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, May 27, 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, May 27, 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

Abstract

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. Sections 4.1 and 4.1 demonstrate the inital real-time latency values gathered for bare metal and virtualisation.

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].

Table 1: Testbed Configuration

CPU	i7-8665U quad core, 1.9 GHz
Memory	2x SO-DIMM DDR4-2400/2133 MHZ, 64GB
GPU	Integrated GPU
BIOS	Version WL37R107
OS	Ubuntu 18.04

3 Salamander 4

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

```
root@sigmatek-core2:~# uname -a
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 1.

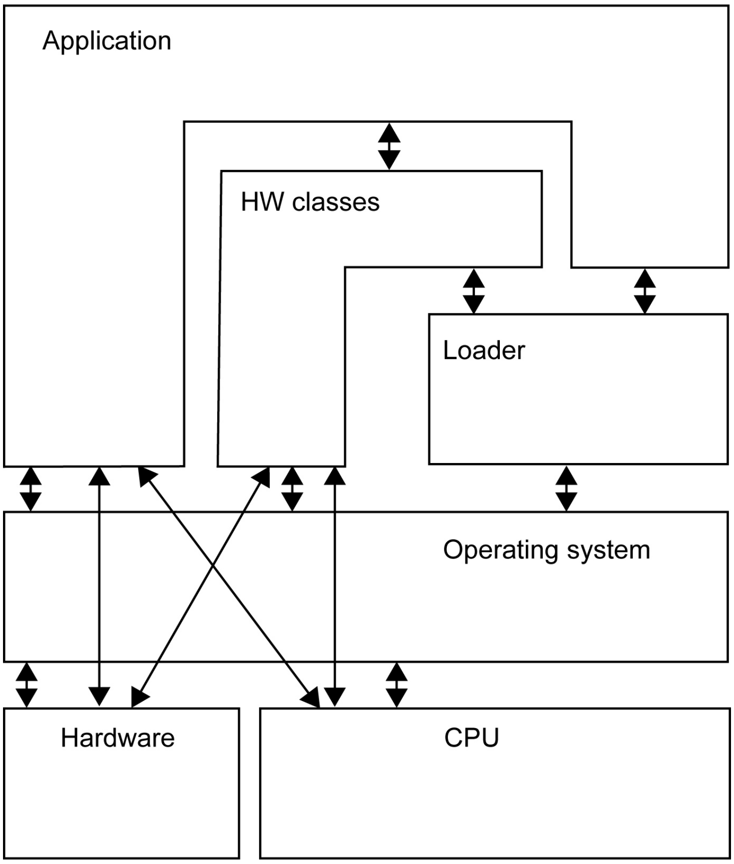

Figure 1: Structure of Salamander 4 CPU

For the sake of completeness, figure 2 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.

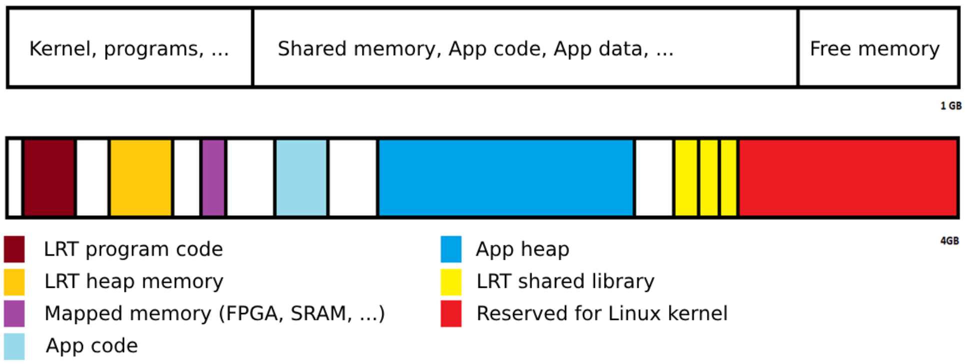

Figure 2: Memory Management

3.1 Xenomai

Xenomai 3 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 3. 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.

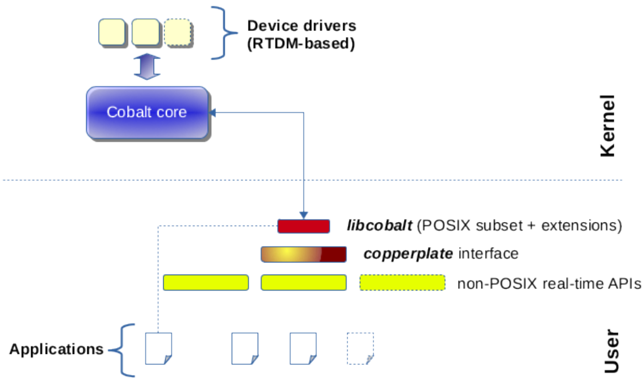

Figure 3: Xenomai Cobalt interfaces

The second approach, called Mercury and shown in figure 4, 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.

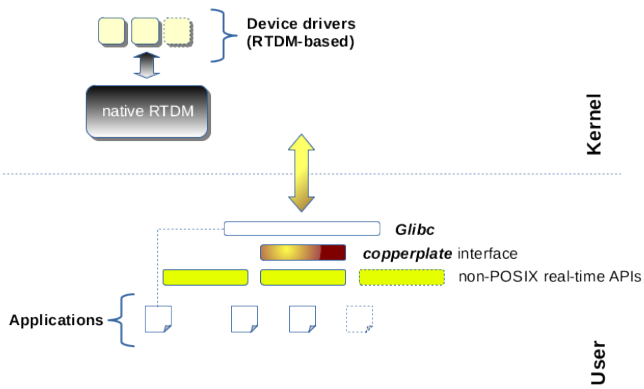

Figure 4: Xenomai Mercury interfaces

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

4.1 Salamander 4 Bare Metal

Salamander 4 Bare Metal refers to the proprietary hardware of SIGMATEK used to employ the custom operating system. The output of the command `uname -a` can be observed in code 2.

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

Code 2: System information

Figure 5 shows latency of hardware Salamander4.

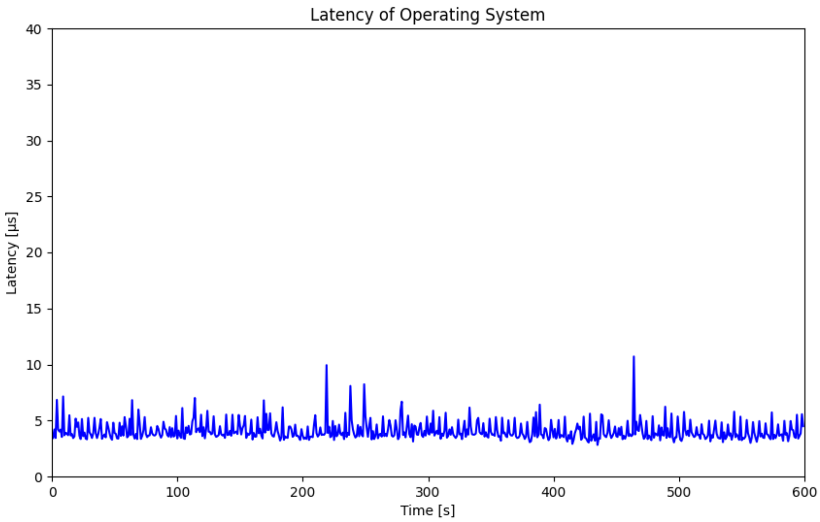

Figure 5: Latency hardware

4.2 Salamander 4 Virtualisation

In addition to providing Salamander 4 on its own hardware, SIGMATEK has also developed a virtualised version of this operating system. It was developped using Yocto, an open source project that allows customised Linux distributions to be created for embedded systems [7]. The virtualisation runs in a QEMU environment, which is an open source tool for hardware virtualisation [8]. 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. Upon generating the necessary files, Yocto generates a QEMU folder with the following components shown in code 3.

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

Code 3: Contents of QEMU folder for Salamander 4

```
#!/bin/sh

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

exec  qemu-system-x86_64 -M pc,accel=kvm -kernel ./bzImage \
-m 2048 -drive file=salamander-image-sigmatek-core2.ext4,format=raw,media=disk \
-append "console=ttyS0 console=tty1 root=/dev/sda rw panic=1 sigmatek_lrt.QEMU=1 ip=dhcp rootfstype=ext4 schedstats=enable" \
-net nic,model=e1000,netdev=e1000 -netdev bridge,id=e1000,br=nm-bridge \
-fsdev local,security_model=none,id=fsdev0,path=drive-c -device virtio-9p-pci,id=fs0,fsdev=fsdev0,mount_tag=/mnt/drive-C \
-device vhost-vsock-pci,guest-cid=3,id=vsock0 \
-drive if=pflash,format=qcow2,file=ovmf.code.qcow2 \
-no-reboot -nographic
```

Code 4: QEMU script for starting Salamander 4 virtualisation

Here is a description of the used components:

- **bzImage**: Compressed Linux kernel image, loaded by QEMU at system start.
- **ovmf.code.qcow2**: Firmware file for QEMU, enables UEFI boot process.
- **qemu_def.sh**: Shell script, starts QEMU with correct parameters to boot Salamamder 4 OS.
- **stek-drive-c-image-sigmatek-core2.tar.gz**: Archive containing files for C drive, unpacked and copied to drive-c/ directory by qemu_def.sh script.
- **drive-c**: Directory serving as C drive for QEMU system, created and filled by qemu_def.sh script.
- **salamander-image-sigmatek-core2.ext4**: Root file system for Salamander 4 OS, used as hard drive for QEMU system.
- **vmlinux**: Uncompressed Linux kernel image, typically used for debugging, contains debugging symbols not present in bzImage.

When the script is started from the host, the QEMU process can be scheduled to run on any available core, as it is noted 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 user level. 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.

Figure 6 shows latency of QEMU default Salamander4.

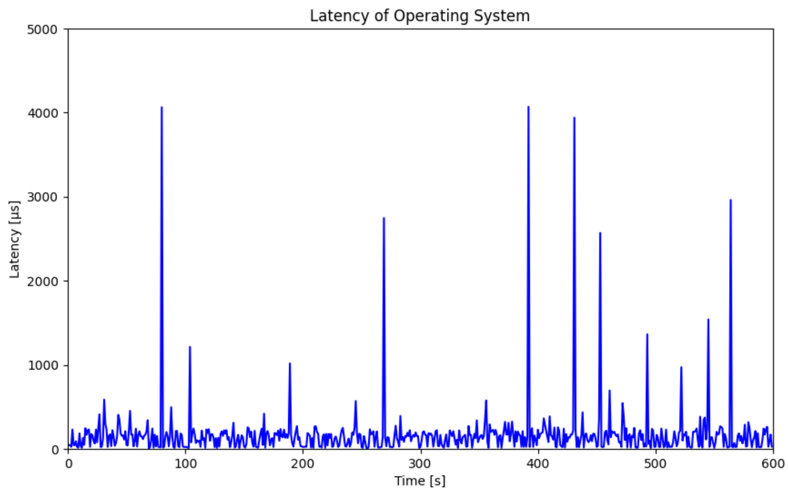

Figure 6: Latency no taskset

Figure 7 shows latency of QEMU taskset Salamander4.

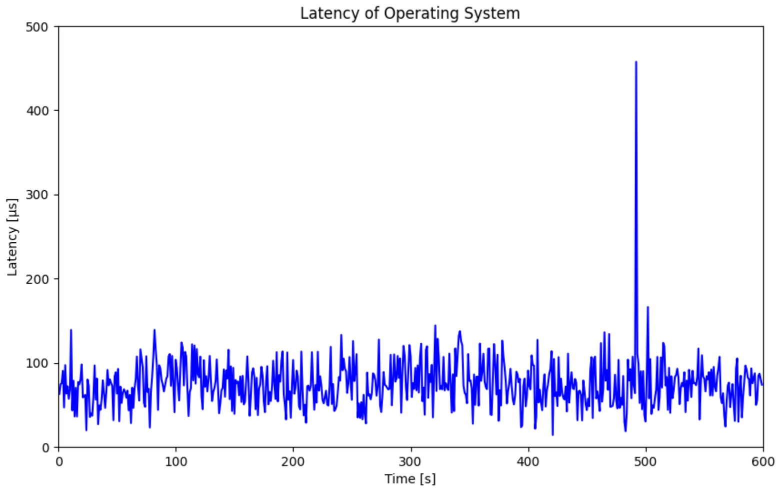

Figure 7: Latency taskset

Figure 8 shows latency of QEMU vapic Salamander4.

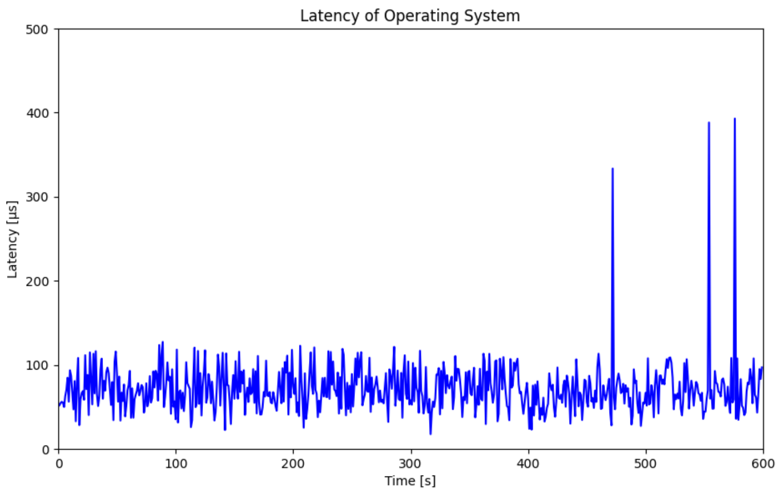

Figure 8: Latency vapic

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 at the user level and does not affect kernel tasks. Consequently, these 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 guestś 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 9 were obtained. Additionally, table 2 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 2: Description of kvm_exit reasons

Figure 9 shows `kvm_exit` frequency with CPU islation.

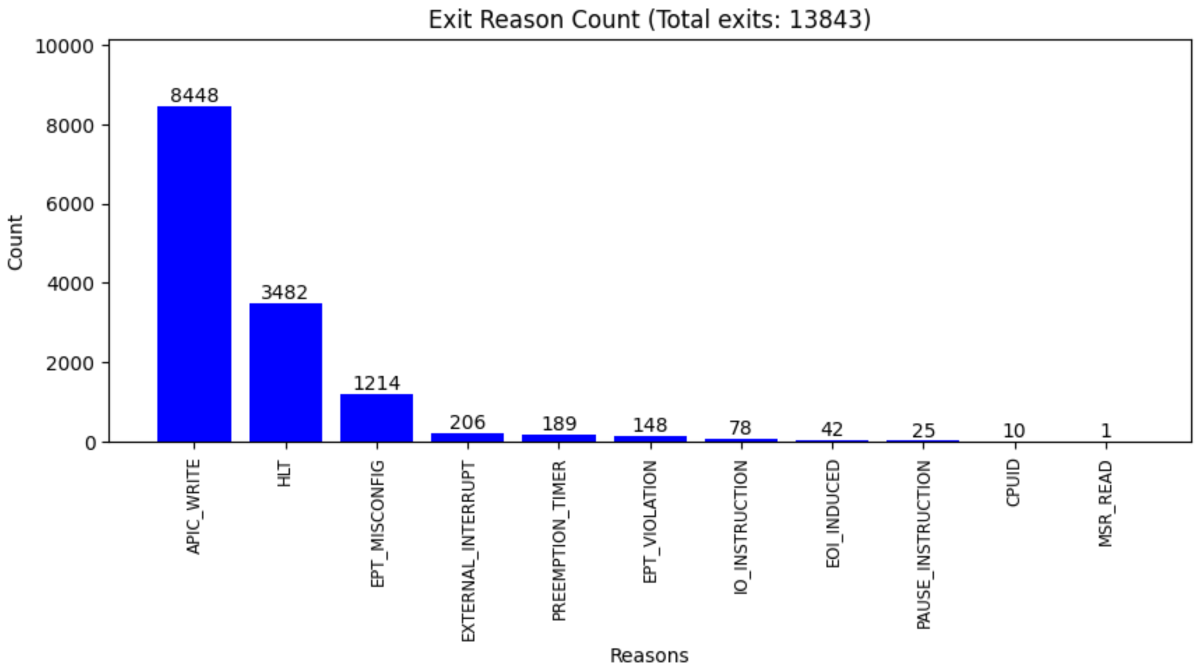

Figure 9: kvm exits

Figure 10 shows `kvm_exit` frequency without CPU islation.

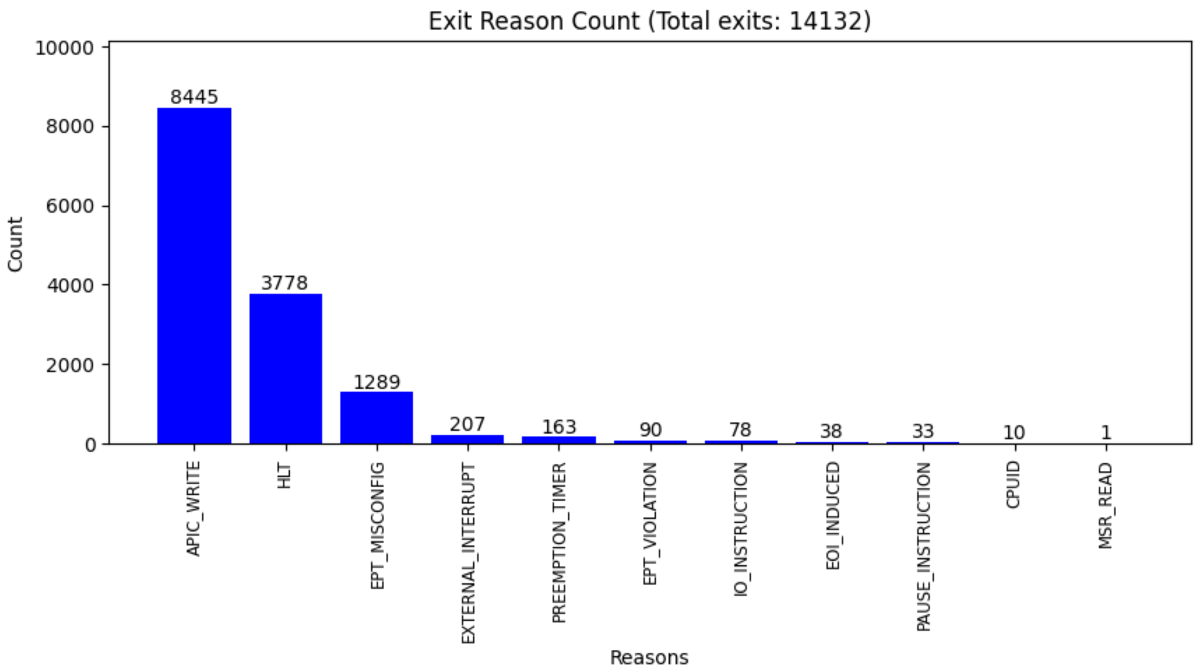

Figure 10: kvm exits default

Table 3: Host report CPU19 (Total of 445.908)

PID	Task	Count
182579	qemu-system-x86	302.748
0	<idle>	112.911
182618	vhost	21.204
182572	qemu-system-x86	7.597
182755	qemu-system-x86	644
182754	qemu-system-x86	643
181870	kworker/19:1	139
3820	kworker/19:1H	16
94	migration/19	6

Table 4: Guest report (Total of 362.370)

PID	Task	Count
0	<idle>	150.744
331	LRT-Main	56.697
377	trace-cmd	48.507
346	CLI	26.426
378	kthreadd	25.837
340	MainTaskLow	19.291
339	<...>	9.980
34	MainTaskHigh	9.185
327	LE-Logger	4.965
369	kworker/0:0	2.793
321	kWorker-LRT	2.542
328	LRTMgr-Main	1.651
34	LrtMgrCyclic	1.220
332	cobalt_printf	1.112
325	LE-System	534
343	TCP-Listen	187
15	rcu_preempt	162
25	kcompactd0	122
58	kworker/0:1H	96
63	kworker/u2:2	89
8	jbd2/sda-8	86
22	kworker/0:1	56
1	init	31
2	kthreadd	25
375	trace-cmd	24
14	ksoftirqd/0	8

In the following, the host and guest tasks along with their impact on system latency are briefly described.

- **qemu-system-x86**: Part of the QEMU process and specifically, this task emulates x86 systems. In table 3, it occurs four times under different PIDs, hence there are four threads of it.

- **<idle>**: This represents the idle time of the CPU, hence it is not being used by any process, allowing to save power. The system halts until the next interrupt, which could be a timer interrupt, I/O interrupt, etc.

- **vhost**: A kernel module which improves virtual input/output (virtio) performance by handling virtqueues in the kernel, thereby reducing context switches and system calls.

- **kworker/19:1**: A kernel worker thread created by the Linux kernel, kworker/19:1 performs work in response to system events. The number after the slash and colon indicate the CPU core and internal ID of the worker thread, respectively.

- **kworker/19:1H**: Similar to kworker/19:1, kworker/19:1H is a kernel worker thread, with the 'H' suggesting that this thread handles hardware interrupts.

- **migration/19**: The migration process is a kernel process that balances load across CPU cores by moving threads from one CPU to another. The number after the slash indicates the CPU core to which the migration process is bound. (URL: https://elixir.bootlin.com/linux/latest/source/kernel/sched/core.c#L2325)

4.2.1 Generic Ubuntu

4.2.2 Real-Time Ubuntu

After analyzing the inital latency of both versions, Trace-cmd and Kernelshark were used to further inspect the reasons that caused this divergence.

4.3 Latency Comparison

In the initial phase, a comparative latency analysis was conducted between the hardware version and the virtualized version of Salamander 4. For this purpose, the latency tool of the Xenomai test suite was used. The latency was measured under two conditions, idle and CPU-stressed. The goal was to optimize the latency of the virtualisation of Salamander 4 OS to closely match that of the bare metal version.

Vorgehensweise von [9]

4.4 KVM exit reasons

4.4.1 APIC_WRITE

The Advanced Programmable Interrupt Controller (APIC) is responsible for the distribution of interrupts in x86 and Itanium-based computer systems. An APIC_WRITE occurs when a guest operating system attempts to write to the APIC registers. Since the APIC is a physical hardware component, KVM must intercept this operation and cause a VM exit. To avoid this, newer Intel processors offer hardware virtualization of the Advanced Programmable Interrupt Controller (APICv). APICv improves virtualized AMD64 and Intel 64 guest performance by allowing the guest to directly access the APIC, dramatically cutting down interrupt latencies and the number of virtual machine exits caused by the APIC. This feature is used by default in newer Intel processors and improves I/O performance.

This can be done by setting the apic flag to 'v' in the VM configuration file.

Figure 11 shows `kvm_exit` frequency with APIC virtualisation. Comparing this to the previous Figures 9 and 10, it can be observed that an APIC_Write no longer occurs.

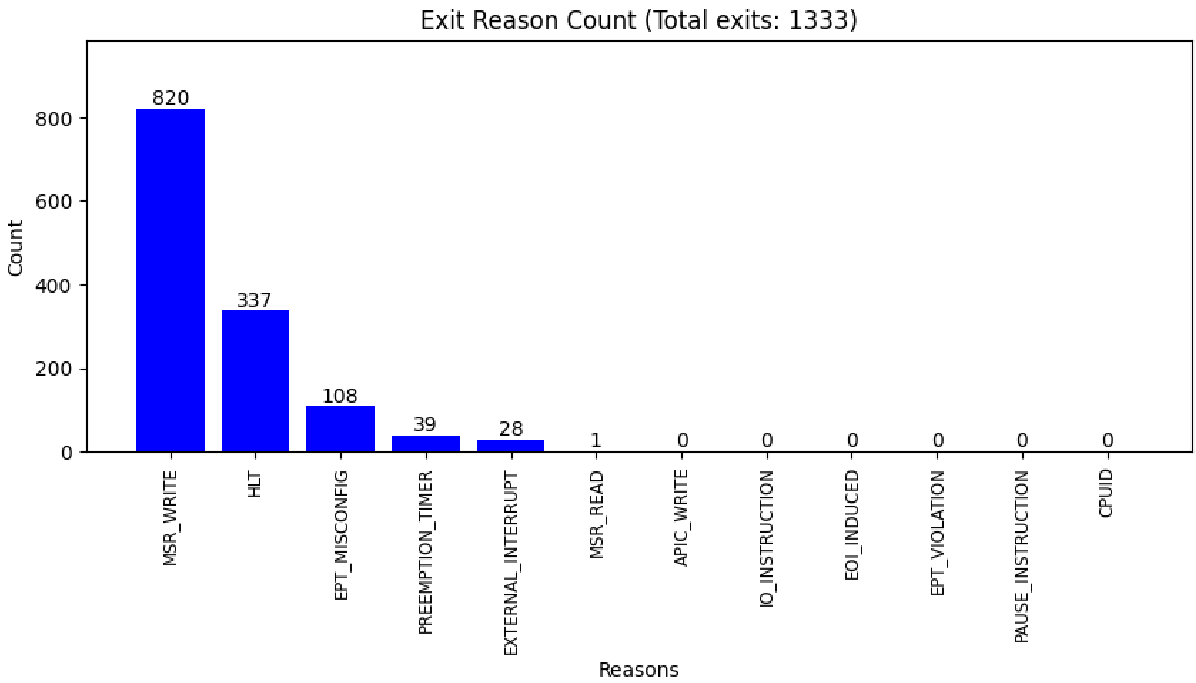

Figure 11: kvm exits default

4.4.2 HLT

4.4.3 EPT_MISCONFIG

4.4.4 PREEMPTION_TIMER

4.4.5 EXTERNAL_INTERRUPT

4.4.6 IO_INSTRUCTION

4.4.7 EOI_INDUCED

4.4.8 EPT_VIOLATION

4.4.9 PAUSE_INSTRUCTION

4.4.10 CPUID

4.4.11 MSR_READ

5 Real-Time Performance Tuning

5.1 BIOS Configurations

5.2 Kernel Configurations

5.3 Host OS Configurations

5.3.1 Tasks and Events

5.3.2 CPU governor

Figure 12 shows ...

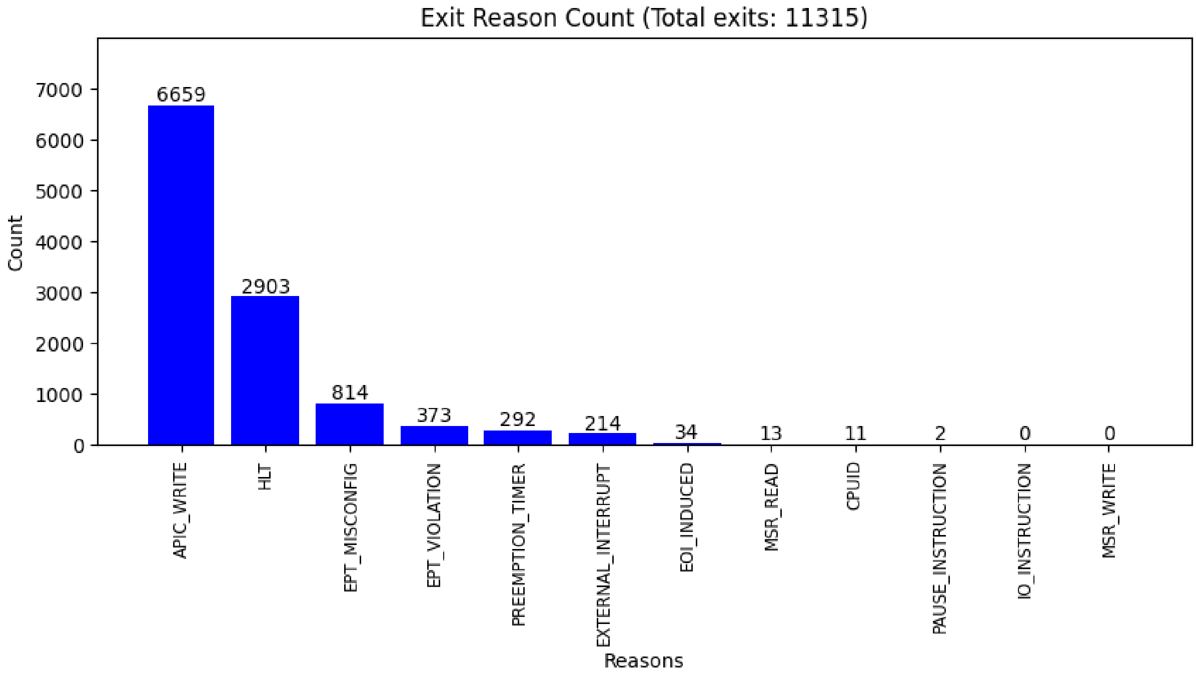

Figure 12: power_saver kvm_exit_count

Figure 13 shows ...

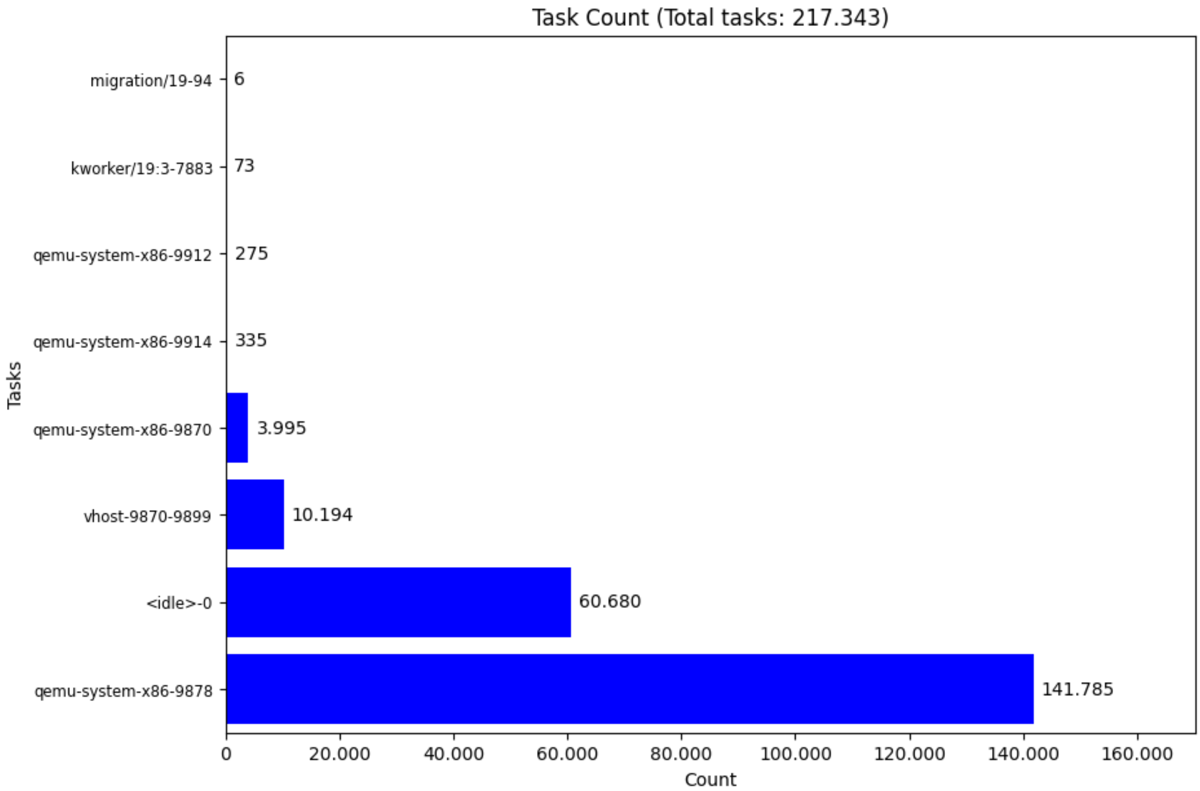

Figure 13: power_saver host report

Figure 14 shows ...

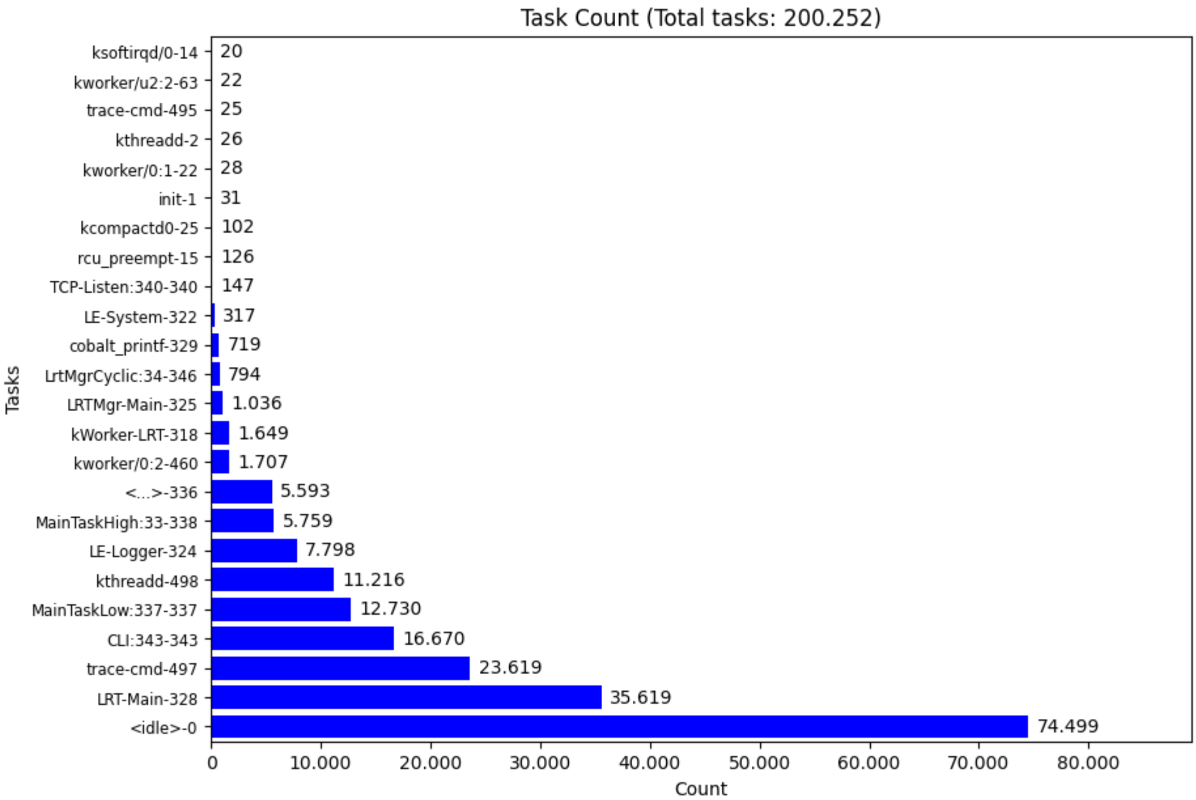

Figure 14: power_saver guest report

Figure 15 shows ...

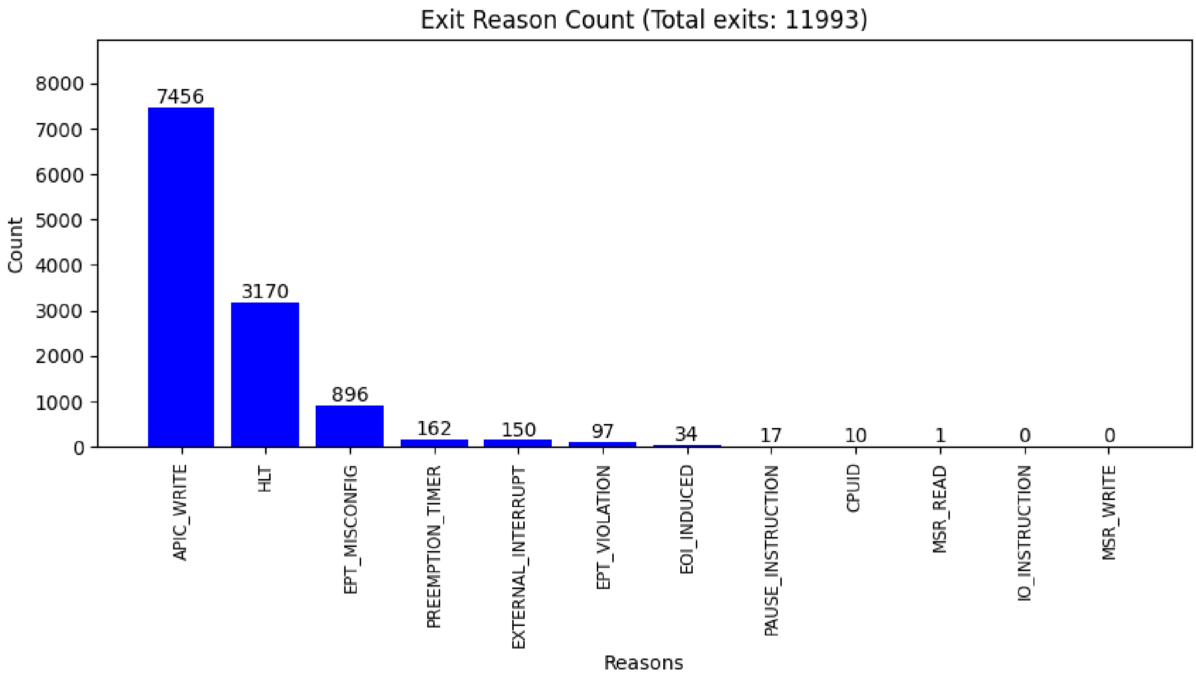

Figure 15: balanced kvm_exit_count

Figure 16 shows ...

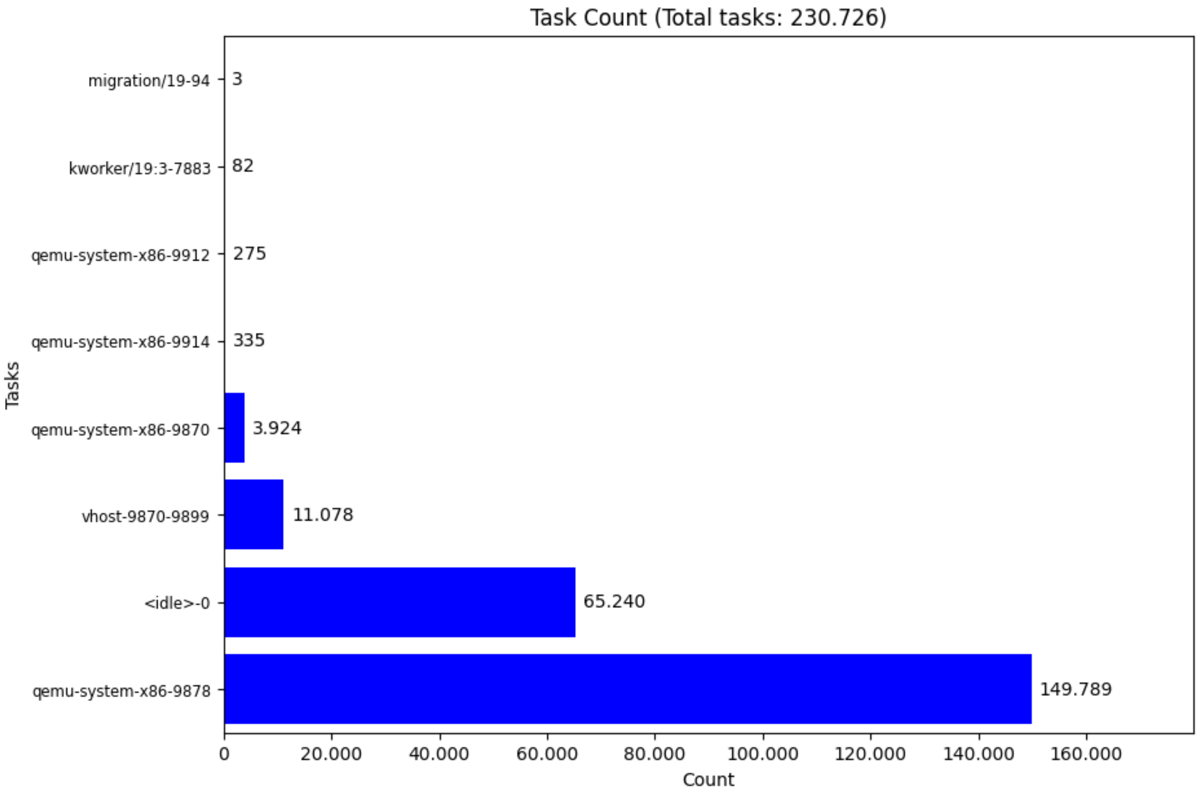

Figure 16: balanced host report

Figure 17 shows ...

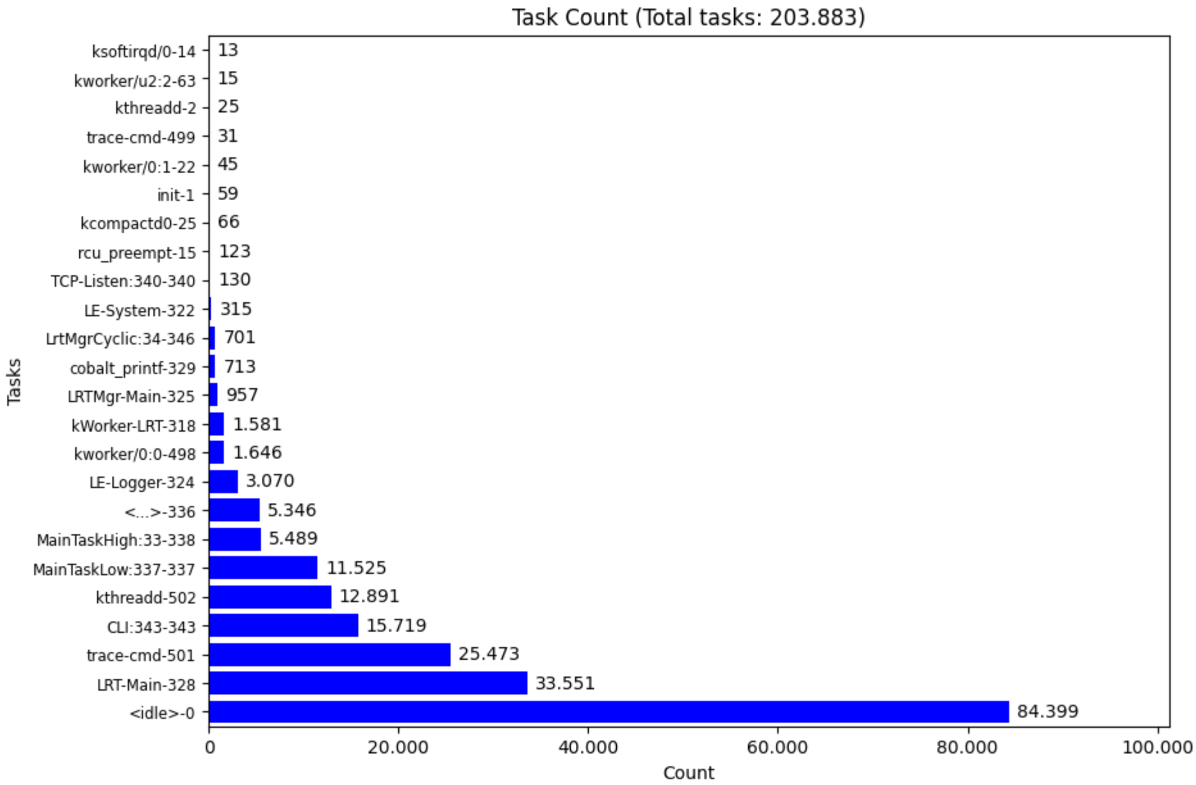

Figure 17: balanced guest report

Figure 18 shows ...

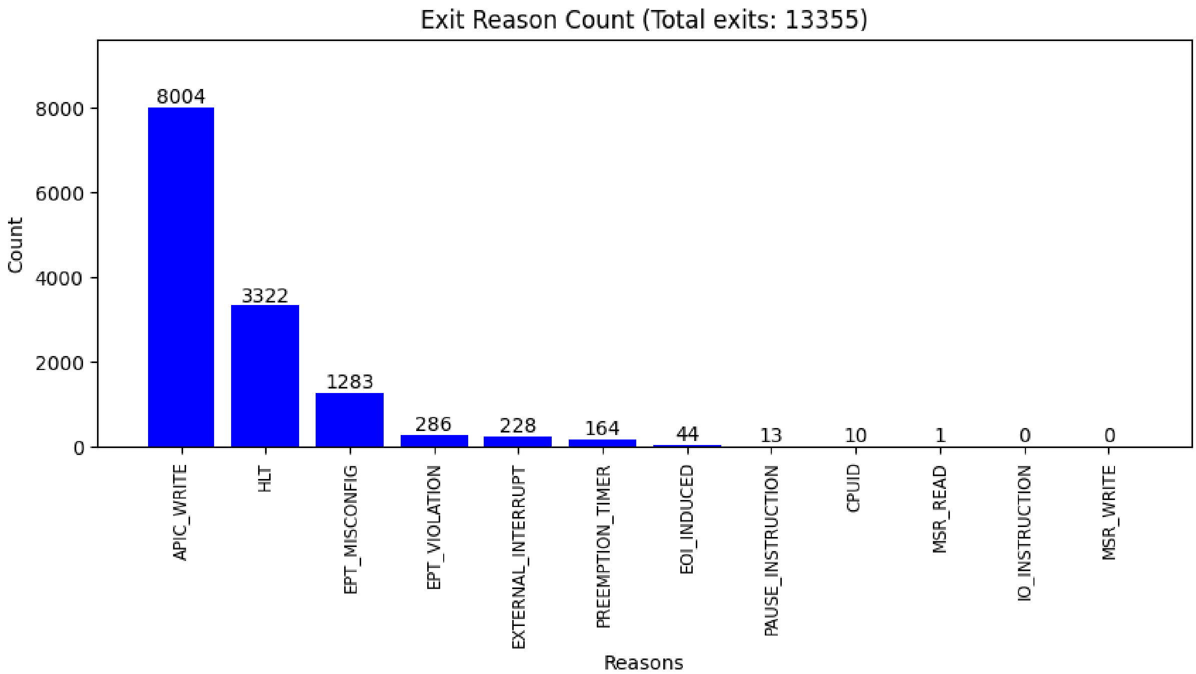

Figure 18: performance _exit_count

Figure 19 shows ...

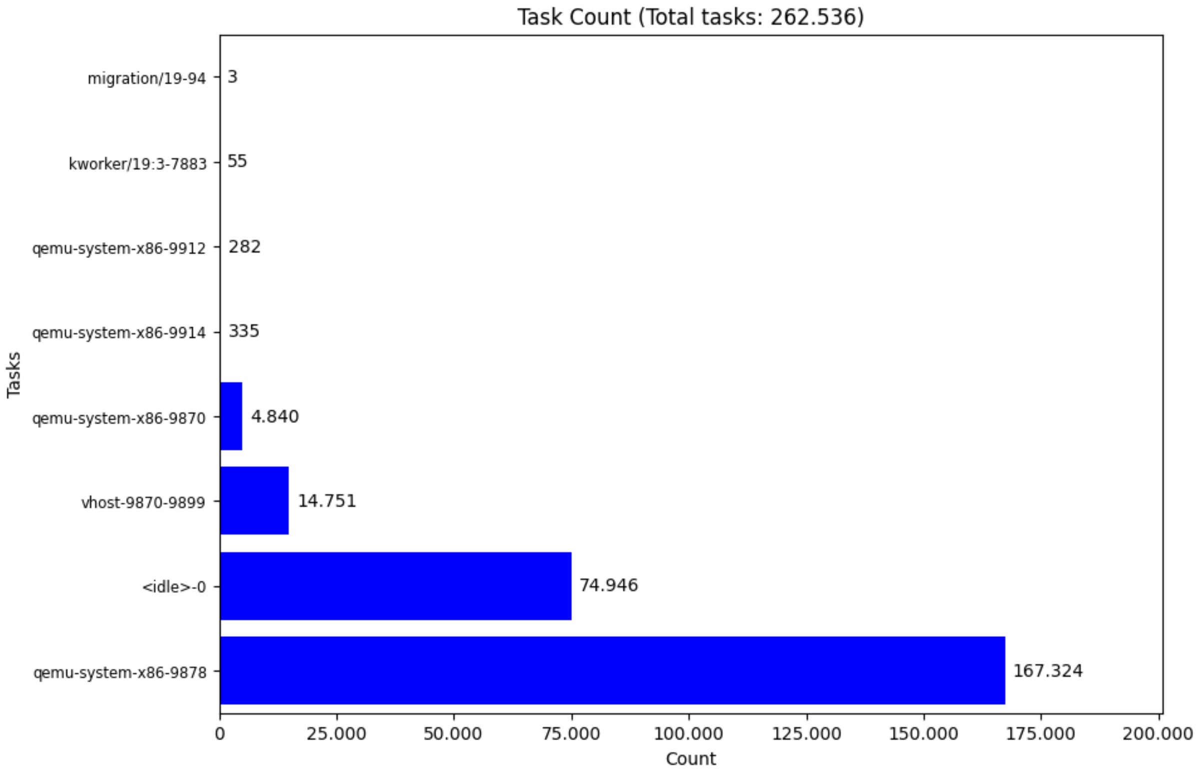

Figure 19: performance host report

Figure 20 shows ...

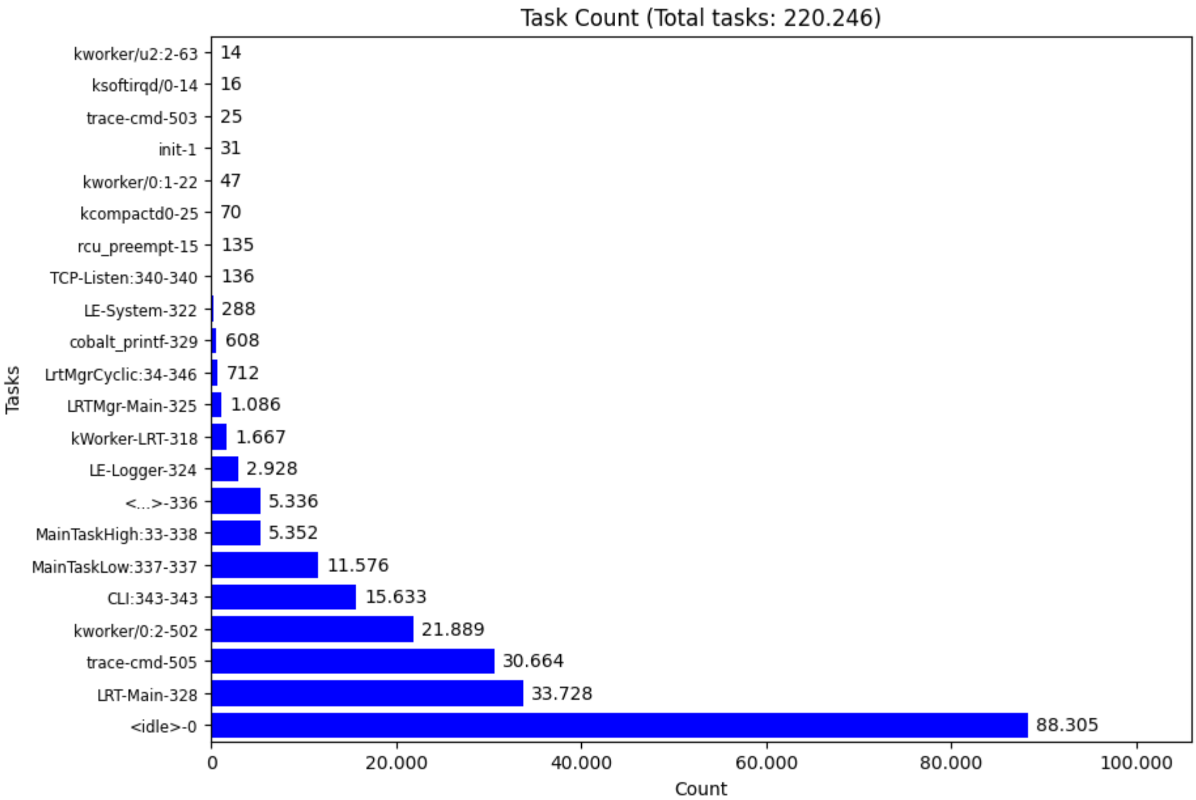

Figure 20: balanced guest report

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.

The gnuplot latency is visible in Figure 21

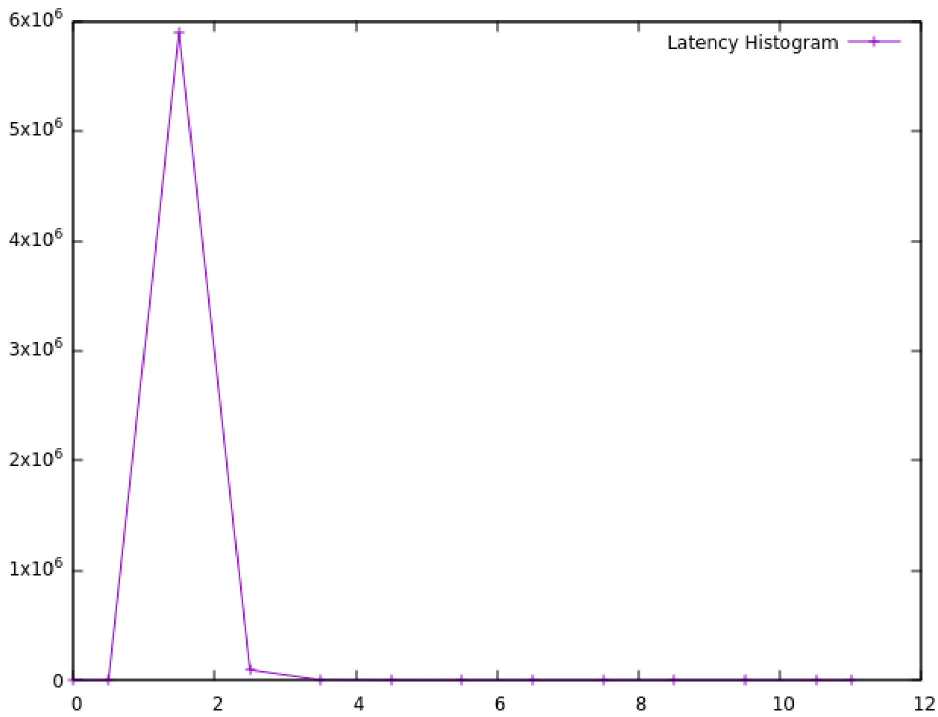

Figure 21: gnuplot latency hardware

Figure 22

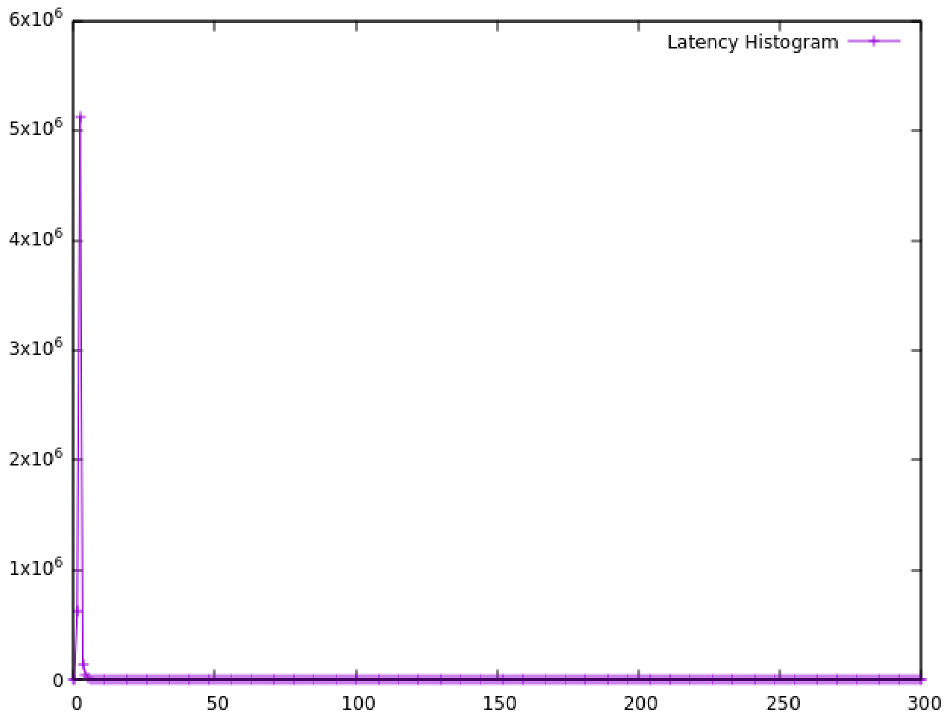

Figure 22: gnuplot latency no taskset

Figure 23

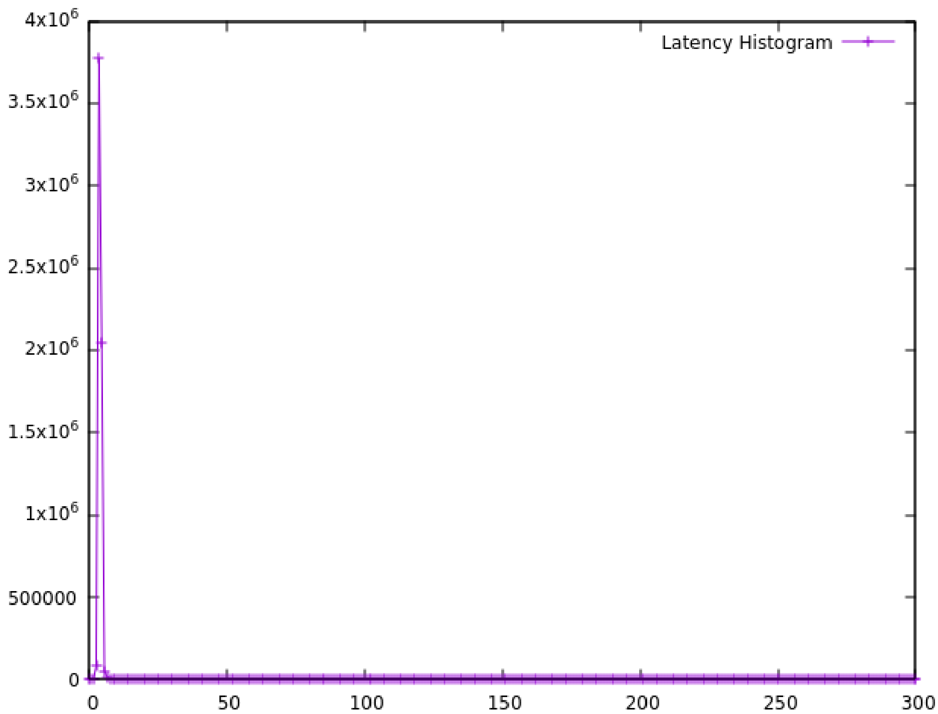

Figure 23: gnuplot latency with taskset

Figure 24

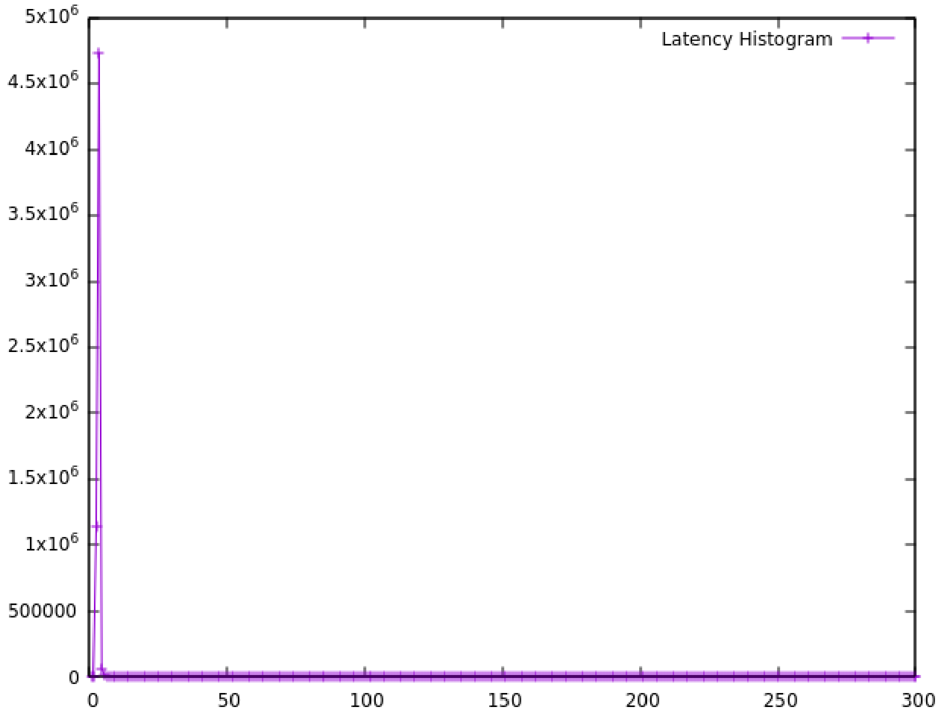

Figure 24: gnuplot latency with taskset

5.3.3 CPU 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. Also, modifying the `proc/irq/IRQ_NUMBER/smp_affinity` property of each Interrupt IRQ_NUMBER in the system is part of this process, as described later in section 5.3.4. Output 5 shows the user and kernel tasks that run on CPU 19. After the isolation, user tasks other than the QEMU process have been removed from running on this CPU. Only few critical kernel threads that are tied to this CPU still take CPU time.

```
sigma_ibo@sigma-ibo:~$ cat /sys/devices/system/cpu/isolated
19
sigma_ibo@sigma-ibo:~$ ps -e -o pid,psr,comm | awk '$2 == 19'
     92  19 cpuhp/19
     93  19 idle_inject/19
     94  19 migration/19
     95  19 ksoftirqd/19
     97  19 kworker/19:0H-events_highpri
   1025  19 irq/205-iwlwifi:queue_7
  17448  19 kworker/19:1H-kblockd
  17499  19 kworker/19:2-events
  18761  19 kworker/19:3-events
  21401  19 qemu-system-x86
```

Code 5: User and Kernel Tasks

5.3.4 Interrupt Requests Handling

Once the CPUs were isolated, Interrupt Requests Handling was the next step. Interrupt Requests are used to send a signal to the CPU, prompting it to 'interrupt' its current task and divert its attention to another task. This allows hardware devices to communicate with the CPU through frequent context switches, which can lead to performance degradation, especially in high-performance computing or real-time scenarios. To mitigate this, the IRQs needed to be removed from the isolated CPUs. This was done by manipulating a file in the proc filesystem, namely `/proc/irq/<IRQ>/smp_affinity`. The value in the `smp_affinity` file is a bitmask in hexadecimal format. Each bit in this mask corresponds to a CPU in the system. The least significant bit (LSB) on the right corresponds to the first CPU (CPU0), and the significance increases towards the left until CPU19. In a system with 20 CPUs, if every CPU was reserved for one IRQ, the value for `smp_affinity` would be FFFFF. The script in code 6 was written to check and log the distribution of Interrupt Requests across each CPU in Salamander 4.

```
#!/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=$1
# Initialize an empty array to store the IRQ numbers
IRQs=()
for IRQ in /proc/irq/*; do
   if [ -f "$IRQ/smp_affinity" ]; then
      # Read the current smp_affinity
      AFFINITY=$(cat "$IRQ/smp_affinity")
      # Check if the bit for the current CPU is set
      if (( (0x$AFFINITY & (1 << CPU)) != 0 )); then
         # Add the IRQ number to the array
         IRQs+=("${IRQ#/proc/irq/}")
      fi
   fi
done
# Sort the array
IFS=$'\n' sorted=($(sort -n <<<"${IRQs[*]}"))
# Print the CPU number
echo "CPU $CPU IRQ affinity:"
# Print the sorted IRQ numbers on separate lines
for irq in "${sorted[@]}"; do
   echo "$irq"
done
```

Code 6: Check distribution of Interrupt Requests across each CPU

Output 7 shows the output of the script above for CPU 19.

```
sigma_ibo@sigma-ibo:~$ ./check_smp_affinity.sh 19
CPU 19 IRQ affinity:
0
2
3
4
5
6
7
10
11
13
15
131
172
188
189
195
```

Code 7: Output of smp_affinity for CPU 19

By changing the values of the `smp_affinity` files of the respective IRQs, the assignment of IRQs was controlled so that they would not be handled by the isolated CPU. This reduced the interruptions caused by IRQs and the isolated CPUs were able to focus more on their assigned tasks.

5.3.5 Real-time patch

5.4 QEMU-KVM Configurations

5.5 Guest OS Configurations

5.5.1 Tasks and Events

```
#!/bin/sh

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

exec  qemu-system-x86_64 -M pc,accel=kvm -kernel ./bzImage \
-m 2048 -drive
    file=salamander-image-sigmatek-core2.ext4,format=raw,media=disk \
-append "console=ttyS0 console=tty1 root=/dev/sda rw panic=1
    sigmatek_lrt.QEMU=1 ip=dhcp rootfstype=ext4 schedstats=enable" \
-net nic,model=e1000,netdev=e1000 -netdev bridge,id=e1000,br=nm-bridge \
-fsdev local,security_model=none,id=fsdev0,path=drive-c -device
    virtio-9p-pci,id=fs0,fsdev=fsdev0,mount_tag=/mnt/drive-C \
-device vhost-vsock-pci,guest-cid=3,id=vsock0 \
-drive if=pflash,format=qcow2,file=ovmf.code.qcow2 \
-no-reboot -nographic
```

Code 8: QEMU script for starting Salamander 4 virtualisation

6 Real-Time Robotic Application

6.1 VARAN

6.2 Robotic Application

7 To Include

The Figure 25 below compares non-optimized guest latency with optimized guest latency and includes optimized bare-metal latency as a reference. The data shows that a 40% reduction in QD1 latency is achievable through system tuning.

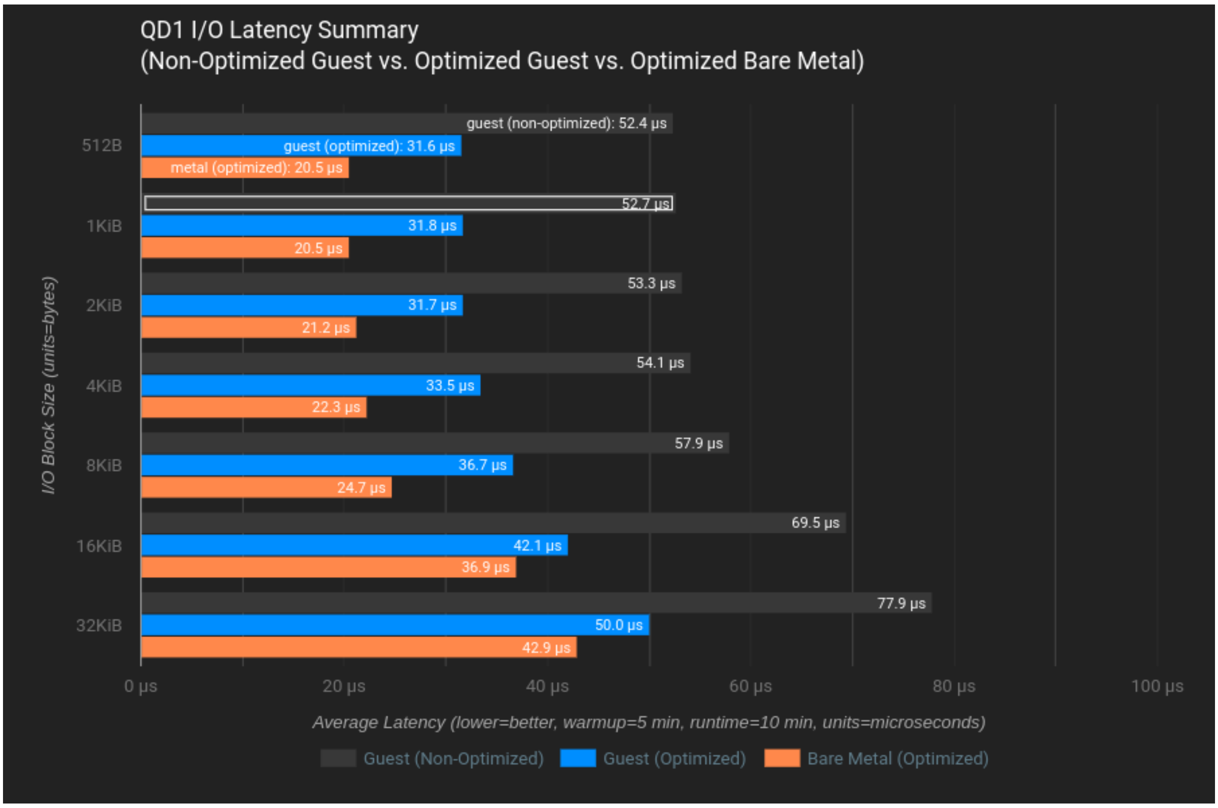

Figure 25: latency comparison

[1] [2] [3] [4] [7] [8] [9] [**abediRTNFPredictableLatency2019**] [10] [**casiniLatencyAnalysisVirtualization2021**] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26]

8 Results

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] Chan-Hsiang Lin and Che-Kang Wu. "Performance Evaluation of Xenomai 3". In: ().

[10] 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).

[11] 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).

[12] 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).

[13] 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).

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

[15] 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).

[16] 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).

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

[18] 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.

[19] 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).

[20] 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).

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

[22] 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).

[23] 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).

[24] 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).

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

[26] 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).

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