University of
South-Eastern Norway

Faculty of Technology, Natural Sciences, and Maritime Sciences

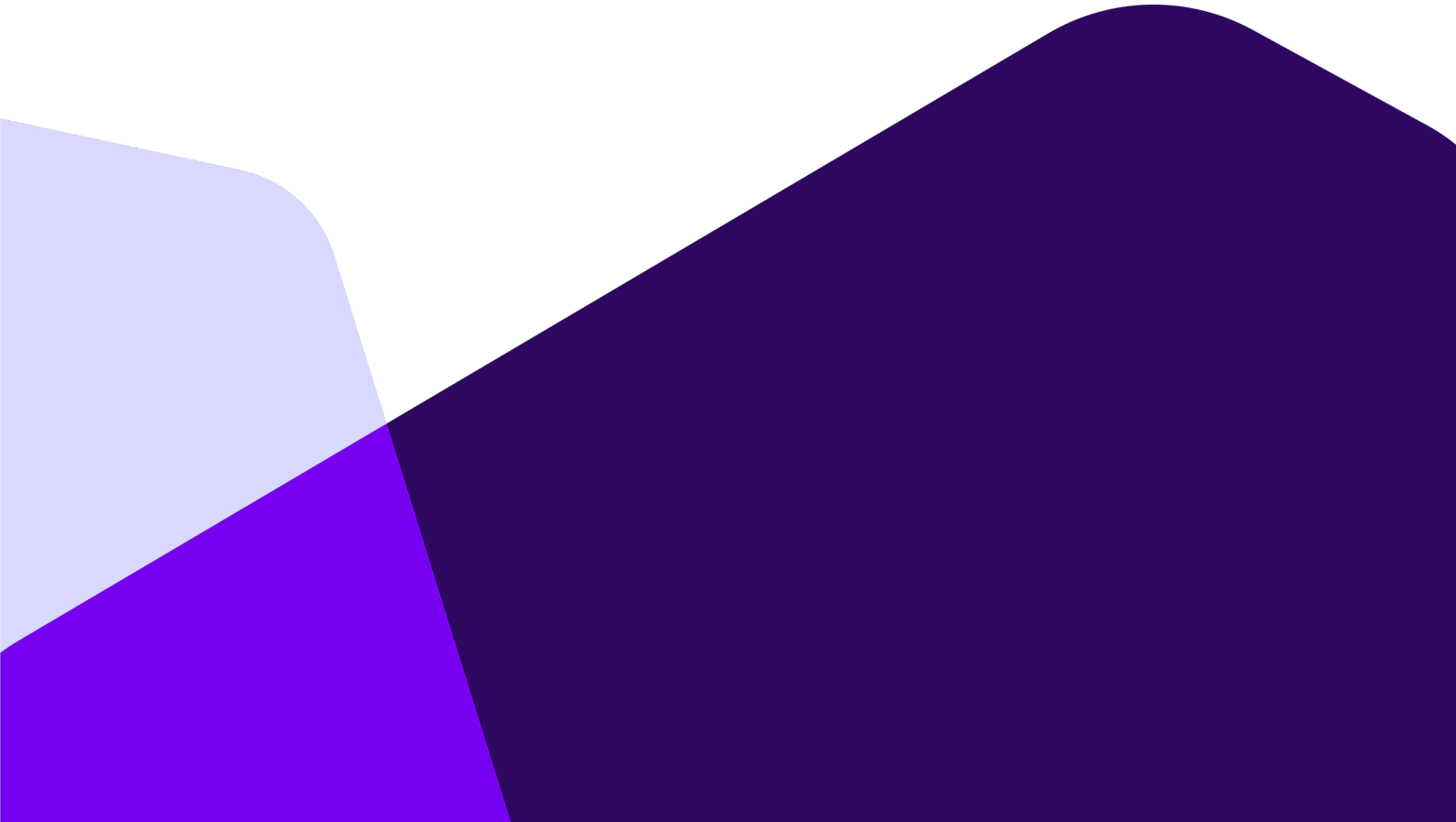
CS5000 - Master's Thesis

Study programme: MACS

Spring 2024

Stian ONARHEIM/ Candidate number: 8508

Using the Linux Kernel PREEMPT_RT Patch in Mixed-Criticality Systems



University of South-Eastern Norway
Faculty of Technology, Natural Sciences, and Maritime Sciences
Department of Science and Industry systems
PO Box 235
NO-3603 Kongsberg, Norway

http://www.usn.no

This thesis is worth 60 study points

University of
South-Eastern Norway

Using the Linux Kernel PREEMPT_RT Patch in Mixed-Criticality Systems

Master's Thesis in Computer Science

Stian ONARHEIM

Academic Supervisor

Prof. António L. L. RAMOS

University of South-Eastern Norway

Faculty of Technology, Natural Sciences and Maritime Sciences

Department of Science and Industry Systems

Campus Kongsberg

May 2024

Abstract

Due to advancements in embedded hardware platforms, implementing subsystems of varying criticality levels on the same hardware platform is a trend for modern real-time systems. The Linux kernel is a common candidate for mixed-criticality designs due to its popularity, versatility, and open-source license. The Linux kernel has seen an increase in interest for real-time usage, and for several years, a patch commonly known as PREEMPT_RT has been developed to improve the kernel's real-time capabilities. This thesis evaluates a Linux-based mixed-criticality system with the PREEMPT_RT patch. The focus is on the system's ability to respond reliably to incoming messages and signals over GPIO, Ethernet, and PCIe in a distributed system. Dedicated measurement systems are designed to generate messages and signals over GPIO, Ethernet, and PCIe and measure the round-trip time.

The Linux kernel's isolation mechanisms effectively lower the round-trip time for GPIO, Ethernet, and PCIe. They also increase the stability but do not provide total temporal isolation. Polling-based implementations are less affected by system load than interrupt-based implementations and produce reasonable results. The variance in the Linux kernel system latencies makes the kernel insufficient for hard real-time systems. Still, the proposed Linux-based mixed-criticality system design can be considered for soft real-time systems.

Acknowledgements

I want to express my gratitude to my supervisor, Professor António L. L. Ramos at the University of South-Eastern Norway (USN), for his guidance and support throughout the process of writing my master's thesis. His encouragement has been instrumental in pursuing this research topic. The process of working on my master's thesis has been an immensely rewarding and enlightening experience.

Additionally, I sincerely thank my family, colleagues, and classmates, whose encouragement has been an invaluable source of motivation. Furthermore, I am grateful for the financial support I have received from my company, which enabled me to attend the Embedded Open Source Summit 2023 in Prague. This experience inspired me to research real-time Linux and contribute back to the Linux community.

Lastly, a special thanks to my classmate Kent Odde for being an outstanding sparring partner throughout my academic journey and constantly motivating me to strive for greatness.

Grammarly [1] has been used to receive suggestions for improving language and writing style.

Stian Onarheim
Kongsberg, Norway, May 24, 2024

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
BCC	BPF Compiler Collection
BSP	Board Support Package
C2H	Card to Host
CBS	Constant Bandwidth Server
CLI	Command-line Interface
CPU	Central Processing Unit
DPDK	Data Plane Development Kit
EDF	Earliest Deadline First
FIFO	First-in, first-out
FPGA	Field-Programmable Gate Array
GPIO	General-Purpose Input/Output
GPOS	General-Purpose Operating System
GUI	Graphical User Interface
H2C	Host to Card
HPC	High-Performance Computing
IDE	Integrated Development Environment
IPC	Inter-process Communication
IPI	Inter-processor interrupts
IP	Intellectual Property
IRQ	Interrupt Requests
LLC	Last-Level Cache
MMU	Memory Management Unit
NIC	Network Interface Card
PCIe	Peripheral Component Interconnect Express
PID	Process Identifier
POSIX	Portable Operating System Interface
QoS	Quality of Service
RAM	Random Access Memory
RCU	Read-Copy-Update
RISC	Reduced Instruction Set Computer
RTOS	Real-Time Operating System
RTT	Round-trip Time
SMP	Symmetric Multi-Processing
SUT	System Under Test
SoC	System on a Chip
TLB	Translation Lookaside Buffer
UART	Universal Asynchronous Receiver-Transmitter
VFS	Virtual File System

VM Virtual Machine
WCET Worst Case Execution Time

Chapter 1

Introduction

The concept of utilizing the Linux kernel for real-time systems is nothing new [2]–[4]. However, interest has increased as newer systems become more complex and are met with both safety-critical and non-safety-critical requirements. A system with applications of different criticality levels on a shared hardware platform is called a mixed-criticality system. Designing a mixed-criticality system is becoming more achievable with modern multi-core hardware platforms [5]. Isolation between the different criticality levels on a shared hardware platform is attainable to a certain degree by hardware and software-based virtualization solutions, where hardware resources such as CPU cores, memory, and peripherals are partitioned among the criticality levels [6].

Linux is classified as a general-purpose operating system (GPOS) and has mainly been developed for high throughput and versatility. It is used in various application domains, such as servers, desktop computers, high-performance computing (HPC), and embedded platforms. On the other hand, real-time operating systems (RTOS) provide a framework for developing reliable and deterministic systems at the expense of throughput. Kernel developers have since 2005 been working on a patch commonly known as the PREEMPT_RT patch, which improves the kernel's real-time capabilities without breaking compatibility with existing Linux applications. The patch is still a work in progress and aims to be fully merged into the mainline Linux kernel [7], [8].

The Linux kernel's popularity, versatility, open-source license, and extensive hardware support make it an attractive platform for embedded systems. With the additional advancing improvements by the PREEMPT_RT project, there is a growing interest in utilizing the Linux kernel for real-time systems. In recent years, the Enabling Linux in Safety Applications (ELISA) project has been established to work towards guidelines for using the Linux kernel in safety-critical application domains such as aerospace and automotive [9].

1.1 Motivation and Problem Statement

Linux-based embedded systems with the PREEMPT_RT patch have previously been evaluated for real-time usage with varying results [10]. Compared to a *vanilla* Linux kernel, the PREEMPT_RT patch manages to effectively lower the system latency at the expense of throughput [11]–[13]. This is mainly achieved by reducing the amount of non-preemptible code sections, allowing high-priority tasks to more frequently preempt tasks with lower priority.

Hard real-time systems have traditionally been evaluated by calculating a Worst Case Execution Time (WCET) to ensure the system meets critical timing requirements [14]. Repeated application execution on a Linux-based system has shown that a `read(2)` system call can take 646 distinct paths in the kernel, making it infeasible to calculate a WCET for a Linux-based system [15]. However, a Linux-based system with the PREEMPT_RT patch can be considered for soft real-time systems where meeting the deadline most of the time is sufficient.

Several performance evaluations of PREEMPT_RT measure low system latency when applying a specific load or no load to the system [13], [16], [17]. High-priority tasks are significantly impacted by system load [18]–[20], indicating that system latencies are highly dependent on the type of system load. Performance evaluations of PREEMPT_-RT with no variance in system load may give false indications, as realistic systems are subject to load.

1.2 Proposal

The thesis aims to evaluate a Linux-based mixed-criticality system with state-of-the-art practices on a modern hardware platform. The goal is to produce relevant results for modern real-time systems, which often are part of larger distributed systems. This is achieved by measuring how reliably a Linux-based mixed-criticality system can respond to incoming messages and signals over the common peripheral interfaces GPIO, Ethernet, and PCIe.

Dedicated measurement systems will be developed to send messages and signals over GPIO, Ethernet, and PCIe while additionally measuring the round-trip time. While other performance evaluations of PREEMPT_RT have used the system under test (SUT) to measure the SUT [13], [18], [21]–[24], an approach with dedicated measurement systems ensures that the measurements will not be affected by applied stress to the SUT.

The thesis will focus on the following:

- Research the state-of-the-art best practices for designing a Linux-based real-time system.
- Design and implement independent measuring systems for GPIO, Ethernet, and PCIe.
- Design a Linux-based mixed-criticality system.
- Measure round-trip times for GPIO, Ethernet, and PCIe and showcase the impact of isolation techniques and applied system load.

1.3 Outline

The rest of this thesis is structured as follows. Chapter 2 introduces the necessary background information regarding the PREEMPT_RT patch and virtualization techniques for mixed-criticality systems. The chapter also includes a literature review on related works regarding Linux and the PREEMPT_RT patch, highlighting the state-of-the-art practices and challenges. Chapter 3 provides the methodology with a description of

the testing environment, including the dedicated measurement systems, the proposed Linux-based mixed-criticality system, custom measuring and tracing software, and the testing strategy. Chapter 4 presents the results for round-trip-times over GPIO, Ethernet, and PCIe, focusing on applied system stress and isolation techniques. The chapter also discusses the results and the proposed Linux system configuration. Chapter 5 concludes the thesis by summarizing its objective and findings, as well as discussing potential future work.

Chapter 2

Background and Related Work

This chapter covers the basic aspects of real-time Linux, with a deep dive into the most important modifications introduced by the PREEMPT_RT patch. It further covers the state-of-the-art best practices for implementing a Linux-based mixed-criticality system and discusses the challenges and related work.

2.1 Real-Time Operating Systems

An operating system plays a crucial role in the design of systems for complex application domains. In safety-critical systems, where 'safety' primarily refers to functional safety rather than security, the system needs to be deterministic, ensuring predictable and reliable behavior. Determinism in this context means that system responses are consistent and dependable, a critical requirement for the integrity of safety-critical operations. Any unexpected behavior could result in fatal consequences. Real-time operating systems purposefully ensure that tasks meet their deadlines and find use in safety-critical systems.

Applications that fall under the real-time classification generally have a requirement regarding the execution time, commonly expressed as a deadline. Real-time applications are usually divided into three classifications based on the validity of the result and the consequence of a missed deadline.

- **Soft real-time:** Frequently missed deadlines are acceptable as long as the Quality of Service (QoS) is kept at an acceptable level. The result can still be seen as useful after the deadline.
- **Firm real-time:** Missing a deadline will cause the result to be invalid.
- **Hard real-time:** A missed deadline will result in a system failure, which can have fatal consequences.

Commercial RTOSs like Wind River VxWorks and Green Hills Integrity have a long-standing presence in critical application domains such as aerospace, automotive, defense, and medical devices. These operating systems require licenses and often come with their own compilers, debuggers, and integrated development environments (IDE). Their source code is proprietary, with limited online resources, and, in most cases, requires support from the distributor. Compared to GPOSs, RTOSs have limited support for hardware platforms and peripherals.

2.2 Real-Time Linux

The Linux kernel has primarily been developed for general-purpose computing. While an RTOS focuses on being deterministic, a general-purpose operating system focuses on maximizing throughput. The idea of utilizing the Linux kernel for real-time applications is nothing new. RTLinux (1997) [2], RTAI (2000) [3], and Xenomai (2002) [4] are well-known open-source approaches based on the co-kernel concept [10].

2.2.1 Co-Kernel Approach

With a co-kernel design, the Linux kernel can co-exist with a real-time kernel. The Linux kernel acts as an *idle* task for the real-time kernel and is scheduled when no higher-priority task is running. The real-time kernel or a separate micro-kernel handles the hardware interrupts and forwards them to the correct kernel. A typical co-kernel system design would schedule applications with real-time requirements on the real-time kernel while scheduling non-real-time sensitive applications on the Linux kernel. Figure 2.1 describes a general co-kernel architecture.

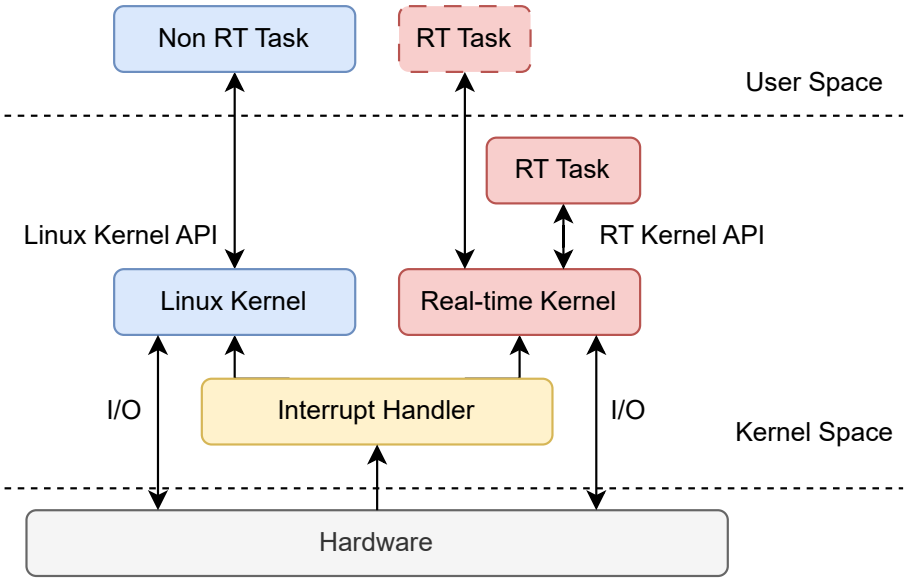


FIGURE 2.1: Co-kernel architecture with the Linux Kernel and a dedicated real-time kernel.

Applications written for Xenomai can be executed in user and kernel space, while RTLinux and RTAI only allow applications to run in kernel space. A major drawback of the co-kernel approach is that applications running on the real-time kernel must use the real-time kernel's API. This means that existing Linux applications and drivers have to be rewritten [10].

2.2.2 Single-Kernel Approach (PREEMPT_RT)

The official single-kernel approach for real-time Linux started in 2005 with the release of the Linux kernel PREEMPT_RT patch. The objective of the patch is to further develop the existing Linux kernel to become more real-time compatible by reducing the portion of kernel sections that are non-preemptible [7], [8]. This implies that, unlike

the co-kernel approach, existing Linux drivers and applications are compatible and do not need to be rewritten.

The PREEMPT_RT project has been kept alive by various funders throughout its lifetime. From military contracts to part-time funding from Red Hat. In 2014, the project lost all funding and was considered a hobbyist project. Since 2015, the project has been funded by the Linux Foundation [7]. To support the rising interest in the PREEMPT_-RT patch, the Linux Foundation has founded the Enabling Linux in Safety Applications (ELISA) project. The project consists of working groups specialized in different domains, such as aerospace and automotive. Their work focuses on defining processes and tools for developing Linux-based safety-critical systems [9].

Preemption

A fundamental characteristic of an RTOS is the ability to preempt tasks to execute code with higher priority, whether it is an interrupt handler or a task. Figure 2.2 describes a scenario with the traditional Linux preemption model.

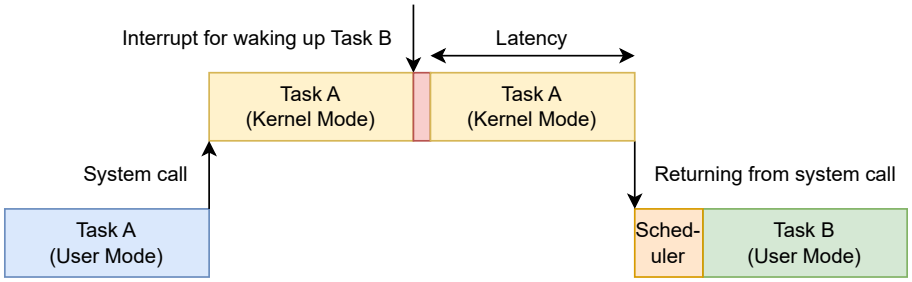


FIGURE 2.2: Scenario with no forced preemption.

While Task A executes a system call in kernel mode, an interrupt for waking up the more important Task B occurs. With the traditional preemption model, Task A has to complete its code execution in kernel mode and return from the system call before Task B can be scheduled [25]. This adds an unbounded latency that varies based on the duration of the system call.

The PREEMPT_RT patch introduces a fully preemptible Linux kernel, allowing preemption in kernel mode except for a few critical sections. Figure 2.3 describes the previous scenario, only now, with a fully preemptible Linux kernel.

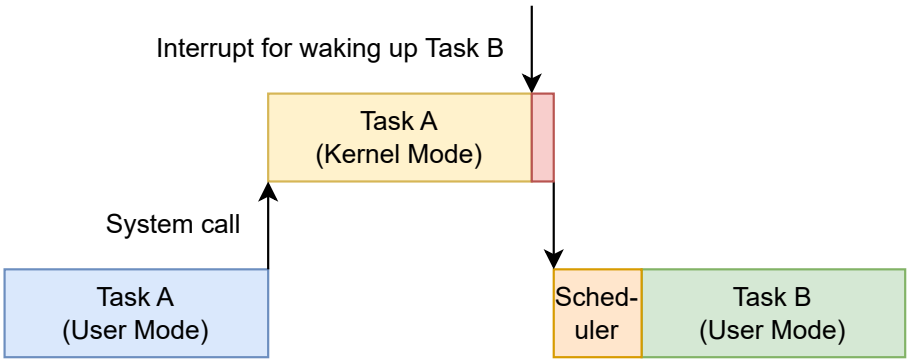


FIGURE 2.3: Scenario with a fully preemptible kernel.

While Task A processes a system call in kernel mode, the interrupt for waking up the more important Task B will implicitly trigger the scheduling routine. The scheduler chooses to schedule Task B as it is ready to run and is more important than Task A, regardless of Task A executing code in kernel mode. This negatively impacts Task A's execution time but results in a more deterministic system, as higher-priority tasks can preempt lower-priority tasks whether they are in user or kernel mode.

Locking Primitives

In multi-threaded applications, locking primitives prevent shared objects from being accessed or modified by multiple threads simultaneously. The most common example is a mutually exclusive lock. While one thread obtains the lock, the others are blocked while trying to acquire it. The Linux kernel offers two implementations of the blocking mechanism: Either the task is suspended until the lock is freed, or the task will busy-wait, also known as spinning.

Spinning locks, as illustrated in Figure 2.4, will implicitly disable preemption and keep the CPU core *hostage* while busy-waiting until the lock is acquired. Consequently, other tasks are prevented from running on the same CPU core until the lock has been acquired and released. This is unfortunate in situations where high-priority tasks have to wait for low-priority tasks that are spinning.

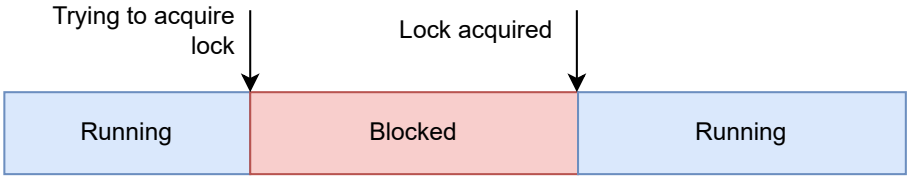


FIGURE 2.4: Spinning locks.

Sleeping locks will instead put the task to sleep while waiting for the lock to be available. This allows other tasks to run in the meantime. Figure 2.5 describes the typical events of a sleeping lock. Sleeping locks are introduced to extra processing overhead as they are scheduled out and in.

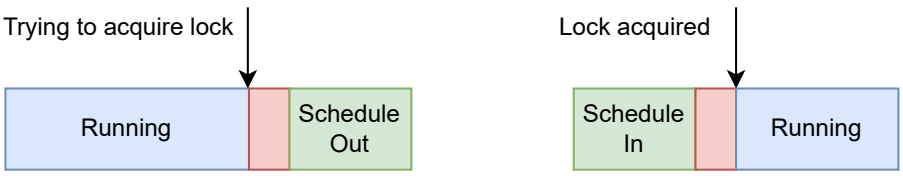


FIGURE 2.5: Sleeping locks.

Although busy-waiting can be seen as wasting CPU cycles, the lock will be acquired much faster than by suspending the task and waiting to be rescheduled when the lock is free. The overhead caused by scheduling and context switching is expensive. Busy-waiting locks, therefore, have a purpose in critical sections.

With the PREEMPT_RT patch, spinning locks are converted to sleeping locks. This decreases the amount of non-preemptible sections in the kernel. Critical code sections that require preemption and interrupts to be disabled can utilize the spinning lock `raw_spinlock_t`, as it will still act as a traditional spinning lock even with a PREEMPT_RT patched Linux kernel [26].

Priority Inversion and Inheritance

When tasks with different priorities share a lock, one might experience a situation known as the *priority inversion* problem. When a high-priority task tries to acquire a lock that is shared and obtained by a task of lower priority, it will have to wait until the lower-priority task has freed the shared lock. During this period, tasks with a higher priority than the low-priority task might run, increasing the waiting time for the high-priority task. Figure 2.6 describes a scenario with three tasks of varying priority where the priority inversion problem is present.

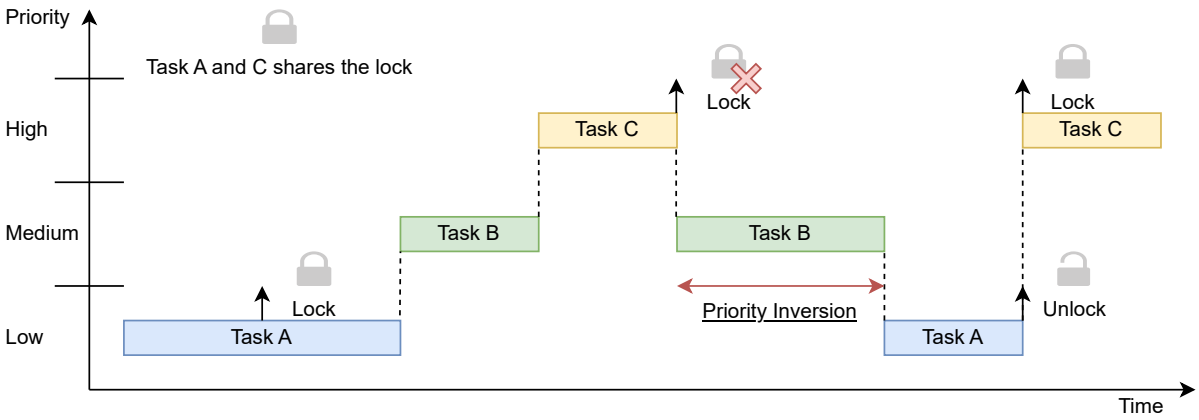


FIGURE 2.6: Scenario with priority inversion.

The scenario starts with the low-priority Task A acquiring a lock that it shares with the high-priority Task C. Before Task A has finished processing and freed the shared lock, it is preempted by Task B and later Task C as they have higher priority. Task C then tries to acquire the shared lock but fails as Task A still possesses the shared lock. Task C is considered blocked and is scheduled out. Instead of prioritizing scheduling Task

A so it can complete its processing and free the shared lock so Task C can be scheduled, Task B is chosen by the scheduler as it has a higher priority than Task A. It is not until Task B is finished that Task A can free the lock so Task C can continue its execution. This scenario has only one task with a priority level between Task A and Task C. With many tasks in between, the priority inversion problem only increases.

The PREEMPT_RT patch implements *priority inheritance* to solve the priority inversion problem. When a task is waiting for a shared lock to be freed by a task with lower priority, the lower-priority task's priority is temporarily *boosted* to the same priority as the higher-priority task. This functionality is only implemented with the `rt-mutex` lock. Figure 2.7 describes the previous scenario in Figure 2.6, now with priority inheritance.

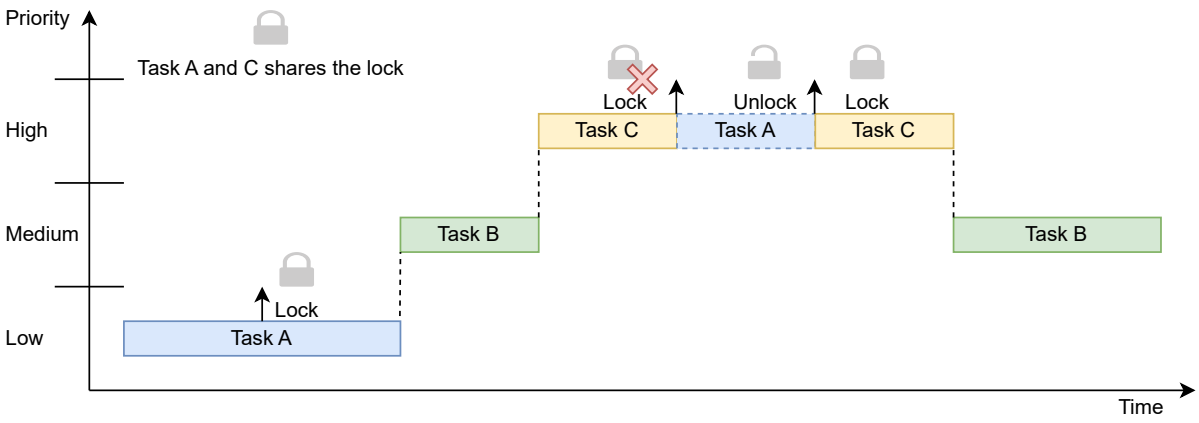


FIGURE 2.7: Scenario with priority inheritance.

When Task C attempts to acquire the shared lock, currently obtained by the lower-priority Task A, Task A's priority is boosted to the same priority level as Task C. This ensures that Task C is unblocked as soon as possible. After Task A has freed the lock, it is downgraded to its previous priority level.

Interrupts

Interrupt routines with variable execution time are a common source of unwanted latencies in the Linux kernel. Interrupt routines have precedence over system processes and will block other pending interrupts. Interrupt routines should, therefore, ideally be as short as possible. Devices will often trigger an interrupt that requires a large amount of work to be processed, resulting in lengthy interrupt routines. The Linux kernel fixes this problem by providing mechanisms for splitting interrupt handlers into two halves. A *top-half* for critical code that has to be executed right away, and a *bottom-half* that can run at a later stage. Maskable interrupts are only disabled during a top-half routine, meaning that bottom-half routines can be interrupted.

Bottom-half routines have several implementations in the Linux kernel. With a vanilla Linux kernel, it is possible to *raise* a bottom-half routine at the end of a top-half routine and have it executed right after the interrupt handler. User tasks may experience lengthy delays caused by bottom-half routines that in a system design context are less important.

The PREEMPT_RT patch forces all top-half routines that are not explicitly declared with the `IRQF_NO_THREAD` flag to run in a threaded context with the scheduling policy `SCHED_FIFO` and a default priority of 50. System processes with a higher priority than 50 will, therefore, have precedence. The priority of threaded interrupts can be set individually, allowing the system architect to correctly prioritize interrupt routines concerning real-time applications [27].

Scheduling Policies

The scheduler plays a critical role in an operating system. Its job is to decide which thread the CPU shall run next. Each thread has a *state*, *scheduling policy*, and a *static priority*, which is information the scheduler uses during its decision-making. The scheduling policies can be divided into *real-time* and *normal*. The following scheduling policies `SCHED_OTHER`, `SCHED_IDLE`, and `SCHED_BATCH` are categorized as normal, and their static priority value is set to 0. The real-time scheduling policies `SCHED_FIFO` and `SCHED_RR` have a static priority inside a range of 1 - 99, where higher is better.

The scheduler maintains a queue of *runnable* threads for each static priority value. It iterates over the queues and selects the thread at the front of the queue with the highest static priority as the next thread to be scheduled. When a thread with a higher priority than the currently running thread becomes runnable, the scheduler will preempt the currently running thread and place it back into the queue for its respective static priority. Depending on the thread's scheduling policy, it is placed at the front, back, or somewhere in the middle of the queue.

The following scheduling policies fall into the real-time category.

`SCHED_FIFO`: Threads are inserted in a first-in, first-out (FIFO) scheme into its static priority queue. Threads will run until they are blocked, preempted, or by explicitly calling the system call `sched_yield(2)`.

`SCHED_RR`: Threads are given an equal time slice. When a thread reaches the end of its allocated time slice, it is placed back at the end of the queue. This scheduling policy allows threads to share the CPU fairly.

`SCHED_DEADLINE`: The scheduling policy is based on the earliest deadline first (EDF) and constant bandwidth server (CBS) algorithms, where global EDF is implemented on multi-core systems to allow threads to migrate between cores. In its basic essence, the thread with the shortest remaining relative deadline is scheduled to run next.

Each thread is configured with three additional parameters: *period*, *runtime*, and *deadline* [28]. The parameters are represented in Figure 2.8.

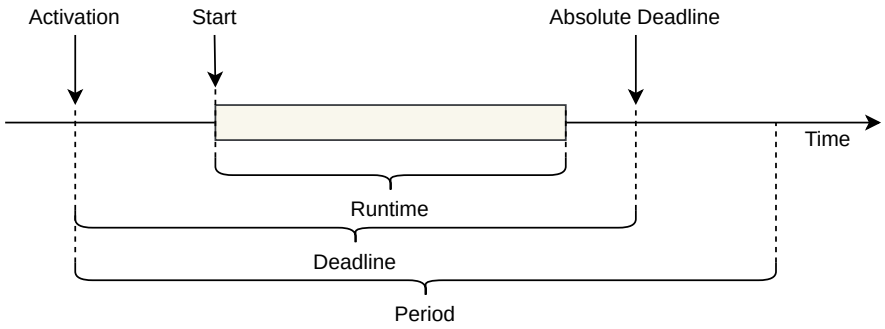


FIGURE 2.8: SCHED_DEADLINE parameters.

Threads scheduled with the `SCHED_DEADLINE` policy are given the highest priority in the system [28].

High Resolution Timers

Normal timers in the Linux kernel are tick-driven. This implies that the timer resolution is derived from the tick frequency, which normally is configured between 100 Hz to 1000 Hz. With a tick frequency of 10 Hz, the timer resolution would be as low as 10 milliseconds.

With the introduction of the high-resolution timers subsystem, it is possible to request a timer with a resolution of one nanosecond [29]. Applications in user space can utilize high-resolution timers by using `posix timers`, `itimers`, or `nanosleep` as these are re-implemented with high-resolution timers when the kernel configuration option `CONFIG_HIGH_RES_TIMERS` is set [30].

2.2.3 Tools

The vast amount of tools available on Linux for kernel tracing, debugging, networking performance, stress tests, and real-time analyses makes Linux a great platform for software development. It is important for software development to be able to perform analyses to find bugs and performance bottlenecks.

Stressors

When tuning the kernel for real-time workloads, it is interesting to analyze how well a real-time application can withstand noise generated by other processes. Tools commonly known as *stressors* generate a simulated workload.

`stress-ng` is a powerful stress-testing tool with over 310 stressors. The tool includes stressors that stress specific areas such as memory/CPU cache, file system, virtual memory, scheduling, high-resolution timers, and networking. The tool is cited in over 80 academic research papers [31], where many are real-time related.

`hackbench` is another popular stressor within the real-time community and is a part of the `rt-tests` test suite [32]. The tool focuses on the repeated setup and teardown of threads and inter-process communication between them [33].

Although stressors can simulate much of the workload generated by real-time systems, they cannot do so accurately. Most real-time systems communicate with external hardware such as sensors and actuators, which stressors cannot simulate accurately.

Tracing Tools

The official tracer for the Linux kernel is called `ftrace`. It can be used for recording the execution flow of kernel functions with additional information such as CPU ID, PID, and timestamp. The tracer offers a great filter selection for specific events such as scheduling, interrupts, and lock mechanisms. This information helps create a picture of what is happening inside the kernel.

The tracer is accessible within the filesystem `tracefs`, which is mounted on `/sys/kernel/tracing` and automatically within `debugfs` in `/sys/kernel/debug/kernel` for backward compatibility with older applications. Interfacing the tracer by writing and reading several files in the filesystem is impractical, which prompted the development of the command line interface (CLI) program `trace-cmd`, a front-end application to `ftrace` [34].

Reports generated by `trace-cmd` can be used together with the graphical user interface (GUI) application `KernelShark` [35]. This allows for a visual representation of the trace data with graph plots.

Tracers must be explicitly enabled when configuring the kernel and will result in a performance loss. Due to their extra overhead, it is recommended that kernel tracers be turned off in production systems.

Although still not widely used in the Linux real-time community, the up-and-coming *eBPF*-technology [36] makes it possible to modify and extend the kernel during runtime. This technology can be used to create better tools for analyzing real-time applications. A proof of concept was presented at the Embedded Open Source Summit 2023 by John Ogness [37].

Benchmarking Tools

Benchmarking tools are necessary to determine how well a system behaves and are especially helpful in measuring the impact of small tuning changes.

`cyclictest` is the most commonly used benchmarking tool for real-time systems. It is designed to measure latencies caused by the hardware and operating system. The tool starts a pre-defined number of threads with real-time priorities that are woken up periodically. The time difference between the configured and actual wake-up time is considered to be the scheduling latency. The tool displays the minimum, average, and maximum latency. The test is by default implemented with the `clock_nanosleep(2)` system call, which implies that the timer wake-up routine is executed in a hard interrupt context. Therefore, applications that rely on threaded interrupts will experience a longer wake-up latency than those measured with `cyclictest` [38].

`rtla`, which is short for real-time Linux analysis tool, is a set of tools for analyzing the kernel's real-time behavior. The tools utilize the kernel's tracing capabilities to record latencies caused by hardware and operating system noise and the timer latency separately at the `IRQ` and `Thread` handler [39].

`perf` is a powerful profiling tool that monitors hardware and software events. The tool counts kernel events such as cache misses, page faults, context switches, and CPU migrations. The tool can also analyze lock events, memory accesses, and scheduling latencies.

2.3 Mixed-Criticality Systems

An ever-increasing trend for real-time systems is integrating multiple subsystems of varying criticality levels onto the same multi-core hardware platform. Mixed-criticality systems (MCS) use a common hardware platform to implement subsystems of different criticality levels, such as safety-critical and non-safety-critical [40]. Mixed-criticality systems' main challenge is isolating and reducing interference between the different criticality levels. Total isolation is important for safety-critical systems to guarantee meeting their deadlines. Isolation solutions usually refer to the following three properties.

- **Fault isolation:** Faults within one partition shall not affect others.
- **Spatial isolation:** Memory reserved for one partition shall not be accessible to others.
- **Temporal isolation:** Usage of hardware such as CPU, network and disk shall not cause serious delays for other partitions.

2.3.1 Virtualization

Virtualization is a widely used solution for mixed-criticality systems. It enables multiple operating systems and bare-metal applications to run concurrently on a shared hardware platform. Isolation is achieved by statically or dynamically partitioning the hardware's resources. Virtualization is not limited to systems with safety-critical requirements, leading to a wide variety of available solutions.

Software-based virtualization solutions are typically classified into three categories: Type I Hypervisor, Type II Hypervisor, and container-based. Type I Hypervisors run directly on the hardware, while Type II Hypervisors run on top of a host OS. Containers rely on the isolation capabilities provided by an operating system and are not used for running multiple operating systems. Figure 2.9 shows a visual representation of the three software-based virtualization classifications.

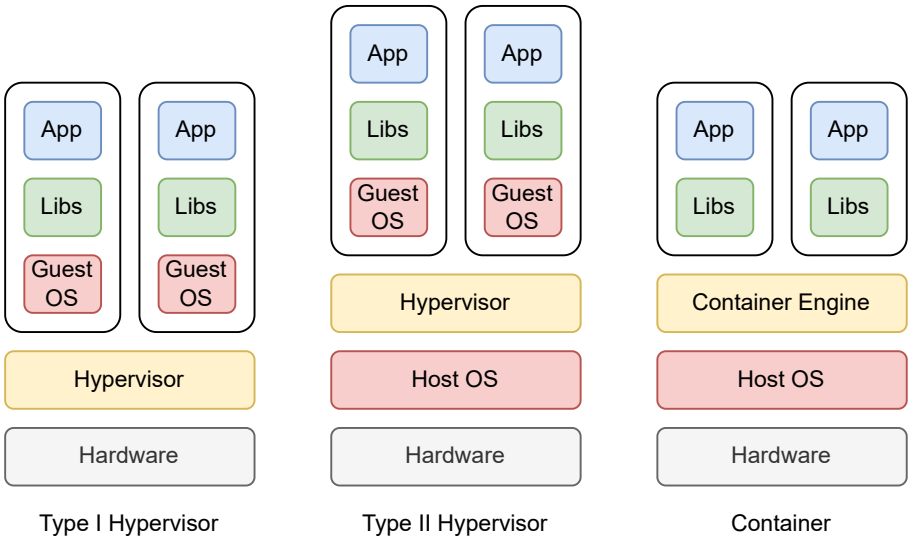


FIGURE 2.9: Software-based virtualization methods.

In a safety-critical context, latency caused by interference from other containers or virtual machines (VMs) has to be deterministic or non-existent. This imposes strict requirements from application domains such as avionics, automotive, and railway for virtualization solutions regarding isolation properties. Hypervisors classified as *separation kernels*, a small bare-metal hypervisor utilizing hardware virtualization extensions to partition VMs statically and nothing more, and lightweight real-time *microkernel* hypervisors that extend the concept of a separation kernel by additionally managing memory, device drivers and scheduling for the VMs, are preferred for safety-critical systems due to their compliance with safety standards. Hypervisors classified as general-purpose are generally best suited for server and HPC domains but have recently seen improvements regarding their real-time capabilities [6].

The two most relevant open-source general-purpose hypervisors for embedded Linux are Xen [41] and KVM [42]. Xen is a bare-metal hypervisor (Type I) sponsored by the Linux Foundation that, in recent years, has seen improvements regarding its real-time capabilities. They have introduced static partitioning and cache-coloring, a concept where the last level cache is partitioned among the VMs [43]. Their current objective is to get the hypervisor certified for the Automotive Safety Integrity Level (ASIL) [44]. KVM extends the Linux kernel by allowing it to host isolated VMs while acting as the host OS (Type II). KVM utilizes QEMU [45] for I/O hardware emulation for the VMs [46].

The results of experimental analyses comparing latencies introduced by Xen and KVM indicate that a KVM host kernel configured with PREEMPT_RT can achieve a worst-case hypervisor latency of less than 100 microseconds when the system is under no load, while Xen performs considerably worse [21]. Another performance analysis of KVM showed a worst-case hypervisor latency of less than 200 microseconds when deploying two virtual machines [47]. In the pursuit of designing an embedded mixed-criticality system that can safely meet a 1 kHz periodic task's 100 microseconds deadline, Li et al. [18] evaluated KVM with different types of stressors and found that their system design managed to meet the deadline 99% of the time with 96% of the

stressors. Most deadline misses were caused by hardware interference when running bus-intensive stressors.

When selecting and evaluating an embedded hypervisor for the automotive industry, E. Hamelin et al. [48] ruled out open-source solutions like Xen due to their lack of safety standard appliances. After considering 23 different embedded hypervisors, they chose the commercial separation kernel-based hypervisor PikeOS. In their performance benchmark results with PikeOS, they discovered that resource sharing can still lead to significant interference between virtual machines, with a 90% reduction in network bandwidth and a 60% reduction in memory.

Due to the shared hardware resources, limiting interference among isolated partitions in a mixed-criticality system is extremely challenging. Valsan et al. [49] found that cache-coloring, a popular technique for partitioning shared last-level caches (LLC) between partitions, does not guarantee predictable cache access timing.

The literature indicates that modern virtualization methods for safety-critical mixed-criticality systems can not guarantee temporal isolation.

2.3.2 Resource Partitioning in Linux

The Linux kernel provides a few techniques for partitioning resources among processes. Control groups, commonly known as `cgroups`, is a mechanism that can group processes and all their future *children* into hierarchical groups. Each group contains a set of *controllers* that can distribute a specific type of system resource to a group's processes. The supported controllers are `cpu`, `memory`, `io`, `cpuset`, `rdma`, `hugeTLB`, and `misc` [50]. Control groups are most famously used with the popular container platform Docker for resource distribution between containers [51].

The `cpu` controller can partition the CPU's bandwidth within a group. As a requirement, all tasks within the group need to be scheduled with non-RT scheduling policies. This implies that processes with real-time scheduling policies can not be placed in a group that uses the `cpu` controller. M. Thiyyakat et al. [52] have performed a study using the CPU controller to partition 95% of the total CPU bandwidth across all cores for a group reserved for critical tasks. They find that under heavy load, `cgroups` alone cannot guarantee the configured CPU bandwidth. They recommend using the real-time scheduling policy `SCHED_DEADLINE` instead of the `cgroup cpu` controller to meet CPU utilization requirements.

Interrupt requests (IRQ) in symmetric multi-processing (SMP) systems can be configured to run on specific CPUs if the IRQ controller supports it. This can be configured in `procfs` for each IRQ number or during boot with the kernel command-line parameter `irqaffinity` [53]–[55]. In a mixed-criticality system design, moving IRQ processing away from the real-time CPU cluster can halve the scheduling latencies for real-time processes during heavy system load [17].

The kernel command-line parameter `isolcpu` can isolate CPUs from disturbances caused by general SMP balancing, *managed* interrupts, and in some cases, scheduling clock ticks. The latter option is referred to as `NO_HZ`. The settings set with `isolcpu` during boot-time are irreversible and cannot be changed once the system is up and

running. A runtime alternative to `isolcpu` is to partition the CPU cores with `cpuset` within `cgroups`. This is the recommended approach [55].

2.4 Tuning and Best Practices for Real-Time Linux

Linux is, first of all, a general-purpose operating system. However, the kernel and user applications can be tuned to perform better for specific domains and workloads, whether high-performance computing, networking, low latency, or real-time. The Linux kernel's versatility is one of the reasons for its widespread usage.

2.4.1 Memory

The Linux kernel divides its virtual memory into pages, usually with a size of 4096 bytes. It is considered a "Page Fault" when a program tries to reference a virtual address currently not mapped to a physical address in RAM by the Memory Management Unit (MMU). This causes the CPU to be interrupted by the MMU to run the page fault handler routine in kernel mode. Page faults are considered to be *major* when the page is not located in RAM and has to be loaded from disk, and *minor* when the page has to be allocated in RAM or is already present but needs to be re-mapped. This occurs regularly as the MMU's Translation Lookaside Buffer (TLB) is often smaller than the total number of mappings that the kernel stores [56].

Latencies introduced by major page faults can be mitigated by pre-faulting the stack and heap during program initialization and by locking the program's virtual address space into RAM to avoid swapping.

Memory locking is achieved by calling the `mlockall(2)` system call. A combination of the `MCL_FUTURE` and `MCL_CURRENT` flag arguments ensures that the application's future and current memory pages are locked into RAM [57]. This method is a common practice for real-time applications [58]–[61].

When a memory page is referenced for the first time, a minor page fault occurs and the memory page has to be allocated in RAM. If an application's maximum stack and heap usage is known at program initialization, pre-faulting the stack and heap is done by iterating over the number of memory pages and referencing them once. This ensures that all memory pages are allocated in RAM and have been *touched* once, reducing the unpredictable delay caused by minor page faults during program execution [61].

2.4.2 Timer APIs

There are some timer APIs to avoid when writing applications for real-time Linux. The Linux kernel offers several ways to configure a timer. Timers configured with `timerfd` can be monitored through a file descriptor with the `select(2)`, `poll(2)`, and `epoll_wait(2)` system calls. POSIX timers can issue a *signal* when a timer has elapsed. With the PREEMPT_RT config, all timers, whether standard or high-resolution, will run in a threaded interrupt context. The only exception is `clock_nanosleep(2)`, where the high-resolution timer's wakeup function is executed in hard interrupt context [30].

For real-time applications, it is recommended to use the `clock_nanosleep(2)` system call with the `CLOCK_MONOTONIC` and `TIMER_ABSTIME` parameters to achieve fast response times and avoid drift [62].

2.4.3 Inter-Process Communication

It is common for processes to communicate with each other. The Linux kernel offers several mechanisms for inter-process communication (IPC). System designs that involve IPC between processes with different priorities will not automatically get *priority boosted* by applying the PREEMPT_RT patch. This can only be achieved explicitly with `rt-mutexes` [63]. The developer has to be aware of this when considering IPC mechanisms.

2.4.4 The Current State of Official PREEMPT_RT Guidelines

At the recent Linux Real-Time Summit 2023, it was pointed out that there is no official documentation describing best practices concerning memory management, IPC, and APIs to avoid [37]. This makes it even more challenging for developers to write applications for real-time Linux, as it requires a deep understanding of the Linux kernel to avoid common pitfalls.

At the same conference, some basic guidelines for real-time Linux were recommended by various authors. The recommendations include locking current and future memory pages into physical RAM, isolating CPU cores for sensitive real-time workloads, enabling adaptive-tick mode, disabling lockup detectors, disabling CPU frequency scaling, and disabling real-time throttling [19], [60].

2.5 Workloads on Linux

The Linux kernel divides its virtual address space into two parts with different levels of privileges: user and kernel space. Processes running in user mode cannot access kernel space data or execute code stored there. Processes running in user mode can request services provided by the kernel through system calls. This can be everything from simple file operations to interfacing hardware device drivers [64].

On Arm and other reduced instruction set computer (RISC) architectures, a system call will cause a software interrupt to switch to kernel mode. On modern x86 architectures, system calls have been implemented differently and will instead call a `SYSCALL` instruction, which is faster as it eliminates the overhead caused by an interrupt [65].

A system call follows a *synchronous* execution model in the sense of waiting for the kernel to complete the system call before continuing user-mode execution. Switching context in and out of kernel mode comes with a cost. Soares and Stumm [66] find in their analysis of the system call's footprint that the cache is heavily polluted during a system call.

A typical Linux system is disrupted at a typical rate of 100 to 1000 times per second on each CPU. The kernel has to perform critical housekeeping tasks such as scheduling, read-copy-update (RCU) callbacks, updating the kernel time, and executing work

queues and bottom-half interrupt handlers. This periodic "tick" is referred to as the periodic timer interrupt. Interrupting applications at such a high rate will naturally impact performance. The Linux kernel offers configuration options to make the kernel borderline tickless on designated CPUs. The default option `CONFIG_NO_HZ_IDLE` disables the tick on idle CPUs, while the `CONFIG_NO_HZ_FULL` option enables *full tickless* mode, also referred to as adaptive-tick mode.

There are a few requirements for the adaptive-tick mode to take effect. The CPUs must be designated at boot time with the `nohz_full=` kernel command-line parameter, where one CPU must remain in non-adaptive-tick mode to handle various housekeeping tasks. Adaptive-tick CPUs will only enter adaptive-tick mode if a single process is running or the CPU is idle [67], [68]. When adaptive-tick CPUs have to be woken up, the adaptive-tick CPU will receive an inter-processor interrupt (IPI) from another CPU.

Akkan et al. [69] have experimented with reducing OS noise in Linux by utilizing adaptive-tick mode on their application cores. They find a slight increase in performance; however, their system is much more deterministic regarding runtime variance. The increase in determinism is partly due to the reduction of L1 cache misses, as each *tick* causes the CPU to switch context from user mode to kernel mode, bringing new data structures into the cache and causing cache eviction for application code.

2.5.1 Ethernet

The Linux kernel supports a wide range of network interface cards (NIC), making it attractive for systems that rely on Ethernet communication. Applications in user mode interact with the kernel's networking stack through file descriptors provided by the virtual file system (VFS). The VFS makes it possible to interface the kernel through system calls such as `read(2)` and `write(2)` [70].

Zhang et al. [71] have analyzed the shortcomings of the Linux kernel's network stack and found that more than half of the time spent sending a packet is used for socket-related operations. This includes the overhead caused by system calls and by copying memory from user space to kernel space and vice versa. When the network stack builds a packet or receives data, it will allocate temporary buffers in kernel space. Frequent memory allocations will result in extra overhead caused by page faults.

Due to the inefficiencies of the native Linux kernel's networking stack, alternative networking stacks have been created targeting hard real-time networking. RTnet [72] is an open-source project that provides a hardware-independent software framework for hard real-time Ethernet communication. It was mainly written for RTAI, and later ported to Xenomai. A. Duca et al. [73] have created a port of RTnet for the Linux kernel version 5.9 with the PREEMPT_RT patch. The patch is over 70,000 lines and introduces new RTnet-specific system calls for sending and receiving UDP packets. Compared with the native Linux kernel networking stack, their RTnet port reduces the average RTT by half, with a worst-case RTT of 643 microseconds compared to 33,000 microseconds. Unfortunately, the port is not mainlined or maintained, but it shows the potential of utilizing an alternative networking stack.

Networking systems that handle a great deal of networking traffic or require faster packet processing, such as in the Financial Service Industries domain, can benefit from

the use of the open-source Data Plane Development Kit (DPDK) [74]. With a DPDK-supported NIC, it is possible to bypass the Linux kernel's networking stack completely by directly accessing the NIC and processing the data in user space. This eliminates the extra overhead caused by the native Linux kernel's networking stack. Li, Zongyao [75] has experimented with DPDK when creating a high-performance software-based router. He achieves a throughput increase of 8 - 10 times higher than when utilizing the native Linux kernel network stack. Xu et al. [76] has created a DPDK-based DDS solution suited for distributed real-time applications. Compared to the state-of-the-art FastDDS that utilizes the Linux networking stack, their solution increases the data transmission throughput by 51% and reduces the average RTT by 56%. Not all network interface cards can benefit from DPDK, as only a limited amount is supported. Mainly NIC for x86 systems are supported, but there exists a few supported Arm processors [77].

The common trend for achieving high-performance Ethernet with Linux involves either rewriting the networking stack or bypassing the Linux networking stack entirely.

To avoid latency-inducing interrupts for incoming packets, networking sockets in Linux can be configured with the `SO_BUSY_POLL` option. This enables the socket layer to poll the incoming packet queue directly. This removes the overhead caused by interrupts and context switching [78]. While this feature improves the overall networking performance, it has been disabled with PREEMPT_RT to avoid wrong locking context problems [79].

2.6 Related Work With PREEMPT_RT

The current literature on real-time Linux is spread out in research articles, developer conferences, wiki pages, and mailing lists. The majority of the core literature on real-time Linux with the PREEMPT_RT patch, which is written by kernel developers, can be considered to be non-academic. However, several academic papers have been produced with experiments with PREEMPT_RT. Table 2.1 shows statistics for the search "PREEMPT_RT" from several online databases that are available to the University.

Database	Number of Results
IEEE Xplore	25
ScienceDirect	44
Scopus	60
Springer Link	117

TABLE 2.1: Database Results for PREEMPT_RT.

Brown and Martin [80] have conducted a study to measure and compare the latencies between a Xenomai kernel, a native Linux kernel, and a PREEMPT_RT patched Linux kernel. They implement an application that responds to a GPIO signal in both user and kernel space, resulting in six combinations. As some real-time tasks have less strict timing requirements, where meeting the deadline 95% of the time is considered good enough, the results differentiate the worst-case latency when 100% of the results

are accounted for and only 95%. They find that Xenomai has lower latencies and less jitter for hard real-time tasks than the Linux kernel. However, when the top 5% of latency results are removed, the application implemented in kernel space with the native Linux kernel slightly outperforms Xenomai. The authors recommend to *only* consider designing a system with Xenomai if 95% is not considered good enough.

A WCET and schedulability analysis is necessary for hard real-time systems to verify their timing correctness. In a safety-critical context, failing to meet a deadline can be catastrophic [14]. Such analyses are increasingly more difficult for modern processors due to the variable execution time caused by pipelines, caches, and branch predictions [81]. Introducing Linux, a complex operating system, makes the analyses even harder to perform, if not unfeasible. Tasks can experience unbounded interference caused by kernel housekeeping, lock synchronization between tasks, and device interrupts, to name a few. A timing analysis on PREEMPT_RT [82] finds a great variance in the time spent scheduling and activating a task. This significantly impacts tasks waiting to acquire a sleeping lock as it involves a lot of rescheduling.

Okech, Peter, et al. [83] find in their experiments that repeated application executions might take different paths in the kernel. A repeated `read(2)` system call can follow a path that includes anywhere from 92 - 888 function calls [15]. These uncertainties make it infeasible to conduct static code analysis, a technique the IEC 61508 safety standard highly recommends. Allende et al. describe the Linux kernel as a "composition of several interdependent state machines, which are asynchronous concerning each other, and thus, a given software execution path can exhibit stochastic behavior due to this asynchronicity." [84]. Probabilistic theory can partially solve this problem. Allende et al. [84] acknowledge that having unknown paths is a safety risk, as the paths are consequently untested. They propose a method to quantify the uncertainties the Linux kernel provides by estimating the percentage of unknown paths.

Xuebing Chen [85] evaluates the Linux kernel PREEMPT_RT patch for a Loongson 3A3000 processor. To reduce interference from housekeeping tasks and peripheral interrupts, the kernel command-line argument `isolcpu` is used to reserve a single CPU core for real-time applications. Task switching, interrupt response, and scheduling latency are similar on a system with no load compared to one with a load that stresses IO operations, network communication, graphics rendering, and inter-process communication. C. Huang and C. Yang [23] finds that their custom application, meant to resemble a real-time robot control system, manages to halve its worst-case latency by pinning the application onto an isolated CPU core. The same application is implemented in kernel space, effectively removing the overhead caused by system calls. The kernel space variant's worst-case latency is four times lower than the user-space equivalent. Adam et al. [13] perform a similar experiment where a GPIO application is implemented in both user and kernel space. Similar to Huang and Yang's results, the kernel space variant performs noticeably better than the user space equivalent.

Although user-space applications heavily relying on system calls will perform better if implemented in kernel space, they can not be linked with the C-library and other user-space libraries. Linux kernel modules have a more limited set of APIs and are harder to debug than user space applications. Maintaining user-space applications is simpler since deprecated Linux kernel system calls remain backward compatible with newer Linux kernel versions. However, this is not the case for kernel-space APIs. These

APIs are frequently modified and removed, meaning kernel modules must be ported to newer Linux kernel versions.

2.7 Summary

The Linux kernel PREEMPT_RT patch improves the kernel's real-time capabilities, mainly by increasing the portion of preemptible sections. The PREEMPT_RT patch performs worse than co-kernel approaches suited for hard real-time systems. However, its compatibility with existing Linux applications and device drivers makes it preferable for system designs with soft real-time requirements.

Applying the PREEMPT_RT patch does not magically improve real-time performance. The kernel has to be fine-tuned for the system's real-time applications, which depends on their characteristics. There are no official guidelines, and recommendations are spread out in research articles, mailing lists, conferences, and news articles. The kernel's latencies are greatly affected by implementation and system load, making it challenging to provide general performance numbers. Configuring a real-time Linux system requires a deep understanding of the kernel and the characteristics of the system load.

Achieving temporal isolation on mixed-criticality systems is challenging, even with modern virtualization methods. Certified embedded hypervisors for safety-critical application domains experience the same interference problems caused by resource sharing between virtual machines as general-purpose hypervisors. The Linux kernel's isolation capabilities show promising results regarding their effectiveness in reducing system latencies when subjected to load.

Chapter 3

Methodology

This chapter discusses the methodology used to design and implement a Linux-based mixed-criticality system, covering the hardware setup and software implementation. The testing strategy and its goals are presented along with the Linux configuration.

3.1 Testing Environment

The thesis aims to implement a Linux-based mixed-criticality system and evaluate the Linux kernel with the PREEMPT_RT patch for real-time usage with state-of-the-art best practices. It is common for real-time systems to be a part of a larger distributed system where they receive and produce data for other sub-systems. Therefore, it is worth evaluating the Linux kernel with a focus on common high-speed transfer protocols. Ethernet and PCIe are chosen as they are commonly used and available on several modern hardware platforms. In addition, similar experiments are conducted for GPIO due to its simplicity to determine if there are large latency differences between simple and complex protocols.

Several studies use `cylictest`, the de facto standard measurement tool for real-time Linux, to measure the system latency [13], [18], [21]–[24]. This implies that the system under test (SUT) is used to measure the SUT. To ensure that the test results are not affected by applied stress to the SUT, dedicated hardware platforms are designed to produce and receive data over GPIO, Ethernet, and PCIe and, most importantly, measure the round-trip time (RTT).

The custom software is made open-source, and commercial off-the-shelf (COTS) hardware is used so that others can reproduce the experiments. The software referenced in this chapter is available on GitLab [86].

3.1.1 Hardware Setup

The integrated hardware setup consists of three independent hardware platforms. Two are used as dedicated measurement systems, while the last one runs Linux and is considered the SUT. Figure 3.1 shows the basic system architecture and highlights each hardware platform's operating system and the connected peripherals.

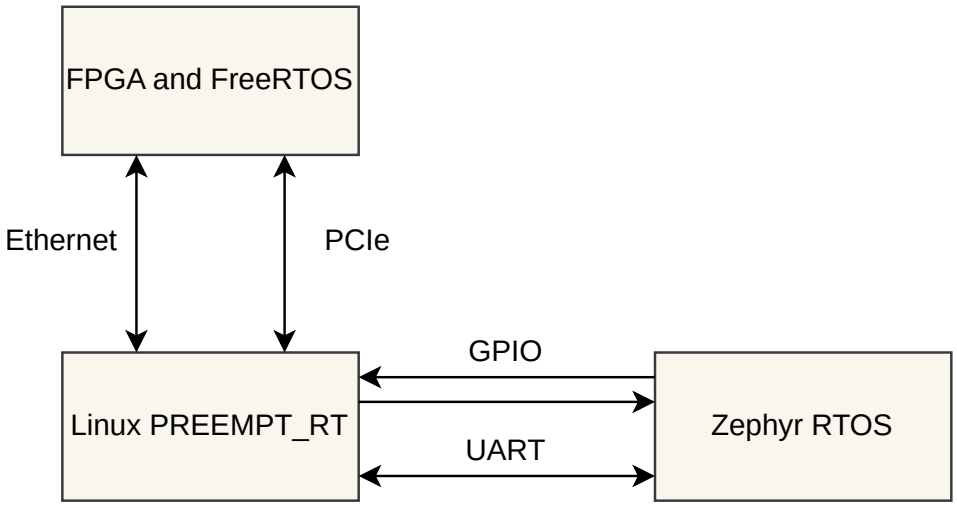


FIGURE 3.1: Hardware Architecture.

The Linux-based system is connected to the dedicated measurement systems via. GPIO, Ethernet, PCIe, and UART. Unlike GPIO, Ethernet, and PCIe, UART is not subject to testing and is only used to transfer testing parameters and results between the Linux-based system and the measurement system for GPIO.

The dedicated measurement system for GPIO was initially used as a proof-of-concept in the early stages of the thesis. This is why it is a separate measurement system and not embedded into the measurement system for Ethernet and PCIe.

The Linux-Based System

The Linux-based system uses the Texas Instruments SK-AM69 evaluation board, based on the AM69 AI vision processor. The processor consists of two 64-bit quad-core Arm Cortex-A72 microprocessor clusters and PCIe, Ethernet, and GPIO peripherals. The processor is marketed for smart vision camera applications [87]. An image of the evaluation board is shown in Figure 3.2.

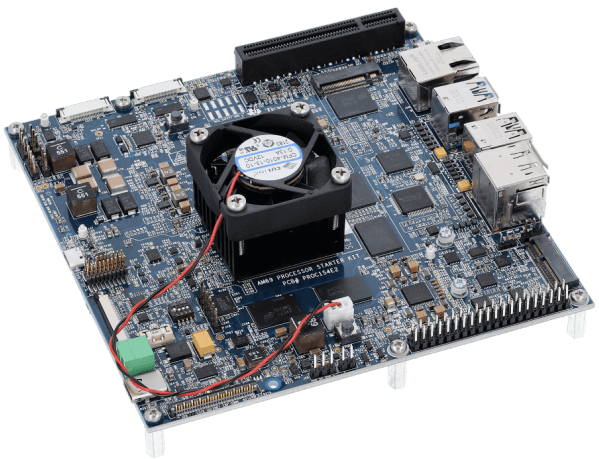


FIGURE 3.2: Texas Instruments SK-AM69 Evaluation Board [88].

The Linux kernel and `rootfs` are built using Buildroot [89] with Texas Instruments' Linux kernel fork [90]. The fork contains a branch for the 6.1 version of the Linux kernel with the PREEMPT_RT patch version 21.

The hardware platform is ideal for mixed-criticality system designs as it consists of two CPU clusters with individual shared last-level CPU caches. Having separate shared last-level CPU caches is beneficial, as sharing the last-level CPU cache between isolated partitions has negative implications [49].

Measurement System for Ethernet and PCIe

The measurement system for Ethernet and PCIe uses the Xilinx ZC706 Evaluation Kit, which is based on the Zynq-7000 SoC. An image of the evaluation kit is shown in Figure 3.3.

FIGURE 3.3: AMD Zynq 7000 SoC ZC706 Evaluation Kit [91].

The SoC features several peripherals, where the 10/100/1000 tri-speed Ethernet MAC and the eight-lane PCIe Gen2 are used for the experiments in this thesis. The SoC comprises an integrated Processing System (PS) and Programmable Logic (PL). The PS has two Arm Cortex-A9 CPUs and access to the Ethernet interface. The PCIe interface is only accessible from the PL. The PS and PL communication is done through a high-speed AXI interface [92].

The measurement system is connected to the Linux-based system via. Ethernet and PCIe. The measurement system is idle until it receives commands from the Linux-based system to start measurement runs for PCIe or Ethernet. The Ethernet interface is used during measurement runs for Ethernet and for receiving commands and test parameters from the Linux-based system. Test results and test statistics are also sent over the Ethernet interface. The PCIe interface is only used to transfer dummy data during measurement runs. Figure 3.4 shows the system FPGA architecture.

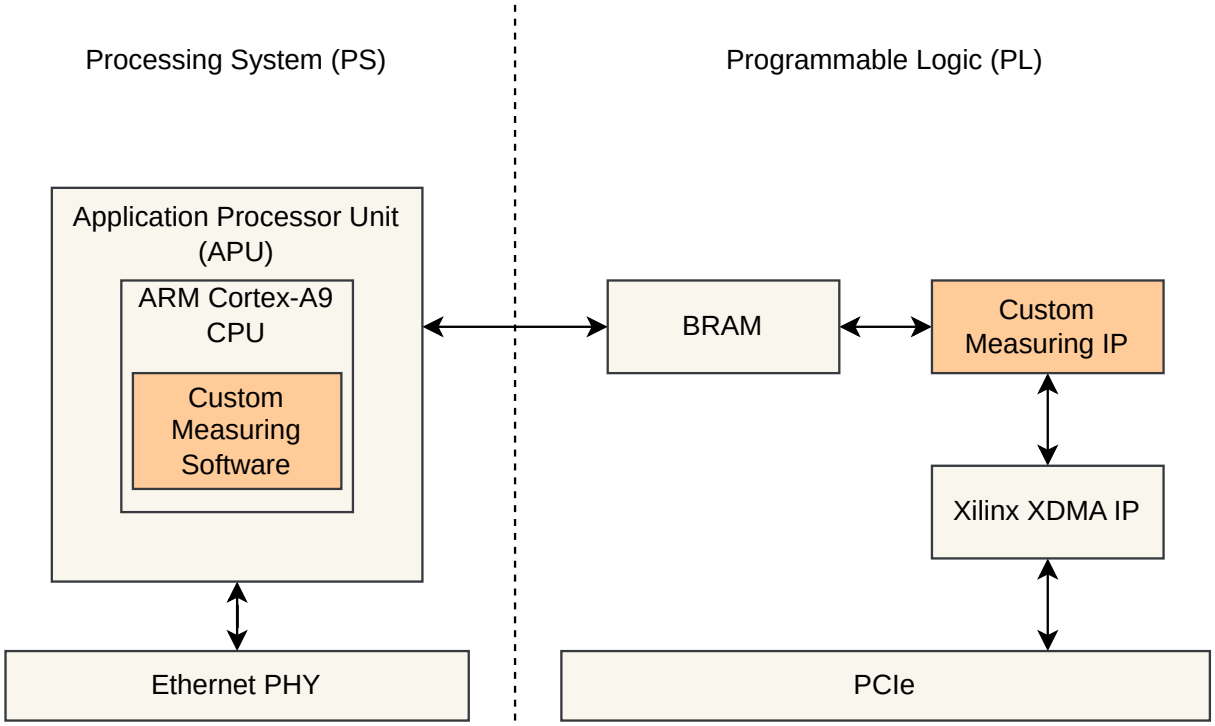


FIGURE 3.4: FPGA System Architecture.

The PS runs the FreeRTOS operating system [93] with custom software for handling measurement runs and communication with the PL. FreeRTOS was chosen due to Xilinx's official support for the operating system. Xilinx's customized Light-weight IP (LwIP) networking stack for embedded systems [94] is used for both TCP/IP and UDP, and Xilinx's standalone BRAM driver [95] is used for communication with the PL.

PS-PL communication is done by reading and writing to a shared memory map. The memory map is defined and implemented in the PL as a Block RAM (BRAM) and is accessible from the PS. The memory map is partitioned into 32-bit registers for commands, status messages, parameters, results, and statistics. The registers are defined in Table 3.1.

Address	Description	Valid Values
0x00	Command	START, STOP, LOAD_CONFIG
0x04	Status	MEASURING, LOAD_CONFIG_DONE, DONE
0x08	Number of Samples	0 - UINT32_MAX
0x0C	Period Duration in us	0 - UINT32_MAX
0x10	FPGA Version	0 - UINT32_MAX
0x14	Mode	TEST_RESULTS, TEST_STATISTICS
0x18	Total Time Upper	0 - UINT32_MAX
0x1C	Total Time Lower	0 - UINT32_MAX
0x20	Result: Highest	0 - UINT32_MAX
0x24	Result: Lowest	0 - UINT32_MAX
0x28	Result: Timeouts	0 - UINT32_MAX
0x2C ...	Result Data	0 - UINT32_MAX

TABLE 3.1: FPGA BRAM Registers.

The FPGA implementation uses a state-machine with the following simplified states: `IDLE`, `LOAD CONFIG`, `MEASURE`, `WRITE RESULTS`, `UPDATE RESULTS`, `WAIT FOR REMAINING PERIOD` and `DONE`. Due to the finite amount of RAM, it is not possible to store the full test results during longer measurement runs. In these scenarios, the FPGA can be configured to continuously update the lowest, highest, and average test statistics instead of storing the test results.

Figure 3.5 shows a simplified example of the FPGA state machine during a measurement run.

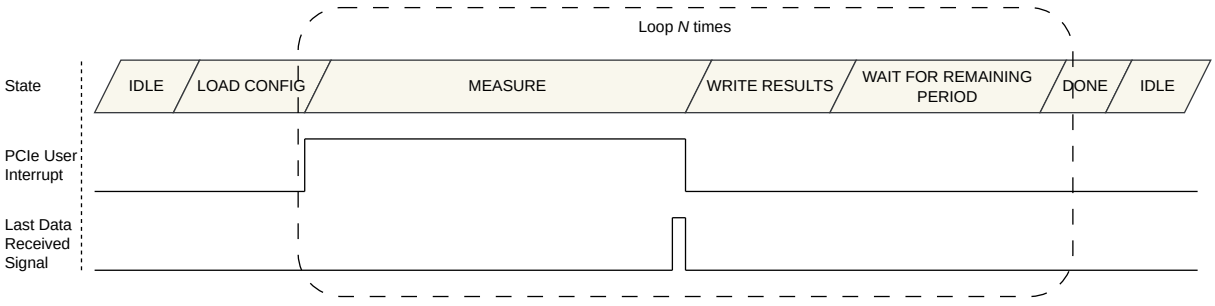


FIGURE 3.5: Simplified example of the FPGA state machine during a measurement run for PCIe.

The FPGA is idle until the PS writes the `Mode`, `Number of samples`, and `Period` configuration option to the shared memory map and commands the FPGA to load the configuration. When the FPGA has finished loading the configuration options, the PS commands it to start a measurement run. Each measurement starts by asserting the PCIe user interrupt and initiating a counter with a microsecond resolution. When the Linux-based system has reacted to the PCIe user interrupt and finished transferring

the dummy data over PCIe, the FPGA asserts the `Last Data Received Signal`. The counter is then stopped, and the results are either written to RAM, or the lowest, highest, and average test statistics are updated depending on the `Mode` configuration option. If the RTT is less than the period, the FPGA waits until the next period before repeating the measuring routine. After completing N measurements, the FPGA notifies the PS and switches to the `IDLE` state. The test results are then available in the shared memory map for the PS to retrieve and transfer to the Linux-based system.

Measurement System for GPIO

The measurement system for GPIO uses the Nordic Semiconductor nrF52 Development kit, which is based on the nRF52832 SoC. The SoC has an Arm Cortex-M4 CPU running at 64MHz, support for GPIO, and hardware timers capable of running at 16MHz [96]. The microcontroller runs ZephyrRTOS, and communication with the Linux-based system is done over UART. Figure 3.6 shows an image of the development kit.

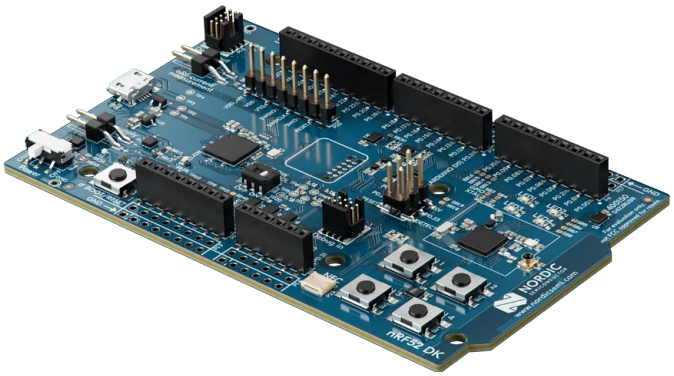


FIGURE 3.6: nRF52 DK [97]

The 16MHz hardware timers allow for sub-microsecond precision, which is more than sufficient for achieving a measuring resolution of one microsecond.

Two GPIOs are connected to the Linux-based system. One is configured as output, while the other is configured to trigger on a rising-edge input. During a measurement run, the output GPIO is asserted, signaling the Linux-based system to respond by asserting the measuring system's input GPIO. In the meantime, a hardware timer is used to measure the round-trip time.

The measurement system is idle until it is commanded by the Linux-based system to initiate a measurement run. The final test statistics are transferred to the Linux-based system at the end of a measurement run.

3.1.2 Measurement Software on the Linux-Based System

A command-line tool has been developed to orchestrate measurement runs with the dedicated measurement systems. The measurement tool is executed on the Linux-based system and commands the dedicated measurement systems to initiate measurement runs.

Testing parameters are set by the user and transferred to the respective dedicated measurement system before each measurement run. The testing parameters consist of the period between each message/signal in microseconds and the minimum testing time in seconds. In cases where the RTT is greater than the period, the total testing time will exceed the configured minimum. The dedicated measurement systems are low on RAM, which makes them incapable of storing the full test results during longer measurement runs. The tool has an option for longer measurement runs where the dedicated measurement systems produce test statistics instead of storing the raw test results. The test statistics consist of an average, highest, and lowest measured RTT.

The tool responds to incoming messages and signals from the dedicated measurement systems over Ethernet, PCIe, and GPIO. After a measurement run, the dedicated measurement systems will send the final test statistics or the raw test results.

Ethernet

The measurement tool's Ethernet implementation utilizes the native Linux networking stack. Although DPDK-based networking stack solutions have been proven to provide better results [75], [76], they have been dismissed due to not being supported by the Linux-based system's hardware platform [77].

A TCP/IP connection is established with the measurement system for Ethernet and PCIe and is used for transferring test parameters and test results. The same connection is used when testing the Linux kernel's TCP/IP implementation. An additional UDP connection is established when measuring the Linux kernel's UDP implementation.

PCIe

The PCIe implementation uses Xilinx's XDMA Linux driver for data transfers over PCIe [98]. The FPGA PCIe IP provided by Xilinx does not allow the FPGA to initiate C2H (Card to Host) transfers. Due to this design limitation, the Linux-based system must initiate both H2C (Host to Card) and C2H transfers. The FPGA can, however, initiate a PCIe user interrupt. Instead of measuring the time it takes for the FPGA to send and receive data over PCIe, the total round-trip time is determined by measuring the time interval between the FPGA generating a PCIe user interrupt and receiving data from the Linux-based system.

Xilinx's XDMA driver has been slightly modified with new custom `ioctl(2)` routines for receiving interrupts initiated by the FPGA and starting data transfers to the FPGA. The measurement tool uses the new `ioctl(2)` routines to catch PCIe user interrupts and respond with dummy data over PCIe.

When inserting the Xilinx XDMA device driver into the kernel, the module can be configured to be in either *polling* or *interrupt mode*. In both cases, initiating a data transfer from the host to FPGA will result in the calling process signaling another process to process the data transfer and wait for its completion before returning to user space.

In polling mode, the device driver creates per-CPU `kthreads` to handle data transfers. However, the `kthreads` are immediately put to sleep as a consequence of waiting on a `waitqueue`. When a data transfer is submitted, a `kthread` is signaled to wake up

and process the data transfer. Once the data transfer is processed, the `kthread` wakes up the task initiating the data transfer.

By configuring the module to be in interrupt mode, an interrupt is issued to process data transfers. The interrupt's top-half routine will schedule the bottom-half processing work onto the per-CPU kernel-global `workqueue`. When the data transfer has been completed, the task initiating the data transfer is woken up.

With both modes, the process that handles the data transfers is scheduled with the `SCHED_OTHER` policy with a nice value of 0. Therefore, data transfers over PCIe will not be prioritized higher than normal tasks.

Two custom bottom-half interrupt service routines have been developed to handle PCIe interrupts issued by the FPGA. One raises a flag and is suited for a polling-based implementation where the measurement tool repeatedly calls a `ioctl(2)` routine that checks if the flag has been raised and lowers it if it has. The other routine is suited for an interrupt-based implementation and wakes up a `waitqueue` that the measurement tool is blocked waiting on. The polling-based implementation checks the flag value every five microseconds.

GPIO

Communication with the GPIO measurement system is done over UART and is used to transfer testing parameters and statistics. Due to the measurement system's memory limitations, the system will only transfer test statistics, not raw test results.

The measurement tool utilizes the Linux userspace ABI found in `<linux/gpio.h>` to read and write to the GPIOs. Both polling and interrupt-based implementations are created.

The interrupt-based implementation will register a rising edge event on the `INPUT_GPIO` and wait for the event with the `poll(2)` system call. The polling-based implementation will instead actively read the `INPUT_GPIO` value every five microseconds.

The GPIO driver's interrupt routine is split into a top and bottom half. The bottom-half routine is explicitly configured as a threaded interrupt. Threaded interrupts will run with a `SCHED_FIFO` scheduling policy and a static priority of 50.

3.1.3 Linux Configuration

All experiments are performed on a Linux kernel version 6.1 with the PREEMPT_RT patch version 21. As of writing this thesis, the long-term Linux kernel version 6.6 has been released, but the kernel driver needed for interfacing the Xilinx PCIe XDMA FPGA IP has not been ported. Experiments comparing the vanilla Linux kernel against the PREEMPT_RT version are not considered for this thesis as they have already been widely researched. The PREEMPT_RT patch has a slightly negative effect on throughput in favor of lower latencies [11]–[13].

Partitioning the Linux-based system with an embedded hypervisor has been dismissed due to the lack of hardware support. Instead, the system is partitioned using the kernel's isolation techniques.

Kernel Configuration

The Linux kernel can be configured in numerous ways by enabling and disabling `Kconfig` options and kernel command-line parameters. Testing the effect of `Kconfig` options for real-time scenarios is important and an interesting research topic but out of scope for this thesis.

The kernel configuration used in the experiments is based on recommendations from real-time kernel developers [19], [60] and Texas Instruments' real-time kernel configuration [99].

Listing 3.1 describes the kernel command line parameters and some noteworthy kernel configuration options.

```
Kernel command line: isolcpus=nohz,domain,managed_irq,4-7 skew_tick=1
    nosoftlockup nowatchdog nohz_full=4-7 rcu_nocbs=4-7 rcu_nocb_poll
    cpuidle.off=1 cpufreq.off=1 processor.max_cstate=0 processor_idle.
    max_cstate=0

Noteworthy kernel configuration options:
CONFIG_PREEMPT_RT=y
CONFIG_NO_HZ_FULL=y
CONFIG_CPUSETS=y
CONFIG_CPU_FREQ=n
CONFIG_RT_GROUP_SCHED=n

CONFIG_GPIO_SYSFS=y

CONFIG_PCI=y
CONFIG_PCI_MSI=y
CONFIG_PCI_J721E_HOST=y
CONFIG_PCI_J721E=y
```

LISTING 3.1: Linux Kernel command-line parameters and noteworthy kernel configuration options.

Timer migration has been turned off with the `sysctl -w kernel.timer_migration=0` command. This ensures that timers will be armed on the same CPU as the applications requesting the timer. Otherwise, one might experience a scenario where CPU #0 is tasked to wake up an application on CPU #7.

The measurement tool running on the Linux-based system is configured with `mlockall(2)` with the `MCL_FUTURE` and `MCL_CURRENT` flag arguments to ensure that the application's future and current memory pages are locked into RAM.

CPU Partitioning

The Linux-based system has two quad-core Arm Cortex-A72 microprocessor clusters. Figure 3.7 represents the CPU partitioning and configuration.

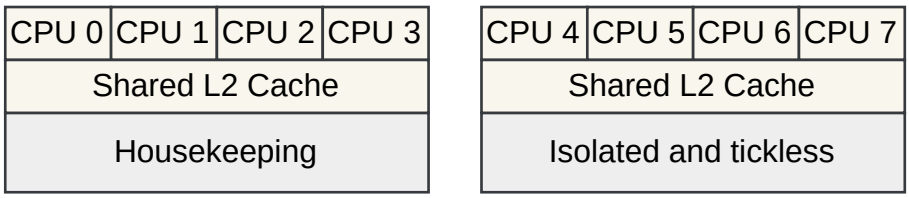


FIGURE 3.7: CPU Partitioning.

The first CPU cluster is used for housekeeping and non-critical tasks, while the second is reserved for real-time applications and is isolated from general scheduling algorithms and SMP balancing. Applications and managed interrupts must be explicitly configured to run on the isolated cores. The isolated cores are configured to be tickless when the cores are idle or have a single task running. Each CPU cluster has a shared L2 Cache, meaning the housekeeping and isolated CPU clusters do not share the last-level cache (LLC). A shared LLC for housekeeping and real-time applications is shown to cause extra latencies [23].

The CPU clusters are partitioned with the `isolcpus` kernel command-line parameter instead of the more complex `cgroup` alternative. CPU partitioning with `cgroups` is better suited for grouping multiple real-time applications. The thesis will only focus on running one real-time application.

The CPU partitioning environment allows for various configurations as the measurement tool and corresponding interrupts can be pinned to either cluster.

3.1.4 Stressors

The `stress-ng` and `hackbench` stressors are chosen to simulate workload and produce interference. They are executed on the housekeeping CPU cluster to simulate a mixed-criticality system. The stressors are scheduled with the `SCHED_OTHER` scheduling policy with a nice value of 0. The variety of stressors aims to highlight how well the kernel performs under different workloads.

`hackbench` (Version 2.50) simulates inter-process communication between threads and repeated setup and teardown. The stressor primarily affects the kernel's scheduler. The stressor's intensity is adjustable by changing the number of concurrent tasks. By default, UNIX sockets are used for inter-process communication, but the stressor can be configured to use pipes instead [33].

`stress-ng` (Version 0.15.07) has a large collection of stressors, with 242 out of 318 available for the Linux-based system's CPU architecture and Linux configuration. The number was determined by running all stressors and removing those that failed.

3.2 Testing Strategy

The initial experiments will use a set of parameters for the measurement tool, aiming to capture the following aspects.

- The effect of pinning interrupts and applications to isolated CPU cores.

- The performance difference between polling and interrupt-based implementations.
- Whether changes in the frequency of incoming messages and signals affect the round-trip time.

Table 3.2 shows the initial testing parameters for the measurement tool. The testing duration for each combination is set to a relatively short period of one minute due to the many combinations.

I = IRQ, P = Polling, WQ = Waitqueue, HC = Housekeeping cluster, IC = Isolated cluster

Protocol	Read Input	IRQ Pinning	App Pinning	Period	Duration
TCP/IP	I	HC, IC	HC, IC	1ms, 10ms, 1s	1m
UDP	I	HC, IC	HC, IC	1ms, 10ms, 1s	1m
PCIe	I+WQ,I+P	HC, IC	HC, IC	1ms, 10ms, 1s	1m
GPIO	I,P	HC	HC, IC	1ms, 10ms, 1s	1m

TABLE 3.2: Initial testing parameters.

The GPIO driver does not allow GPIO interrupts to be pinned to any other CPU core than CPU #0. However, the interrupts associated with the other protocols can be pinned to both CPU clusters. In all cases, the device driver used for PCIe will generate an interrupt for incoming data. The `I+WQ` label means that a `waitqueue` is used in the interrupt-handler to wake up the measurement tool, while the `I+P` label means that the measurement tool regularly polls a flag that is set by the interrupt-handler.

The measurement tool runs with the `SCHED_FIFO` scheduling policy with a static priority of 80. This ensures that the measurement tool has a higher priority than threaded interrupts and the stressor threads produced by `hackbench` and `stress-ng`.

Initial prototyping has found the `hackbench` stressor to be quite effective at stressing the system and generally produces worse latencies than `stress-ng`. The initial test runs will utilize four combinations of `hackbench` with two intensity levels and two IPC mechanisms. The intensity level is adjusted by changing the number of concurrent tasks. Ten groups and 20 file descriptors result in 400 concurrent tasks while increasing the number of file descriptors to 200 results in 4000 tasks. The number of loops has been set arbitrarily high, so the stressor runs throughout the testing period. The `hackbench` command line parameters for each combination are shown in Table 3.3. The labels are used in the test result graphs in section 4.

Label	Parameters
h1	hackbench –datasize 100 –groups 10 –fds 20 –loops 99999999
h2	hackbench –datasize 100 –groups 10 –fds 20 –loops 99999999 –pipe
h3	hackbench –datasize 100 –groups 10 –fds 200 –loops 99999999
h4	hackbench –datasize 100 –groups 10 –fds 200 –loops 99999999 –pipe

TABLE 3.3: Initial hackbench parameters.

3.2.1 Additional Measurement Runs

When the optimal configuration has been determined, additional measurement runs with `stress-ng` stressors are performed to show how well the configuration works against a wide range of stressors. The frequency of incoming signals and messages is configured to be 10 ms, and the test duration is one minute.

The collection of stress-ng stressors used in the measurement runs is shown in Listing B.1. The list has been compiled by extracting all available stress-ng stressors and removing those unsupported by the AArch64 CPU architecture, system configuration, or caused the system to break. The list of excluded stress-ng stressors is shown in Listing B.2.

3.2.2 Baseline Tests With Cyclictest

The `cyclictest` measurement tool is considered the de facto standard benchmarking tool for real-time Linux. It is widely used in research papers to provide an idea of the system's worst-case scheduling latency. Baseline tests with `cyclictest` are performed to produce comparable results.

Listing 3.2 shows the `cylictest` command used during the baseline tests.

```
cyclictest -a <7,3> -m -p 80 -l 100000 --secaligned --default-system --
    quiet
```

LISTING 3.2: Cyclictest command.

3.3 BPF Programs

Custom BPF programs have been developed to trace the kernel's inner workings. The eBPF technology allows custom code to be attached to kernel hooks during runtime.

The BPF programs use the BPF-side APIs provided by `libbpf`. To avoid the complications and runtime overhead that come with using the popular toolkit BCC (BPF Compiler Collection) [100], I have taken advantage of `libbpf`'s BPF CO-RE concept [101] that allows for BPF programs to be compiled once and executed anywhere.

3.3.1 Inter-Processor Interrupts

The inter-processor interrupts (IPI) program keeps track of all IPI calls with the origin processor, PID, IPI type, and destination processor. In cases where the IPI call happens in a hard interrupt context, the interrupt name is extracted instead of the task name. An example is shown in Listing 3.3.

CPU	PID	Name	IPI	CPU0	CPU1	CPU2	CPU3
0	0	arch_timer	IPI_CALL_FUNC		61		
0	15	ktimers/0	IPI_RESCHEDULE		1	9	8
0	20	rcuc/0	IPI_RESCHEDULE				1
0	184	kworker/0:3	IPI_RESCHEDULE		1	1	1
0	398	kworker/u16:2	IPI_RESCHEDULE			1	
0	401	kworker/u16:7	IPI_RESCHEDULE		1		1
1	0	arch_timer	IPI_CALL_FUNC			3	
1	28	ktimers/1	IPI_RESCHEDULE	1		1	2
2	0	arch_timer	IPI_CALL_FUNC		3		
2	37	ktimers/2	IPI_RESCHEDULE	7	4		1
2	197	kworker/2:1	IPI_RESCHEDULE		1		
2	401	kworker/u16:7	IPI_RESCHEDULE	1	1		
3	0	arch_timer	IPI_CALL_FUNC		10		
3	46	ktimers/3	IPI_RESCHEDULE	10	1	3	
3	398	kworker/u16:2	IPI_RESCHEDULE	1			

LISTING 3.3: IPI BPF Program example output.

3.3.2 Softirqs

Softirq is one of the kernel's bottom-half interrupt routine implementations and is reserved for the most time-critical interrupt handlers. Softirqs are usually raised by an interrupt's top-half routine and handled in the return from a hardware interrupt [102] or by the per-CPU `ksoftirqd` daemon [57]. Each softirq is placed into one of the following classifications: `HI`, `TIMER`, `NET_TX`, `NET_RX`, `BLOCK`, `IRQ_POLL`, `TASKLET`, `SCHED`, `HRTIMER`, and `RCU`.

With the PREEMPT_RT patch, softirqs with the `TIMER` or `HRTIMER` classification are handled by the per-CPU `ktimers` daemons that run with a higher priority than the `ksoftirqd` daemons. Softirqs that are raised in hard interrupt context are handled in the per-CPU `ksoftirqd` daemon, while softirqs raised in threaded interrupt handlers are handled after the threaded interrupt handler [103].

Two independent BPF programs have been developed to capture the raising and handling aspects of softirqs.

Softirq Raising

The Softirq Raising BPF Program creates an overview of the processes and hard interrupt handlers that raise softirqs, and statistics for the time between a softirq is raised and the beginning of its handling. An example is shown in Listing 3.4.

```
-------------------------------------------------------------------------
CPU  PID   NAME                VECTOR   COUNT    HIGHEST  LOWEST   AVERAGE (ns)
-------------------------------------------------------------------------
0    0     arch_timer          TIMER    528      9145     4800     5179
0    0     arch_timer          SCHED    2515     17680    4365     4611
0    0     arch_timer          HRTIMER  54       5270     4215     4503
0    217   irq/68-46000000     NET_RX   3        3740     1960     2610
0    218   irq/70-46000000     TIMER    1        23340    23340    23340
0    218   irq/70-46000000     NET_RX   21       2410     1600     1814
0    292   irq/218-2880000     SCHED    1        3295     3295     3295
1    0     IPI_CALL_FUNC       SCHED    2646     6980     4195     4719
1    0     arch_timer          TIMER    60       18600    5935     9079
1    0     arch_timer          SCHED    58       15805    7325     9304
1    0     arch_timer          HRTIMER  60       7820     4550     5351
1    2     kthreadd            SCHED    1        65625    65625    65625
2    0     IPI_CALL_FUNC       SCHED    9        8550     4410     5740
2    0     arch_timer          TIMER    84       10540    5820     6628
2    0     arch_timer          SCHED    6        10990    4775     8229
2    0     arch_timer          HRTIMER  50       6510     4905     5337
-------------------------------------------------------------------------
```

LISTING 3.4: Softirq Raise BPF Program example output.

Softirq Handling

The Softirq Handling BPF Program keeps track of the number of softirqs handled by each PID for each classification, with statistics for the duration in nanoseconds. An example is shown in Listing 3.5.

```
VECTOR    PID  NAME              COUNT    HIGHEST  LOWEST   AVERAGE (ns)
-------------------------------------------------------------------
TIMER     15   ktimers/0         64       30675    1430     2425
TIMER     28   ktimers/1         7        4950     2855     3736
TIMER     37   ktimers/2         15       6610     1965     3935
TIMER     46   ktimers/3         10       6520     2290     3629
NET_RX    216  irq/68-46000000   1        12630    12630    12630
NET_RX    217  irq/70-46000000   3        31710    8590     16676
SCHED     14   ksoftirqd/0       313      6645     1760     1954
SCHED     29   ksoftirqd/1       80       9645     690      2306
SCHED     38   ksoftirqd/2       2        2780     855      1817
SCHED     47   ksoftirqd/3       4        1995     1020     1403
SCHED     15   ktimers/0         1        1445     1445     1445
SCHED     28   ktimers/1         7        2350     970      1877
SCHED     37   ktimers/2         13       1290     900      1047
SCHED     46   ktimers/3         9        1425     880      1080
HRTIMER   15   ktimers/0         5        3220     2350     2792
HRTIMER   28   ktimers/1         2        3850     3330     3590
HRTIMER   37   ktimers/2         6        5180     2480     3200
HRTIMER   46   ktimers/3         8        3700     2350     2823
-------------------------------------------------------------------
```

LISTING 3.5: Softirq Handle BPF Program example output.

3.3.3 Hardirqs

The Hardirq BPF Program captures all hard interrupt routines with count and timing statistics. An example is shown in Listing 3.6.

```
------------------------------------------------------------
CPU  IRQ  NAME              COUNT   HIGHEST  LOWEST  AVERAGE (ns)
------------------------------------------------------------
0    1    IPI_RESCHEDULE    19      1160     325     485
0    11   arch_timer        5008    6465     710     1839
0    70   46000000.ethern   4       915      365     547
0    218  2880000.serial    6       5295     325     1170
1    1    IPI_RESCHEDULE    14      970      505     640
1    2    IPI_CALL_FUNC     82      1500     530     684
1    11   arch_timer        20      6905     765     2480
2    1    IPI_RESCHEDULE    13      1600     535     721
2    2    IPI_CALL_FUNC     5       1300     685     885
2    11   arch_timer        12      4210     845     3328
3    1    IPI_RESCHEDULE    20      1240     455     681
3    2    IPI_CALL_FUNC     1       1230     1230    1230
3    11   arch_timer        39      6325     770     2495
------------------------------------------------------------
```

LISTING 3.6: Hardirq BPF Program example output.

Chapter 4

Results and Discussion

This chapter presents the test results obtained by following the testing strategy outlined in section 3.2. The test results focus on the effects of pinning real-time applications and their associated interrupt routines to an isolated CPU cluster configured as tickless. It also examines how system loads of varying intensity and characteristics impact the GPIO, Ethernet, and PCIe implementations.

4.1 Baseline Test With Cyclictest

The `cyclictest` benchmarking tool effectively measures the system wake-up latency by repeatedly going to sleep and calculating the time difference between the expected and actual wake-up times. `cyclictest` is scheduled with a static priority of 80, and routinely woken up using `nanosleep(2)`.

The initial baseline test with `cyclictest` aims to capture the difference in scheduling latency, also called wake-up latency, for applications running on the isolated and housekeeping CPU clusters. The experiments are conducted with and without load on the housekeeping CPU cluster to highlight the impact of system load and the kernel's isolation capabilities.

Figure 4.1 illustrates the variation in system wake-up latency for the housekeeping and isolated CPU clusters when subjected to high load, as opposed to no load.

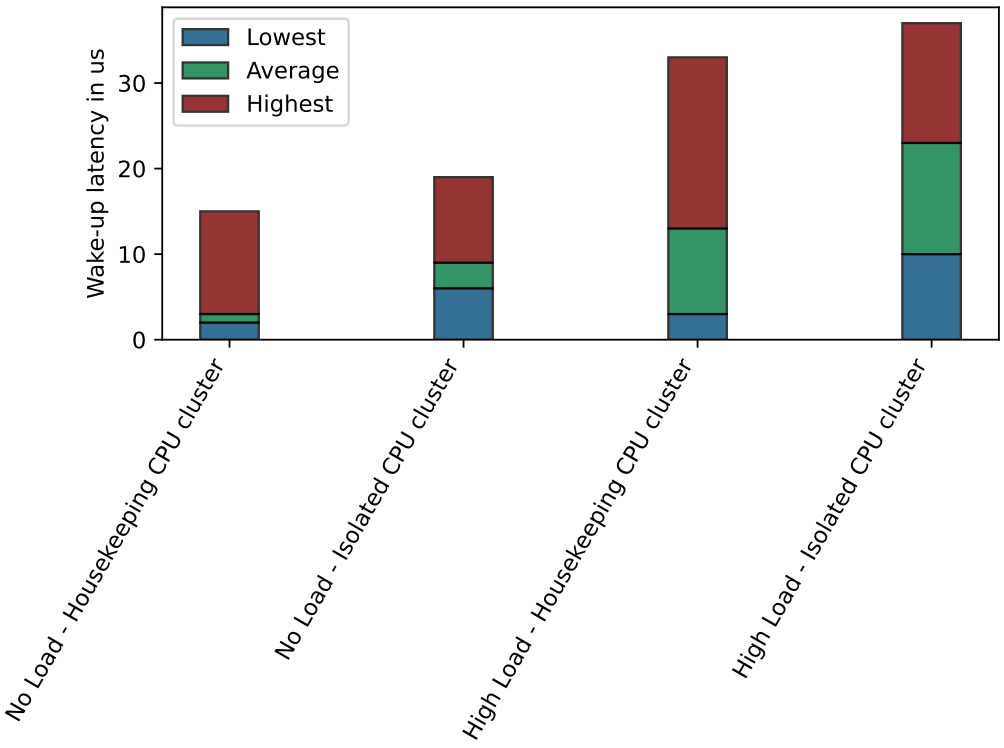


FIGURE 4.1: System wake-up time with timer migration enabled.

Applying a high load to the housekeeping CPU cluster increases the wake-up latency for both CPU clusters. Applications running on the isolated CPU cluster are not exempt from interference from the high load on the housekeeping CPU cluster.

The wake-up latency is higher on the isolated CPU cluster, regardless of the system load. This is a side effect of configuring the isolated CPU cluster as tickless. It takes longer to wake up an idle tickless CPU core as it includes inter-processor communication.

The default configuration of Linux has *timer migration* enabled. This is unfortunate for applications running on tickless CPU cores as the timer wake-up routine may be executed on a different CPU core. This leads to extra scheduling overhead, as the timer wake-up routine must issue a `rescheduling` inter-processor interrupt to the CPU core running the application. Disabling timer migration ensures that timers will be armed on the same CPU core as the application requesting the timer. Applications running on tickless CPU cores will, therefore, not offload timer work to a different CPU core. Figure 4.2 shows the same scenario as Figure 4.1, only now with timer migration disabled.

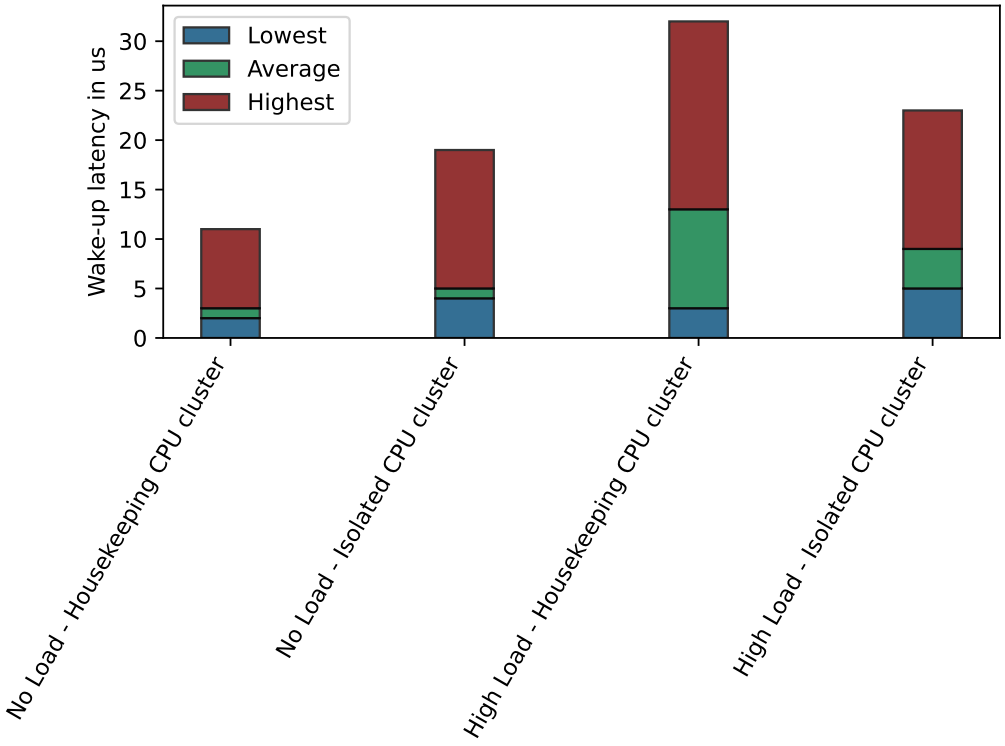


FIGURE 4.2: System wake-up time with timer migration disabled.

Similar to the results with timer migration enabled, the wake-up latency is higher when `cyclictest` is executed on the tickless isolated CPU cluster, and no load is applied to the housekeeping CPU cluster. However, the trend changes when system load is applied to the housekeeping CPU cluster. With timer migration disabled and subjected to high load, the highest measured and average wake-up latency is lower on the isolated CPU cluster.

Figure 4.3 shows a direct comparison of the highest measured wake-up latency from the previous results in Figure 4.1 and Figure 4.2, highlighting the effect of disabling timer migration.

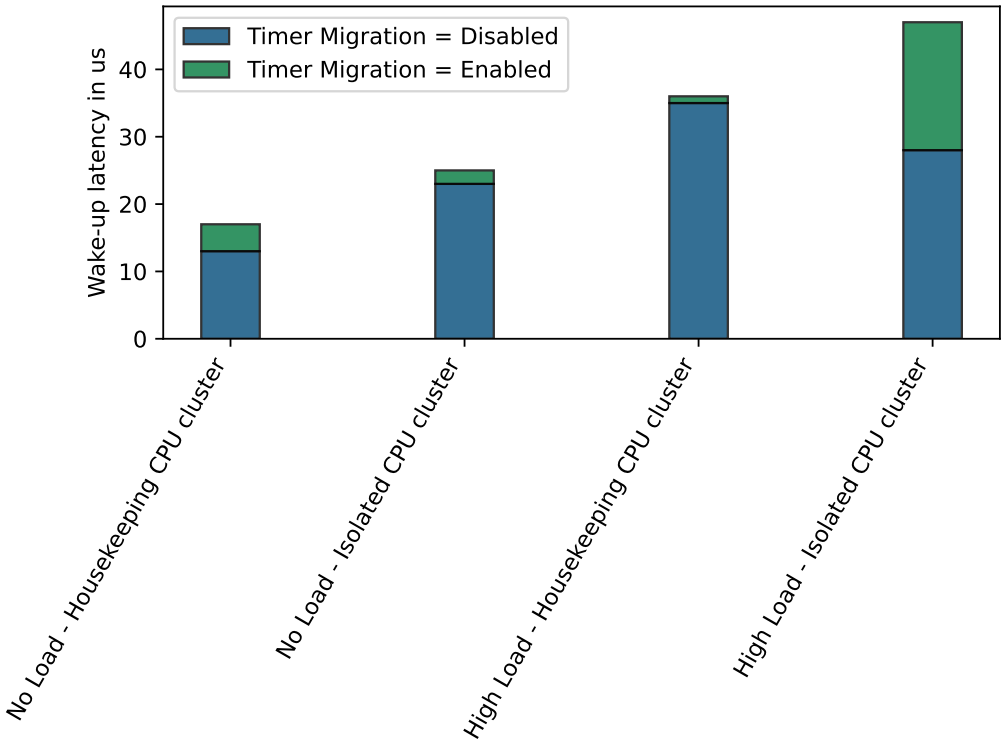


FIGURE 4.3: Comparison of system wake-up time with timer migration enabled and disabled.

Disabling timer migration has a positive impact in all scenarios, especially when a high load is applied to the housekeeping CPU cluster and `cyclictest` runs on the tickless configured isolated CPU cluster.

The results from the baseline tests with `cyclictest` indicate that the system's scheduling latency is greatly affected by general system load. Pinning timing-critical applications that wake up periodically, similar to `cyclictest`, to a tickless configured isolated CPU cluster is only beneficial when a general load is applied and timer migration is disabled.

4.2 GPIO

The measurement runs for GPIO focus on two different implementations regarding how the incoming GPIO signal is registered. One implementation is interrupt-based, where the measurement tool waits to be woken up by the GPIO interrupt handler. The other implementation is polling-based, polling the GPIO value every five microseconds.

The GPIO interrupt handler can not be pinned to any other CPU core than CPU #0. This implies that the interrupt handler will, in all cases, be executed on the housekeeping CPU cluster. However, the measurement tool can be pinned to a CPU core in both CPU clusters. The experiments and corresponding test results highlight the impact of assigning GPIO-based applications to an isolated CPU cluster and how stress impacts the Linux kernel's ability to respond to a GPIO signal.

4.2.1 GPIO Driver With No Modifications

The initial measurement runs use an unmodified version of the native Linux GPIO driver. The native Linux GPIO driver splits the GPIO interrupt handler into a top half and a threaded bottom half. The top-half routine is forcibly converted to a thread as the Linux kernel is configured with the PREEMPT_RT patch. Both interrupt halves run as threads with a `SCHED_FIFO` scheduling policy and a static priority of 50. The interrupt threads are scheduled on the housekeeping CPU cluster as the isolated CPU cluster is reserved against general SMP balancing. Even though the interrupt threads run on the same CPU cluster as the stressor threads, they have a higher scheduling priority.

Figure 4.4 shows the highest measured RTT results with the interrupt-based GPIO implementation. The results highlight the wake-up time difference between the two CPU clusters. All abbreviations used in the figures are introduced in section 3.2. In this example, `CPU=IC` means that the measuring tool is pinned to the isolated CPU cluster.

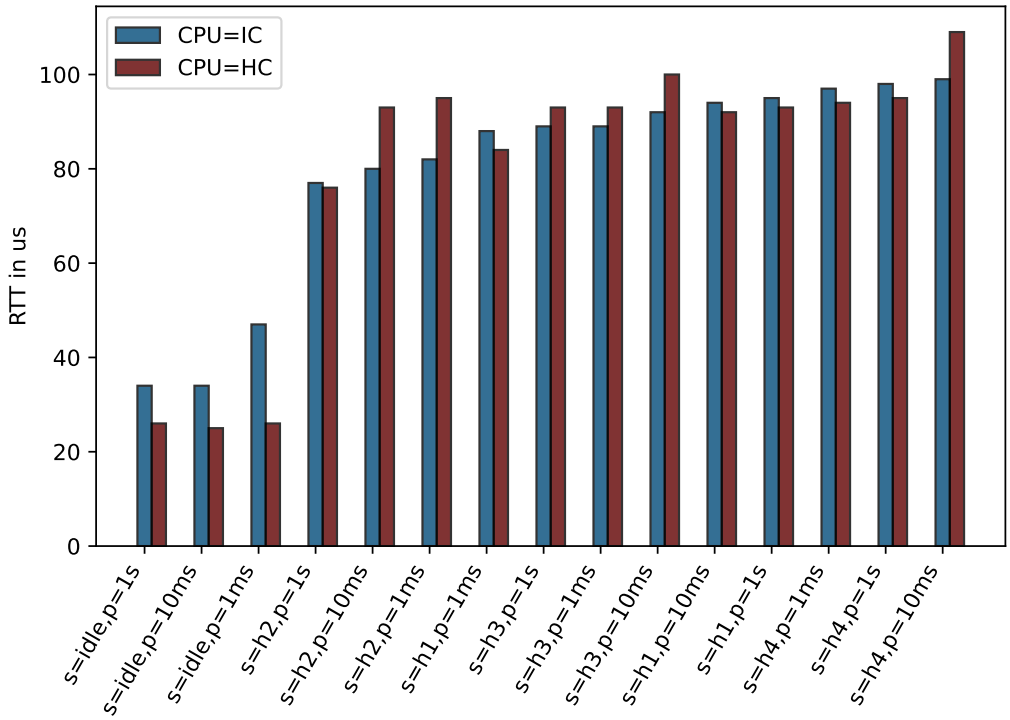


FIGURE 4.4: Highest measured RTT results for the interrupt-based GPIO implementation with an unmodified GPIO driver.

When no additional load is applied to the housekeeping CPU cluster, the RTT is noticeably higher when the measurement tool is pinned to the tickless isolated CPU cluster. The RTT increases when an additional load is introduced, but the differences between the two CPU clusters are evened out. This behavior complies with the results from the baseline tests with `cyclictest` described in section 4.1. There are no obvious differences between the period configurations.

Figure 4.5 shows the highest measured RTT results with the polling-based GPIO implementation.

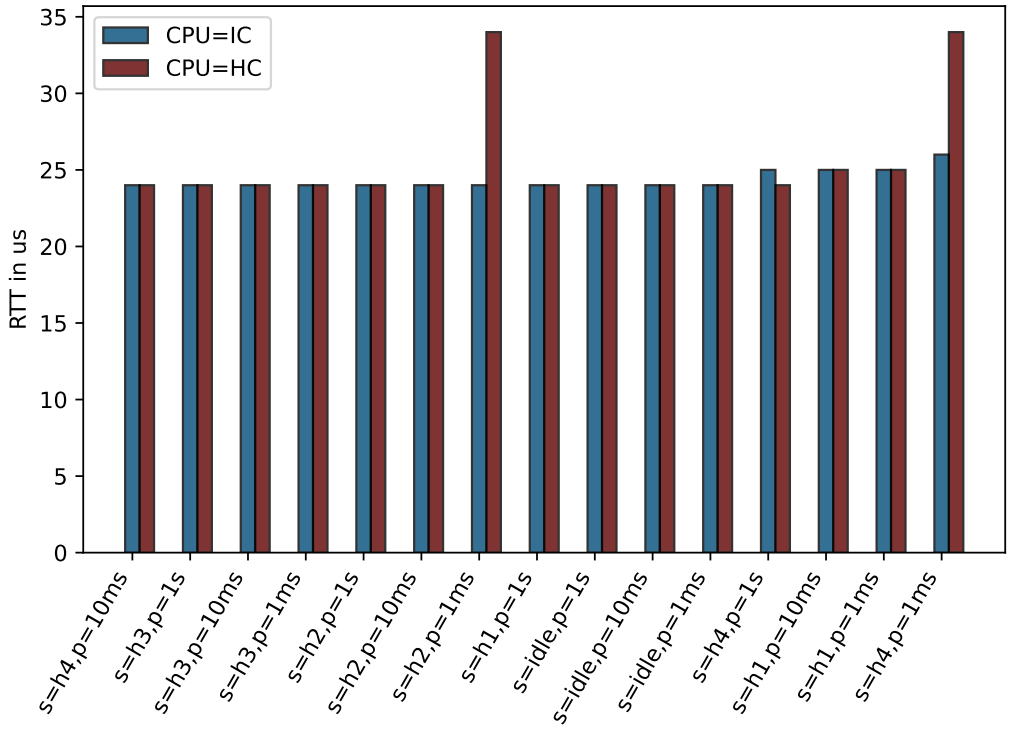


FIGURE 4.5: Highest measured RTT results for the polling-based GPIO implementation with an unmodified GPIO driver.

The results are relatively stable, with two spikes when the measurement tool is pinned to the housekeeping CPU cluster. The polling-based GPIO implementation does not seem to be heavily affected by the stressors. It is worth noting that both spikes occurred when the period was one millisecond. This is the lowest period used in the experiments.

4.2.2 Modified GPIO Driver With IRQF_NO_THREAD

Enabling the PREEMPT_RT patch will force all top-half interrupt routines to be converted to threads. This is favorable for high-priority applications as they will experience shorter interference from interrupt handlers. Converting interrupt handlers to threads comes with a reduction in throughput due to extra scheduling overhead. Interrupts can be requested to *not* be converted to threads with the `IRQF_NO_THREAD` flag. This configuration option is used for time-critical interrupts such as timers, per-CPU interrupts, and inter-processor interrupts [27].

The following measurement runs use a modified native Linux GPIO driver, where the interrupt halves have been merged into one, and the interrupt routine is executed immediately in a hard interrupt context.

Figure 4.6 shows the highest measured RTT with the interrupt-based GPIO implementation with the modified GPIO driver. The results are similar to the measurement runs for the interrupt-based GPIO implementation with the unmodified GPIO driver shown in Figure 4.4. When comparing the results, the highest measured RTT is slightly lower under heavy load with the modified GPIO driver. The results are, however, similar when the system is idle.

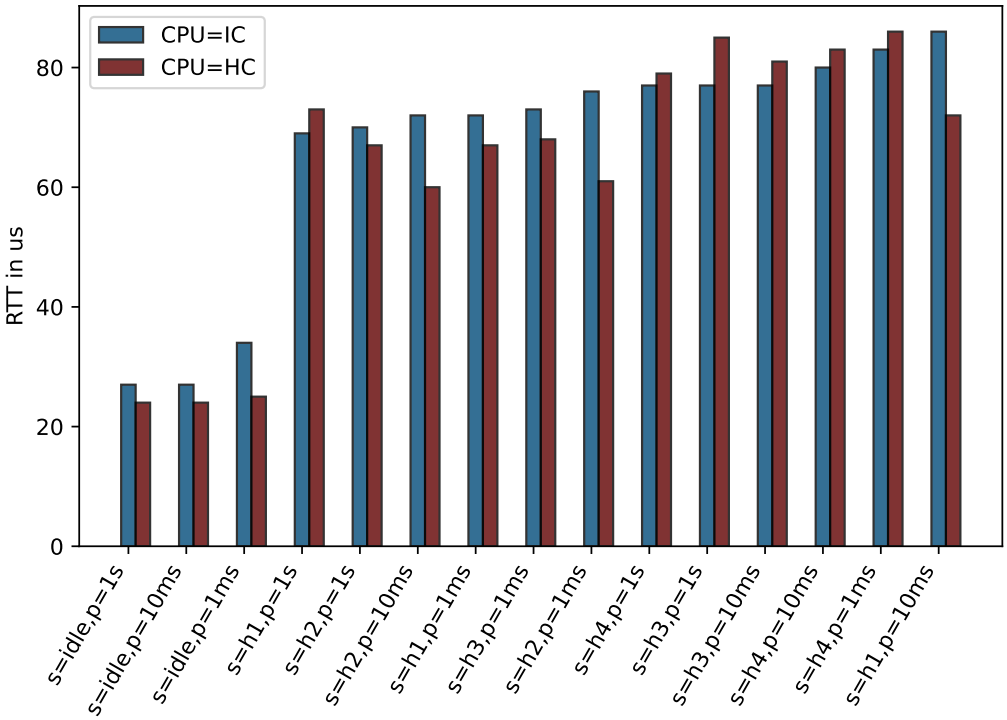


FIGURE 4.6: Highest measured RTT results for the interrupt-based GPIO implementation with a modified GPIO driver with `IRQF_NO_THREAD`.

Figure 4.7 shows the highest measured RTT with the polling-based GPIO implementation with the modified GPIO driver.

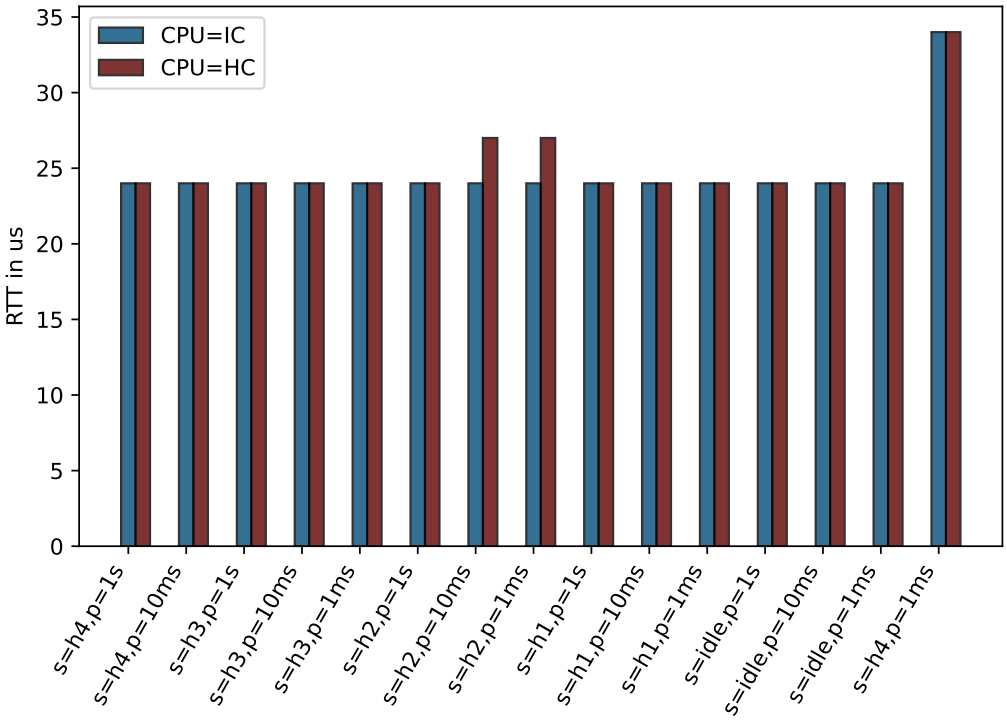


FIGURE 4.7: Highest measured RTT results for the polling-based GPIO implementation with a modified GPIO driver with `IRQF_NO_THREAD`.

The results seem unchanged compared to the previous run with the unmodified GPIO driver. However, they might be considered worse, as pinning the measurement tool to the isolated CPU cluster did not help mitigate a spike.

Modifying the native Linux GPIO driver results in slightly better RTT results for the interrupt-based GPIO implementation. The polling-based implementation is much less affected by the stressors than the interrupt-based implementation. However, it is not immune to spikes when configured with a short period and additional load is applied to the housekeeping CPU cluster. Pinning the measurement tool to the isolated CPU cluster has little impact on the polling-based implementation and varying positive impact on the interrupt-based implementation in terms of the highest measured RTT.

The average RTT results for the interrupt-based implementation with the unmodified GPIO driver are shown in Figure A.1, and the results with the modified GPIO driver are shown in Figure A.2. Pinning the measurement tool to the isolated CPU cluster is slightly preferable based on the average RTT. Similar results for the housekeeping and isolated CPU clusters show that the Linux scheduler does a great job honoring the measurement tool's high-priority scheduling policy.

4.2.3 The Impact of CPU Configuration and Load on the Cross-CPU Wake-Up Mechanism

During tracing sessions with `ftrace` and `KernelShark`, introduced in section 2.2.3, the duration of the GPIO interrupt routine varies greatly based on system load and CPU configuration. The last step of the GPIO interrupt routine is to wake up threads waiting on the GPIO event. A custom BPF program is created to trace the time taken to wake up other threads.

Previous results indicate that the wake-up time increases when a tickless CPU core is involved. It is also interesting to look at the time difference between waking up an application running on the same CPU core as the GPIO interrupt routine, compared to a different one.

Figure 4.8 shows the lowest, average, and highest measured time needed to complete the GPIO interrupt routine's wake-up procedure. The application waiting on the GPIO event is assigned to three different CPU cores with unique characteristics. CPU #0 is the same CPU core used to process the GPIO interrupt, CPU #1 is in the housekeeping CPU cluster, and CPU #7 is in the isolated CPU cluster and is configured as tickless.

Waking up CPU #1 and CPU #7 from CPU #0 requires an inter-processor interrupt, introducing extra overhead. To highlight the impact of load, the experiments are repeated with an additional load on the housekeeping CPU cluster.

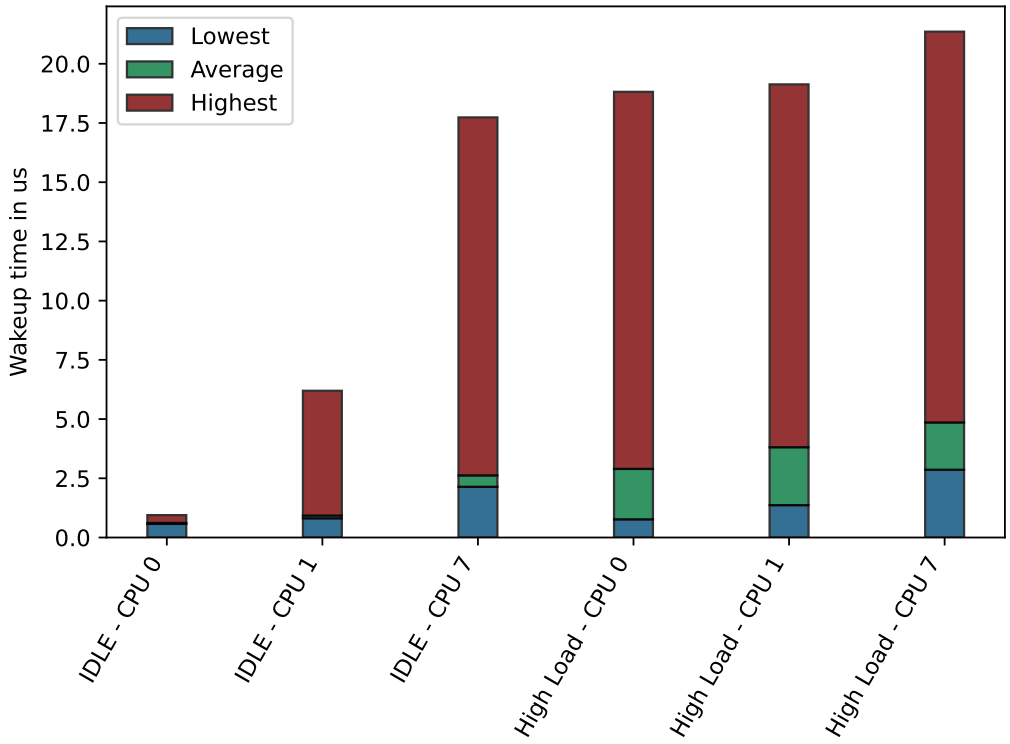


FIGURE 4.8: GPIO wake-up time with different CPU configurations and load level.

It takes longer to wake up applications running on a different CPU core than the GPIO interrupt, especially if the CPU core is configured as tickless. The differences in the highest measured time are evened out when the system is subjected to high load. However, the average wake-up time follows a similar trend regardless of load.

The custom IPI BPF Program introduced in section 3.3.1 can track the inter-processor interrupts that occur when the measurement tool runs on a different CPU core than the GPIO interrupt handler. Listing 4.1 shows the custom IPI BPF program's output during a measurement run with 150000 samples where the measurement tool was pinned to CPU #7.

```
CPU  PID    Name                IPI                  CPU0  ...  CPU7
------------------------------------------------------------------
0    0      gpio-input-risi     IPI_RESCHEDULE                 14986
```

LISTING 4.1: Output from the IPI BPF Program, focusing on IPIs generated by the GPIO interrupt routine when the measurement tool runs on a different CPU.

14986 rescheduling inter-processor interrupts for CPU #7 were initiated by the GPIO interrupt routine running on CPU #0. The remaining samples were implicitly covered by other rescheduling interrupts that were triggered by other means. The output from the BPF Program proves that inter-processor interrupts are used to wake up the measurement tool when a GPIO is triggered.

4.2.4 Additional Measurement Runs for GPIO

Additional measurement runs are performed with the best-performing GPIO configuration. The stressors used during the measurement runs are described in section 3.2.1.

Modifying the GPIO driver to force the GPIO interrupt routine to execute immediately in a hard interrupt context results in a slightly lower average and highest measured RTT for the interrupt-based GPIO implementation. However, choosing whether to pin the measurement tool to the isolated or housekeeping CPU cluster is not obvious for the interrupt-based GPIO implementation. Looking at the average RTT results in Figure A.2, pinning the measurement tool to the isolated CPU cluster has a slight advantage when subjected to high load. The highest measured RTT results in Figure 4.6 indicate that pinning the measurement tool to the housekeeping CPU cluster is more beneficial when idle or during the lesser aggressive stressor configurations. Pinning the measurement tool to the isolated CPU cluster is slightly better when the most aggressive stressors are active. The benefits of pinning the measurement tool to the isolated CPU cluster are thus worth taking advantage of.

The best-performing GPIO configuration for the interrupt and polling-based implementations uses the modified GPIO driver as described in section 4.2.2 with the measurement tool pinned to the isolated CPU cluster.

Figure 4.9 shows a histogram of the average and highest measured RTT for additional measurement runs with the interrupt-based GPIO implementation.

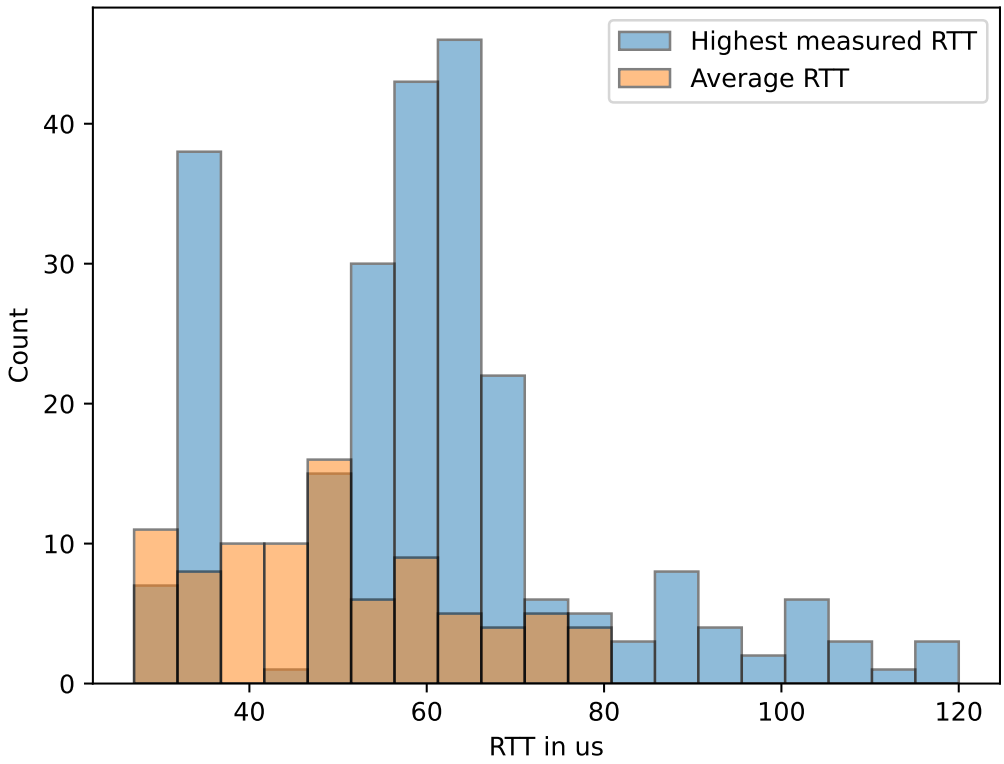


FIGURE 4.9: Histogram of the average and highest measured RTT for the additional measurement runs with the interrupt-based GPIO implementation.

As indicated by the results from the previous measurement runs with the `hackbench` stressor configurations, the interrupt-based GPIO implementation is heavily affected by system load. The majority of the round-trip time results with the `stress-ng` stressors are lower than 70 microseconds, while the results with the `hackbench` stressor are higher. This indicates that the interrupt-based GPIO implementation is more vulnerable to intense scheduling stress than general stress.

Figure 4.10 shows a histogram of the average and highest measured RTT for additional measurement runs with the polling-based GPIO implementation.

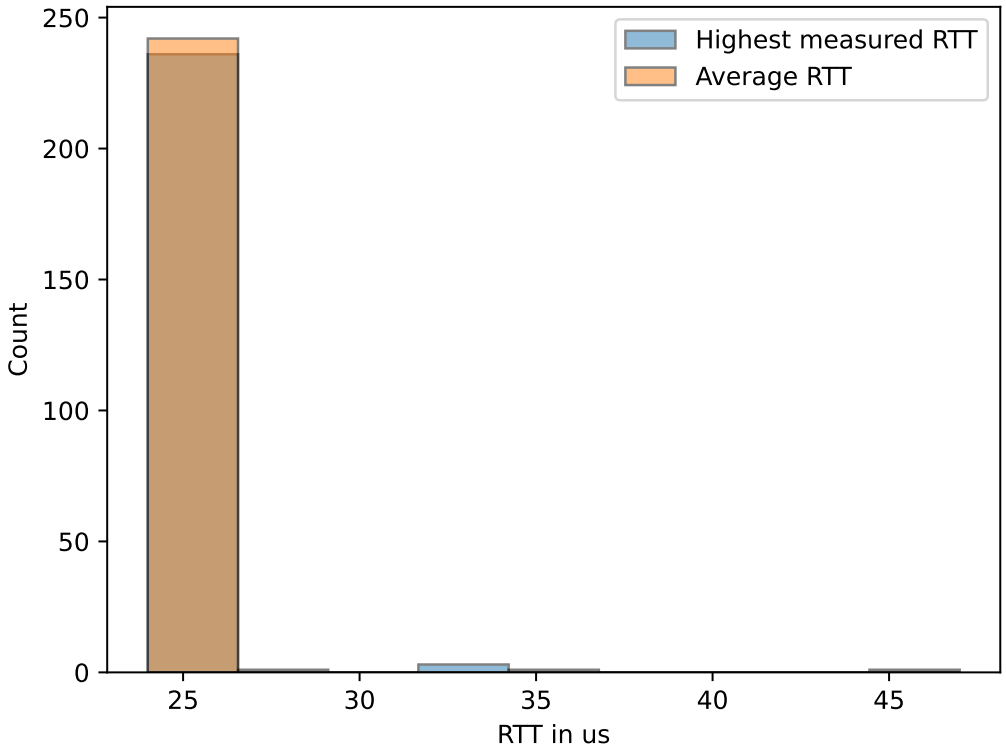


FIGURE 4.10: Histogram of the average and highest measured RTT for the additional measurement runs with the polling-based GPIO implementation.

The average RTT is remarkably stable and remains unaffected by all stressors. Although a few stressors impact the highest measured RTT, it remains relatively stable, with 98% of the results being under 30 microseconds.

By comparing the determinism of the interrupt and polling-based GPIO implementations, it is clear that the polling-based GPIO implementation is the better choice. The RTT results with the interrupt-based GPIO implementation vary greatly by the type of stressor, but a highest measured RTT of 120 microseconds is usable in many system designs. Although the polling-based implementation produces the best results, it requires more CPU time, negatively affecting other processes.

4.3 Ethernet

The measurement runs for Ethernet use the native Linux kernel networking stack with no modifications. Incoming and outgoing Ethernet packets trigger an interrupt that can be pinned to any given CPU core. The interrupt's bottom-half routine is implemented as a softirq, and the interrupt's top-half routine is responsible for raising the softirq at the end of its routine. As observed in the output from the custom Softirq BFP Program in Listing 3.5, most softirqs are handled in dedicated per-CPU `ksoftirqd` and `ktimers` processes. However, the Ethernet softirq is handled in the return of the interrupt's top-half routine as it is a threaded interrupt. This implies that the Ethernet softirq is processed quicker than other softirqs.

Each combination of stressor and period is run with the four possible CPU cluster pinning combinations. To reduce the amount figures, two primary colors with two shades are used to represent the four CPU cluster pinning combinations. Blue represents the Ethernet interrupt being pinned to the isolated CPU cluster, while red represents being pinned to the housekeeping CPU cluster. The darker shade represents the measurement tool running on the isolated CPU cluster and the lighter shade represents the housekeeping CPU cluster.

Measurement runs for both UDP and TCP/IP are performed to determine the different characteristics of the protocols.

4.3.1 Measurement Runs for UDP

Figure 4.11 shows the highest measured RTT results for the UDP implementation.

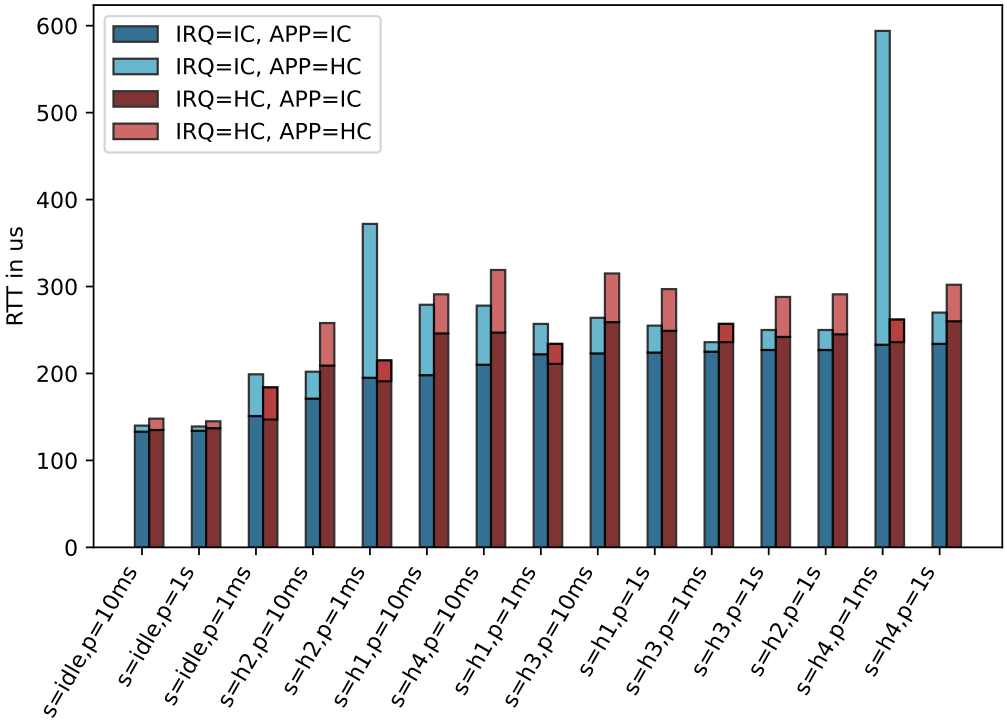


FIGURE 4.11: Highest measured RTT results for UDP.

The CPU pinning combinations represented by the darker shades are better in all cases. This means that pinning the Ethernet interrupts or measurement tool to the isolated CPU cluster has a beneficial effect regardless of load. This is opposed to the trend indicated by the baseline tests with `cyclictest` and the test results for the interrupt-based GPIO implementation. The UDP implementation benefits from running on a CPU core configured as tickless regardless of load.

A heavy load and a short period of one millisecond can result in spikes when the measurement tool is pinned to the housekeeping CPU cluster and the Ethernet interrupts are pinned to the isolated CPU cluster. It is not the top-half interrupt routines or the bottom-half softirq routines that cause the spikes, but rather the processing of the `read(2)` and `write(2)` system calls to the networking stack.

Excluding the combination that caused the two spikes, the highest measured RTT is similar for all periods and stressor configurations. CPU pinning has a less noticeable impact on an idle system but shows a noticeable beneficial impact during heavy loads and shorter periods. The average RTT shown in Figure 4.12 is much more affected by the period.

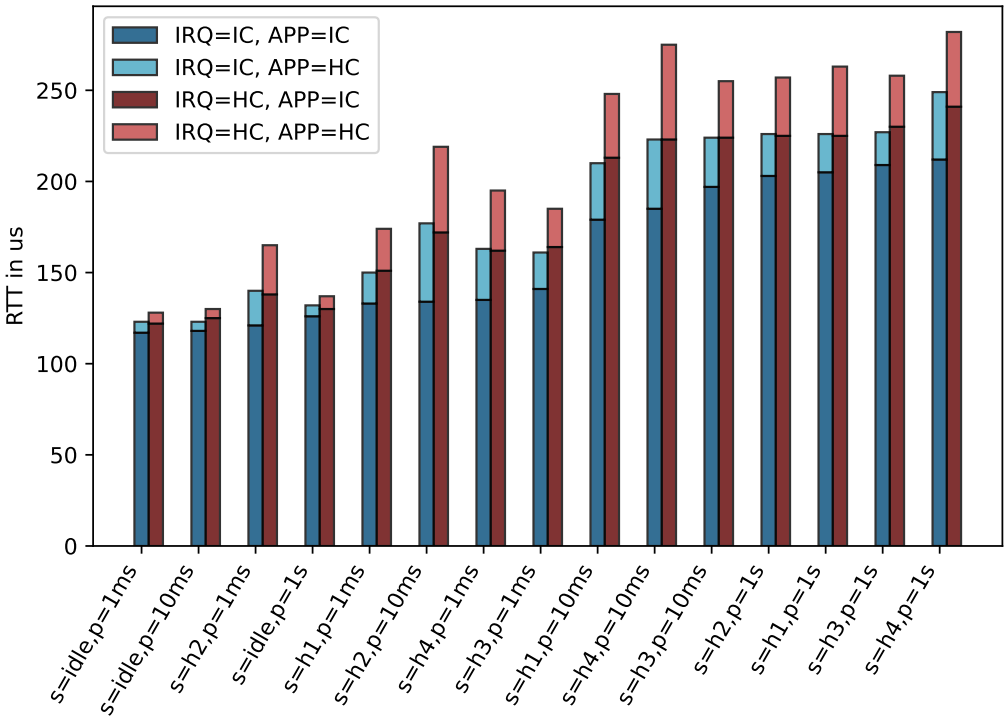


FIGURE 4.12: Average RTT results for UDP.

The four worst results are all from measurement runs with a one-second period. The average RTT is more affected by period than stressor intensity. Pinning the measurement tool and Ethernet interrupts to the isolated CPU cluster does not change the period trend. Overall, pinning the measurement tool and the corresponding Ethernet interrupts to the isolated CPU cluster is preferred in all cases.

The stressor configurations use either *sockets* or *pipes* for IPC between the stressor threads. Figure 4.13 shows the two IPC mechanisms' impact on the highest measured RTT for UDP with the measurement tool and Ethernet interrupts pinned to the isolated CPU cluster.

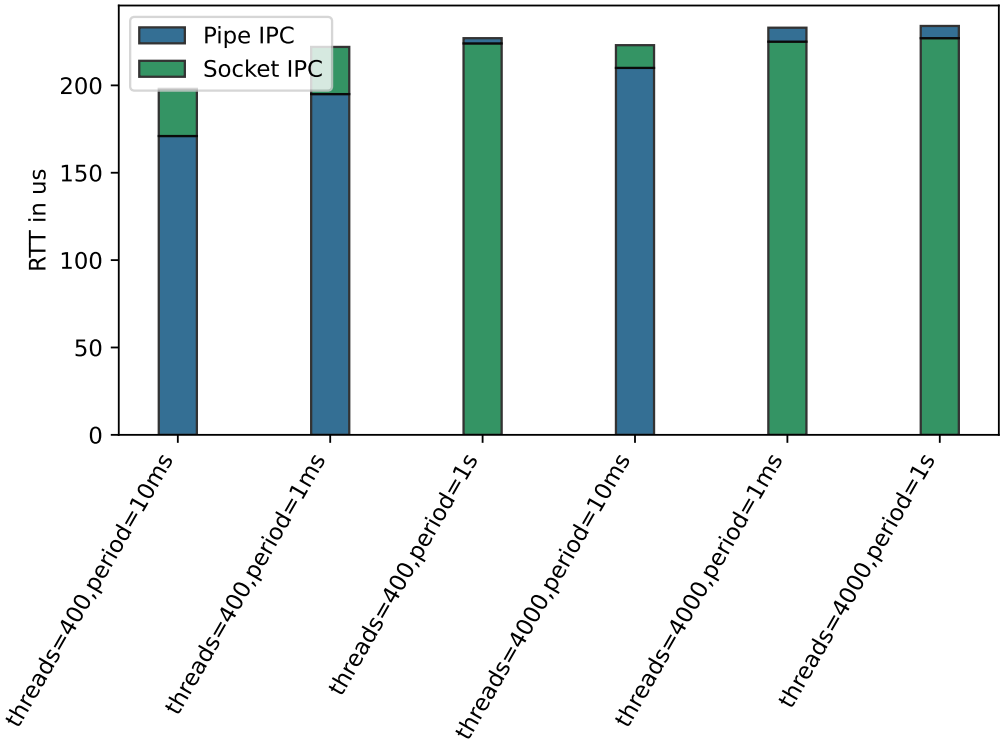


FIGURE 4.13: Comparison of the impact of socket and pipe-based IPC stress for UDP.

The highest measured RTT for UDP is similar for both IPC mechanisms. The pipe IPC mechanism is preferable when 400 stressor threads are applied to the housekeeping CPU cluster during shorter periods. However, when the number of stressor threads is ten-folded, sockets are preferred. Based on these results, predicting the best-performing IPC mechanism during other load types is challenging.

Additional Measurement Runs for UDP

Additional measurement runs are performed with the best-performing UDP configuration. The stressors used during the measurement runs are described in section 3.2.1.

The results for the highest measured RTT in Figure 4.11 and average RTT in Figure 4.12 indicate that pinning the Ethernet interrupt and measurement tool to the isolated CPU cluster is the best-performing configuration for the UDP implementation. This configuration is thus used in the additional measurement runs.

Figure 4.14 shows a histogram of the highest measured and average RTT for the additional measurement runs with the UDP implementation.

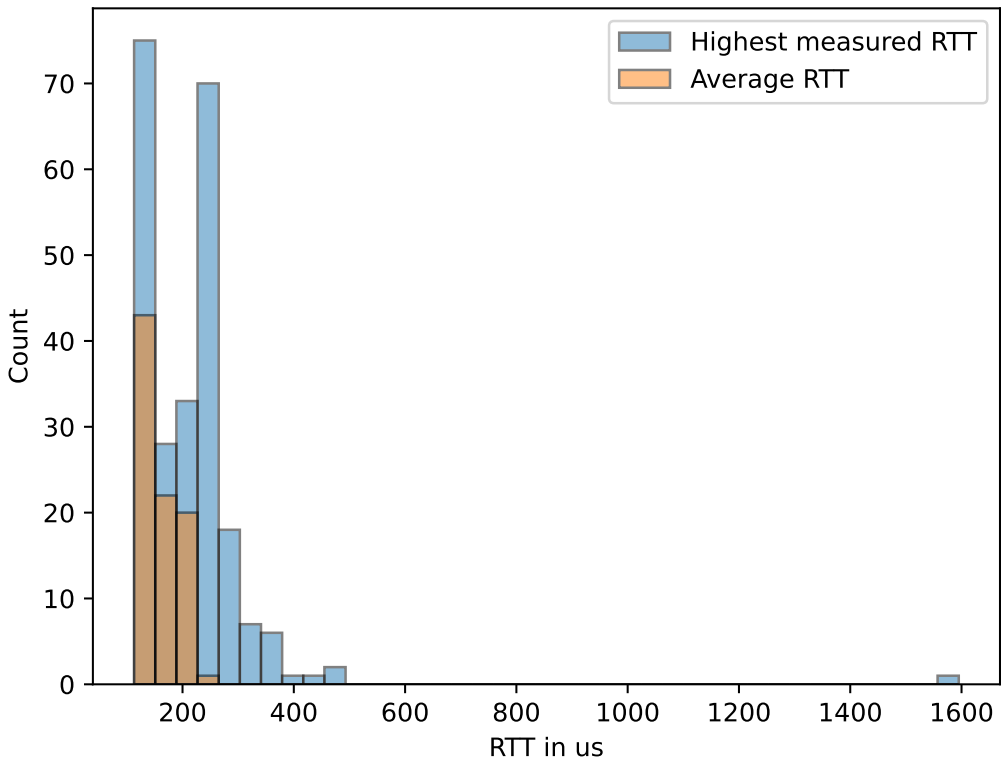


FIGURE 4.14: Histogram of the average and highest measured RTT for the additional measurement runs with the UDP implementation.

The Linux kernel's native UDP implementation is relatively stable but experiences a spike with the `stress-ng` stressor `procfs`. Appendix A.5 shows a detailed description of the worst-performing stressors. With 95% of the stressors, the UDP implementation achieves a round-trip time of under 350 microseconds.

More than half of the `stress-ng` stressors produced worse round-trip times than the most intense `hackbench` configuration with 4000 concurrent tasks. This indicates that other kernel characteristics affect the UDP implementation more than an increase in scheduling delay.

4.3.2 Measurement Runs for TCP/IP

The following test results focus on the Linux kernel's TCP/IP implementation. The TCP/IP protocol is more comprehensive than UDP, requiring more processing time. The TCP/IP protocol is similarly implemented as the UDP protocol, utilizing interrupts and softirqs for incoming and outgoing Ethernet packets.

The highest measured RTT results for the TCP/IP measurement runs are shown in Figure 4.15. Similar to the results for the UDP implementation described in section 4.3.1, pinning the measurement tool and corresponding Ethernet interrupts to the isolated CPU cluster provides better results. The TCP/IP implementation is vulnerable to spikes when the measurement tool is pinned to the housekeeping CPU cluster and the Ethernet interrupts are pinned to the isolated CPU cluster.

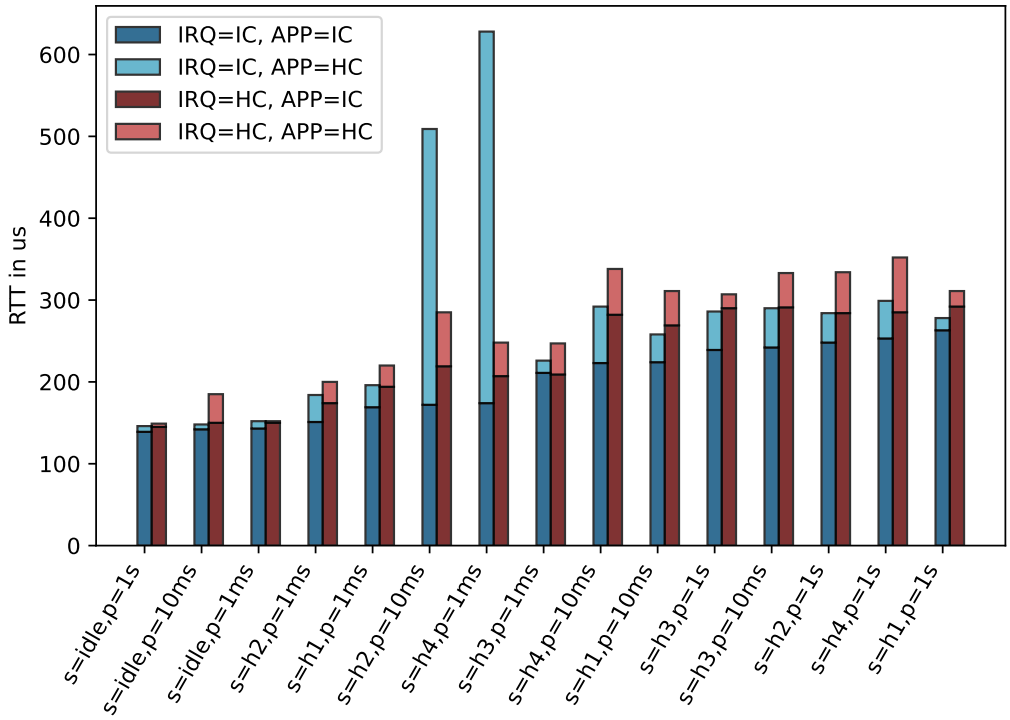


FIGURE 4.15: Highest measured RTT results for TCP/IP.

The average RTT results for the TCP/IP implementation are shown in Figure 4.16.

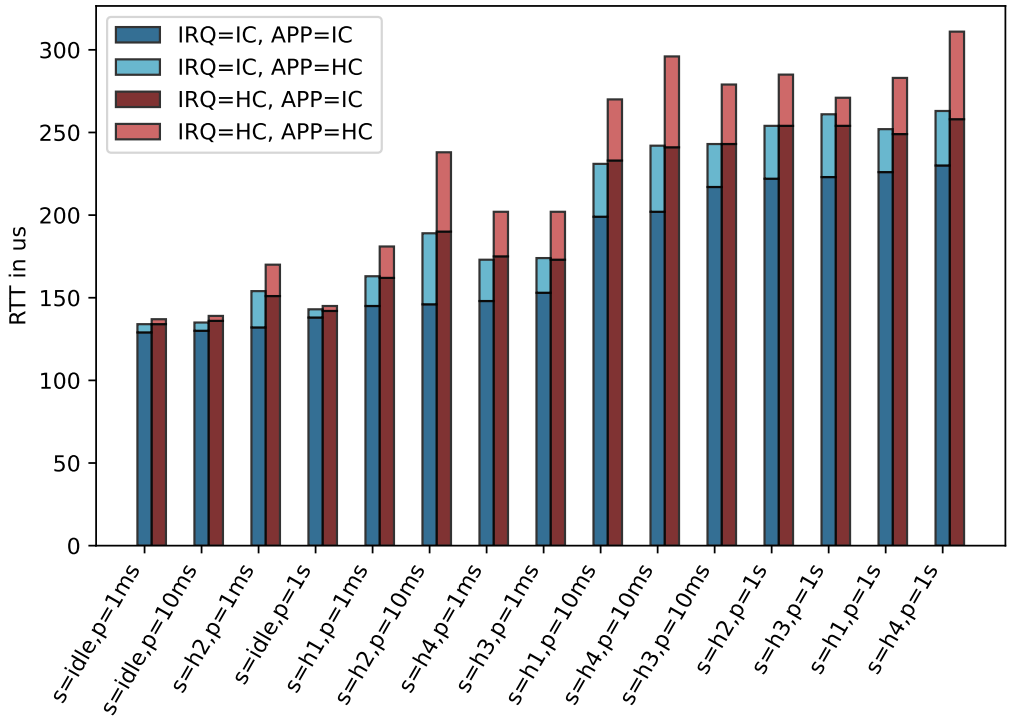


FIGURE 4.16: Average RTT results for TCP/IP.

The average RTT results for TCP/IP follow the same trend as the UDP implementation, where the RTT is more affected by period than stressor configuration.

Figure 4.17 highlights the difference in the highest measured RTT for TCP/IP when the stressor threads use sockets rather than pipes for IPC.

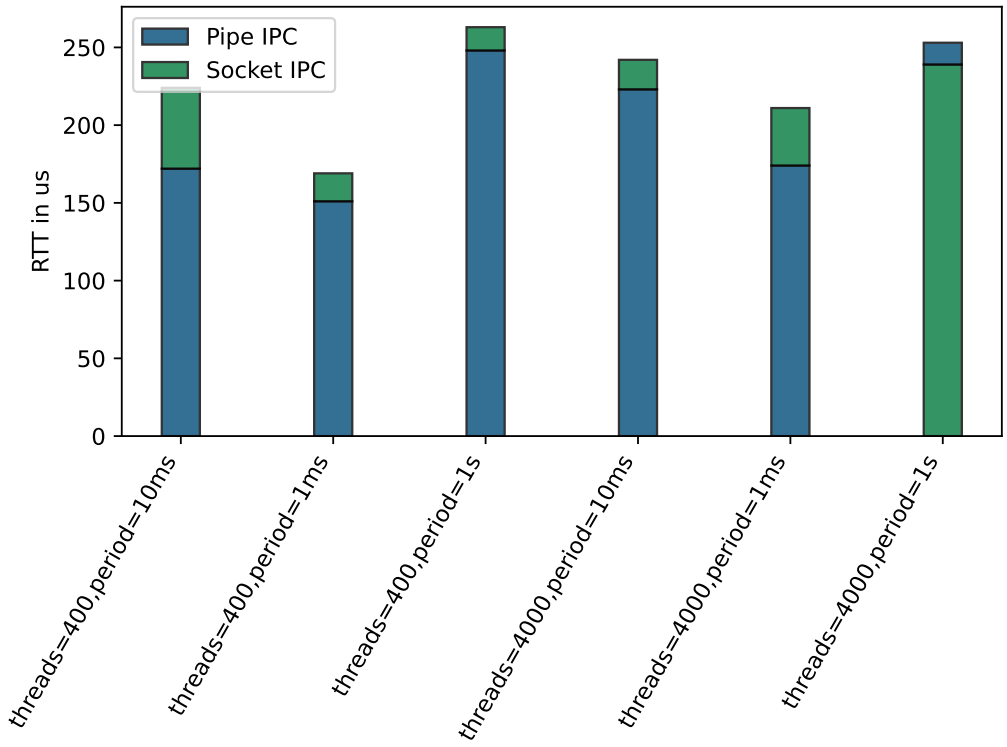


FIGURE 4.17: Comparison of the impact of socket and pipe-based IPC stress for TCP/IP.

In most cases, the highest measured RTT for TCP/IP is higher when the stressor threads use sockets instead of pipes for IPC. This differs from the UDP implementation, where it was unclear whether sockets or pipes were preferred. The choice of IPC mechanism for tasks running on the housekeeping CPU cluster affects the Linux kernel's TCP/IP implementation, even when assigning all network-related processing to the isolated CPU cluster.

Additional Measurement Runs for TCP/IP

Additional measurement runs are performed with the best-performing TCP/IP configuration. The stressors used during the measurement runs are described in section 3.2.1.

Like the UDP implementation, pinning the measurement tool and the Ethernet interrupt to the isolated CPU cluster is the best-performing configuration. The results for the additional measurement runs for TCP/IP are shown in Figure 4.18.

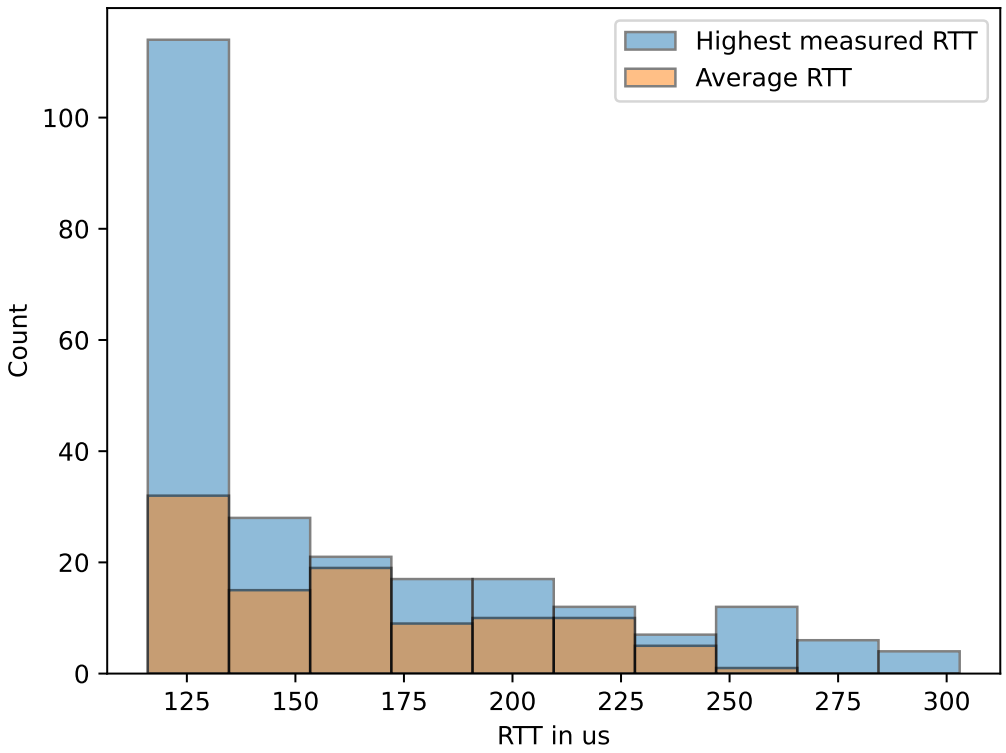


FIGURE 4.18: Histogram of the average and highest measured RTT for the additional measurement runs with the TCP/IP implementation.

The highest measured RTT is equal for half of the stressors, while the average RTT is more varied. Compared to the additional measurement run results for the UDP implementation, the TCP/IP implementation is more stable and did not experience any spikes. It is unknown what caused the spike to happen for the UDP implementation. As the TCP/IP implementation is similar to the UDP implementation, the TCP/IP implementation may probably experience a spike if the experiments were run for a longer duration.

4.3.3 The Impact of Heavy Load on Ethernet Softirqs

The native Linux kernel networking stack uses softirqs to process incoming and outgoing Ethernet packets. A small experiment is conducted to determine if there is any noticeable difference regarding softirq raising and handling during idle and heavy system load. The interrupts responsible for raising the Ethernet softirqs are pinned to a CPU in the isolated CPU cluster. The softirq is processed in the *return* of the interrupt, implying that it is processed on a CPU in the isolated CPU cluster. When simulating a system with a heavy load, the `hackbench` config *h4* is performed on the housekeeping CPU cluster.

The Softirq Raise BPF Program introduced in section 3.3.2 can track the time between a softirq is raised, and until it starts processing. Figure 4.19 shows the output from the Softirq Raise BPF Program, where the softirq handling incoming Ethernet packets is tracked during high and no load.

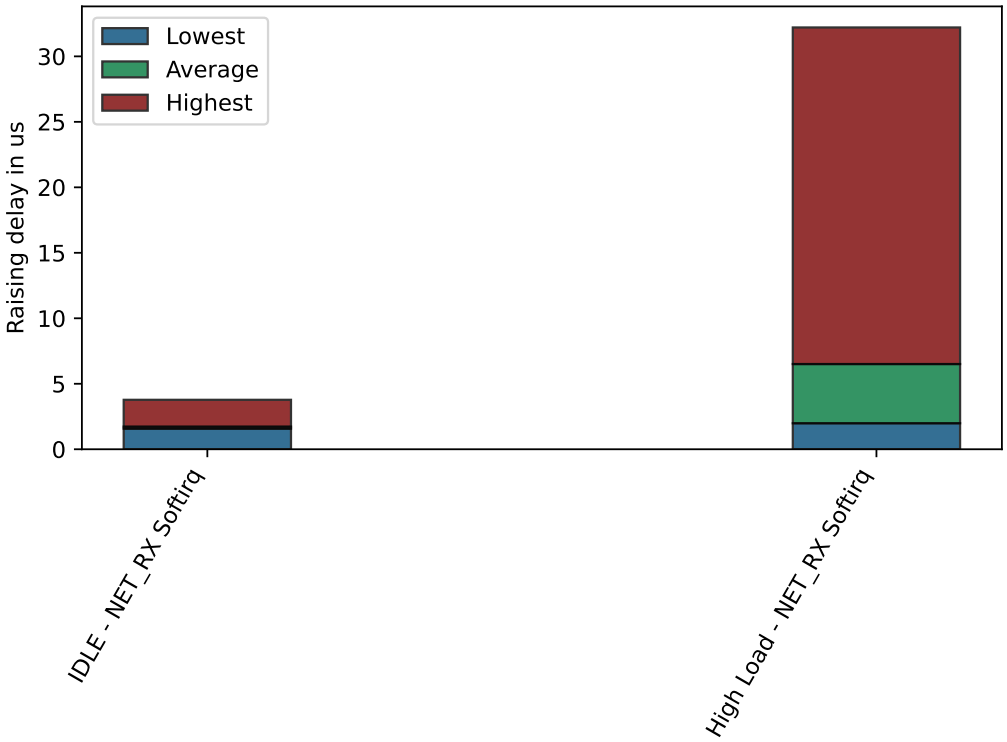


FIGURE 4.19: Difference in UDP softirq raising delay for a system under high and no load.

The difference in softirq raising delay is noticeable. The average delay doubles and the highest measured delay is ten times worse, even though the raising and handling of the softirq are processed on an isolated CPU core.

The softirq processing time is measured with the Softirq Handle BPF Program introduced in section 3.3.2. Figure 4.20 shows the processing time for the softirq handling incoming Ethernet packets during high and no load. The highest measured processing time is similar, but the average time has drastically increased.

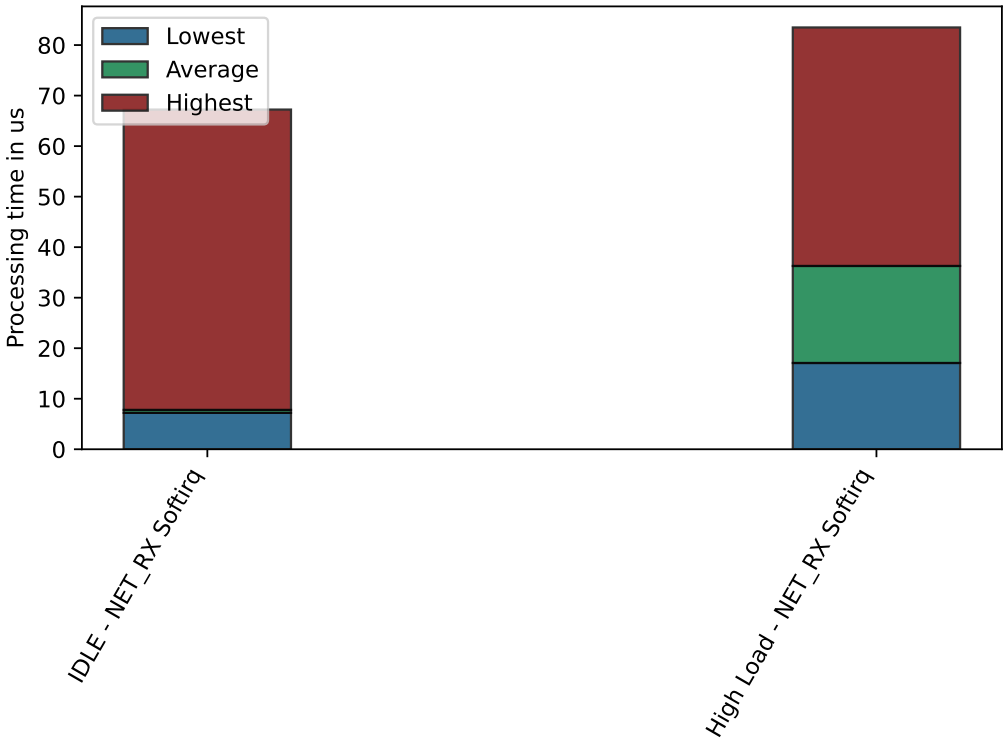


FIGURE 4.20: Difference in UDP softirq processing time for a system under high and no load.

The experiment was also conducted for the TCP/IP implementation. The results for the softirq raising delay are shown in Figure A.3, and the results for the softirq processing time are shown in Figure A.4. The results follow a similar trend to the UDP implementation but with a higher highest measured raising delay with an idle system.

4.4 PCIe

The Xilinx XDMA device driver plays an important role regarding the RTT results for PCIe. The device driver handles all PCIe data transfers to and from the PCIe and Ethernet measurement system. The inner workings of the device driver are discussed in section 3.1.2.

The device driver has two configuration options regarding how data transfers are processed. The default configuration option is called *Interrupt Mode* and issues an interrupt that adds data transferring work to a global kernel `workqueue`. The other option is called *Polling Mode* and wakes up a `kthread` that handles the data transferring work. With both configuration options, the data transferring work is offloaded to a different thread.

The measurement tool has two implementations regarding how it is notified of PCIe interrupts sent by the measurement system. The first implementation has the measurement tool sleeping on a kernel `waitqueue`, where the PCIe interrupt handler is responsible for waking up threads sleeping on the `waitqueue`. This implementation is referred to as the waitqueue-based implementation. The other implementation is polling-based, where the measurement tool polls a flag every five microseconds. The

flag is raised by the PCIe interrupt handler. This implementation is referred to as the polling-based implementation.

Measurement runs are performed for all four combinations to determine distinct performance differences.

4.4.1 Xilinx Driver With no Modifications

The initial measurement runs use an unmodified Xilinx XDMA device driver. Figure 4.21 shows the highest measured RTT results for the measurement tool's waitqueue-based implementation with an unmodified Xilinx XDMA device driver in polling mode.

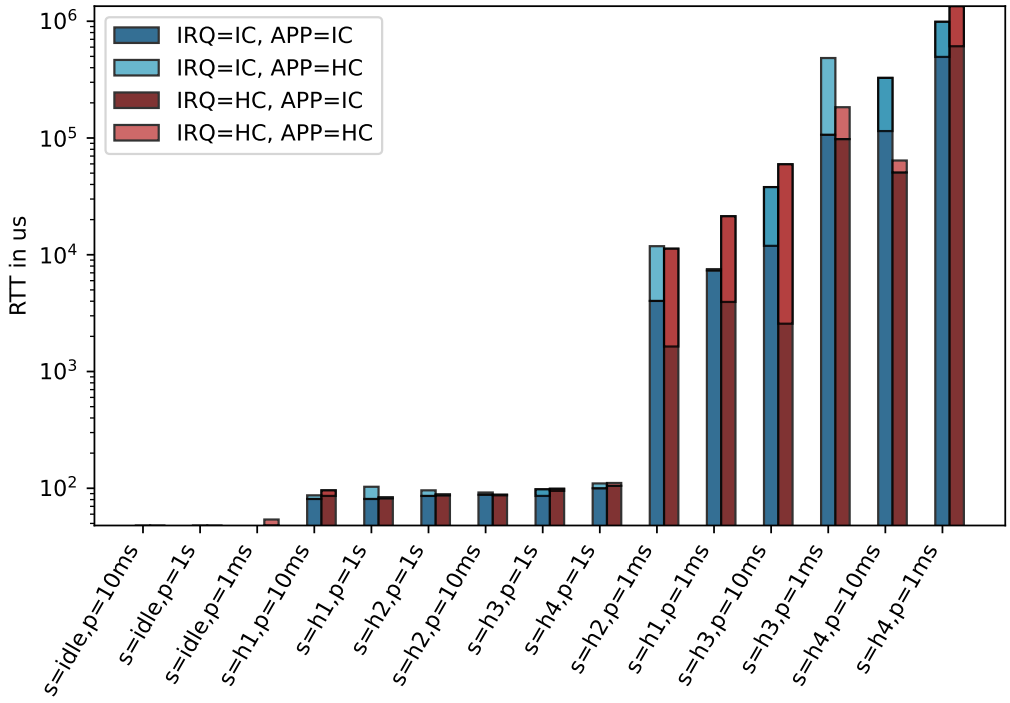


FIGURE 4.21: Highest measured RTT results for the PCIe waitqueue-based implementation with an unmodified Xilinx XDMA device driver in polling mode.

The stressors have a massive impact on the highest measured RTT regardless of the PCIe interrupt and the application's CPU cluster pinning configuration. The six worst-performing configurations have a period shorter than one second. While measurement runs with a one second period are relatively less affected by stress, they are still ten times worse when compared to an idle system. The same goes for measurement runs with a period of 10 ms with the *h1* and *h2* stressor configurations. The *h3* and *h4* stressor configurations spawn ten times more threads than the *h1* and *h2* stressor configurations. There is a direct correlation between the number of threads and the highest measured RTT.

The relatively good performance of the measurement runs with a one-second period is due to the high processing time needed to process a PCIe data transfer. When the Ethernet and PCIe measurement system receives data from the Linux-based system, it might issue a new PCIe interrupt before the Linux-based system is finished processing

a PCIe data transfer. The longer the period between each PCIe interrupt, the more time the Linux-based system has at its disposal to finish the data transfer and prepare itself for incoming PCIe interrupts. With the ten and one millisecond periods, the Linux-based system cannot finish the data transfer processing before the next PCIe interrupt, resulting in higher round trip times.

Figure 4.22 shows the highest measured RTT results for the measurement tool's polling-based implementation with an unmodified Xilinx XDMA device driver in polling mode.

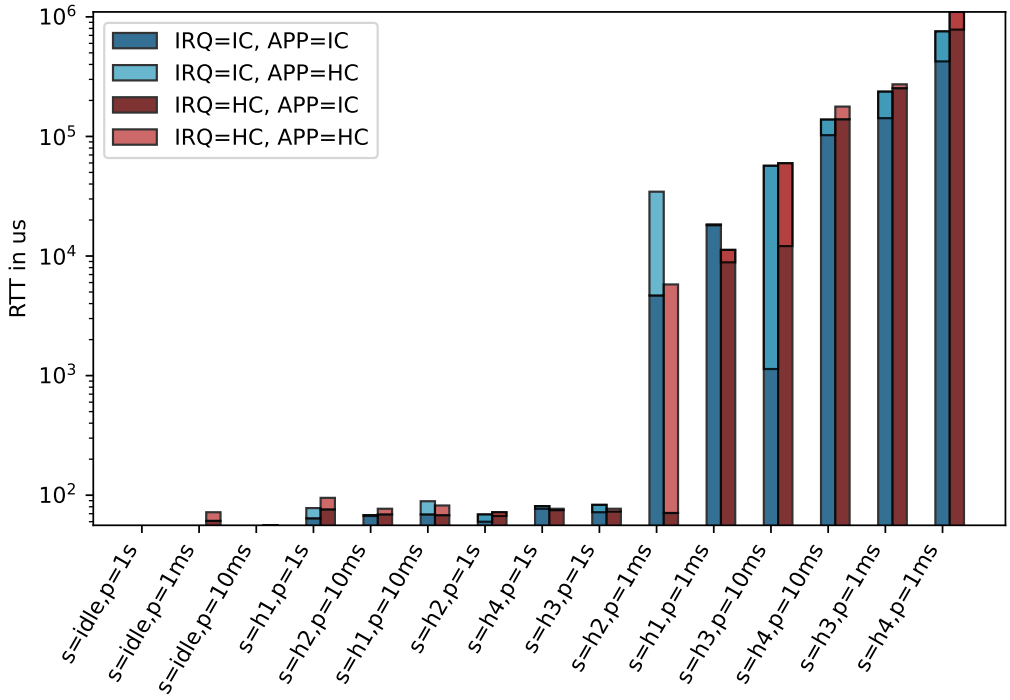


FIGURE 4.22: Highest measured RTT results for the PCIe polling-based implementation with an unmodified Xilinx XDMA device driver in polling mode.

The results are similar to the waitqueue-based implementation results in Figure 4.21. Both results show that the Xilinx XDMA device driver in polling mode provides unstable results, especially during short periods. As discussed in section 3.1.2, the polling mode implementation implies that the data transferring work is handled by a `kthread` with a scheduling policy of `SCHED_OTHER` and a nice value of 0. The threads spawned by the stressor program are scheduled with the same scheduling policy as the `kthread`. Regardless of the PCIe interrupt and measurement tool's high priority and CPU cluster pinning configurations, the data transferring will *always* be handled by a thread with low priority.

Although the Xilinx XDMA device driver initiates a `kthread` for each CPU core, including the CPU cores in the isolated CPU cluster, the data transfer work is always handled by the `kthread` running on CPU #1. CPU #1 is a part of the housekeeping CPU cluster, which implies that the `kthread` has to compete for CPU time with the stressor threads. Pinning the PCIe interrupt or measurement tool to the isolated CPU cluster does not affect which CPU core the `kthread` uses.

Figure 4.23 shows the highest measured RTT results for the measurement tool's waitqueue-based implementation with an unmodified Xilinx XDMA device driver in interrupt mode.

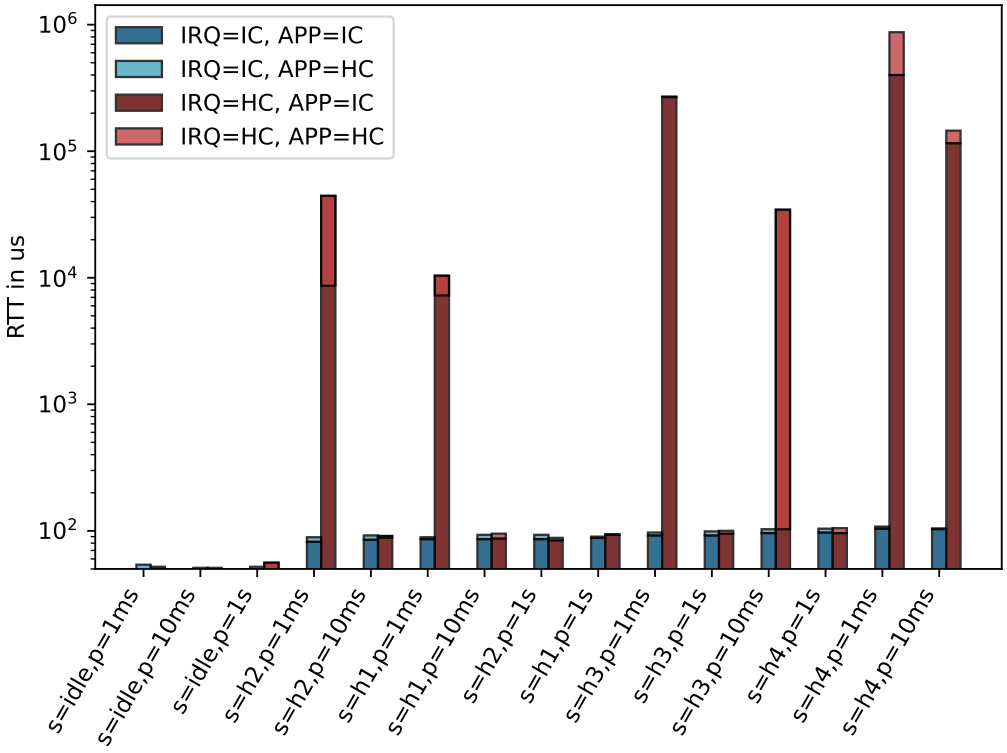


FIGURE 4.23: Highest measured RTT results for the PCIe waitqueue-based implementation with an unmodified Xilinx XDMA device driver in interrupt mode.

Switching the device driver configuration to interrupt mode has an immediate positive impact. Spikes only occur when the PCIe interrupt is pinned to the housekeeping CPU cluster. The measurement runs with the PCIe interrupt pinned to the isolated CPU cluster are not highly impacted by variations in period and stressor intensity. Compared to an idle system, there is, of course, a significant difference.

Figure 4.24 shows the highest measured RTT results for the measurement tool's polling-based implementation with an unmodified Xilinx XDMA device driver in interrupt mode.

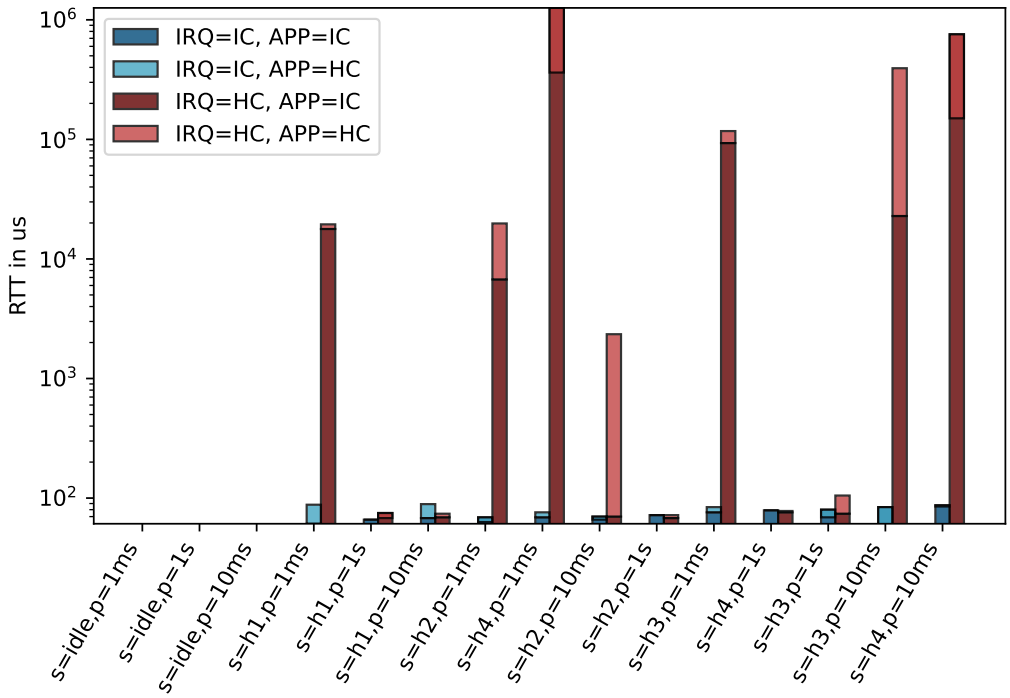


FIGURE 4.24: Highest measured RTT results for the PCIe polling-based implementation with an unmodified Xilinx XDMA device driver in interrupt mode.

Spikes occur when the PCIe interrupt is pinned to the housekeeping CPU cluster. Compared to the previous results in Figure 4.23 with the waitqueue-based implementation, the highest measured RTTs are lower with the polling-based implementation for the configurations that pin the PCIe interrupt to the isolated CPU cluster. This is due to the latencies induced by the kernel's cross-CPU wake-up mechanism. How system load and CPU configuration impact the cross-CPU wake-up latencies is discussed in section 4.2.3.

Pinning the PCIe interrupt to the isolated CPU cluster causes a great reduction in RTT when the period is short and the system is under a heavy load. As discussed in section 3.1.2, configuring the Xilinx XDMA device driver to be in interrupt mode means that the data transfer work is added to the per-CPU global kernel `workqueue`. The `workqueue` is handled by a `kworker` thread with a `SCHED_OTHER` scheduling policy and a nice value of 0. This is the same scheduling policy as the `kthread` used when the device driver is configured to be in polling mode and the threads spawned by the stressor program.

There is a huge performance difference between the device driver's two modes when the PCIe interrupt is pinned to the isolated CPU cluster. This is due to a difference in where the data transferring work is processed. When the device driver is configured to be in interrupt mode, the data transferring work is processed on the *same* CPU core as the PCIe interrupt. Pinning the PCIe interrupt to the isolated CPU cluster will then cause the data transferring work to be processed on the isolated CPU cluster. With the device driver's polling mode configuration, the data transferring work is always processed on the housekeeping CPU cluster regardless of the CPU pinning configuration, meaning that it has to compete for CPU time with hundreds of threads spawned by the stressor program.

4.4.2 Modified Xilinx Driver Revision 1

The previous results indicated that moving the task responsible for processing the data transfers to a CPU core within the isolated CPU cluster provides better results. The Xilinx XDMA device driver has been modified to pin the `kthread` in polling mode and `kworker` in interrupt mode to a CPU core in the isolated CPU cluster. Figure 4.25 shows the highest measured RTT results with the device driver in polling mode and the measurement tool's waitqueue-based implementation.

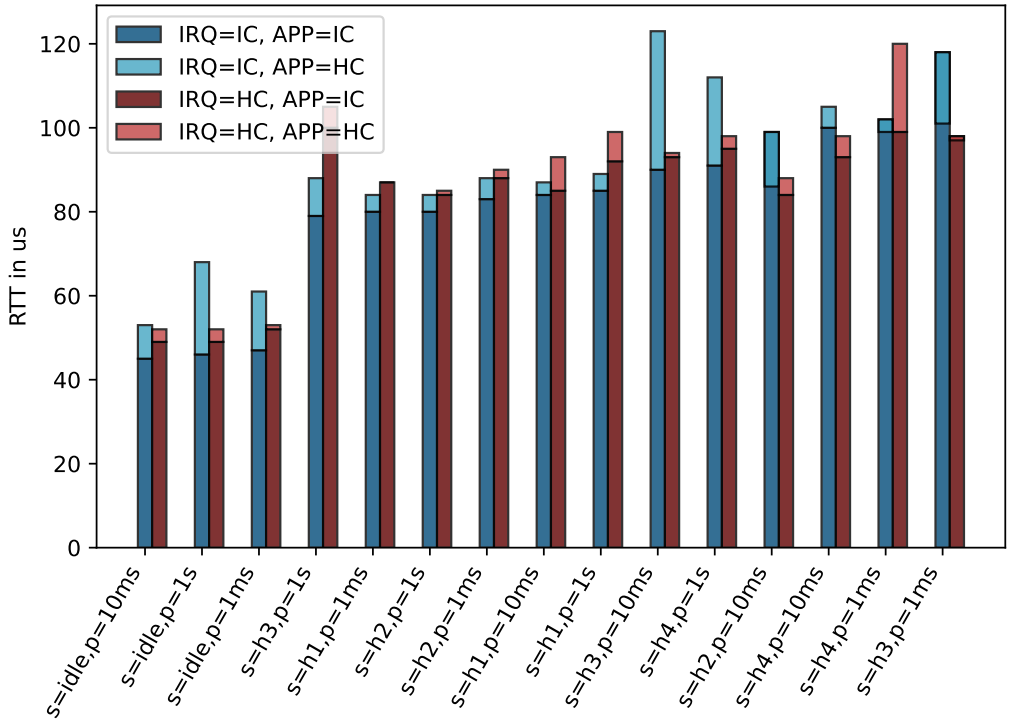


FIGURE 4.25: Highest measured RTT results for the PCIe waitqueue-based implementation with the first revision of the Xilinx XDMA device driver in polling mode.

Compared to the results with the unmodified Xilinx XDMA device driver, forcing the data transfer work to be processed on a CPU core in the isolated CPU cluster has a great impact on all four PCIe interrupt and measurement tool CPU pinning configurations. The stressors still cause a noticeable increase in RTT but are more stable regardless of the period and number of stressor threads.

Figure 4.26 shows the highest measured RTT results with the measurement tool's polling implementation and the same device driver configuration.

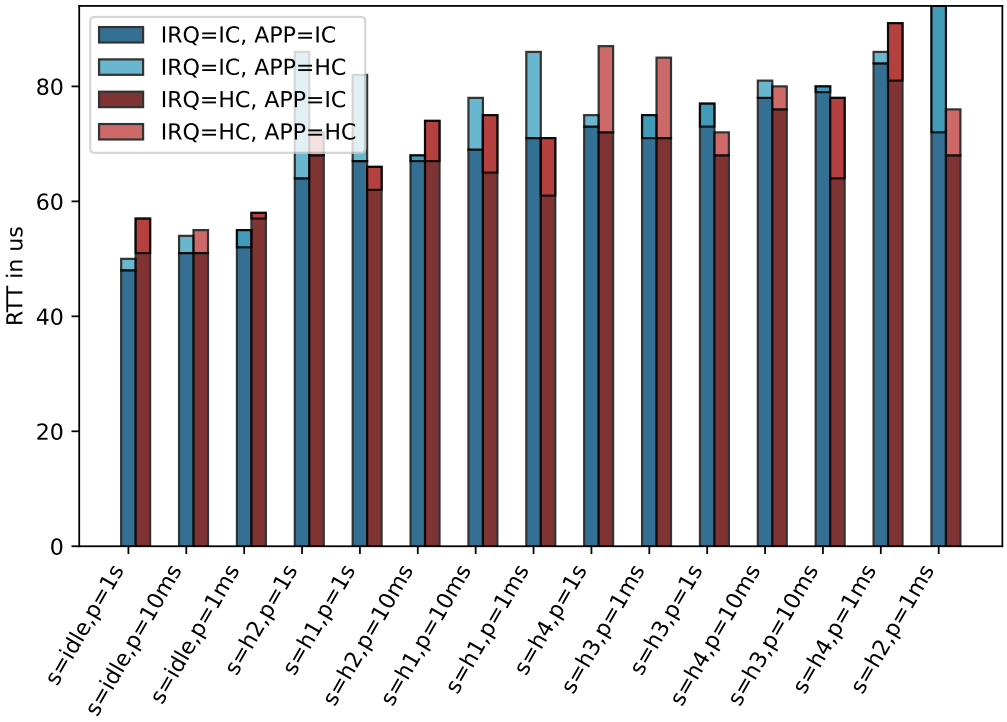


FIGURE 4.26: Highest measured RTT results for the PCIe polling-based implementation with the first revision of the Xilinx XDMA device driver in polling mode.

The polling-based measurement tool implementation yields even more stable results than the waitqueue-based implementation. When focusing on the results where the application is pinned to the isolated CPU cluster, the lowest and highest results are less than doubled.

Changing the device driver's configuration from polling mode to interrupt mode shows a small increase in RTT for both the polling and waitqueue-based implementations as an extra interrupt is introduced. The test results with the polling-based measurement tool implementation are shown in Figure A.6, and results with the waitqueue-based implementation are shown in Figure A.7.

4.4.3 Modified Xilinx XDMA Device Driver Revision 2

The dedicated measurement system will trigger a PCIe user interrupt, regardless of the device driver's polling and interrupt mode configurations. The results for GPIO in section 4.2.2 find that requesting interrupts with the `IRQF_NO_THREAD` flag effectively reduces the round-trip time. The second revision of the Xilinx XDMA device driver is modified to request the PCIe interrupt with the `IRQF_NO_THREAD` flag.

Figure 4.27 shows the highest measured RTT for the measurement tool's waitqueue-based implementation with the second revision of the device driver in polling mode.

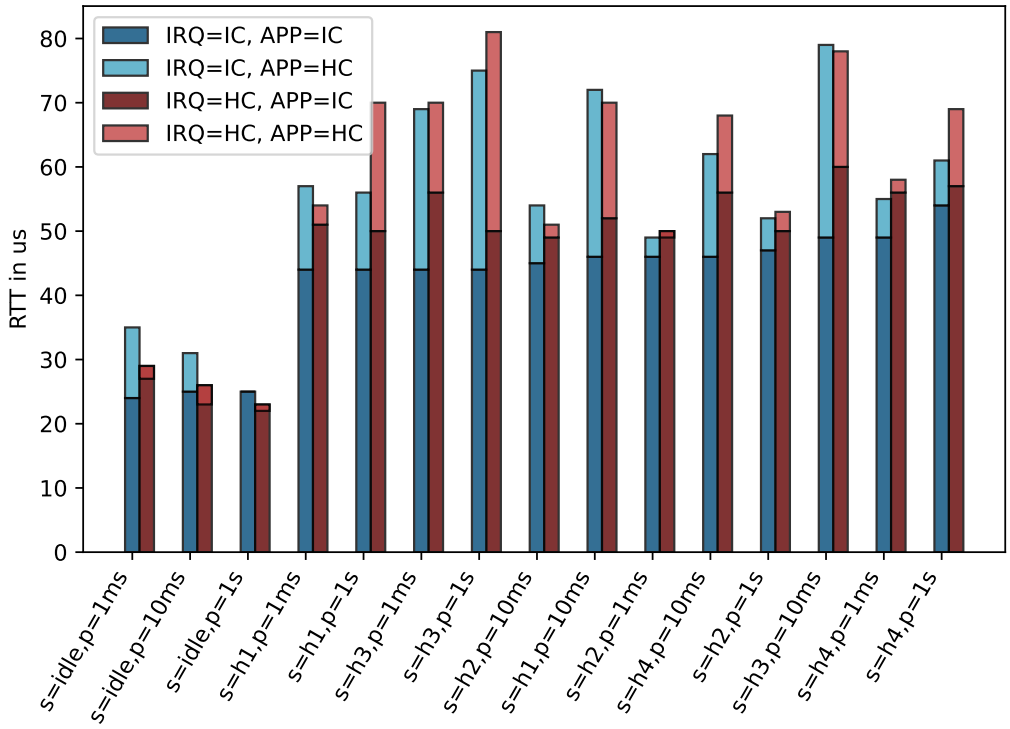


FIGURE 4.27: Highest measured RTT results for the PCIe waitqueue-based implementation with the second revision of the Xilinx XDMA device driver in polling mode.

The impact of pinning the PCIe interrupt and application to the isolated CPU cluster is massive, and pinning them both, as highlighted in dark blue colors, is impressively stable under stress. Changing the device driver's configuration to interrupt mode yields similar results as seen in Figure A.8. With the second revision, the distinction between the two device driver modes is diminished, as interrupt-based implementations experience less scheduling overhead with the `IRQF_NO_THREAD` flag.

Figure 4.28 compares the highest measured RTT between the first and second revisions of the device driver. The measurement tool and PCIe interrupt are pinned to the isolated CPU cluster, and the polling-based measurement tool implementation is used.

The `IRQF_NO_THREAD` modification introduced in the second revision of the device driver significantly improves the RTT results. The RTT is reduced by a third or more and performs similarly regardless of stressor and period configuration. The only exception is when the system is idle.

Figure 4.29 shows the highest measured RTT for the measurement tool's polling-based implementation with the second revision of the device driver in polling mode.

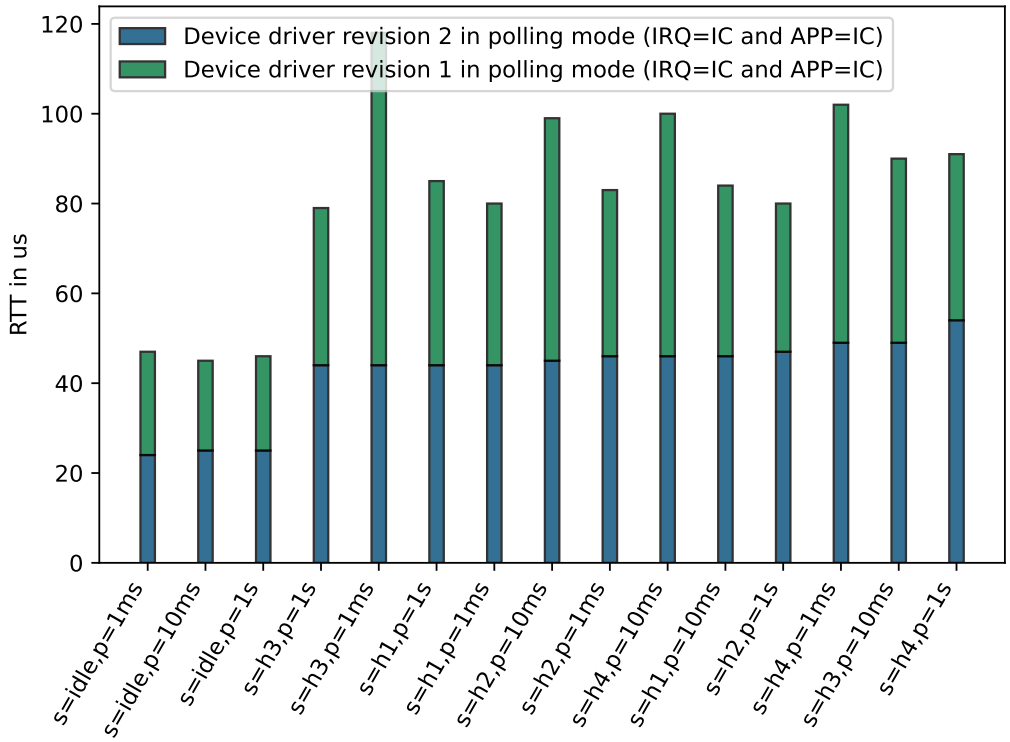


FIGURE 4.28: Comparison of the highest measured RTT results for the PCIe waitqueue-based implementation with the Xilinx XDMA device driver's first and second revisions in polling mode.

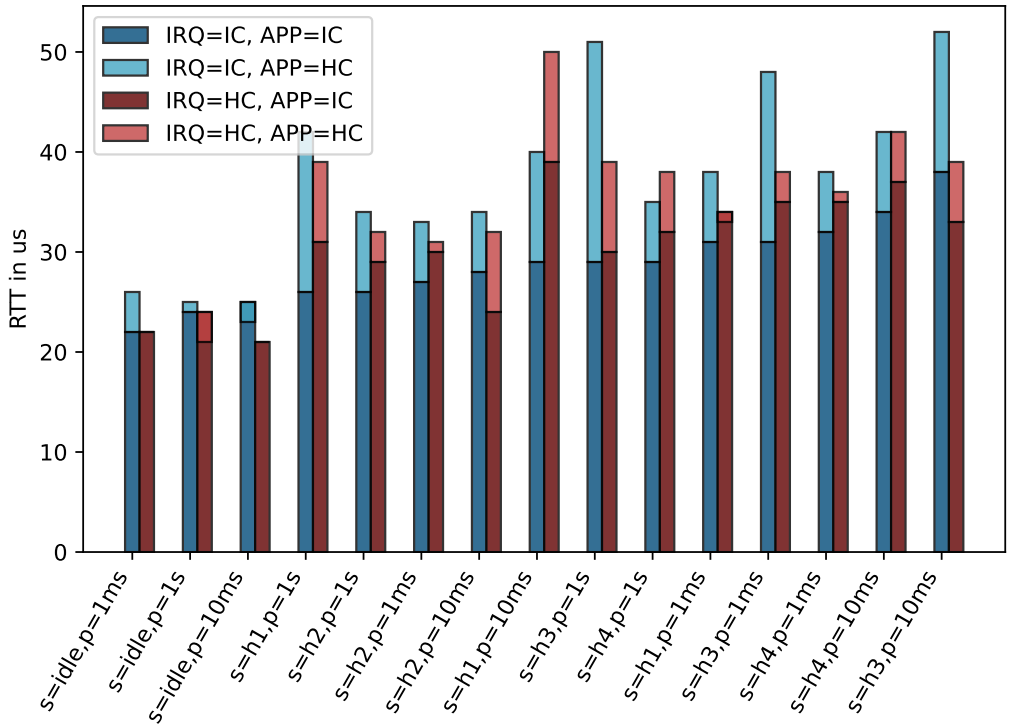


FIGURE 4.29: Highest measured RTT results for the PCIe polling-based implementation with the second revision of the Xilinx XDMA device driver in polling mode.

The polling-based implementation experiences the same reduction in RTT as the waitqueue-based implementation. When focusing on the results highlighted in dark blue, the gap between the lowest and highest RTT results is relatively low. This is unlike the waitqueue-based implementation, where the difference in the highest measured RTT is noticeably different between an idle and stressed system.

Figure 4.30 highlights the differences in the highest measured RTT for the device driver's second revision in polling mode and interrupt mode.

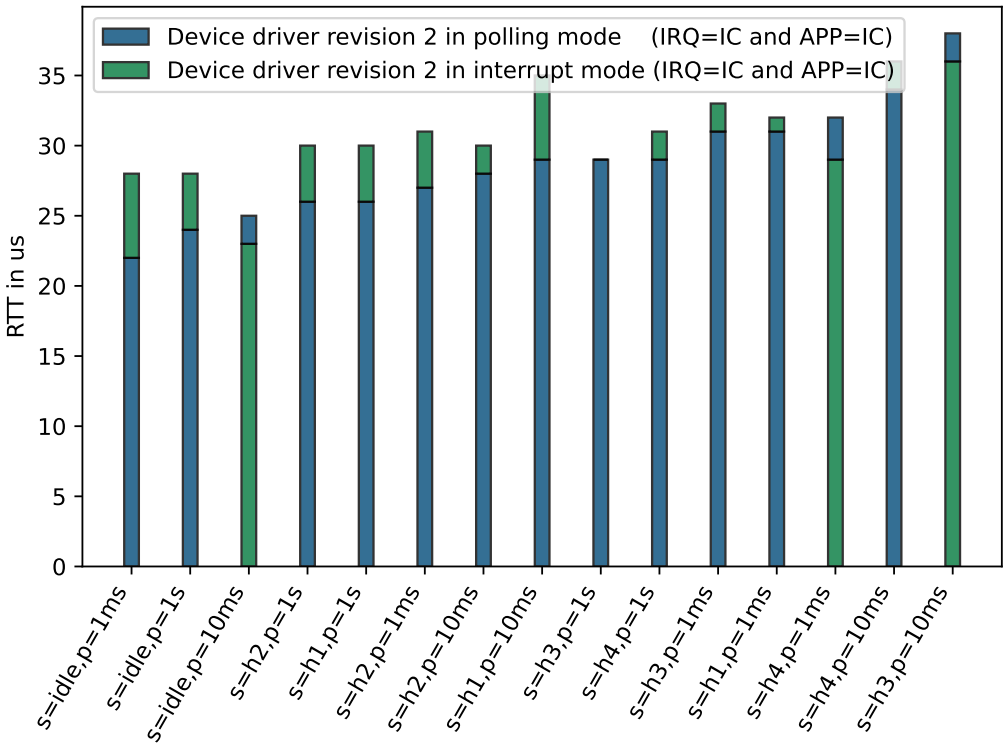


FIGURE 4.30: Comparison of the highest measured RTT results for the PCIe polling-based implementation with the Xilinx XDMA device driver revision 2 in polling mode and interrupt mode.

With the polling-based implementation, the two device driver modes perform similarly. For most stressor configurations, the device driver's polling mode performs slightly better.

Overall, the polling-based implementation is more stable than the waitqueue-based implementation and gives lower RTT results. For this revision, the device driver's polling mode configuration option is slightly preferred for both the waitqueue and polling-based implementations.

4.4.4 Modified Xilinx XDMA Device Driver Revision 3

Whether the Xilinx XDMA device driver is in polling mode or interrupt mode, the data transferring work is handled by a different process than the one requesting the transfer. In the meantime, the process requesting the transfer is blocked until the data transferring work is completed.

In the third revision of the device driver, the process requesting a data transfer is responsible for processing the data transfer instead of offloading the work to a `kthread` or `kworker`. The device driver's polling mode and interrupt mode have previously dictated how the data transfer is processed. These modes are now irrelevant as the data transfer is always processed by the process requesting the data transfer.

Figure 4.31 shows the highest measured RTT results for the waitqueue-based implementation with the third revision of the modified Xilinx XDMA device driver.

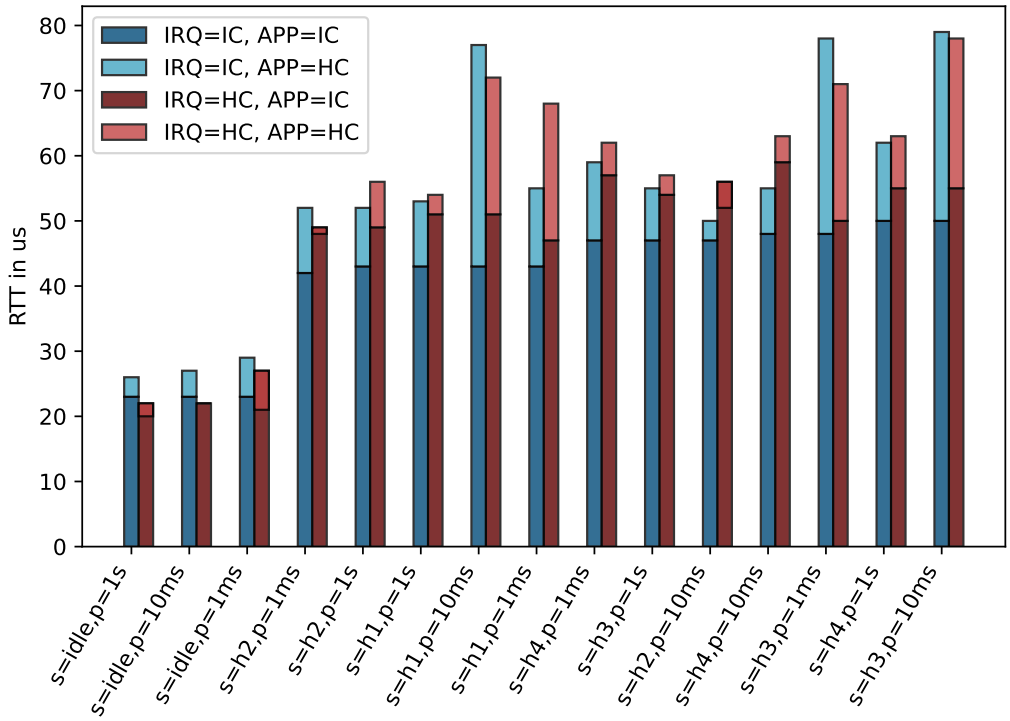


FIGURE 4.31: Highest measured RTT results for the PCIe waitqueue-based implementation with the third revision of the Xilinx XDMA device driver.

Compared to the previous revision, the results are more or less identical. Figure 4.32 shows the difference between the second and third revisions of the device driver with the waitqueue-based implementation.

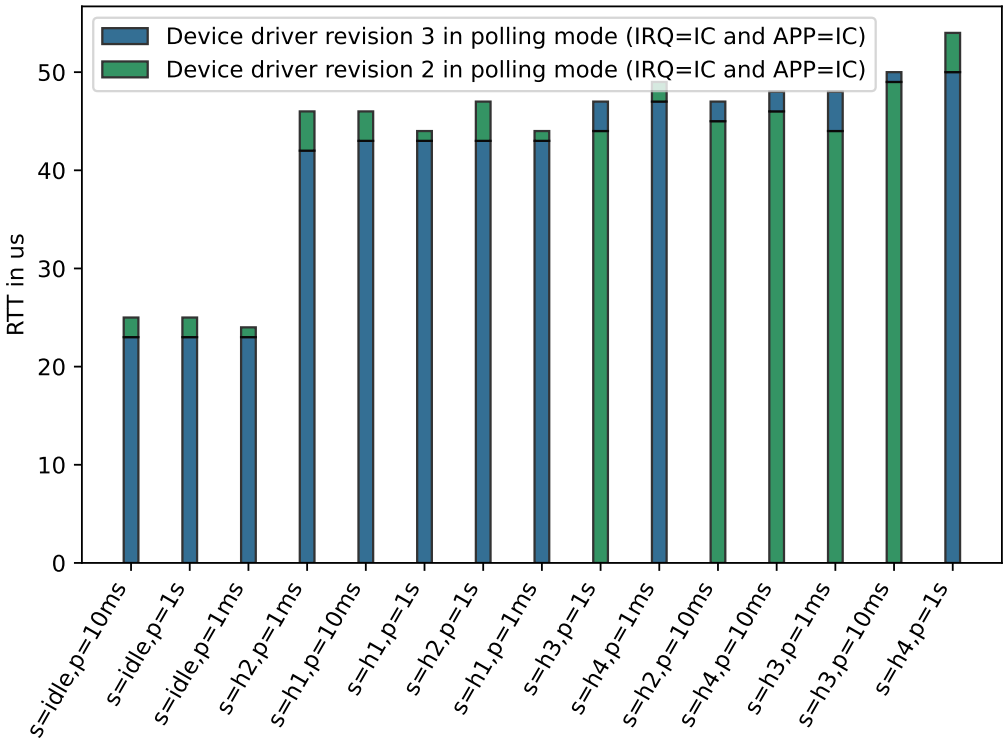


FIGURE 4.32: Comparison of the highest measured RTT results for the PCIe waitqueue-based implementation with the Xilinx XDMA device driver's second and third revisions.

In two-thirds of the results, the third revision performs slightly better. The results for the polling-based implementation as seen in Figure 4.33 are also similar to the results for the previous revision.

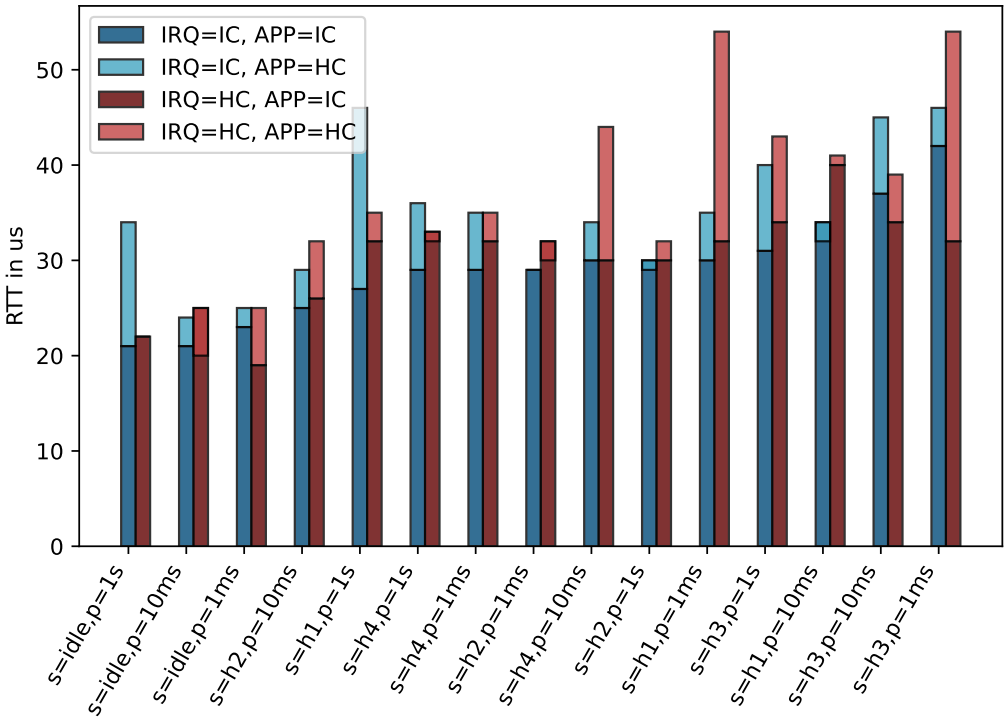


FIGURE 4.33: Highest measured RTT results for the PCIe polling-based implementation with the third revision of the Xilinx XDMA device driver.

The results for the third revision of the device driver show a slight improvement over the second revision, but are similar overall. Applying a few simple modifications to the device driver has drastically reduced the highest measured RTT from over one second to less than 80 microseconds for the waitqueue-based implementation with the PCIe interrupt and measurement tool pinned to the housekeeping CPU cluster. Pinning the PCIe interrupt and measurement tool to the isolated CPU cluster further reduces the highest measured RTT to less than 50 microseconds. More importantly, the results are stable regardless of the `hackbench` stressor configuration.

4.4.5 Additional Measurement Runs for PCIe

Additional measurement runs are performed with the best-performing PCIe configuration. The stressors used during the measurement runs are described in section 3.2.1.

Although the second and third revisions of the Xilinx XDMA device driver produce more or less the same results, the third revision is chosen as the optimal version. The third revision is less complex as the data transfers are processed by the same thread that initiated a data transfer. The other revisions offloaded the data transferring work to another thread, introducing the need for extra scheduling overhead and inter-process communication. Pinning the PCIe interrupt and measurement tool to the isolated CPU cluster is the most consistent combination and has the lowest RTT in most measurements.

The results for the additional measurement runs for PCIe with the waitqueue-based implementation are shown in Figure 4.34.

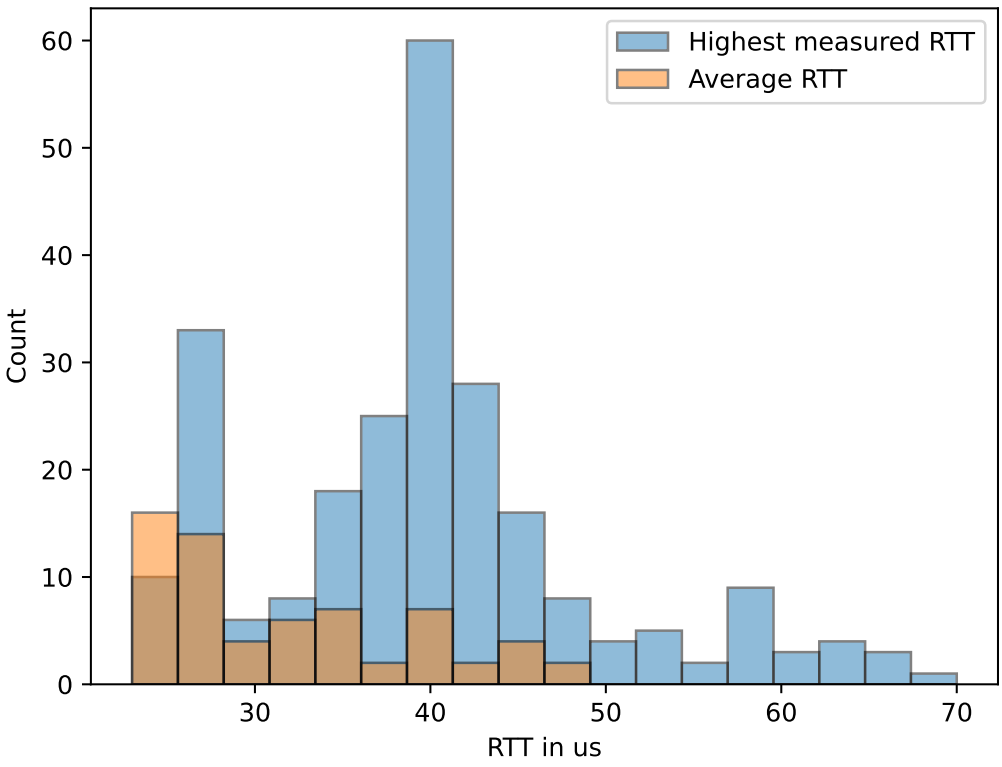


FIGURE 4.34: Histogram of the average and highest measured RTT for the additional measurement runs with the waitqueue-based PCIe implementation.

Although the waitqueue-based implementation was consistent with all four `hackbench` configurations, the same cannot be said for `stress-ng`'s variety of stressors. The highest measured RTT varies from 20 to 70 microseconds, with a 40-microsecond RTT for most stressors. Figure 4.35 shows the additional measurement results for the polling-based implementation.

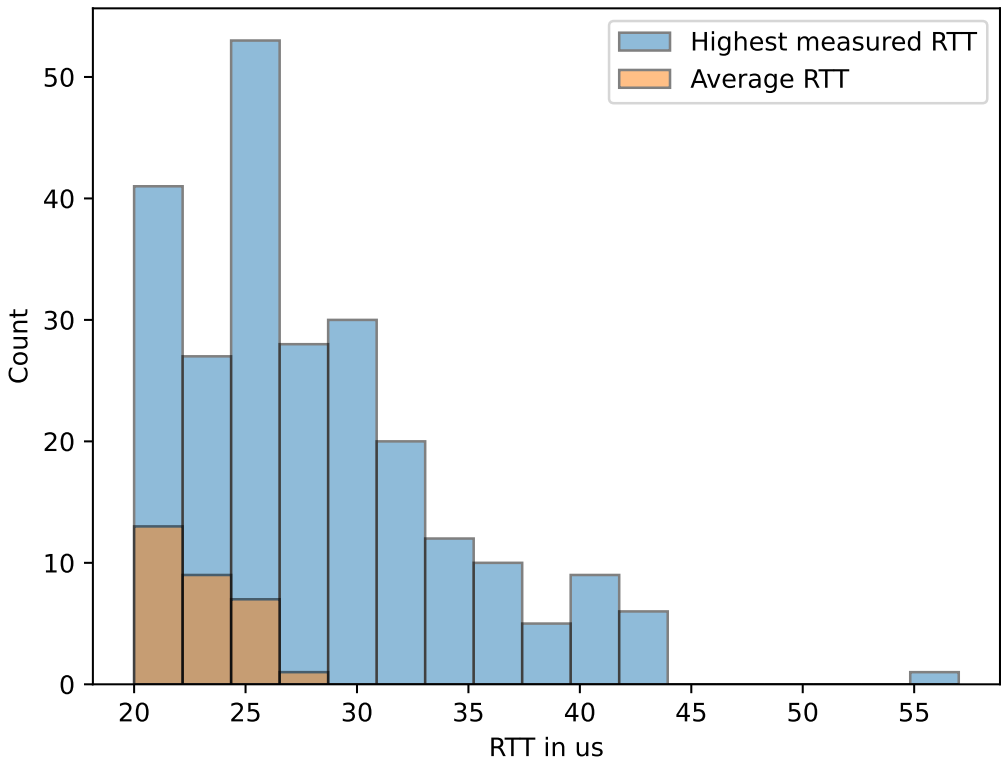


FIGURE 4.35: Histogram of the average and highest measured RTT for the additional measurement runs with the polling-based PCIe implementation.

As expected, the RTT results for the polling-based implementation are lower and less spread out compared to the RTT results for the waitqueue-based implementation. Similar to the results from the additional measurement runs for the polling-based implementation for GPIO described in section 4.2.4, the polling-based PCIe implementation is not exempt from experiencing spikes.

The highest measured round-trip time for the waitqueue and polling-based PCIe implementations is less than 70 microseconds. With most workloads, the highest measured round-trip time is under 50 microseconds.

4.5 Discussion

This section summarizes the various observations obtained from the test results and discusses the most important findings in detail.

4.5.1 Designing Real-Time Applications With Existing Device Drivers

One of the many reasons for designing a system with the Linux kernel instead of a dedicated RTOS is the vast number of available tools, programs, and device drivers. Instead of writing device drivers from scratch, existing open-source device drivers can be incorporated into real-time applications. However, the currently available device drivers for Linux may not have been written with real-time usage in mind [104], [105].

The measurement system for PCIe and Ethernet, designed for this thesis, uses the freely available Xilinx PCIe XDMA FPGA module for PCIe data transfers with the Linux-based system. The FPGA module has a corresponding open-source device driver for Linux, which the measurement tool uses on the Linux-based system. The alternative would have been to write an FPGA module and a corresponding Linux device driver from scratch, requiring expertise and a huge effort. Reusing existing modules saves a lot of time, which is essential for products fighting for time to market.

The measurement runs for PCIe, described in section 4.4, initially use an unmodified version of the Xilinx PCIe XDMA device driver. It is quickly revealed that the device driver is unstable when stress is introduced to the system. CPU pinning configurations that have proven effective for Ethernet and GPIO do not seem to make much of a difference for the unmodified device driver. When investigating the device driver, it is revealed that the process responsible for processing data transfers over PCIe is scheduled with the same priority and scheduling policy as the stressor threads. After three revisions of the device driver, the PCIe data transfers are processed by the initiator, which is the measurement tool, and the PCIe interrupt is forced to be executed immediately in a hard interrupt context instead of being scheduled as a thread. These small modifications drastically improve the device driver's performance for real-time usage.

The experiments and results for PCIe prove that open-source Linux device drivers may not be optimized for real-time usage. On a positive note, the experiment results show that small modifications can sometimes drastically improve the device driver's real-time usability. With the increase of Linux-based real-time systems, open-source device drivers may gradually be optimized for real-time in the coming years.

4.5.2 The Impact of System Stress

Stress negatively affects the round-trip time results for GPIO, Ethernet, and PCIe. Although some implementation modifications and CPU configuration options reduce the impact, the protocols perform considerably worse during a high system load. The protocols are affected to different degrees, with a variance in the gap between the worst and best round-trip times.

The additional measurement runs serve as an effective evaluation of the protocols' stability by measuring against various stressors with different characteristics. These

experiments highlight the inherent instability of the Linux kernel. The round-trip time results exhibit significant variation depending on the stressor. This makes it challenging to predict a worst-case round-trip time when the system load and characteristics of the real-time application are unknown. The test results can only help indicate a *probable* worst-case round-trip time. It is worth noting that the measurement runs were only conducted for a relatively short time duration. Increasing the duration might reveal different results, as the probability of experiencing spikes is higher.

The BPF program created for tracing the softirq raising and handling mechanisms revealed that the kernel's softirq implementation is unstable and quite affected by stress. The same trend is seen for the kernel's wake-up mechanism in the results from the baseline tests with `cyclictest`. These are core elements of the kernel, impacting a wide range of applications. Moving the processing of these core elements to an isolated CPU core does not exempt them from being affected by general system load. This indicates that the kernel's core elements are tightly coupled with all CPU cores, and cannot be completely shielded with the kernel's isolation techniques.

The results from this thesis may not directly translate to other systems and should only be seen as an indication. Getting an accurate indication can only be done by measuring against the *actual* system load.

4.5.3 The effectiveness of the Linux kernel's isolation techniques

The Linux-based system is configured to partition its two quad-core Arm Cortex-A72 microprocessor clusters into a housekeeping and isolated CPU cluster. The isolated CPU cluster is also configured as tickless and excluded from general SMP scheduling. Applications and interrupts must, therefore, be explicitly assigned to the isolated CPU cluster in order to run there.

The initial baseline tests with `cylictest` find that when no additional load is applied to the system, the scheduling latency is higher on the isolated CPU cluster than on the housekeeping CPU cluster. The same trend is observed for PCIe and the interrupt-based GPIO implementation. This negative implication is due to the isolated CPU cluster being configured as tickless.

Configuring a CPU core as tickless comes with a trade-off. High-processing applications will benefit from the tickless CPU configuration as the applications are not periodically interrupted by the system tick, increasing the throughput. Consequently, waking up an application running on a tickless CPU core takes more time, especially if woken up by a different CPU core. This was highlighted in the experiments with timer migration in section 4.1.

During the measurement runs conducted in this thesis, the measurement tool is mostly blocked, waiting to be woken up by an inter-processor interrupt or a periodic timer. Little to no data processing is performed between each cycle. Therefore, in most situations, the measurement tool does not benefit from being assigned to a tickless CPU core and is mostly negatively affected by the CPU configuration. However, the trend turns during the Ethernet experiments as the networking stack requires more data processing. It is essential to know the characteristics of the application when contemplating configuring a CPU core as tickless.

Assigning the measurement tool and the corresponding interrupts to the isolated CPU cluster does not provide temporal isolation. The GPIO, Ethernet, and PCIe implementations are still negatively affected by stress. In return, the round-trip times are lower and more stable. Isolating and dedicating a CPU core for real-time tasks is simple and should be considered for multi-processor real-time Linux system designs.

4.5.4 Interrupt vs. Polling-Based Implemenations

Interrupt and polling-based implementations were compared for GPIO and PCIe. The polling-based implementations yield lower and more stable results than their interrupt-based counterparts. The great performance of the polling-based implementations does not come without consequences. The threads performing the polling will frequently preempt other threads with lower priority. Context switching is generally considered to be expensive, especially at this rate. Dedicating a CPU core for a polling-based real-time application is becoming more achievable as modern SoCs have multiple CPU cores. For smaller and older SoCs with fewer CPU cores, interrupt-based implementations are more compelling as more CPU bandwidth is available for other processes. The interrupt-based implementations are more affected by system load, as shown in the results from the additional measurement runs for GPIO in section 4.2.4 and PCIe in section 4.4.5. How well the interrupt-based implementations perform varies by the different kinds of system load with great variance. The polling-based implementations are more stable but still perform differently based on the kind of system load.

Forcing interrupt handlers to be executed immediately in a hard interrupt context instead of being scheduled as a thread has shown to be quite effective at reducing the RTT and increasing the stability of the GPIO and PCIe implementations. This does not come without consequences, as extending the time in a hard interrupt context negatively affects other tasks, whether high-priority or not. It should only be considered for small interrupt handlers and interrupt handlers that are more important than any other task in the system.

4.5.5 Designing a Linux-based real-time system

Writing time-sensitive applications for Linux with the PREEMPT_RT patch requires a good understanding of the kernel's inner workings. Although the PREEMPT_RT patch introduces priority inheritance for threads that share locking primitives, this functionality does not automatically take effect when enabling the `CONFIG_PREEMPT_RT` kernel configuration option. The shared locking primitives must be explicitly declared with the `PTHREAD_PRIO_INHERIT` protocol attribute [106]. Another great example is the kernel's timer mechanisms. The kernel offers multiple timer mechanisms, but only one is executed in a hard interrupt context and can be considered reliable. The others are forcibly converted to threads, which can lead to unwanted delay.

The open-source Xilinx XDMA PCIe device driver is an excellent example of a driver that needed modifications before it could be considered for real-time usage. It has been necessary to understand the negative implications of its design in order to make the required modifications.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, a comprehensive hardware setup with dedicated measurement systems is developed to measure the Linux kernel's ability to reliably respond to incoming messages and signals over GPIO, Ethernet, and PCIe. The experiments are conducted to determine if the Linux kernel with the PREEMPT_RT patch can provide reliable high-speed serial communication with devices in a distributed system. The Linux kernel's isolation mechanisms are evaluated to determine if the Linux kernel alone can achieve temporal isolation and be used to implement a mixed-criticality system.

The proposed Linux-based mixed-criticality system, designed with state-of-the-art practices, can not provide total temporal isolation. However, the Linux kernel's isolation mechanisms are proven to deliver more stable and lower round-trip times for GPIO, Ethernet, and PCIe.

A fundamental attribute of the PREEMPT_RT patch is its compatibility with preexisting Linux kernel applications and device drivers. An open-source device driver is used to conduct data transfers over PCIe to an FPGA-based measurement system. When stress is applied to the Linux-based system, the device driver's performance degrades significantly. Analyzing and applying a few modifications to the device driver drastically improves its performance and makes it relatively stable during high system load. Pre-existing Linux drivers may require minor modifications to work with real-time applications.

Polling and interrupt-based implementations are compared for GPIO and PCIe. The interrupt-based implementations perform worse, mainly due to the inclusion of interprocess communication and the variance in wake-up latency. Polling-based implementations are proven to be less affected by system stress and provide reliably low round-trip times.

Although the highest measured round-trip times for GPIO, Ethernet, and PCIe can not be regarded as absolute worst-case round-trip times, they indicate that state-of-the-art methods for real-time Linux effectively decrease the round-trip times and increase the stability. However, the Linux kernel is still greatly impacted by stress. The variance in the kernel's system latencies makes the kernel insufficient for hard real-time systems. However, the round-trip time results are stable enough to be considered for soft real-time systems.

5.2 Future Work

When stress is introduced to the Linux-based system, an increase in latency among core Linux kernel mechanisms such as scheduling, softirqs, and the wake-up mechanism is observed. The round-trip time results for GPIO, Ethernet, and PCIe vary greatly depending on the stressor configuration, indicating that the Linux kernel's core mechanisms are not deterministic. It is worth investigating the causes behind the increased system latency and the common denominators of the worst-performing stressor configurations.

The polling-based GPIO implementation was unaffected by 98% of the stressor configurations, indicating that polling-based implementations can obtain temporal isolation with a wide variety of system stressors. It would be interesting to measure the stability of other polling-based implementations under the same stress conditions, such as a DPDK-based Ethernet implementation.

This thesis implements a mixed-criticality system using only the Linux kernel's isolation mechanisms. Although embedded hypervisors can not guarantee total temporal isolation, it would be interesting to reimplement the mixed-criticality system using an embedded hypervisor and compare the results.

The test results indicate that configuring the isolated CPU cluster as tickless had a more negative than positive effect on PCIe and the interrupt-based GPIO implementation. It would be interesting to conduct additional measurement runs without the tickless CPU configuration and compare the impact.

The literature suggests that the Linux kernel's networking stack is not ideal for real-time systems, favoring alternative real-time networking stacks. However, the native Linux kernel networking stack offers various parameters that can be adjusted. Fine-tuning the networking stack could potentially improve the round-trip time and stability of the Ethernet results.

Appendix A

Additional Test Results

A.1 GPIO

Figure A.1 shows the average RTT results for the interrupt-based GPIO implementation with an unmodified GPIO driver.

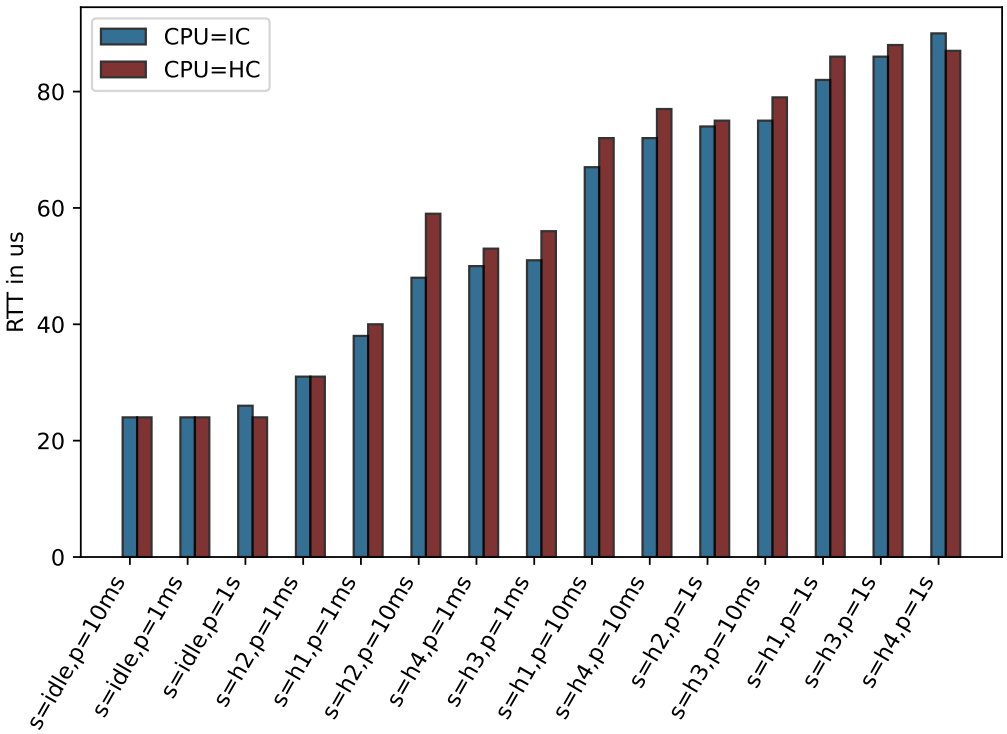


FIGURE A.1: Average RTT results for the interrupt-based GPIO implementation with an unmodified GPIO driver.

Figure A.2 shows the average RTT results for the interrupt-based GPIO implementation with a modified GPIO driver with `IRQF_NO_THREAD`. The modifications are discussed in section 4.2.2.

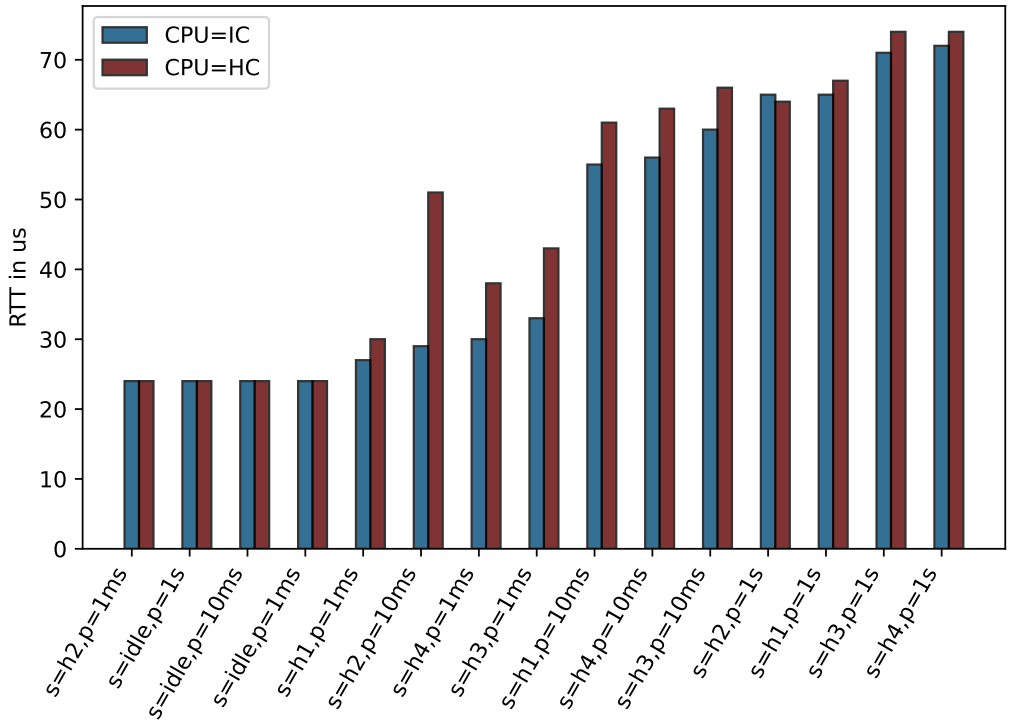


FIGURE A.2: Average RTT results for the interrupt-based GPIO implementation with a modified GPIO driver with `IRQF_NO_THREAD`.

A.2 Ethernet

Figure A.3 shows the difference in TCP softirq raising delay for a system under heavy load and no load.

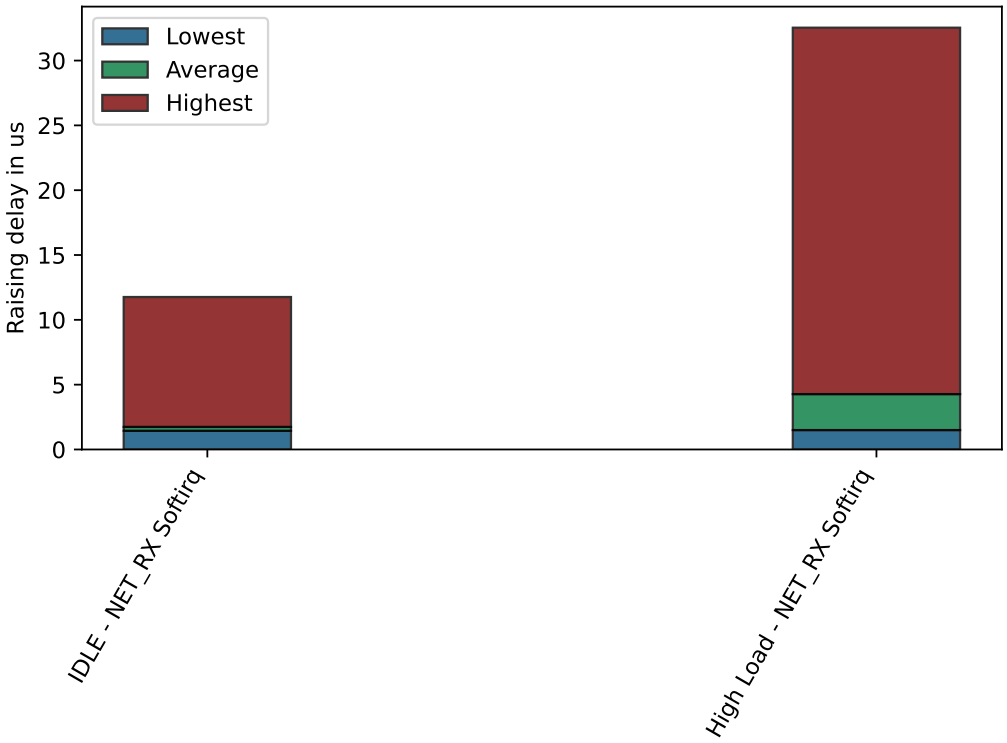


FIGURE A.3: Difference in TCP softirq raising delay for a system under heavy load and no load.

Figure A.4 shows the difference in TCP softirq processing time for a system under heavy load and no load.

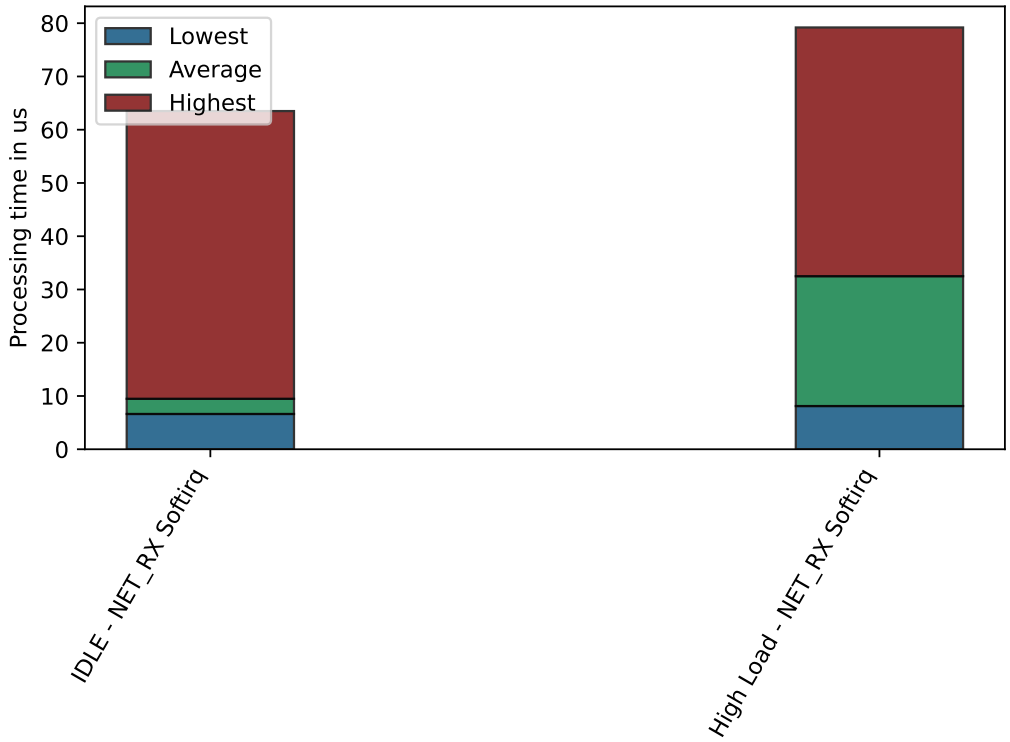


FIGURE A.4: Difference in TCP softirq processing time for a system under heavy load and no load.

Figure A.5 shows the top 20 worst-performing `stress-ng` stressors for the UDP implementation.

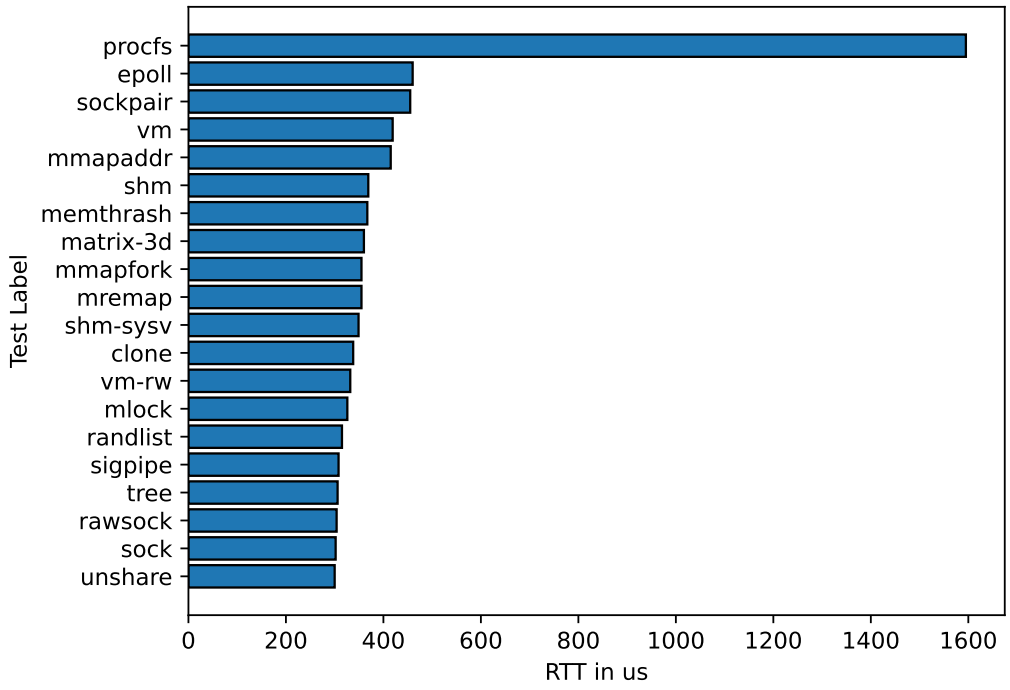


FIGURE A.5: The 20 worst-performing `stress-ng` stressors for UDP.

A.3 PCIe

Figure A.6 shows the highest measured RTT for the measurement tool's waitqueue-based implementation with the first revision of the device driver in interrupt mode.

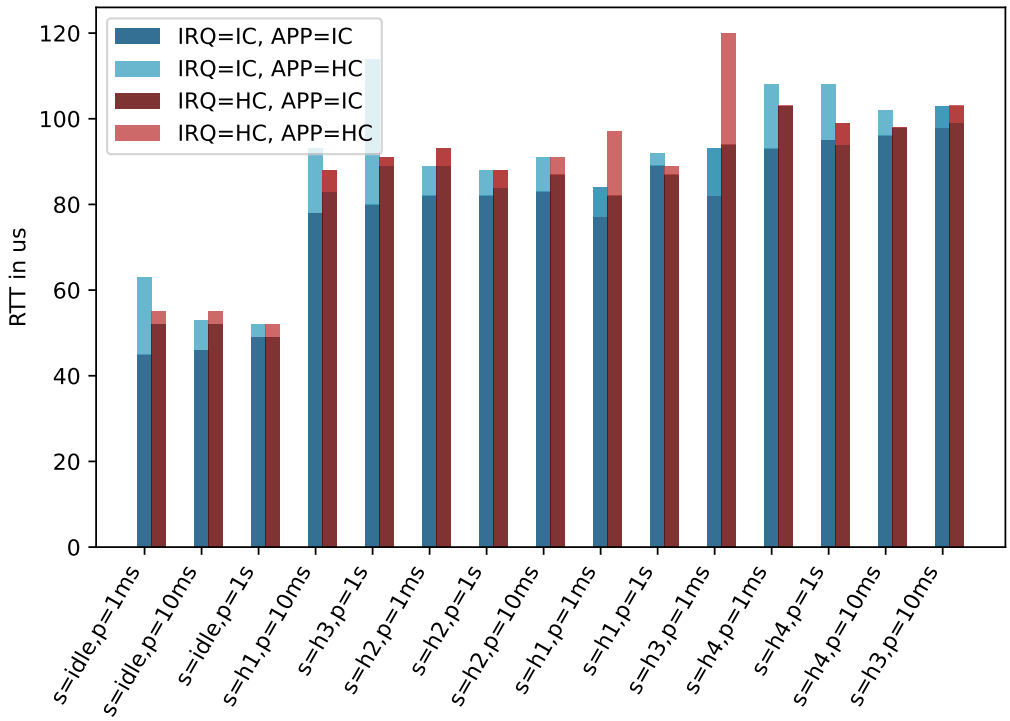


FIGURE A.6: Highest measured RTT results for the PCIe waitqueue-based implementation with the first revision of the Xilinx XDMA device driver in interrupt mode.

Figure A.7 shows the highest measured RTT for the measurement tool's polling- based implementation with the first revision of the device driver in interrupt mode.

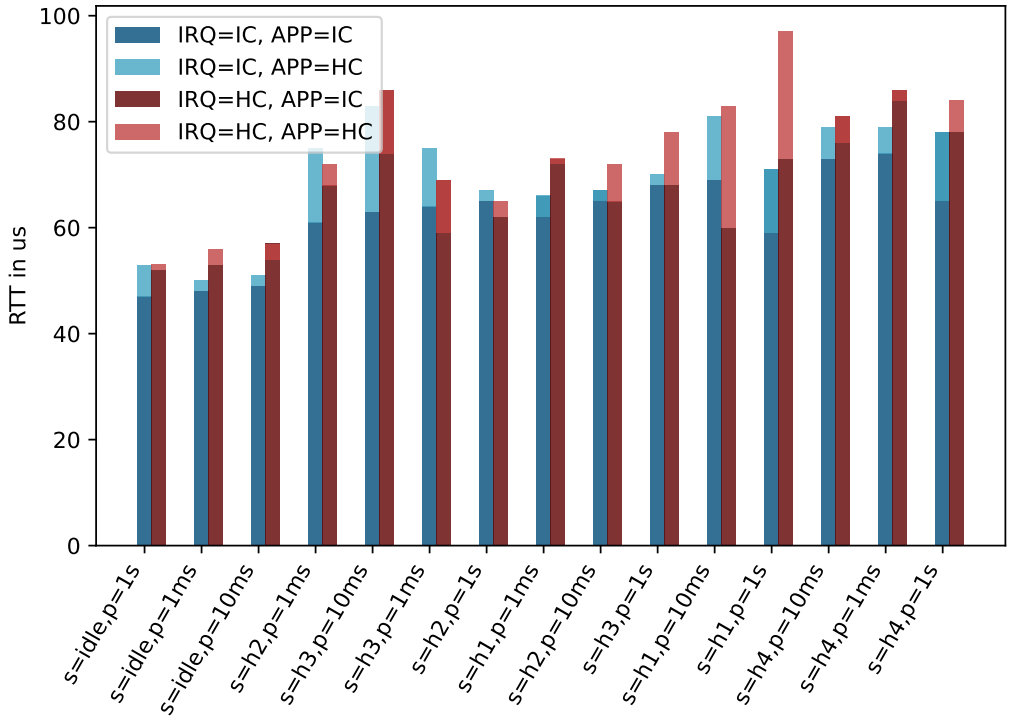


FIGURE A.7: Highest measured RTT results for the PCIe polling-based implementation with the first revision of the Xilinx XDMA device driver in interrupt mode.

Figure A.8 shows the highest measured RTT for the measurement tool's waitqueue-based implementation with the second revision of the device driver in interrupt mode.

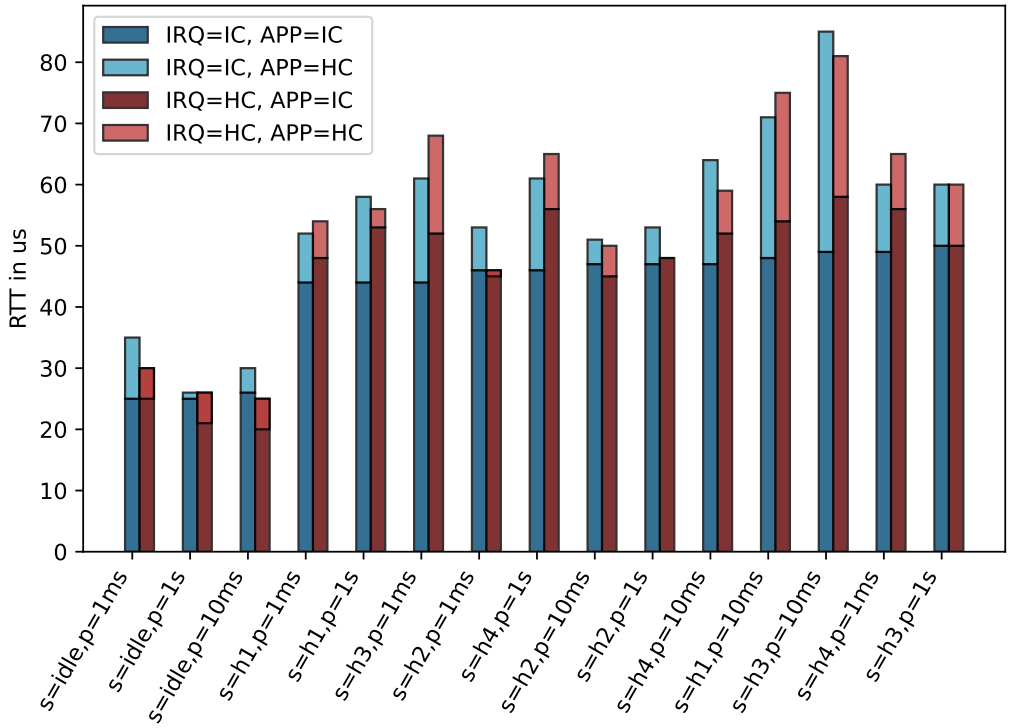


FIGURE A.8: Highest measured RTT results for the PCIe waitqueue-based implementation with the first revision of the Xilinx XDMA device driver in interrupt mode.

Figure A.9 shows the highest measured RTT for the measurement tool's polling- based implementation with the second revision of the device driver in interrupt mode.

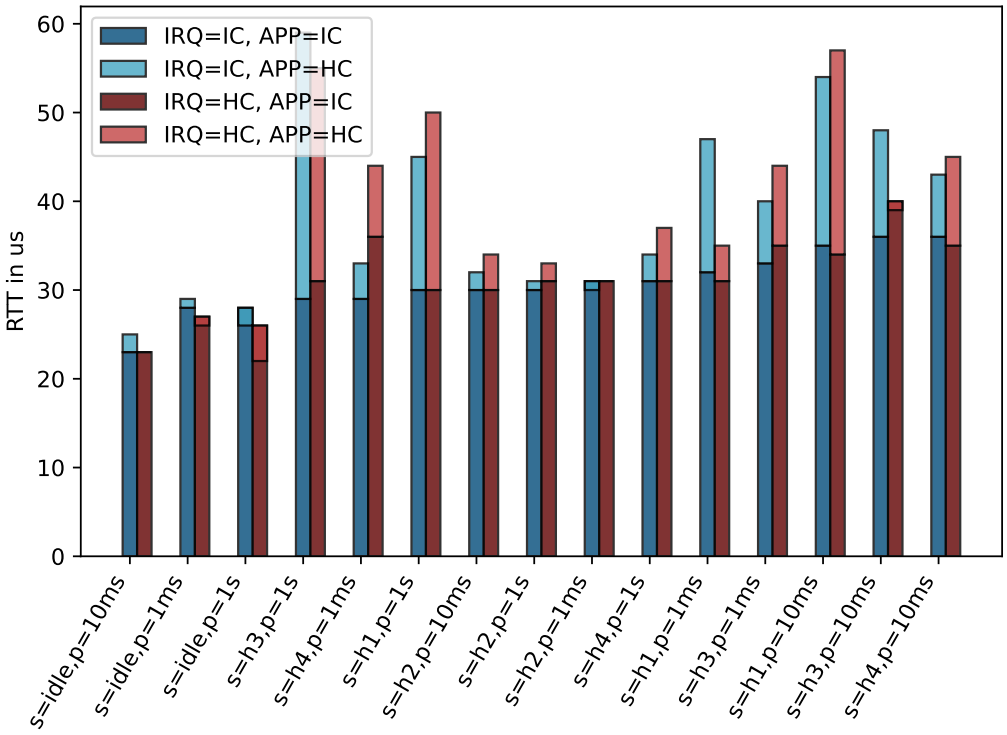


FIGURE A.9: Highest measured RTT results for the PCIe polling-based implementation with the second revision of the Xilinx XDMA device driver in interrupt mode.

Appendix B

Stress-ng Stressors

Listing B.1 shows the list of the stress-ng stressors used in the additional measurement runs in sections 4.2.4, 4.4.5, 4.3.1 and 4.3.2.

```
access affinity alarm atomic bad-altstack bigheap branch brk bsearch cache
cacheline chattr chdir chmod chown chroot clock clone close context
copy-file cpu crypt cyclic daemon dekker dentry dev dev-shm dir dirdeep
dirmany dnotify dup dynlib enosys env epoll eventfd exit-group fallocate
fanotify far-branch fault fcntl fifo file-ioctl filename flock flushcache
fork forkheavy fp fp-error fpunch fsize fstat full funccall funcret futex
get getdent getrandom goto handle hash hdd hsearch icache icmp-flood
inode-flags inotify io iomix ioprio itimer kcmp kill klog lease link list
locka lockbus lockf lockofd longjmp loop lsearch madvise malloc matrix
matrix-3d mcontend memcpy memfd memhotplug memrate memthrash mincore
misaligned mknod mlock mmap mmapaddr mmapfixed mmapfork mmaphuge
mmapmany mprotect mq mremap msg msync msyncmany munmap mutex nanosleep
netdev netlink-task nice nop null numa opcode open pagemove pageswap
pci personality peterson physpage pipe pipeherd pkey poll prctl prefetch
priv-instr procfs pthread ptrace pty qsort randlist ramfs
rawpkt rawsock rawudp readahead reboot regs remap rename resched resources
revio ring-pipe rlimit rmap rotate schedpolicy seal seek sem sem-sysv
sendfile session set shellsort shm shm-sysv sigabrt sigchld sigfd sigfpe
sigio signal signest sigpending sigpipe sigq sigrt sigsegv sigsuspend
sigtrap skiplist sleep sock sockabuse sockdiag sockfd sockpair sockmany
sparsematrix splice stack stackmmap str stream switch symlink sync-file
syncload sysbadaddr syscall sysinfo sysfs tee timer timerfd tlb-shootdown
tmpfs touch tree tsearch tun udp udp-flood umount unshare urandom utime
vdso vecfp vecmath vecshuf vecwide vfork vforkmany vm vm-addr vm-rw
vm-segv vm-splice wait waitcpu yield zero zlib zombie
```

LISTING B.1: List of used stress-ng stressors

Listing B.2 shows the list of excluded stress-ng stressors. The reason for the exclusion is mentioned in section 3.2.1.

```
af-alg aio aiol apparmor bad-ioctl binderfs bind-mount cap cgroup
cpu-online dccp efivar eigen exec fiemap fma gpu heapsort hrtimers
idle-page ioport io-uring ipsec-mb jpeg judy key kvm l1cache landlock
led llc-affinity loadavg membarrier mergesort metamix mlockmany module
mpfr netlink-proc oom-pipe pidfd ping-sock plugin quota race-sched
    radixsort
 rawdev rdrand rseq rtc schedmix sctp seccomp secretmem sigbus sigxcpu
sigxfsz smi softlockup spawn swap sysinval trig tsc uprobe userfaultfd
usersyscall verity vma vnni watchdog wcs workload x86cpuid x86syscall xattr
```

LISTING B.2: List of excluded stress-ng stressors

Bibliography

[1] Grammarly. (2024). Grammarly, Inc. Accessed: May 17, 2024. [Online]. Available: https://www.grammarly.com/

[2] M. Barabanov, "A Linux-based real-time operating system," M.S. thesis, New Mexico Institute of Mining and Technology, Socorro, NM, USA, Jun. 1997.

[3] P. Mantegazza, E. L. Dozio, and S. Papacharalambous, "RTAI: Real time application interface," *Linux Journal*, vol. 2000, no. 72es, 10–es, Apr. 2000, ISSN: 1075-3583.

[4] S. R. Koganti and K. Gowthami, "Implementation of Xenomai framework in GNU/Linux environment to run applications in a real time environment," *Indian Journal of Science and Technology*, vol. 9, May 2016. DOI: 10.17485/ijst/2016/v9i17/93109.

[5] C. Gu, N. Guan, Q. Deng, and W. Yi, "Partitioned mixed-criticality scheduling on multiprocessor platforms," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.305.

[6] M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello, "Virtualizing mixed-criticality systems: A survey on industrial trends and issues," *Future Generation Computer Systems*, vol. 129, pp. 315–330, Apr. 2022, ISSN: 0167739X. DOI: 10.1016/j.future.2021.12.002. arXiv: 2112.06875.

[7] The Linux Foundation. *PREEMPT_RT History*. (2022). Accessed: Dec. 20, 2023. [Online]. Available: https://wiki.linuxfoundation.org/realtime/rtl/blog#preempt-rt-history

[8] Paul McKenney. "A realtime preemption overview." lwn.net. Accessed: Apr. 22, 2024. [Online]. Available: https://lwn.net/Articles/146861/

[9] S. H. VanderLeest and K. Stewart, "Enabling Linux in aerospace applications," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, Oct. 2023, pp. 1–6. DOI: 10.1109/DASC58513.2023.10311338.

[10] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on PREEMPT_RT," *ACM Computing Surveys*, vol. 52, no. 1, Jan. 2019, ISSN: 15577341. DOI: 10.1145/3297714.

[11] R. Rosmaninho, D. Raposo, P. Rito, and S. Sargento, "Time constraints on vehicular edge computing: A performance analysis," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, May 2023, pp. 1–7. DOI: 10.1109/NOMS56928.2023.10154306.

[12] X. Fan, T. Zheng, S. Sun, M. Gidlund, and J. Åkerberg, "Can embedded real-time Linux aystem effectively support multipath transmission? An experimental study," in *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*, Apr. 2023, pp. 1–8. DOI: 10.1109/WFCS57264.2023.10144118.

[13] G. K. Adam, N. Petrellis, and L. T. Doulos, "Performance assessment of Linux kernels with PREEMPT_RT on arm-based embedded devices," *Electronics (Switzerland)*, vol. 10, no. 11, Jun. 2021, ISSN: 20799292. DOI: 10.3390/electronics10111331.

[14] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang, *A Survey of WCET Analysis of Real-Time Operating Systems*. Jan. 2009, p. 72. DOI: 10.1109/ICESS.2009.24.

[15] P. Okech, N. M. Guire, and W. Okelo-Odongo, "Inherent diversity in replicated architectures," Oct. 2015. arXiv: 1510.02086. [Online]. Available: http://arxiv.org/abs/1510.02086.

[16] X. Chen, X. Kong, Y. Ling, and X. Cao, "DDS performance evaluation for PREEMPT_RT Linux," in *2021 International Conference on Computer, Blockchain and Financial Development (CBFD)*, Apr. 2021, pp. 84–89. DOI: 10.1109/CBFD52659.2021.00024.

[17] Y. Wei, "Research on real-time improvement technology of Linux based on multi-core ARM," in *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, Dalian, China: IEEE, Jun. 2021, pp. 1061–1066, ISBN: 978-1-66541-867-6. DOI: 10.1109/ICAICA52286.2021.9498165.

[18] Y. Li, Y. Matsubara, H. Takada, K. Suzuki, and H. Murata, "A performance evaluation of embedded multi-core mixed-criticality system based on PREEMPT RT Linux," *Journal of Information Processing*, vol. 31, pp. 78–87, 2023, ISSN: 18826652. DOI: 10.2197/ipsjjip.31.78.

[19] K. Kozlowski. (2023). Preparing Linux real-time kernel and tuning robotics platform with a modern ARM64 SoC. Presented at the Embedded Open Source Summit, Prague, Czech Republic. [Online]. Available: https://static.sched.com/hosted_files/eoss2023/1d/Linux%20Real-Time%20kernel%20and%20tuning%20robotics%20platform%20-%20Krzysztof%20Kozlowski%2C%20Linaro%20-%20ELCE%202023.pdf

[20] S. Alonso, J. Lázaro, J. Jiménez, U. Bidarte, and L. Muguira, "Evaluating latency in multiprocessing embedded systems for the smart grid," *Energies*, vol. 14, no. 11, p. 3322, Jan. 2021, ISSN: 1996-1073. DOI: 10.3390/en14113322. (accessed Oct. 2, 2023).

[21] L. Abeni and D. Faggioli, "An experimental analysis of the Xen and KVM latencies," in *Proceedings - 2019 IEEE 22nd International Symposium on Real-Time Distributed Computing, ISORC 2019*, Test, Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 18–26, ISBN: 978-1-72810-150-7. DOI: 10.1109/ISORC.2019.00014.

[22] J. Altenberg. (2023). Evaluation of PREEMPT_RT in virtualized environments. Presented at the Embedded Open Source Summit, Prague, Czech Republic. [Online]. Available: https://static.sched.com/hosted_files/eoss2023/51/preempt_rt_virtualization.pdf

[23] C. C. J. Huang and C. F. Yang, "An empirical approach to minimize latency of real-time multiprocessor Linux kernel," in *Proceedings - 2020 International Computer Symposium, ICS 2020*, Institute of Electrical and Electronics Engineers Inc., Dec. 2020, pp. 214–218, ISBN: 978-1-72819-255-0. DOI: 10.1109/ICS51289.2020.00051.

[24] N. Litayem and S. B. Saoud, "Impact of the Linux real-time enhancements on the system performances for multi-core Intel architectures," *International Journal*

of Computer Applications, vol. 17, no. 3, pp. 17–23, Mar. 2011. [Online]. Available: https://ijcaonline.org/archives/volume17/number3/2202-2796/ (accessed May 18, 2024).

[25] The Linux Foundation. *Preemption Models*. (2023). Accessed: Dec. 31, 2023. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/preemption_models

[26] The Linux Foundation. *Lock types and their rules — The Linux Kernel documentation*. (2022). Accessed: Apr. 22, 2024. [Online]. Available: https://docs.kernel.org/locking/locktypes.html

[27] The Linux Foundation. *Threaded interrupt handler*. (2023). Accessed: Jan. 4, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/threadirq

[28] The Linux Foundation. *Sched(7) - Linux manual page*. (2023). Accessed: Jan. 10, 2024. [Online]. Available: https://man7.org/linux/man-pages/man7/sched.7.html

[29] T. Gleixner and D. Niehaus, "Hrtimers and beyond: Transforming the Linux time subsystems," 2006. [Online]. Available: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf.

[30] The Linux Foundation. *How to use high resolution timers?* (2023). Accessed: Jan. 26, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/hr_timers

[31] stress-ng (stress next generation). (2023). C. I. King. Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/ColinIanKing/stress-ng

[32] rt-tests. (2024). The Linux Foundation. Accessed: Mar. 17, 2024. [Online]. Available: https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/

[33] The Linux Foundation. *Hackbench*. (2023). Accessed: Mar. 17, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/hackbench

[34] S. Rostedt. *Ftrace - Function Tracer*. (2022). Accessed: Oct. 12, 2023. [Online]. Available: https://www.kernel.org/doc/html/v6.1/trace/ftrace.html

[35] KernelShark. (2023). The Linux Foundation. Accessed: Oct. 12, 2023. [Online]. Available: https://git.kernel.org/pub/scm/utils/trace-cmd/kernel-shark.git/

[36] Matt Fleming. "A thorough introduction to eBPF." lwn.net. Accessed: Feb. 6, 2024. [Online]. Available: https://lwn.net/Articles/740157/

[37] J. Ogness. (2023). Proposing a new tracer to monitor RT-task behavior. Presented at the Embedded Open Source Summit, Prague, Czech Republic. Accessed: Aug. 23, 2023. [Online]. Available: https://static.sched.com/hosted_files/eoss2023/27/Proposing%20a%20new%20tracer%20to%20monitor%20RT%20task%20behavior%20-%20John%20Ogness.pdf

[38] The Linux Foundation. *Cyclictest*. (2023). Accessed: Jan. 15, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

[39] rtla. (2023). D. B. de Oliveira. Accessed: Oct. 1, 2023. [Online]. Available: https://www.kernel.org/doc/html/next/tools/rtla/rtla.html

[40] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys*, vol. 50, no. 6, pp. 1–37, Nov. 2018, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3131347. (accessed Sep. 29, 2023).

[41] P. Barham, B. Dragovic, K. Fraser, *et al.*, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03, New York, NY, USA: Association for Computing Machinery, Oct. 2003, pp. 164–177, ISBN: 978-1-58113-757-6. DOI: 10.1145/945445.945462.

[42] A. Qumranet, Y. Qumranet, D. Qumranet, U. Qumranet, and A. Liguori, "KVM: The Linux virtual machine monitor," *Proceedings Linux Symposium*, vol. 15, Jan. 2007.

[43] rromoff. "Cache Coloring: Interference-free Real-time Virtualization." xenproject.org. Accessed: Oct. 4, 2023. [Online]. Available: https://xenproject.org/2020/09/03/cache-coloring-interference-free-real-time-virtualization/

[44] The Linux Foundation. "Xen For Automotive / Embedded." xenproject.org. Accessed: Feb. 6, 2024. [Online]. Available: https://xenproject.org/users/automotive-embedded/

[45] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," 2005.

[46] I. Habib, "Virtualization with KVM," *Linux Journal*, vol. 2008, no. 166, 8:8, Feb. 2008, ISSN: 1075-3583. [Online]. Available: https://dl.acm.org/doi/fullHtml/10.5555/1344209.1344217.

[47] Jun Zhang, Kai Chen, Baojing Zuo, Ruhui Ma, Yaozu Dong, and Haibing Guan, "Performance analysis towards a KVM-Based embedded real-time virtualization architecture," in *5th International Conference on Computer Sciences and Convergence Information Technology*, Seoul, Korea (South): IEEE, Nov. 2010, pp. 421–426, ISBN: 978-1-4244-8567-3. DOI: 10.1109/ICCIT.2010.5711095.

[48] E. Hamelin, M. Ait Hmid, A. Naji, Y. Mouafo-Tchinda, and J. Land Rover, "Selection and evaluation of an embedded hypervisor: Application to an automotive platform," 2020. [Online]. Available: https://www.mentor.com/embedded-software/hypervisor/.

[49] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2016, pp. 1–12. DOI: 10.1109/RTAS.2016.7461361.

[50] The Linux Foundation. *Control Group v2 — The Linux Kernel documentation*. (2015). Accessed: Feb. 6, 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.1/admin-guide/cgroup-v2.html

[51] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, 2:2, Mar. 2014, ISSN: 1075-3583.

[52] M. Thiyyakat, S. Kalambur, and D. Sitaram, "Improving resource isolation of critical tasks in a workload," in *Job Scheduling Strategies for Parallel Processing: 23rd International Workshop, JSSPP 2020, New Orleans, LA, USA, May 22, 2020, Revised Selected Papers*, Berlin, Heidelberg: Springer-Verlag, May 2020, pp. 45–67, ISBN: 978-3-030-63170-3. DOI: 10.1007/978-3-030-63171-0_3.

[53] The Linux Foundation. *What is an IRQ? — The Linux Kernel documentation*. (2022). Accessed: Dec. 2, 2023. [Online]. Available: https://www.kernel.org/doc/html/v6.1/core-api/irq/concepts.html

[54] The Linux Foundation. *SMP IRQ affinity — The Linux Kernel documentation*. (2022). Accessed: Dec. 2, 2023. [Online]. Available: https://www.kernel.org/doc/html/v6.1/core-api/irq/irq-affinity.html

[55] The Linux Foundation. *The kernel's command-line parameters — The Linux Kernel documentation*. (2022). Accessed: Dec. 2, 2023. [Online]. Available: https://www.kernel.org/doc/html/v6.1/admin-guide/kernel-parameters.html

[56] The Linux Foundation. *Page Tables — The Linux Kernel documentation*. (2022). Accessed: Apr. 22, 2024. [Online]. Available: https://docs.kernel.org/mm/page_tables.html

[57] M. M. Madden, "Challenges using Linux as a real-time operating system," in *AIAA Scitech 2019 Forum*, San Diego, California: American Institute of Aeronautics and Astronautics, Jan. 2019, ISBN: 978-1-62410-578-4. DOI: 10.2514/6.2019-0502.

[58] H. Austad, E. R. Jellum, S. Hendseth, *et al.*, "Composable distributed real-time systems with deterministic network channels," *Journal of Systems Architecture*, vol. 137, p. 102 853, Apr. 2023, ISSN: 13837621. DOI: 10.1016/j.sysarc.2023.102853. (accessed Oct. 2, 2023).

[59] K. G. Erickson, M. D. Boyer, and D. Higgins, "NSTX-U advances in real-time deterministic PCIe-based internode communication," *Fusion Engineering and Design*, vol. 133, pp. 104–109, Aug. 2018, ISSN: 0920-3796. DOI: 10.1016/j.fusengdes.2018.02.055. (accessed Oct. 1, 2023).

[60] S. Khan and E. Copperman. (2023). RT Linux in safety-critical systems: The potential and the challenges. Presented at the Embedded Open Source Summit, Prague, Czech Republic. Accessed: Aug. 23, 2023. [Online]. Available: https://static.sched.com/hosted_files/eoss2023/fd/RT%20Linux%20in%20Safety%20Critical%20Systems_%20the%20potential%20and%20the%20challenges%20.pdf

[61] D. Duval, "From fast to predictably fast," in *Ottawa Linux Symposium*, Montreal, Quebec Canada, 2009. [Online]. Available: https://www.kernel.org/doc/ols/2009/ols2009-pages-79-86.pdf.

[62] John Ogness. "A checklist for real-time applications in Linux." linutronix.de. Accessed: Apr. 24, 2024. [Online]. Available: https://www.linutronix.de/blog.php

[63] The Linux Foundation. *RT-mutex subsystem with PI support — The Linux Kernel documentation*. (2020). Accessed: Apr. 22, 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.1/locking/rt-mutex.html

[64] W. Mauerer, *Professional Linux Kernel Architecture*. GBR: Wrox Press Ltd., Sep. 2008, ISBN: 978-0-470-34343-2.

[65] M. Bagherzadeh, N. Kahani, C.-P. Bezemer, A. E. Hassan, J. Dingel, and J. R. Cordy, "Analyzing a decade of Linux system calls," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, New York, NY, USA: Association for Computing Machinery, May 2018, p. 267, ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3182518.

[66] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, USA: USENIX Association, Oct. 2010, pp. 33–46.

[67] Jonathan Corbet. "(Nearly) full tickless operation in 3.10." lwn.net. Accessed: Dec. 6, 2023. [Online]. Available: https://lwn.net/Articles/549580/

[68] The Linux Foundation. *NO_HZ: Reducing Scheduling-Clock Ticks — The Linux Kernel documentation*. (2023). Accessed: Dec. 6, 2023. [Online]. Available: https://docs.kernel.org/timers/no_hz.html

[69] H. Akkan, M. Lang, and L. Liebrock, "Understanding and isolating the noise in the Linux kernel," *International Journal of High Performance Computing Applications*, vol. 27, pp. 136–146, May 2013. DOI: 10.1177/1094342013477892.

[70] The Linux Foundation. *Overview of the Linux Virtual File System — The Linux Kernel documentation*. (2023). Accessed: Nov. 27, 2023. [Online]. Available: https://www.kernel.org/doc/html/next/filesystems/vfs.html

[71] H. Zhang, H. Zhang, L. Zhang, and Y. Wu, "FastUDP: A highly scalable user-level UDP framework in multi-core systems for fast packet I/O," *The Journal of Supercomputing*, vol. 77, pp. 1–28, May 2021. DOI: 10.1007/s11227-020-03486-6.

[72] J. Kiszka, B. Wagner, Y. Zhang, and J. Broenink, "RTnet – A flexible hard real-time networking framework," Jan. 2005. DOI: 10.1109/ETFA.2005.1612559.

[73] L.-C. Duca and A. Duca, "Achieving hard real-time networking on PREEMPT_-RT Linux with RTnet," in *2020 International Symposium on Fundamentals of Electrical Engineering (ISFEE)*, Nov. 2020, pp. 1–4. DOI: 10.1109/ISFEE51261.2020.9756165.

[74] DPDK Project. "DPDK." dpdk.org. Accessed: Feb. 7, 2024. [Online]. Available: https://www.dpdk.org/

[75] Z. Li, "HPSRouter: A high performance software router based on DPDK," in *2018 20th International Conference on Advanced Communication Technology (ICACT)*, Feb. 2018, pp. 503–506. DOI: 10.23919/ICACT.2018.8323810.

[76] T. Xu, X. Chen, C. Wu, *et al.*, "3DS: An efficient DPDK-based data distribution service for distributed real-time applications," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, Dec. 2022, pp. 1283–1290. DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00201.

[77] DPDK Project. "DPDK - Supported Hardware." dpdk.org. Accessed: Mar. 18, 2024. [Online]. Available: https://core.dpdk.org/supported/

[78] Julie Cummings, Eliezer Tamir, "Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll," 2013. [Online]. Available: https://caxapa.ru/thumbs/793343/Open_Source_Kernel_Enhancements_for_Low-.pdf.

[79] S. A. Siewior. "[PATCH net-next] net/core: disable NET_RX_BUSY_POLL on PREEMPT_RT - Sebastian Andrzej Siewior." lore.kernel.org. Accessed: Nov. 27, 2023. [Online]. Available: https://lore.kernel.org/all/20211001145841.2308454-1-bigeasy@linutronix.de/

[80] D. J. H. Brown and Martin, Brad, "How fast is fast enough? Choosing between Xenomai and Linux for real-time applications," 2010.

[81] R. Wilhelm, J. Engblom, A. Ermedahl, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, 36:1–36:53, May 2008, ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. (accessed Dec. 25, 2023).

[82] D. B. de Oliveira and R. S. de Oliveira, "Timing analysis of the PREEMPT RT Linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 789–819, 2016, ISSN: 1097-024X. DOI: 10.1002/spe.2333.

[83] P. Okech, N. Mc Guire, and C. Fetzer, "Utilizing Inherent Diversity in Complex Software Systems," pp. 71–78, 2014.

[84] I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve, and R. Obermaisser, "Towards Linux based safety systems—A statistical approach for software execution path coverage," *Journal of Systems Architecture*, vol. 116, Jun. 2021, ISSN: 13837621. DOI: 10.1016/j.sysarc.2021.102047.

[85] X. Chen, "Research on Linux real-time and performance evaluation for Loongson 3A3000 processor," in *Journal of Physics: Conference Series*, Cited By :2, vol. 1453, 2020. DOI: 10.1088/1742-6596/1453/1/012100.

[86] Stian Onarheim / masters-thesis - GitLab. (2024). Stian Onarheim. Accessed: May 17, 2024. [Online]. Available: https://gitlab.com/Feqzz/masters-thesis

[87] Texas Instruments. *AM69x Processors, Silicon*. (2023). Accessed: Oct. 1, 2023. [Online]. Available: https://www.ti.com/lit/ds/symlink/am69.pdf?ts=1696168389669&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FAM69

[88] Texas Instruments. "SK-AM69 Evaluation board." ti.com. Accessed: Apr. 3, 2024. [Online]. Available: https://www.ti.com/tool/SK-AM69

[89] Buildroot. "Buildroot - Making Embedded Linux Easy." buildroot.com. Accessed: Apr. 3, 2024. [Online]. Available: https://buildroot.org/

[90] ti-linux-kernel. (2024). Texas Instruments. Accessed: Nov. 26, 2023. [Online]. Available: https://git.ti.com/cgit/ti-linux-kernel/ti-linux-kernel/

[91] AMD. "AMD Zynq 7000 SoC ZC706 Evaluation Kit." xilinx.com. Accessed: Mar. 30, 2024. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html

[92] Xilinx. *Zynq-7000 SoC Data Sheet: Overview (DS190)*. (2018). Accessed: Aug. 13, 2023. [Online]. Available: https://docs.amd.com/v/u/en-US/ds190-Zynq-7000-Overview

[93] FreeRTOS. "FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions." freertos.org. Accessed: Mar. 30, 2024. [Online]. Available: https://www.freertos.org/index.html

[94] LwIP. (2023). Xilinx. Accessed: Apr. 28, 2024. [Online]. Available: https://github.com/Xilinx/embeddedsw/tree/master/ThirdParty/sw_services/lwip213

[95] BRAM Standalone driver. (2023). Xilinx. Accessed: Apr. 28, 2024. [Online]. Available: https://github.com/Xilinx/embeddedsw/tree/master/XilinxProcessorIPLib/drivers/bram

[96] Nordic Semiconductor. *nRF52832 Product Specification (v1.8)*. (2021). Accessed: Aug. 13, 2023. [Online]. Available: https://infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.8.pdf

[97] Nordic Semiconductor. "nRF52 DK - Development kit for Bluetooth Low Energy and Bluetooth mesh." nordicsemi.com. Accessed: Apr. 3, 2024. [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nRF52-DK

[98] xilinx/dma_ip_drivers. (2024). Xilinx. Accessed: Mar. 16, 2024. [Online]. Available: https://github.com/Xilinx/dma_ip_drivers

[99] ti_rt.config. (2024). Texas Instruments. Accessed: Apr. 24, 2024. [Online]. Available: https://git.ti.com/cgit/ti-linux-kernel/ti-linux-kernel/tree/kernel/configs/ti_rt.config?h=ti-rt-linux-6.1.y

[100] iovisor/bcc. (2024). iovisor. Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/iovisor/bcc

[101] The Linux Foundation. *Libbpf Overview — The Linux Kernel documentation*. (2023). Accessed: Mar. 10, 2024. [Online]. Available: https://docs.kernel.org/bpf/libbpf/libbpf_overview.html

[102] The Linux Foundation. *Unreliable Guide To Hacking The Linux Kernel — The Linux Kernel documentation*. (2022). Accessed: Mar. 16, 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.1/kernel-hacking/hacking.html

[103] S. A. Siewior. (2024). How to not break PREEMPT_RT. Presented at the Embedded Open Source Summit, Seattle, Washington, USA. [Online]. Available: https://static.sched.com/hosted_files/eoss24/b2/EOSS_2024-HowToNotBreakPreemptRT.pdf

[104] L.-C. Duca, A. Duca, and A.-S. Lup, "Real-time Linux drivers and latency evaluation system for TI OMAP4 mcSPI peripheral," in *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, Istanbul, Turkey: IEEE, Jun. 2020, pp. 1–4, ISBN: 978-1-72817-116-6. DOI: 10.1109/ICECCE49384.2020.9179286.

[105] Vaishnav Achath, Vignesh Raghavendra, and Keerthy J. (2023). Tweaking device drivers for achieving real-time performance in embedded systems using real-time Linux. Presented at the Open Source Embedded Summit, Prague, Czech Republic. Accessed: Apr. 15, 2024. [Online]. Available: https://static.sched.com/hosted_files/eoss2023/ce/RT_Linux_Device_Driver.pdf

[106] The Linux Foundation. *Pthread_mutexattr_getprotocol(3) - Linux man page*. (2023). Accessed: Apr. 15, 2024. [Online]. Available: https://linux.die.net/man/3/pthread_mutexattr_getprotocol