal is to virtualize Salamander 4 and approach the performance of bare metal. Salamander 4 is built with Yocto and virtualized through QEMU/KVM. The details of this operating system are explained in chapter 3.

1.2 State of the art

1.3 Problem and task definition

1.4 Objective

The main objective of this work is to create a real-time robot control platform that integrates Salamander OS, Xenomai, QEMU and PCV-521 in the Yocto environment.

2 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 was used for tracing the Linux kernel. 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 [2].

The data that was recorded by trace-cmd was then fed into Kernelshark, which is a graphical front-end tool [3]. 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. 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 [4].

The system configuration is shown in Table 1

Table 1: System configuration

CPU	13th Gen Intel(R) Core(TM) i7-13800H
Memory	2 $\times$ 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 1 is the output of the `lstopo` command and visualizes the hardware nodes of the system, including CPU cores, caches, memory, and I/O devices.

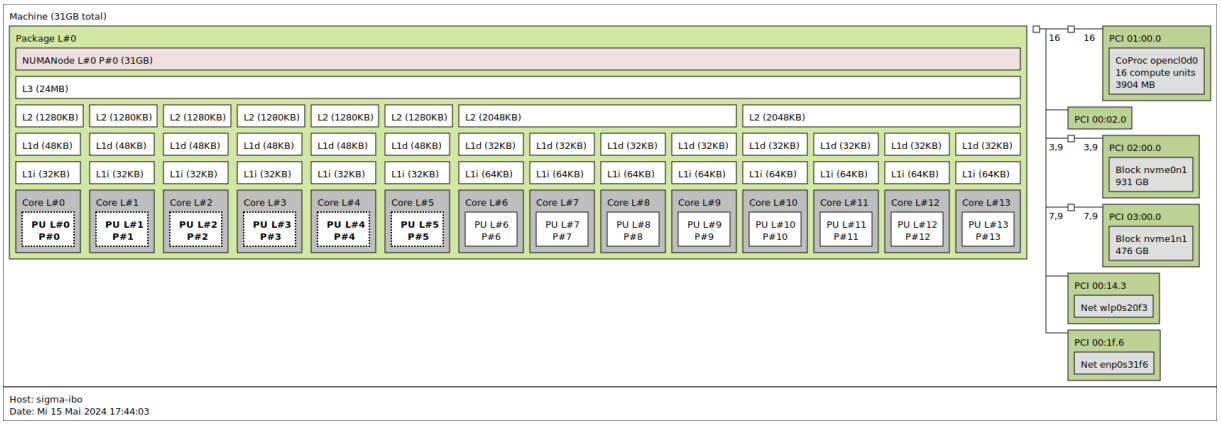

Figure 1: Hardware topology

3 Salamander 4

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

3.1 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 [4]. 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 [5] 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 [6], a solution for automation tasks that supports object-oriented programming and design in compliance with IEC 61131-3.

The interfaces between the individual modules are indicated by an arrow in Figure 2.

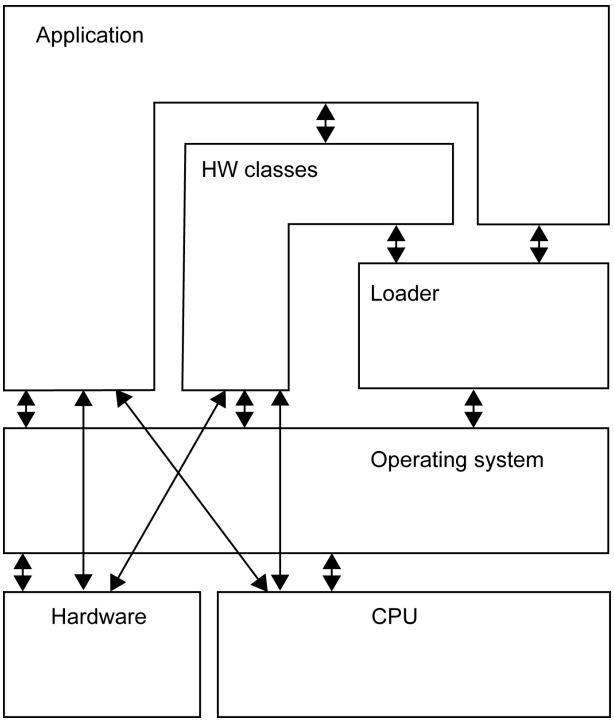

Figure 2: Structure of Salamander 4 CPU

3.2 Memory Management

For the sake of completeness, Figure 3 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.

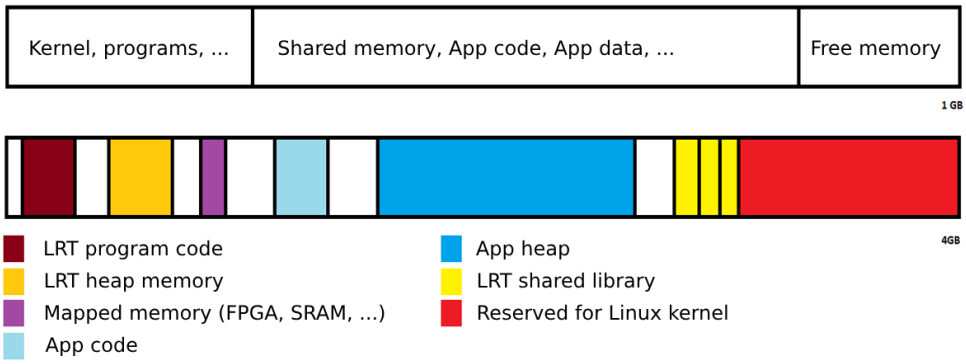

Figure 3: Memory Management

3.3 Xenomai

Xenomai 3 [4] is a 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.

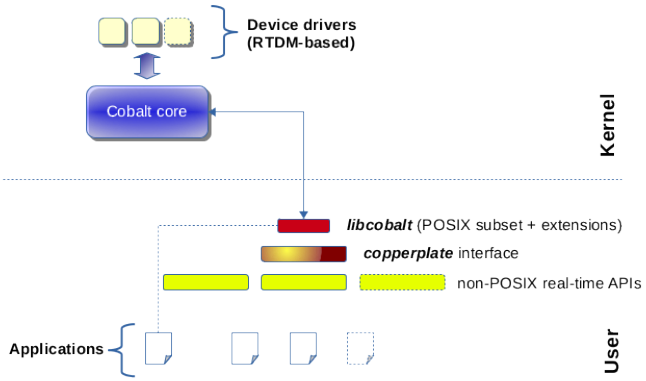

Figure 4: Xenomai Cobalt interfaces

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.

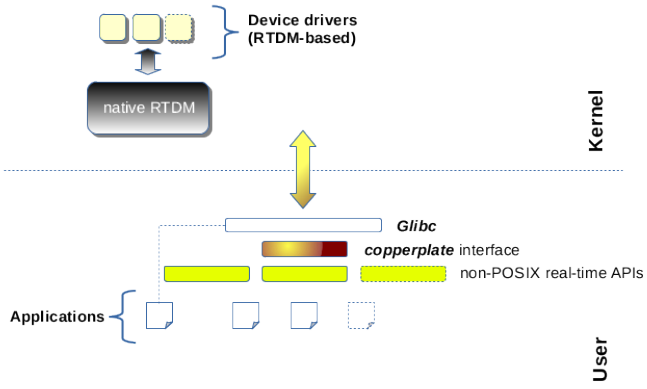

Figure 5: Xenomai Mercury interfaces

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

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 6 shows the gathered latency values in microseconds.

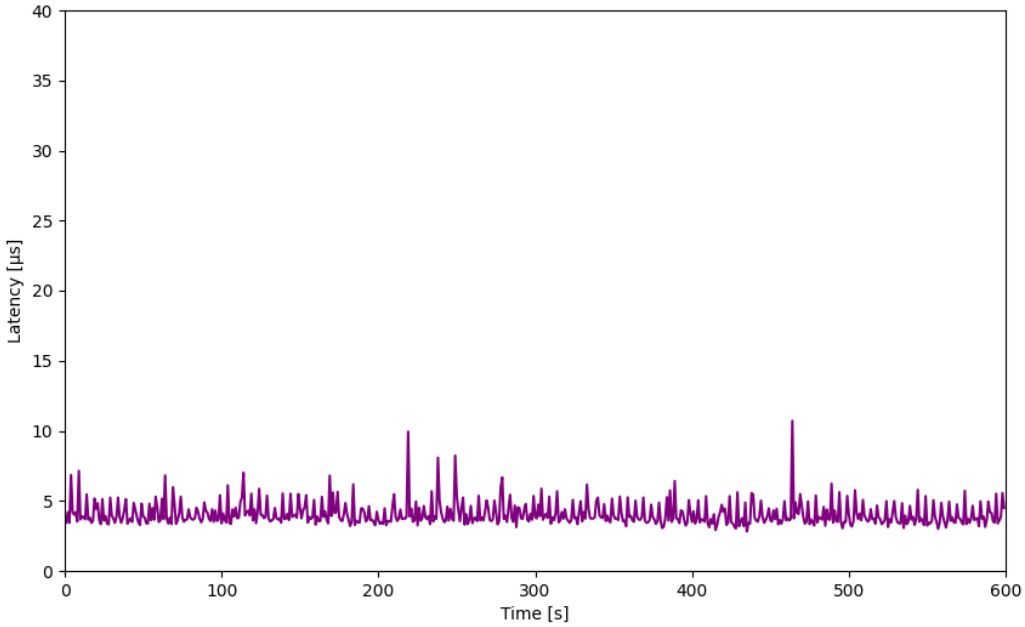

Figure 6: Latency of Salamander 4 bare metal

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

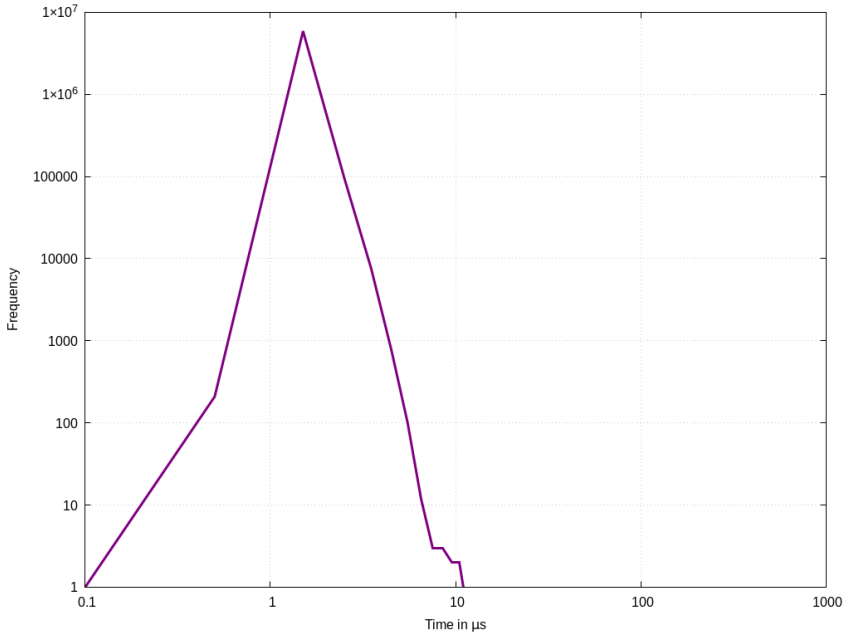

Figure 7: Variation in latency of Salamander 4 bare metal

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 [7]. 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 [8].

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

Code 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 Salamamder 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 inital 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 Salamamder 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 2. The kernel version and other details are presented in Code 5, using the `uname -a` command.

```
root@sigmatek-core2:~# uname -a
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 8 were collected.

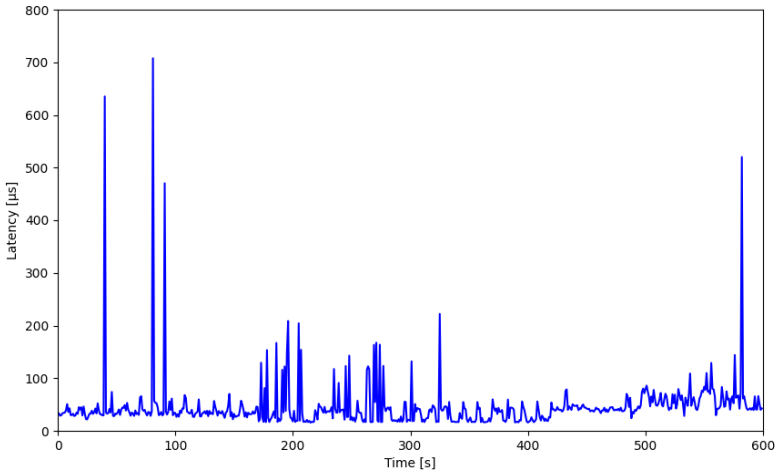

Figure 8: Latency with default settings

The statistics obtained from this measurement are provided in Table 3. 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 9 depicts the variation in latency over the course of said time with default settings.

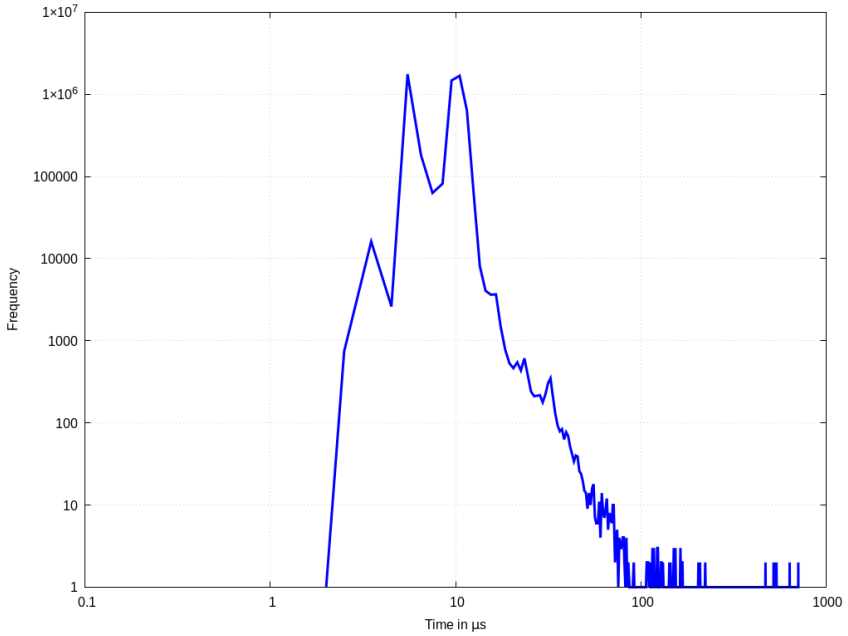

Figure 9: Variation in latency with default settings

Comparing these values to those of bare metal in Figure 10, 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.

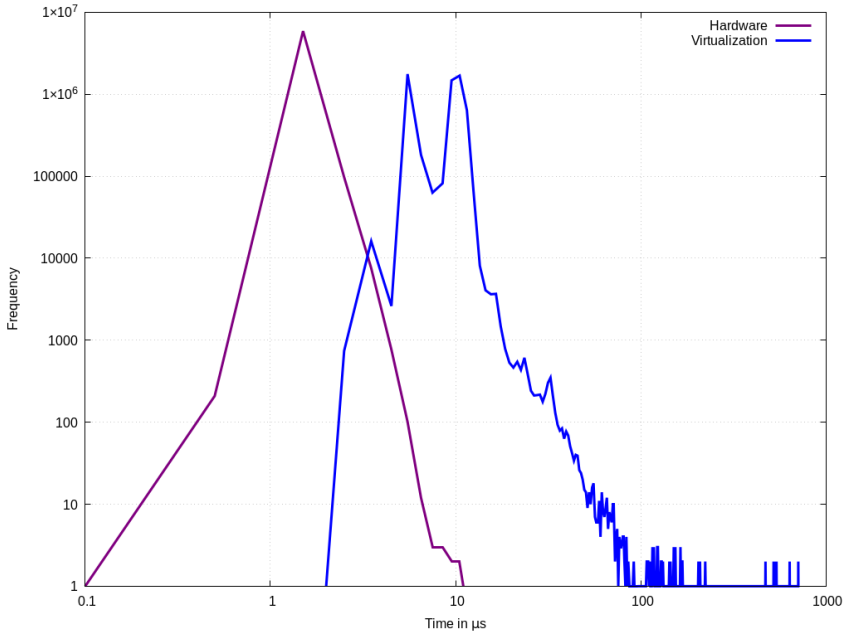

Figure 10: Comparison of variation in latency between hardware and virtualization

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.

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

Table 4: 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. Some of them are essential for real-time performance.

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

Code 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.

- **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 leads 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 kvm-clock 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 5.3.2 dives deeper into Interrupt Request affinity.

5.3 Host OS Configurations

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 [9], including systemd services. To prevent systemd services from running on an isolated CPU, the CPU affinity can be set the 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. 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 running on CPU 4. 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>` with the PID obtained from the previous command. Output 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 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 -c 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. 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.

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

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

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

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

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

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

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

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

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

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

Code 8: Check distribution of interrupt requests across each CPU

```
#!/bin/bash

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

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

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

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

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

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

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

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

Code 9: Change IRQ assignment of a CPU

5.3.3 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.

5.3.4 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 alltogether 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.

5.3.5 RT-priority

5.3.6 No unexpected RT processes

It is important that there are no other unexpected real time processes running on the system. A high priority RT process could occasionally be created by the host operating system at boot, hence disabling the offending RT process is advised if it is not needed. The process of disabling these RT processes is usually described in the documentation for the chosen Linux distribution.

5.3.7 Set CPU Affinity of IRQ thread to CPU 0

5.3.8 Set Device Driver Work Queue to CPU 0

5.3.9 Stop Certain Services

5.3.10 Disable Machine Check

There are other configurations that may be relevant depending on your use case, some of which are documented in [this talk](https://www.youtube.com/watch?v=NrjXEaTSyrw) and [this other talk](https://www.youtube.com/watch?v=w3yT8zJe0Uw). Additionally, quality-of-life configurations, such the variables in /etc/security/limits.conf, may need to be tuned as well.

“korrekt”

“This LaTeX.”

5.4 QEMU-KVM Configurations

5.4.1 Tune lapic timer advance

5.4.2 Set QEMU options for real-time VM

5.4.3 Set CPU affinity and scheduling policy of QEMU CPU threads

5.4.4 Passthrough PCI devices into the VM

5.5 Guest OS Configurations

6 KVM exit reasons

6.1 APIC_WRITE

6.2 HLT

6.3 EPT_MISCONFIG

6.4 PREEMPTION_TIMER

6.5 EXTERNAL_INTERRUPT

6.6 IO_INSTRUCTION

6.7 EOI_INDUCED

6.8 EPT_VIOLATION

6.9 PAUSE_INSTRUCTION

6.10 CPUID

6.11 MSR_READ

7 Real-Time Robotic Application

7.1 VARAN

7.2 Robotic Application

8 Results

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30]

9 Discussion

10 Summary and Outlook

Bibliography

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

[2] *Trace-Cmd*. https://trace-cmd.org/. (Visited on 03/25/2024).

[3] *KernelShark*. https://kernelshark.org/. (Visited on 03/25/2024).

[4] *Xenomai :: Xenomai*. https://xenomai.org/. (Visited on 03/21/2024).

[5] *CPU-Einheiten - SIGMATEK*. https://www.sigmatek-automation.com/de/produkte/steuerungssysteme/cpu-einheiten/cp-841/. (Visited on 05/27/2024).

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

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

[8] *QEMU*. https://www.qemu.org/. (Visited on 03/27/2024).

[9] Ruhui Ma et al. “Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System”. In: ().

[10] Chan-Hsiang Lin and Che-Kang Wu. “Performance Evaluation of Xenomai 3”. In: ().

[11] 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. (Visited on 04/18/2024).

[12] Marcello Cinque et al. “Virtualizing Mixed-Criticality Systems: A Survey on Industrial Trends and Issues”. In: *Future Generation Computer Systems* 129 (Apr. 2022), pp. 315–330. ISSN: 0167739X. DOI: 10.1016/j.future.2021.12.002. arXiv: 2112.06875 [cs]. (Visited on 03/25/2024).

[13] 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. (Visited on 03/25/2024).

[14] 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. (Visited on 03/25/2024).

[15] “Hard Real Time Linux* Using Xenomai* on Intel® Multi-Core Processors”. In: ().

[16] Diogenes Javier Perez et al. “How Real (Time) Are Virtual PLCs?” In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. Stuttgart, Germany: IEEE, Sept. 2022, pp. 1–8. ISBN: 978-1-66549-996-5. DOI: 10.1109/ETFA52439.2022.9921545. (Visited on 03/25/2024).

[17] 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. (Visited on 03/25/2024).

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

[19] 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.

[20] 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. (Visited on 03/25/2024).

[21] 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. (Visited on 03/25/2024).

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

[23] 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. (Visited on 05/06/2024).

[24] 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. (Visited on 03/25/2024).

[25] 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. (Visited on 03/25/2024).

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

[27] 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. (Visited on 05/08/2024).

[28] 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. (Visited on 05/16/2024).

[29] Daniel Bristot de Oliveira et al. “Demystifying the Real-Time Linux Scheduling Latency”. In: ().

[30] “Real-Time Performance Tuning Best Practice Guidelines for KVM-Based Virtual Machines”. In: (2022).

List of Figures

List of Tables

List of Code

List of Abbreviations

CPU Central Processing Unit

QEMU Quick Emulator

IRQ Interrupt Request

A Anhang A

B Anhang B