CHALMERS
UNIVERSITY OF TECHNOLOGY

Embedded Linux as Platform Enabler for Real-Time Sensor Control Applications

Master's thesis in Embedded Electronic System Design

Alexandra Hirschfeldt
Johannes Hjalmarsson

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025

Master's thesis 2025

Embedded Linux as Platform Enabler for Real-Time Sensor Control Applications

Alexandra Hirschfeldt
Johannes Hjalmarsson

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2025

Embedded Linux as Platform Enabler for Real-Time Sensor Control Applications
Alexandra Hirschfeldt, Johannes Hjalmarsson

Supervisor: Per Larsson-Edefors, Department of Microtechnology and Nanoscience
Company advisor: Johan Friberg, Saab AB
Examiner: Lena Peterson, Department of Microtechnology and Nanoscience

Master's Thesis 2025
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Embedded Linux as Platform Enabler for Real-Time Sensor Control Applications
Alexandra Hirschfeldt, Johannes Hjalmarsson
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

With the recent introduction of the PREEMPT_RT patch into the mainline kernel, the open-source operating system Linux is becoming a more formidable real-time platform. The patch introduces the fully preemptible kernel mode, which can be enabled when building the Linux kernel. This mode increases real-time performance by improving the way real-time tasks preempt lower priority tasks. Sensor control systems are generally dependent on real-time performance, and thus requires such optimizations to meet time constraints. This project considers a series of fully preemptible Linux setups as a prospective platform for sensor control systems, and compares them to reference setups using previous preemption modes. To evaluate the setups, a series of latency measurements have been made at different background load conditions using different schedulers, as well as the impact of CPU-isolation. The measurements show that significant gains in real-time performance can be made using the fully preemptible kernel mode, but also shows that the mode increases overhead for non-real-time tasks.

Keywords: Real-time performance; Linux; PREEMPT_RT; Kernel configuration; Wake-up latency; Embedded platform; Sensor control applications.

Acknowledgements

We would like to thank our industrial supervisors Johan Friberg, Jerker Bengtsson, and David Wilkins for their invaluable support throughput the project. During our weekly meetings, they shared their expertise, offered valuable advice, and guided us through theoretical discussions and practical implementation approaches.
Furthermore, we would like to thank Rasmus Kallio Mattsson and Benjamin Hallenrud at Saab for taking the time to help us set up the hardware platform in the lab and for demonstrating how to build and compile the Linux kernels.
We are also deeply grateful to our academic supervisor at Chalmers, Per Larsson-Edefors, for the valuable insights and feedback he provided on the report as well as the discussions on methodology, which offered us new perspectives.

Alexandra Hirschfeldt, Johannes Hjalmarsson, Gothenburg, June 2025

Acronyms

RT	Real-Time
RTOS	Real-Time Operating System
GPOS	General Purpose Operating System
OS	Operating System
CPU	Central Processing Unit
I/O	Input/Output
CFS	Completely Fair Scheduler
FIFO	First-In-First-Out
ISR	Interrupt Service Routine
RCU	Read-Copy-Update
SMT	Simultaneous Multithreading
IRQ	Interrupt Request
APU	Application Processing Unit
RPU	Real-Time Processing Unit
L1	Level 1
L2	Level 2
FSBL	First Stage Boot Loader
IQR	Inter-Quartile Range
EEVDF	Earliest Eligible Virtual Deadline First

Contents

1

Introduction

Real-time systems are computer systems designed to reliably meet real-time constraints. For example, a real-time system running a sensor control application may read and process inputs from a sensor at specific times separated by a time interval. For such a sensor control application, it is important that the sensor task starts execution in time, and finishes execution before the next time interval. Another example of real-time systems are video streaming applications, in which the frames of a video must be displayed at a set frame-rate.

As discussed, there are various types of real-time systems, which often vary in importance of meeting their real-time constraints. To that end, there are two categories of real-time systems; soft real-time systems in which failing to meet a constraint only results in a temporarily diminished quality of the application's service, and hard real-time systems where failing to meet a constraint is considered a catastrophic event. Many real-time applications, especially hard real-time systems, run on dedicated operating systems known as real-time operating systems (RTOS). Such operating systems are designed with the handling of hard real-time applications in mind, which means that they prioritize time determinism and concurrency [1]. This is in stark contrast to more general operating systems, such as Linux, where attributes such as fairness, speed and good resource management are paramount, much to the detriment of its real-time capabilities [1].

Because of its general nature, the operating system Linux features a rich and extensive code base due to its widespread usage and open-source license. This means that modifying Linux to gain real-time capabilities could potentially serve as a platform for real-time applications, such as sensor control applications, while simultaneously leveraging that code base. Due to the open-source nature of Linux, its usage could also provide financial benefits in comparison to licensed competitors.

One way to add real-time capabilities to a Linux-based platform is through real-time patches, such as the PREEMPT_RT patch, which modifies the Linux kernel with enhanced real-time capabilities through lower worst-case latencies and improved time determinism [2].

The goal of this project is to investigate the real-time capabilities of a Linux-based real-time platform using the PREEMPT_RT patch, and to determine to what extent the platform is feasible to use in sensor control applications.

To determine this, a Linux-based real-time platform must first be implemented. To determine its capabilities, it must then be tested and compared to relevant baselines, such as non-RT Linux or modified variants of the existing platform.

1.1 Related Work

There have been multiple attempts to modify Linux for real-time performance. A survey in [3] details the problems with using Linux for real-time applications and the solutions adopted by various Linux-based platforms. Broadly, it describes two main approaches: patching the mainline kernel to improve preemptibility and reduce latency; or using a dual-kernel setup, where a real-time kernel handles time-critical tasks while Linux runs as a low-priority process. While dual-kernel solutions may provide lower latency and better determinism, the single-kernel approach has become more desirable due to its compatibility with mainline Linux, simpler setup, and easier application development.

The work in [4] provides a detailed survey on the PREEMPT_RT patch, which enhances kernel preemptibility through features like threaded interrupt handlers, high-resolution timers, and priority inheritance. However, the authors emphasize that these capabilities from the patch must be complemented with proper system configuration to achieve reliable low-latency performance. Techniques such as locking memory pages, setting processor and interrupt request affinities are critical, particularly in multicore processors where shared resources increase complexity and unpredictability. The authors also highlight that evaluating real-time capabilities remains difficult due to the complexity of a general-purpose operating system (GPOS) like Linux, inconsistent benchmarking, limited reproducibility, and strong dependence on hardware platforms.

A study that evaluates the real-time performance of the PREEMPT_RT patch on ARM-based platforms is [2]. The authors tested three Linux distributions (Debian, Ubuntu, and Arch Linux), measuring response latencies. Although not targeting a specific real-time application, the study used an upper latency threshold to assess the suitability of real-time performance.

The test setup in [2] involved a master-slave configuration with GPIO-triggered interrupts, and both custom measurement modules and the *cyclictest* (a benchmarking tool developed by mainline kernel developers) to validate their results. The results showed that PREEMPT_RT significantly reduced worst-case latencies across all distributions, with little variations between platforms. However, the study pointed out that the worst-case latency might still not be enough for certain real-time applications with strict timing constraints.

Interestingly, the authors of [2] did not apply common tuning strategies such as setting processor affinities, despite acknowledging that this could reduce inter-core interference. Although memory locking and high-priority scheduling were used, it remains unclear which, if any, additional kernel parameters were configured beyond simply enabling full preemption provided by the patch. This contrasts to [4], which emphasizes the importance of carefully tuning the system parameters to achieve optimal real-time performance.

Additional relevant studies, [5], [6], and [7], also evaluate the performance of the PREEMPT_RT patch using the *cyclictest* benchmark to measure worst-case scheduling latency under various conditions.

The study in [5] examines single- and multi-core ARM-based computers, showing that multi-core platforms can yield better real-time responsiveness than single-core ones under load. However, neither system met the strict latency requirement of the target control application in the study.

In [6], the authors evaluate a dual-core Intel system, comparing several kernel configurations, including the PREEMPT_RT and the standard Linux kernel. Their results showed that the patched kernel outperformed the standard kernel. However, like [5], this study provides limited information on kernel settings and workload intensity, which limits reproducibility and makes comparisons less reliable.

The most comprehensive evaluation is presented in [7], which compares PREEMPT_RT against a standard Linux kernel on a 16-core Intel Xeon platform. This work uses processor affinity to restrict one thread per core, real-time scheduling with highest priority, memory locking, and disables features such as multithreading, frequency scaling and debugging options. Three workload scenarios were tested: idle, memory-intensive, and interrupt-intensive. PREEMPT_RT showed clear advantages under load, particularly in the interrupt-heavy scenario, where the standard kernel yielded latencies in the millisecond range.

1.2 Purpose and Goal

The primary objective of this project is to evaluate to what extent Linux, enhanced with real-time capabilities through the PREEMPT_RT patch, can meet the demands of modern applications with real-time constraints, particularly in the context of sensor control systems. Specifically, the goal is to develop a methodology for assessing the capabilities of the patched real-time Linux kernel through a software tool that gathers data about the system's real-time capabilities. Key performance metrics include latency bounds, with a focus on worst-case and average values, as well as jitter, which reflects variability in latency across repeated measurements.

A central challenge in evaluating real-time performance is setting up test scenarios that accurately reflect realistic operating conditions. To address this, we aim to conduct latency measurements under a range of representative workload conditions.

Building on this, we will design structured test setups that allow us to systematically investigate how different features offered by the kernel of the operating system (OS), identified from the literature, affect latency. These setups should isolate the impact of each contributing factor, including aspects of the OS itself, the specific processor, and the test applications, to clearly identify the significance of each factor and ensure meaningful and consistent results.

In addition to testing real-time modified Linux, standard Linux (without real-time patches) will be evaluated on the same hardware for comparison. Comparing these platforms will enable us to assess the impact of Linux with the PREEMPT_RT patch and determine to what extent the measured latencies meet the needs of real-time applications.

Since acceptable latency thresholds depend on the specific application, the results will be interpreted in the context of potential target real-time tasks, which, for this

project, includes sensor control applications.

1.3 Delimitation

Due to time constraints, only a limited number of test setups can be implemented and tested within the scope of this project. Thus, the test setups chosen for implementation must be prioritized to only those which provide the most valuable data in order to evaluate the project's thesis.

Since only a single hardware platform has been made available for this project, that hardware platform also acts as a limiting factor for what software can be utilized, and thus what test setups are possible to implement.

Furthermore, since most of the work done as part of this project will be completed inside Saab's internal network, any software used for the project must also meet Saab's strict security requirements. To gain access to external software, that software must first be examined for licenses and harmful code, and subsequently approved by the company for use. This means that the software available for use in this project is limited to the subset of software that meets Saab's security requirements.

1.4 Thesis Outline

This thesis is organized as follows. Chapter 1 introduced the background and purpose of the work, namely to evaluate the real-time capabilities of Linux using the PREEMPT_RT patch. Chapter 2 presents a literature review, starting with an overview of real-time computing followed by fundamental concepts of operating systems and additional features offered by the PREEMPT_RT patch. Chapter 3 describes the resource method, including the hardware platform, software versions, and tools used. It also presents the development of the test software for latency measurement and the approach for setting up benchmarks and other kernel setups. Chapter 4 details further design specifications of the test software and kernel setups. Chapter 5 presents the measurement results obtained from the various test scenarios. Chapter 6 includes a comprehensive discussion of the results reflecting back on the initial goals of the thesis. It also provides suggestions for future work. Finally, Chapter 7 concludes the thesis by summarizing the important findings.

2

Technical Background

In this chapter, we present the key concepts of real-time computing and describe the characteristics of real-time operating systems and how they differ from a GPOS, specifically focusing on Linux. Next, we describe existing approaches to improve the real-time capabilities of Linux, including the use of the PREEMPT_RT patch. Finally, relevant Linux kernel parameters and configurations to optimize real-time performance are introduced.

2.1 Real-Time Systems

Real-time systems is a term that refers to systems in which producing correct results depend not only on delivering logical correctness, but doing so within specified time constraints [8]. Based on the severity of the consequences of a missed deadline, real-time systems are classified into two categories, hard real-time systems and soft real-time systems.

A system is classified as soft real-time system if a missed deadline is tolerable to some extent. In such cases, the broken time constraint leads to a temporary loss in the quality of service of the system [9]. A soft real-time system could be for example a multimedia processing software, such as a process synchronizing audio and video streams. In contrast, a system is considered hard real-time if missing a deadline would lead to unacceptable consequences. These systems must be carefully designed to ensure that they maintain the time constraints, even in a worst-case scenario. Examples of a hard real-time system could be for example a flight-control system aboard an aircraft, where a failure of the system could potentially lead to a crash.

A common misconception about real-time systems is they should always be as fast as possible [10]. Rather than speed, the key aspect of real-time computing is maintaining time determinism, ideally ensuring that there is no timing variation between identical tasks. This guarantees predictable behavior, which is essential for the system to function correctly and consistently meet timing constraints. Both specialized hardware and software design are crucial in real-time systems to achieve this reliability.

2.1.1 Response Time, Latency and Jitter

The response time in real-time systems refers to the time it takes for the system to react to an event or input and produce the corresponding output or response [10]. Latency is a term similar to response time, also denoting a time difference between an event and a response, but is typically used when discussing internal delays in a system, such as a scheduling latency or network latency [4].

Another important term is jitter, which denotes the deviation of a response time or latency from the expected value, reflecting uncertainty in the system. Jitter could for example be denoted as bounds in the form of a minimum and maximum value of latency in a series of measurements. To preserve temporal correctness, a real-time system must ensure that the system's jitter is bounded by its corresponding time constraints.

2.1.2 Hardware for Real-Time Systems

Real-time systems can be built using various hardware platforms, but are most commonly implemented on microprocessors [11]. The hardware in these systems generally consist of three main components; a central processing unit (CPU), memory, and input/output (I/O) devices, all interconnected via a system bus. Such I/O devices may include sensors, transducers, displays, or switches.

Real-time systems may be implemented using either a single-processor or a multiprocessor architecture [11]. In multiprocessors, the system supports multiple CPUs, enabling them to schedule and execute tasks in parallel. However, multiprocessing also introduces significant challenges, such as task synchronization, scheduling complexity, and shared resource management, all of which can introduce time uncertainty.

Instructions are fetched sequentially from the main memory, decoded by the CPU, and executed [11]. In modern CPUs, computations are much faster than memory, meaning that accessing memory can slow down execution. To address this, modern computing systems use a memory hierarchy, where several levels of progressively smaller and faster memories known as caches exist in-between the main memory and the CPU cores. Figure 2.1 shows a memory hierarchy in a quad-core CPU with three cache levels; L1, L2, and L3. In this instance, each core has individual level one and two caches, while all cores share a common third level.

When the CPU requests a piece of data from the memory, it checks each level of cache in order, ideally finding the data saved in one of the lower level caches from a previous memory request, rather than in the slower main memory. Each time the CPU checks a cache unsuccessfully, it is called a cache miss, with each subsequent cache miss adding an additional delay to the memory request. Once the data has been found, it is saved at all levels of the memory hierarchy in the hopes that the next request for that data will cause fewer cache misses.

Even more severe are the delays caused by page faults, which occur when the data being accessed is not in main memory and must be fetched from secondary memory, such as a hard drive. These delays are much longer and can significantly affect

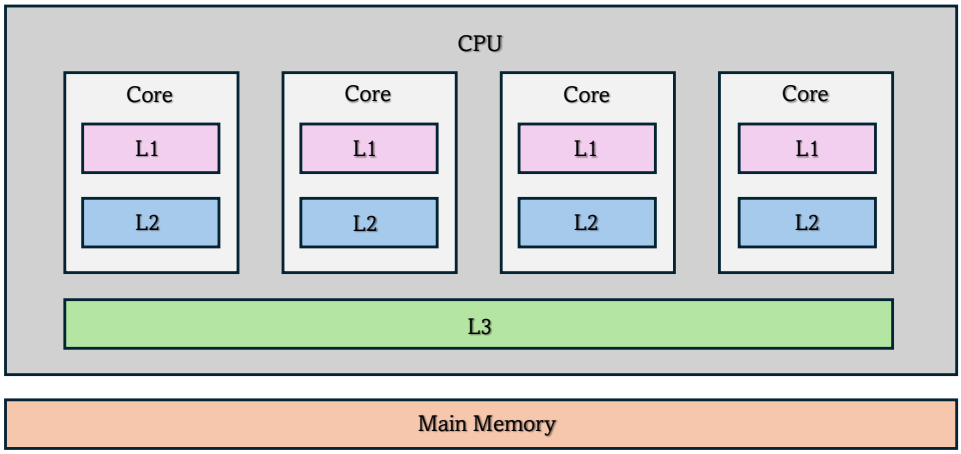


Figure 2.1: Example of a memory hierarchy in a quad-core processor. Each core has a private L1 and L2 cache, while all share a common L3 cache. Main memory is accessed outside the CPU.

performance [8]. However, page faults can be avoided by using memory locking, which allows the system to lock specific parts of the main memory, typically related to a real-time application, and prevent it from being overwritten when the main memory is out of capacity.

2.2 Real-Time Operating Systems

A common way way to improve the real-time performance of a real-time system is to use a specialized operating system, called a real-time operating system (RTOS). The main purpose of an RTOS is to provide time-determinism to a real-time system through the upper bounding of execution time, with low latency and high throughput being positive but less important traits [4]. To provide this, RTOS must use certain techniques, such as specialized schedulers.

The scheduler is the part of an operating system used to divide computational resources between software tasks on a concurrent computer system [12]. In general-purpose operating systems, schedulers are typically constructed with scalability, interactivity and fairness in mind. As one example, the default Linux scheduler Completely Fair Scheduler (CFS) uses soft priorities to provide both control and fairness to scheduled processes.

In another instance, static-priority preemption real-time schedulers, such as the built-in `SCHED_FIFO` real-time scheduler in Linux, assign a priority to each task running on the system, and allow higher-priority tasks to preempt lower-priority tasks, ceding the CPU to the higher priority task in a context switch [13]. By assigning real-time tasks with time constraints a higher priority, the real-time scheduler ensures that real-time tasks are not delayed by other general tasks. Figure 2.2 illustrates a simplified view of preemptive scheduling, where higher priority tasks preempt lower priority ones, forcing them to wait or pause execution.

One of the main ways schedulers impact real-time performance is through context

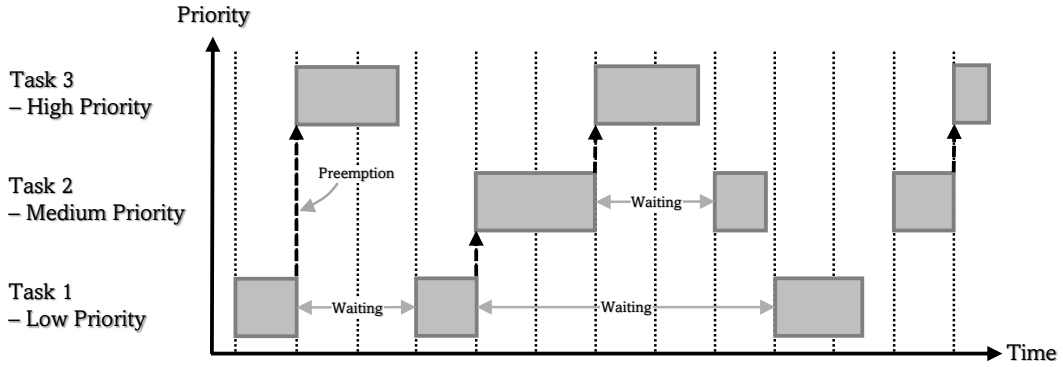


Figure 2.2: Simplified timeline of preemptive scheduling across tasks with different priorities. The scheduler is shown as being invoked periodically (e.g., by timer ticks), marked by dashed vertical lines. In practice, it may also be invoked aperiodically by task events or interrupts.

switching, which occurs each time the system switches from one task to another, including during preemption. Context switches typically require some time in order to prepare the system for the incoming task, adding to the latency of real-time tasks. This also means that a high number of context switches during execution reduces the throughput of the system.

Computer systems generally rely on two timers for their operation; a hardware timer that controls instruction cycles, and an operating system timer that controls CPU time [14]. The operating system timer in particular is used for things such as preemption, which means that the period of that timer becomes the resolution at which timing can be controlled. Thus, a higher resolution operating system timer should improve a system's real-time capabilities.

2.2.1 Real-Time Linux

Several successful attempts have been made so far to develop a Linux-based real-time system. Most such systems are constructed using a so-called dual-kernel approach. The dual-kernel approach uses a second kernel called a co-kernel, which acts as a layer between the Linux kernel and the hardware [4]. The co-kernel can then handle real-time tasks at an elevated level, ensuring that the time constraints of such tasks are followed. Examples of dual-kernel systems are RTLinux, RTAI and Xenomai [4]. While the dual-kernel approach has been shown to achieve a remarkable level of real-time performance, its applicability is limited by the fact that it requires special system calls to be used [4]. This means that the pre-existing Linux code base cannot be utilized without significant modification.

A simpler approach is the single-kernel approach, in which the Linux kernel itself is modified in order to enhance its real-time capabilities. Following the single-kernel approach, the PREEMPT_RT patchset for Linux was developed in order to increase preemptibility of kernel code [4]. In contrast to the dual-kernel approach, the PREEMPT_RT patch can be used in conjunction with the preexisting Linux code base.

2.2.2 Features of the PREEMPT_RT Patch

The PREEMPT_RT patch reduces non-preemptible sections in the kernel code through several features, including threaded interrupt service routines (ISRs), priority inheritance, and the replacement of spinlocks with mutexes [15]. With threaded interrupts, interrupt handling can be scheduled according to thread priorities, reducing the interrupt delay for high-priority threads. Figure 2.3 illustrates how an incoming interrupt affect an already running high-priority task, comparing the behavior with and without threaded interrupts. Upon arrival of an interrupt with support for threaded interrupts, it only requires masking the interrupt and waking the associated thread, rather than fully servicing the interrupt in a non-preemptible context. However, some ISRs must still run in an interrupt context and cannot be threaded, such as the timer interrupt.

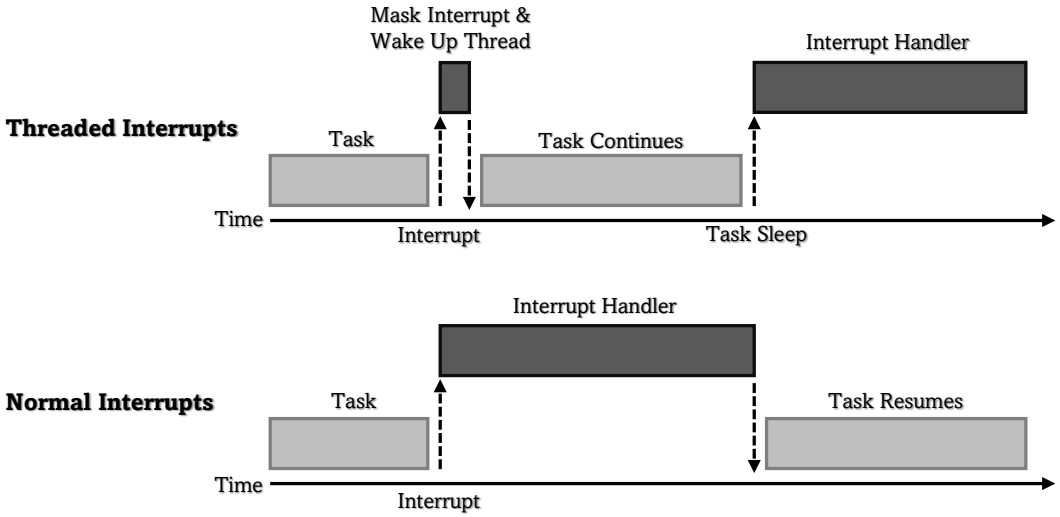


Figure 2.3: Execution timeline of a processor illustrating the difference between normal and threaded interrupts upon an incoming interrupt.

Another important feature is the replacement of most spinlocks with sleeping spinlocks (or mutexes). Spinlocks are used to protect critical sections in the kernel by disabling preemption and preventing concurrent access, causing threads that attempts to access a locked section to busy-loop (spin) [15]. This can lead to unbounded latencies for high-priority threads, even if those threads do not require access to the locked section, since preemption remains temporarily disabled. To resolve this, the PREEMPT_RT patch replaces spinlocks with mutexes, causing threads that attempt to access already occupied critical sections to sleep until the mutex is released, allowing preemption to continue for other high-priority threads.

Furthermore, the patch implements priority inheritance to address priority inversion [4], [15]. Priority inversion, as illustrated in Figure 2.4, occurs when a high-priority thread is blocked by a lower-priority thread for holding a mutual resource, and the lower-priority thread is itself preemped by a medium-priority thread, indirectly preempting the high-priority thread as well. The priority inheritance mechanism resolves this by letting the lower-priority thread temporarily inherit the higher priority of the blocked thread.

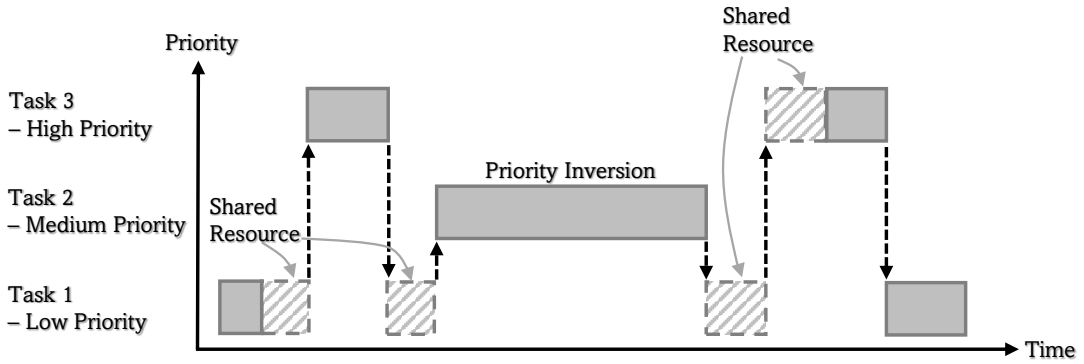


Figure 2.4: Illustration of priority inversion with three tasks. A lower-priority task blocks a higher-priority one due to shared resources, while a medium-priority task preempt the lower-priority task.

2.3 Linux Kernel Parameters and Configurations

The Linux kernel provides a wide range of configurable options for customization and optimization to suit different use cases [16]. This is particularly relevant for real-time purposes, as most Linux distributions favor throughput over latency and responsiveness by default. Additionally, they often include unnecessary features that can be removed to improve performance and efficiency.

When adjusting the kernel with the desired configuration options before compilation, it is common to build upon a default kernel configuration setup and adjust it to match the target hardware and requirements [16]. After compiling the kernel, parameters can be passed via the bootloader at startup or adjusted dynamically at runtime. These parameters can, for instance, optimize how the kernel utilizes hardware resources, such as how workloads are distributed across CPUs and managed by the scheduler. Most boot parameters are associated with specific kernel subsystems and take effect only if the corresponding features are built into the kernel [17].

One of the most challenging aspects of customizing a kernel is identifying the necessary drivers and configuration options for a given system [16]. Common build options and kernel parameters relevant for real-time performance are discussed in the following subsections.

2.3.1 Kernel Build Options for Real-Time Linux

Linux prioritizes throughput over latency and determinism [16], making it unsuitable for real-time applications by default. However, it can be configured to better accommodating real-time tasks by applying the PREEMPT_RT patch [18], which has already been integrated into the mainline kernel as of version 6.12. This patch enables full preemption through the `CONFIG_PREEMPT_RT` kernel build option.

Traditional Linux typically runs with no support for preemption in the kernel (`CONFIG_PREEMPT_NONE`), which maximizes throughput at the cost of increased latency [16], [18]. With this mode enabled, there cannot be any time-bounded guarantees from the kernel [1]. The mainline Linux kernel does include options that support

preemption within the kernel, such as CONFIG_PREEMPT [18], which was introduced as early as version 2.6. While CONFIG_PREEMPT allows most kernel code to be preempted, the CONFIG_PREEMPT_RT option makes the kernel code fully preemptible. This improves determinism within the kernel and reduces maximum latency.

Simply enabling full preemption is not necessarily enough to achieve real-time performance [19]. Additional kernel build options should be carefully selected to minimize latencies and ensure deterministic behavior. One key feature is high-resolution timers (CONFIG_HIGH_RES_TIMERS), which improve the precision of system timers beyond the jiffy-based resolution controlled by CONFIG_HZ [1], [20].

The CONFIG_HZ parameter determines the system's clock tick rate, with selectable values 100 Hz, 250 Hz, 300 Hz, or 1000 Hz, affecting scheduling and traditionally also timing accuracy. However, with high-resolution timers enabled, the timer precision of some temporal events becomes independent of CONFIG_HZ, allowing for microsecond or nanosecond precision, depending on the hardware [1].

This means that setting a higher timer frequency is not necessary for better resolution, which could be beneficial in some situations. While a higher timer frequency improves system responsiveness, it may also increase jitter and reduce performance due to excessive interrupts from the software clock [21]. Choosing the right frequency depends on the specific system and use case.

The impact of the timer interrupt on the system also depends on the selected timer tick mode [21]. The Linux kernel provides three build options: periodic mode (CONFIG_HZ_PERIODIC), tickless idle mode (CONFIG_NO_HZ_IDLE), and full tickless mode (CONFIG_NO_HZ_FULL) [22]. The periodic mode ensures consistent timer interrupts at the configured frequency. This gives a predictable interrupt pattern, which is described as the preferred setting for real-time systems [22], [23]. In contrast, tickless idle mode disables timer ticks on idle CPUs to reduce power consumption, but doing so increases wake-up latency. The full tickless mode almost completely removes timer interrupts on specified CPUs when there is only one running task on the processors [1]. This mode is particularly useful for single-threaded, latency-sensitive tasks since it eliminates unnecessary timer interrupts, as the main purpose of the timer is to manage scheduling of processes to ensure multitasking [1], [22]. However, the full tickless mode requires, during boot time, to specify which cores to disable the timer interrupt as not all cores are allowed to run in this mode. Additionally, read-copy-update (RCU) callback offloading (CONFIG_RCU_NOCB_CPU) must be enabled to allow full tickless mode.

RCU is a synchronization mechanism in the Linux kernel that enables reader to access data without being blocked by concurrent updates [21]. Once all read operations have completed, an RCU callback is invoked to free or update resources. RCU callbacks can queue up on CPUs and allocate significant memory when they run, leading to large latencies and OS jitter [24]. By enabling CONFIG_RCU_NOCB_CPU, these callbacks can be offloaded on processors at boot time, preventing them from interfering with critical real-time tasks on specified cores [19], [24].

Another important factor is CPU frequency scaling (CONFIG_CPU_FREQ), which allows the processor clock speed to be dynamically changed at run-time to optimize

power consumption in relation to the workload [16]. This option requires a government policy to be enabled by default to control the scaling. However, frequency scaling introduces unpredictable delays, making it undesirable during real-time processing [21]. To ensure maximum and stable performance, it is recommended to use the performance governor (`CONFIG_CPU_FREQ_DEFAULT_GOV_PERFORMANCE`) [23], which enforces the CPU to run at its highest clock speed.

The symmetric multiprocessor (SMP) option (`CONFIG_SMP`) and the core isolation option (`CONFIG_CPU_ISOLATION`) are two straightforward options to enable, as they support multiprocessing for systems with multiple CPUs or multicore CPU and allow for the isolation of cores, respectively. It is often desirable to isolate critical cores so that dedicated real-time tasks can run on them without being disturbed by non-critical tasks [1], [19].

To further improve preemptibility, `CONFIG_IRQ_FORCED_THREADING` option allows interrupt handlers to run as kernel threads by default, except for interrupt handlers explicitly marked not to [20]. This ensures that user-space threads with higher priority can preempt lower-priority interrupt handlers, bounding maximum scheduling latency for critical threads. However, this option is automatically applied when the PREEMPT_RT patch is enabled.

There are certain kernel features that should be disabled as they can negatively impact real-time performance. Power management options typically trade performance for power savings, which introduces variability in response and unwanted latencies [1], [21]. Debugging options should also be avoided, as they typically include heavy logging that can cause significant delays. Hyper-threading, or simultaneous multithreading (SMT), can also negatively impact real-time performance as it gives unpredictable behavior and increases jitter.

2.3.2 Kernel Boot Parameters for Real-Time Linux

Build options enable specific functionality and set many kernel parameters by default. However, some kernel parameters must be explicitly set at boot time to take effect. For real-time tuning, several of these boot-time parameters (or kernel arguments) can be useful for optimizing performance.

One important aspect of real-time tuning is ensuring that critical tasks run on dedicated CPU cores without interference [19]. This involves isolating certain cores from the general process scheduler and pinning specific tasks to them, such as real-time tasks [17], [25]. Figure 2.5 illustrates an example setup in which one core is isolated in a quad-core processor. The *isolcpus* boot parameter is used to isolate processors by specifying a list of cores to remove from SMP balancing and scheduling algorithms, preventing the kernel from assigning user-space processes to them. Only user-space processes or threads explicitly pinned to these isolated cores by setting CPU affinity will run on them, which is useful when real-time and non-real-time processes coexist [1]. By specifying a process or thread to run on an isolated processor, it minimizes context switching, which is a common source of latency.

Additional flags can refine this isolation further. The *domain* flag disables scheduler balancing on isolated cores and is the default if no other flags are specified [17], [19].

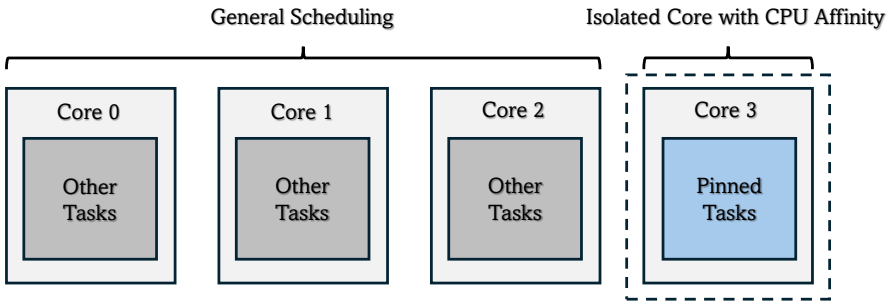


Figure 2.5: Task scheduling across CPU cores, where core 3 is isolated from the general OS scheduler and can only run tasks explicitly pinned through CPU affinity.

The *nohz* flag stops the periodic scheduler tick when only a single task is running, reducing unnecessary CPU interruptions from the timer. The *managed_irq* flag indicates that the kernel should try and prevent external interrupts from targeting the isolated cores, basically affecting interrupt routing decisions [17].

The *isolcpus* parameter only guarantees isolation from other user space processes, but not, for example, from certain kernel threads or IRQ handlers [1]. Further isolation can be achieved using the *kthread_cpus* boot parameter, which restricts kernel threads to operate on selected cores [19]. Kernel threads can arise from various background tasks and maintenance processes that could interfere with real-time threads if assigned to the same cores. However, this parameter is not currently a part of the mainline Linux kernel, and thus applying a certain kernel patch is necessary for it to be supported.

Managing interrupts is critical in real-time systems, as frequent and unpredictable interrupts from the kernel introduce jitter [21]. The *irqaffinity* parameter allows interrupt requests (IRQs) to be assigned to specific cores by default [25], [26]. If CPUs are isolated for real-time tasks, IRQ affinity can be configured to avoid interrupting them, directing IRQs to other CPUs. Alternatively, since most real-time processes are initiated by external interrupts, IRQs generated by critical hardware devices can be assigned to a specific CPU, with the real-time tasks that depend on these interrupts scheduled to run on the same CPU [1]. This improves efficiency and potentially leads to better performance. To achieve this, the *smp_affinity* file must be modified to direct the specific IRQ to the desired CPU [26].

Another useful parameter for interrupt handling is *threadirqs*, which forces most interrupt handlers to run as kernel threads instead of executing immediately in an interrupt context [27]. This puts a priority on these interrupt threads under a scheduling policy, allowing certain application threads to be enabled to run with higher priority than the threaded interrupt handler. This parameter is especially useful for mainline Linux kernels that lack real-time support, since kernels with the PREEMPT_RT setup already enforce threaded interrupt handling [21], [27]. Although, when handling threaded interrupts, it is essential to understand the system's workload and priorities. If a user task has a higher priority than the interrupt handler for a critical function, e.g., networking, it could cause performance issues if the task is waiting for network data.

As mentioned in the build options, offloading RCU callbacks prevents them from executing on specified CPUs, reducing unnecessary workload in these cores. This is done using the *rcu_nocbs* boot parameter, where critical cores are listed [24]. Additionally, the *rcu_nocb_poll* parameter can be applied to shift the responsibility of waking up RCU offload threads from the CPU to a periodic timer.

To enable full dynamic tickless mode, the *nohz_full* boot parameter must be used to specify a list of cores [22]. This mode eliminates unnecessary kernel overhead caused by timer interrupts when only a single task is running on the core. The *nohz* boot parameter can also be used to enable or disable dynamick tick mode without recompiling the kernel.

In systems with real-time tasks, a potential issue is that if they are always ready to run, they can fully occupy the CPU, blocking other tasks [21]. To prevent this, the kernel includes a mechanism called real-time throttling, which limits how long real-time threads are allowed to occupy the CPU before non real-time threads are given runtime. This is controlled by the *sched_rt_runtime_us* parameter [19]. By default, it allows real-time threads to use up to 95 % of the CPU. While this mechanism is useful for maintaining overall system responsiveness, it is typically disabled in strict real-time setups where CPUs are isolated.

3
Methods

To evaluate the impact of PREEMPT_RT, a number of OS binaries were compiled both with and without the patch. Each test setup consisted of one OS binary running a number of standardized tests using test software that was developed specifically for this project.

This chapter further outlines the methodology, detailing the tools used, the development and application of the test software, and the build process of the kernel setups, each discussed in separate sections.

3.1 Hardware Platform and Software

The hardware platform used for this project was the AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [28]. It features a 64-bit quad-core ARM Cortex-A53 (Application Processing Unit or APU) and a dual-core ARM Cortex-R5F (Real-Time Processing Unit or RPU) [29]. Only the APU was used in this project, where the operating system was set up to run on it. The quad-core processor has a cache hierarchy consisting of level 1 (L1) caches with a 32 KB instruction cace and a 32 KB data cache per core, and a shared level 2 (L2) cache of 1 MB.

The hardware platform was flashed with various configurations of a stripped-down Linux distribution binary, which was then controlled using a TeraTerm UART terminal on a laboratory PC.

The test software was written in C using standard Linux and POSIX C libraries using a VIM editor. The software was then cross-compiled to run on the hardware platform ARM architecture using a GCC compiler. To perform tests, the test software binary was transferred to the hardware platform using the XMODEM protocol, and run locally.

The output test data was then returned to the laboratory PC using terminal logging. Once all the test data had been gathered, it was imported into the numeric computing environment MATLAB, analyzed and plotted.

3.2 Test Software and Methodology

The test software that was developed for this project consists of a single, compact executable binary. The binary was specifically cross compiled to transfer and run

on the operating system and hardware used in this project. When run, the binary performs a single or a series of real-time tests, the data of which can then be recovered from the test platform and analyzed.

A test consists of a main process taking a specific number of latency measurements at a specific period. Specifically, the test measures wake-up latency, which is defined in Section 4.3.1.

By starting a number of separate load processes in the background before the test is run, the test software can also test the impact of background load on real-time performance. The load processes work by continuously copying a certain amount of memory between two allocated partitions of memory of a specific size. When performing a so-called sweep, the test software performs a series of tests, each with an incremental number of load processes.

The test software is also able to change its scheduler to allow the main process to preempt other processes on the system, including the load processes. The test software can also change the CPU affinity of the main process in order to allow for CPU isolation.

Once recovered, the data consisted of a series of latency measurements in nanoseconds ready for data analysis. The results of this analysis are presented in Chapter 5 and discussed in Chapter 6.

3.2.1 Standard Test Series

To allow for comparisons between individual test setups, three standardized test series were established. All three standard test series consist of a sweep of nine tests each, varying between 0-8 load processes.

Each individual test used 60,000 measurements taken at a period of 1 ms for each test, resulting in a total of 540,000 measurements over a total of nine minutes. The load processes were configured at their standard copy size of 1 MB of memory.

The three standard test series used varied as follows:

- **Preemption.** This standard test series changed the scheduler of the main process to a real-time scheduler, allowing it to preempt all non-real-time tasks on the system. This test series was used to determine the effect of a test setup on real-time processes.
- **No preemption.** This standard test series did not change the scheduler of the main process to a Linux real-time scheduler, meaning that the main process made its measurements without preemption. This test series served as a baseline for comparing the effect of a test setup on non-real-time processes.
- **Preemption and CPU Isolation.** This standard test series used preemption, but also CPU affinity in order to confine the main process on an isolated CPU. This setup required additional special preparations, see the CPU Isolation section.

3.2.2 CPU Isolation

In an attempt to further optimize real-time performance, a test series was conducted in which the main process was scheduled on an isolated CPU core. The aim was to minimize scheduling and interrupt-related disturbances by confining "all" other system activity, including background processes, interrupts, and RCU callbacks, to the remaining three cores of the quad-core processor.

These settings were implemented via kernel boot parameters passed through the U-Boot bootloader, which was used to initialize and boot the operating system's startup behavior. Specifically, the custom parameters were appended to the *bootargs* environment variable within the U-Boot *bootcmd* macro. The following kernel arguments were applied:

- *isolcpus=managed_irq,domain,3 rcu_nocbs=3 rcu_nocb_poll irqaffinity=0-2*

These arguments instructed the Linux kernel to exclude the isolated core (specifically the fourth core with index 3) from normal task scheduling, offload RCU callbacks from the isolated core, and redirect interrupt handling to the non-isolated cores. Additionally, at runtime, CPU affinity was explicitly set for the main process to ensure the real-time process ran solely on the isolated core.

In one of the kernel configurations (see further details in Section 4.1), full dynamic tickless mode was enabled in addition to CPU isolation. This mode aims to suppress periodic timer interrupts on the isolated core. The following additional boot parameters were applied:

- *nohz=on nohz_full=3*

With these settings, core 3 operated without regular timer ticks, except for an unavoidable 1 Hz tick. As noted in Section 2.3.1, full tickless mode is particularly advantageous for systems running a single real-time task on an isolated core, as it minimizes interruptions from the system clock.

3.3 Linux Kernel Setups and Build Process

To evaluate the impact of different Linux kernel configurations on real-time performance, several kernel binaries were compiled and built for the target hardware. All setups were based on a scaled-down version of the Linux 6.1.40 kernel, with many non-essential drivers and features removed. Most kernel setups were further patched with the PREEMPT_RT patch (version 6.1.38-rt13-rc1), which makes changes to the kernel's source files to enable preemption capabilities beyond what is available in the standard mainline kernel. Due to the lack of a patch that matched the exact kernel version used, the closest available patch version was applied to maintain compatibility. The patch was applied to the Linux kernel using the *patch* program within the kernel source tree as follows:

- *patch -p1 < ../patch-6.1.38-rt13-rc1.patch*

The desired configurations was set using *make menuconfig*, a user-friendly tool for configuring the kernel prior to compilation. After configuration, the kernel was

built using the GCC x86 to ARM cross compiler. The resulting binary image, along with a corresponding First Stage Boot Loader (FSBL), was then, through Vivado's Hardware Manager, flashed to the non-volatile memory on the hardware platform.

The kernel setups were designed primarily to explore how different levels of kernel preemption influence system responsiveness under various operating conditions. Three main configurations were created, the first used a non-preemptive setup, the second enabled standard low-latency preemption, and the third used full preemption through the PREEMPT_RT patch. These configurations allow for comparison between the baseline Linux kernel and its real-time counterpart with more advanced preemption capabilities introduced by the patch. This makes it possible to determine the extent to which preemption cababilities within the kernel improve (or potentially degrade) the worst-case latency and jitter for a latency-sensitive process under different operating conditions provided by the software test.

Each setup was derived from a common default configuration. A selected group of kernel configuration options, identified in the literature as critical for low jitter and latency performance, were explicitly enabled or disabled to optimize for a deterministic behavior. These configurations were kept consistent between all setups. While the initial plan was to incrementally test each configuration change in isolation, starting from the default setup and modifying one variable at a time, this approach proved infeasible due to time constraints. Each kernel build and flash process was time-intensive, so the focus was narrowed down to evaluating setups that already included the most critical configuration parameters identified in the literature as important for real-time optimization. Beyond these configurations, the remaining configuration settings were left at their defaults and consistent between all setups to isolate the effect of preemption-related changes in the kernel.

In addition to these baseline configuration setups focused solely on comparing preemption modes, additional kernel setups were created to investigate the impact of system clock parameters. These builds continued to use the PREEMPT_RT patch with full preemption but varied in terms of timer interrupt frequency and timer modes, including periodic ticking, idle tick suppression, and full tickless operation. This comparative analysis of kernel setups with subtle configuration differences was conducted to identifying the most effective setup. Importantly, performance was evaluated across different software test scenarios, as a configuration setup optimized for one context may be less effective in another.

4

Design

The design and implementation of both the Linux kernel and the application software are detailed in this chapter. Optimizing an operating system is complex as it involves several layers of tuning. One layer concerns the hardware platform, which in this project is fixed and limited to a single platform. More relevant to our focus, however, are the additional layers of tuning: the OS kernel, where configuration options and kernel arguments are used to modify system behavior; and the software applications, where aspects such as task priority further influence performance.

In Section 4.1, the different kernel builds and their respective configuration settings are presented. Section 4.2 introduces our definition of a sensor control application, which our test software was based on. Lastly, Section 4.3 provides a more detailed explanation of the test software constructed, including the functions used in the POSIX C libraries and the runtime flags implemented.

4.1 Kernel Configuration Setups

To evaluate how different kernel configurations affect performance, we designed and tested several Linux kernel setups with slightly varying configurations. The primary goal was to assess how the available preemption modes in the kernel influence latency and jitter for real-time tasks. The preemption modes evaluated in this project included no forced preemption (`CONFIG_PREEMPT_NONE`), the preemptible kernel (`CONFIG_PREEMPT`) available in the standard Linux kernel, and the fully preemptible kernel (`CONFIG_PREEMPT_RT`), which becomes available by applying the PREEMPT_RT patch.

In addition to preemption, the effect of timer interrupt configuration on system performance was also examined. The kernel timer supports three operational modes: periodic mode (`CONFIG_HZ_PERIODIC`), tickless idle mode (`CONFIG_NO_HZ_IDLE`), and full tickless mode (`CONFIG_NO_HZ_FULL`). Two timer frequencies were considered, specifically 250 Hz and 1000 Hz. Higher timer interrupt rates result in faster responsiveness but also increase system overhead [21], creating a trade-off that must be considered when adjusting the timer frequency. Therefore, this makes it an interesting parameter to investigate.

Table 4.1 summarizes the key differences in configuration across all built kernel setups. Setup 1 and 2 contain configurations available in the standard Linux kernel without the real-time patch, while setup 3 through 6 include the PREEMPT_RT

patch to enable full kernel preemption. Setup 1, 2 and 3 were designed to isolate the effect of preemption modes, using identical timer configurations consisting of periodic mode with 250 Hz frequency. This timer setup was selected to ensure consistent interrupt intervals and because 250 Hz is the default in many major Linux distributions, including the one used for these experiments. Setup 4, 5, and 6 expanded the investigation to explore the impact of timer mode and frequeny, while keeping the fully preemptible kernel option enabled. This approach was chosen to be able to identify the optimal kernel configuration best suited for the real-time tasks in our test environment.

Table 4.1: Linux Kernel Setups and Their Distinct Configuration Parameters

Setup	Preemption Mode	Timer Mode	Frequency
1	CONFIG_PREEMPT_NONE	CONFIG_HZ_PERIODIC	CONFIG_HZ_250
2	CONFIG_PREEMPT	CONFIG_HZ_PERIODIC	CONFIG_HZ_250
3	CONFIG_PREEMPT_RT	CONFIG_HZ_PERIODIC	CONFIG_HZ_250
4	CONFIG_PREEMPT_RT	CONFIG_NO_HZ_IDLE	CONFIG_HZ_250
5	CONFIG_PREEMPT_RT	CONFIG_HZ_PERIODIC	CONFIG_HZ_1000
6	CONFIG_PREEMPT_RT	CONFIG_NO_HZ_FULL	CONFIG_HZ_250

In summary, the setups in Table 4.1 allow us to evaluate the effect of preemption mode by comparing setups 1, 2, and 3; the impact of timer mode by comparing setups 3, 4, and 6; and the influence of timer frequency by comparing setups 3 and 5.

All setups listed in Table 4.1 were built with power management features disabled to eliminate response time variability caused by power saving mechanisms. Several kernel configuration settings were kept consistent across all setups. The performance governor (`CONFIG_CPU_DEFAULT_GOV_PERFORMANCE`) was enabled, as it maintains maximum CPU frequency. Similarly, the RCU callback offloading feature (`CONFIG_RCU_NOCB_CPU`) was enabled in all builds, althoguh it has no runtime effect unless the relevant kernel arguments are provided at boot. Other important options, such as high-resolution timers (`CONFIG_HIGH_RES_TIMERS`), CPU isolation (`CONFIG_CPU_ISOLATION`), and the disabling of simultaneous multithreading (`CONFIG_SCHED_SMT`), were also consistently maintained across all configuration setups. Moreover, no significant debugging options were enabled.

4.2 Sensor Control Systems

This project is concerned with investigating Linux as a platform enabler for sensor control systems. Since no formal or generally agreed upon definition or model for a "sensor control system" exists in literature, one must be established for the purposes of this project.

We define a sensor control system as an embedded real-time system designed to control one or more sensors while handling the inputs. More specifically, we define a sensor control system to have two main responsibilities:

- Periodically initiate sensor readings.
- Read, pre-process and pass on sensor data.

With this definition in hand, we can construct a simple model for the software running on a sensor control system. We define this model as a periodically activated real-time task responsible for the initiation of sensor readings. The real-time performance of the task is impacted by some amount of load caused by inputs, outputs and data processing from processes running in the background.

In this simple model, if the real-time performance of the system must be good enough to satisfy the time constraints of the real-time task, then the system is feasible from a real-time perspective.

4.3 Test Software

The test software used in this project was developed to evaluate the real-time performance of a system under varying amounts of background load using a real-time performance metric. We define this metric as wake-up latency, see Section 4.3.1.

To measure latency under a certain level of background load, the test software is divided into a main process which sets up the test and measures latency, and a number of load processes which apply load to the system in the background. The main process begins by creating a specific number of load processes using `fork()`, before executing a periodic latency measurement routine.

4.3.1 Defining and Measuring Wake-up Latency

To evaluate our setups, we must first define a performance metric, and a test able to measure it. In this project, we based this test on the *cyclictest* latency benchmarking software for Linux, and define the latency measured as the wake-up latency of the measuring process.

A real-time process calls the `clock_nanosleep()` standard C library function and asks to sleep until time t_t, called the target time. When the sleep is complete, and the process wakes back up, it immediately uses `clock_gettime()` standard C library function to measure the current time t_w, called the wake-up time. In this case:

$$t_w - t_t = \Delta t, \tag{4.1}$$

where Δt is the measured wake-up latency. As a performance metric, the wake-up latency of different test setups can be compared in order to establish their relative performances. Since each measurement is affected by the state of the system at the time of measurement, a large set of wake-up latency measurements should be used when analyzing a setup.

The main process then repeats this measurement periodically for a set number of iterations, before returning from the measurement routine and terminating the background loads. Each wake-up latency measurement is then logged into a data file

and formatted to be compatible with typical data analysis software. A pseudocode breakdown of test software execution with standard library C function calls can be found in Algorithm 1.

Algorithm 1 Test Software Main Process

```
1: function TEST(T, period, iterations, loads, copysize, scheduler, affinity)
2:     for i ← loads do
3:         P[i] ← fork();
4:         if child then
5:             LOADPARENT(copysize)
6:         end if
7:     end for
8:     (set scheduler to scheduler)
9:     (set CPU affinity to affinity)
10:    wakeup ← clock_gettime();
11:    for i ← iterations do
12:        target ← wakeup + period
13:        clock_nanosleep(target);
14:        wakeup ← clock_gettime();
15:        T[i] ← wakeup − target
16:    end for
17:    (reset scheduler)
18:    (reset CPU affinity)
19:    for i ← loads do
20:        kill(P[i]);
21:    end for
22: end function
```

By default, the periodic wake-up latency measurement routine is performed using the default Linux scheduler, with the system default CPU affinity. In order to use preemption, the main process can switch to the Linux real-time `SCHED_FIFO` scheduler, allowing the main thread to preempt all non-real-time applications on the system. In order to use CPU isolation, the main process can also change its CPU affinity to run on an isolated CPU for the duration of the test.

In order to streamline testing, the test software is capable of performing a sweep of tests in series using varying numbers of load processes.

The test software is compiled into a single executable binary, and can be controlled using flags in the Linux terminal. Flags give the user the option to control most aspects of test execution, such as the number of measurements, measurement period, preemption, CPU affinity, number of loads, etc.

4.3.2 Load Process

Since the background load of a typical system consists of many tasks performing a large variety of operations, great care must be taken when defining what a unit of load on a system actually means. In this project, a decision was made to define load

in terms of a number of identical load processes. In that decision, a load process was defined as a Linux process that continuously consume the resources of a single CPU-core.

Initially, a load process was developed that consisted of a single continuously running process. Since the load process is not a real-time application, it uses the default Linux scheduler CFS. This had the unintended effect of throttling the load processes, since the CFS scheduler is designed to de-prioritize singular, resource hungry scheduler entities by tracking CPU-time. In practice, this resulted in the effect of load processes diminishing over time, rather than applying a time-consistent load on the system.

To work around this without modifying or changing the scheduler, an alternative parent-child load process pair was developed instead. To apply a time-consistent load, the parent process continuously creates a single child process using `fork()` and waits for it to finish executing using `wait()`. The child process then performs a series of resource demanding operations, before finishing execution. Since each new child is created as a blank slate, the operations of the previous child have no impact on its scheduling. See Algorithm 2 for a pseudocode breakdown of the parent-child load process pair with standard library C function calls.

Algorithm 2 Test Software Load Process

```
1: function LoadParent(copysize)
2:     loop
3:         child ← fork();
4:         if child then
5:             LoadChild(copysize)
6:             exit();
7:         end if
8:         wait(child);
9:     end loop
10: end function
11: function LoadChild(copysize)
12:     A ← malloc(copysize);
13:     B ← malloc(copysize);
14:     memcpy(A,B)
15: end function
```

In this implementation of a load child process, the decision was made to perform memory allocation (`malloc()`), memory deallocation (`free()`), and a memory copy operation (`memcpy()`). This, as well as the creation and destruction of child processes by the parent process serves as the actual load imposed on the system.

Once created, the parent process runs continuously until it receives a signal to stop execution, either from the user or the main process of the test software.

5

Results

This chapter presents the results of the latency measurements collected through the test software. The specific latency measured is the wake-up latency of a process, which is defined in Section 4.3.1. Three standardized test series were applied across the various kernel setups, with each series sweeping the number of background load processes and recording latency, as defined in Section 3.2.1.

The result from each test series is presented in its own section. Table 5.1 below summarizes the test series and kernel configurations used in each case. Section 5.1 presents the results from the baseline test series, where the main process is scheduled using the regular scheduler without real-time priority. Section 5.2 presents the results from using the real-time scheduler, allowing the main process to be prioritized and preempt lower-priority tasks. Finally, Section 5.3 presents the results from applying CPU affinity and letting the main process run on an isolated core.

Table 5.1: Overview of Test Series and Kernel Setups

Test Series	**Target Kernel Setups**	**Kernel Arguments at Boot**
No preemption, Regular scheduling	Kernels with different preemption modes	– (default only)
Preemption, Real-time scheduling	All kernel configurations	– (default only)
Preemption and CPU Isolation, Real-time scheduling with CPU affinity	All kernel configurations, with special handling for full tickless-capable kernel	**Isolation:** *isolcpus*, *rcu_nocbs*, *rcu_nocb_poll*, *irqaffinity* **Full Tickless:** *nohz*, *nohz_full*

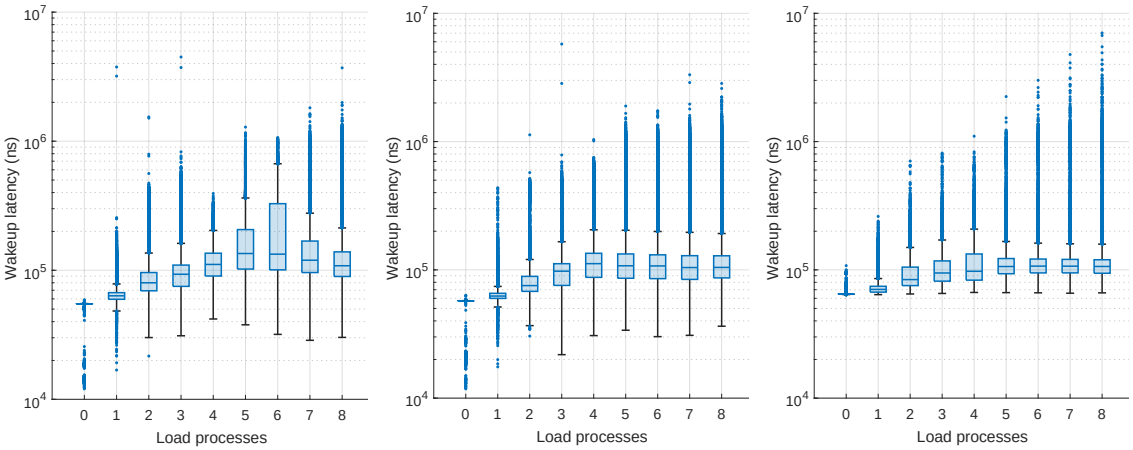
The results are presented in a number of figures below, in the form of box plots with the vertical axis representing latency and the horizontal axis the number of background load processes used.

The box plots in the figures show distributions of measured wake-up latency. Each box is bounded by the test's first and third quartile, with the median marked as a line in the middle. The whiskers of the box represent the upper and lower bounds of the measurements excluding outliers, which are presented as dots above and below the boxes. Outliers are in this case defined as any value further away from the box than $1.5 \cdot IQR$, with IQR being the inter-quartile range.

In the figures, the results of different tests are compared side-by-side using identical base 10 logarithmic scales for wake-up latency.

5.1 Wake-Up Latency without Preemption

Figure 5.1 shows results for kernel setups 1, 2, and 3 from Table 4.1. These represent increasing levels of preemption support. Each setup is shown in a separate subfigure to facilitate comparison under the default, non-real-time scheduling conditions, which is to say that the main process is scheduled with the same default Linux CFS scheduler as the load processes.



a) Non-preemptible kernel. b) Preemptible kernel. c) Fully preemptible kernel.

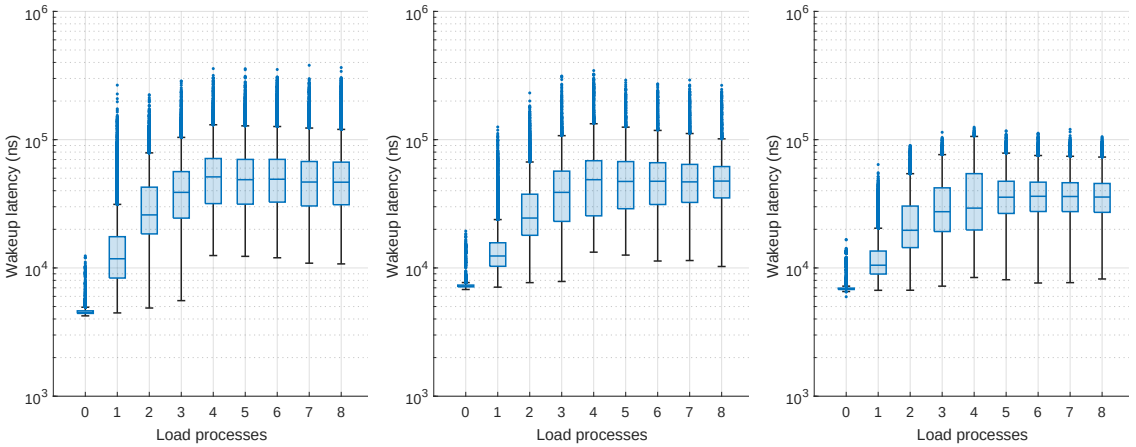
Figure 5.1: Wake-up latency under varying number of load processors (from 0 to 8) using the no preemption test series, where the main process runs under the regular scheduler. Subfigures a, b, and c corresponds to three different kernel setups.

The result for the normal non-preemptible and preemptible kernel displayed in Figures 5.1a and 5.1b show a clear diminishing of quality as the load increases. With the PREEMPT_RT patch applied and a fully preemptible kernel setup, Figure 5.1c shows a similar response to an increasing load, but with a very consistent lower bound much higher than what is seen in either Figures 5.1a or 5.1b. Since the test measures the wake-up latency of a non-preempting process, the increased lower bound can be interpreted as the result of increased overhead in the scheduling of non-preempting processes. This result indicates that non-real-time processes suffer from reduced responsiveness when using a fully preemptible kernel setup.

5.2 Wake-Up Latency with Preemption

Figure 5.2 revisits kernel setups 1, 2, and 3, with the real-time scheduler enabled. That is to say that the main process is using the Linux FIFO real-time scheduler, allowing it to be scheduled ahead of and preempt all other non-real-time processes.

Compared to the results seen in Figure 5.1, we can clearly see that all three setups see benefits to wake-up latency when using a real-time scheduler in Figure 5.2, whether the setup supports preemption or not. However, comparing Figure 5.2a-b to Figure 5.2c also shows that the PREEMPT_RT setup has by far the best real-time performance, with a lower average wake-up latency and a lower worst-case in



a) Non-preemptible kernel. b) Preemptible kernel. c) Fully preemptible kernel.

Figure 5.2: Wake-up latency under varying number of load processors (from 0 to 8) using the preemption test series, where the main process runs under the real-time scheduler. Subfigures a, b, and c corresponds to three different kernel setups.

all tests with one or more load processes.

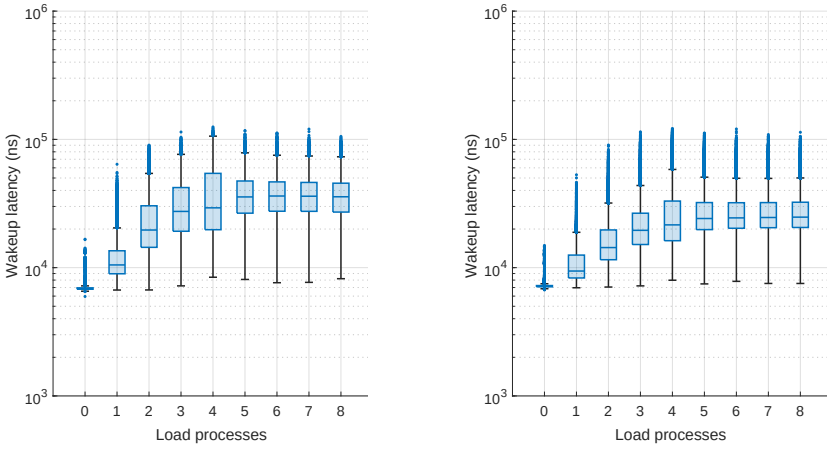
Notably, the impact of additional load processes in Figure 5.2 start diminishing after four processes and above. This is most likely because the tests are run on a CPU with four cores. Since the main process is allowed to preempt the load processes, it is presumably the processes actually running on the CPU at wake-up time that have an impact on the measurements.

Interestingly, as the number of load processes increases, the standard preemptible kernel in Figure 5.2b does not consistently outperform the non-preemptible one in Figure 5.2a. Its performance varies depending on the load. While it performs slightly better in some cases, it performs worse in others, such as with three or four background load processes. This inconsistency is somewhat unexpected. One possible explanation is that the system was not sufficiently stressed to expose clear performance gaps between the two kernels. Regardless, our result indicate that the standard preemptible kernel does not guarantee improved real-time performance over the non-preemptible kernel.

In contrast to this, the fully preemptible kernel in Figure 5.2c enabled by the PREEMPT_RT patch consistently achieves better worst-case latency in all tested scenarios involving background load. Its performance is also more predictable in comparison to the other kernels, as the jitter is significantly reduced, and best-case latency remains relatively stable across different load levels. This consistency highlights the effectiveness of full preemption and indicates a more deterministic system behavior. These test series show the clear benefits of using the PREEMPT_RT patch for real-time processes on a busy system.

Furthermore, Figure 5.3 visualizes the effect of the timer frequency by comparing setups 3 and 5 from Table 4.1. These kernels are configured with full preemption using PREEMPT_RT and are identical except for the timer interrupt rate, with setup 3 using 250 Hz while setup 5 uses 1000 Hz. Here, we can see the benefits

of increasing system clock frequency, with a much lower average wake-up latency. Interestingly, the change of clock frequency had little to no impact on the system's jitter.



a) Frequency 250 Hz. b) Frequency 1000 Hz.

Figure 5.3: Wake-up latency under varying number of load processors (from 0 to 8) using the preemption test series, where the main process runs under the real-time scheduler. Subfigures a and b corresponds to two different kernel setups.

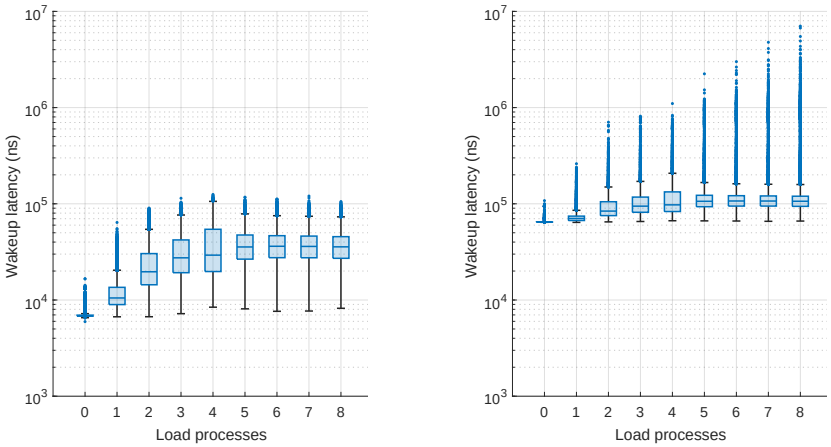
As the high-resolution timers configuration option, which is consistently enabled across all setups, ensures better accuracy and responsiveness in waking up tasks, the system clock, and thus timer frequency, should have little impact on wake-up latency. This is because wake-ups of the main process are driven by hardware timers and not by the periodic ticks of the system clock.

However, the slightly lower average latency observed in Figure 5.3b, with the higher frequency configuration, may be explained by other kernel tasks and system housekeeping tasks still depend on the system clock. A higher timer interrupt frequency value allows for these background tasks to be scheduled more often and processed in smaller portions, which can indirectly reduce average latency by minimizing scheduling delays.

Additionally, the increase in timer frequency also results in more frequent timer interrups (four times as many in the case of 1000 Hz compared to 250 Hz). This can introduce additional interrupt noise, which slightly increases the chance of interrupting or delaying the immediate scheduling of the high-priority task if its wake-up coincide with a timer interrupt. This could explain the increased number of outliers observed in Figure 5.3b compared to Figure 5.3a.

A final comparison is shown in Figure 5.4, which contrasts kernels of different timer modes. Here, setups 3 and 4 from Table 4.1 are compared. Both uses a 250 Hz tick rate with full preemption enabled, but setup 3 runs with periodic tick interrupts while setup 4 runs with tickless idle and specifically stops tick interrupts during idle periods.

In the periodic timer kernel shown in Figure 5.4a, latency remains low and consistent. Both average and worst-case latencies are well bounded and show little variation



a) Periodic timer mode.

b) Tickless idle mode.

Figure 5.4: Wake-up latency under varying number of load processors (from 0 to 8) using the preemption test series, where the main process runs under the real-time scheduler. Subfigures a and b corresponds to two different kernel setups.

once the background load exceeds four processes. This is likely due to the regular timer interrupts, which keep the scheduler and system active, improving overall responsiveness.

In contrast, the tickless idle kernel in Figure 5.4 shows a different behavior. Although the average and best-case latency remains relatively stable as load increases, they are consistently higher than in the periodic kernel. This suggests that operating in tickless idle mode introduces a fixed overhead in wake-up latency. More notably, the worst-case latency increases exponentially with the number of background processes, making the system more prone to severe outliers and unpredictable delays.

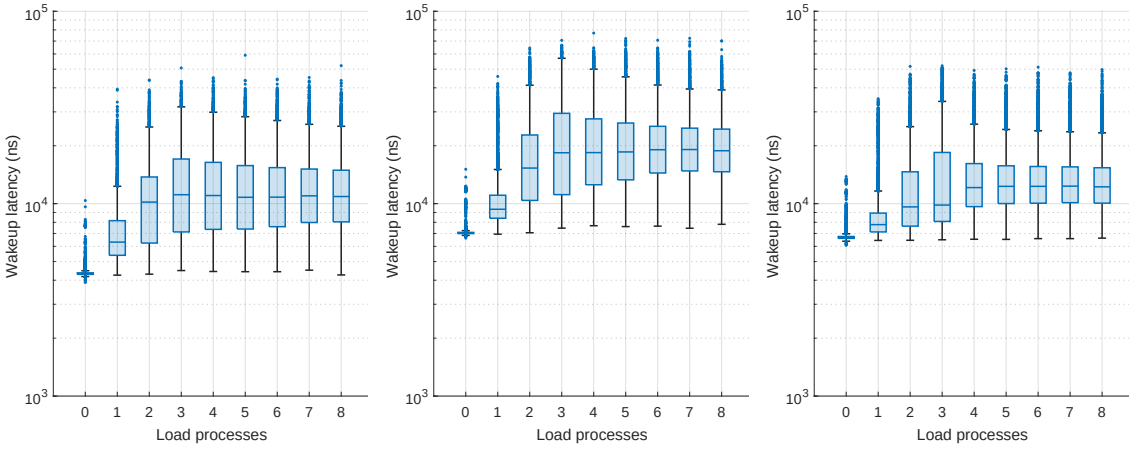
During idle periods, tickless idle mode disables the periodic timer ticks and makes CPU cores enter deep sleep states. While this improves power efficiency, it increases latency when waking the CPU. As load increases, one might expect tickless idle mode to behave more like periodic mode, since the CPU is often busy and not in deep sleep. However, the observed exponential increase in worst-case latency suggests otherwise.

These measurements may also be influenced by the test setup. Before each new test when the load is increased, all background processes are stopped and restarted simultaneously. If all CPU cores are in deep sleep prior to this, wake-up delays may stack up. When many cores wake up at once and processes or other kernel tasks begin competing for CPU time, the scheduler becomes heavily loaded. Once the system is fully awake, the extreme delays disappear and latency stabilizes as indicated by the average latency levels. However, identifying the exact causes of these worst-case latencies would require further tracing and debugging, which is outside the scope of this project.

5.3 Wake-Up Latency with CPU Isolation

This section presents the results from the test series where preemption is enabled and the main process uses CPU isolation. This means that in addition to using the Linux FIFO real-time scheduler, the main process has an entire CPU core to itself where no other processes are allowed to execute.

Figure 5.5 show the results of kernel setups 1, 2, and 3 from Table 4.1, which represent the three different levels of kernel preemption support.



a) No forced preemption. b) Preemptible kernel. c) Fully preemptible kernel.

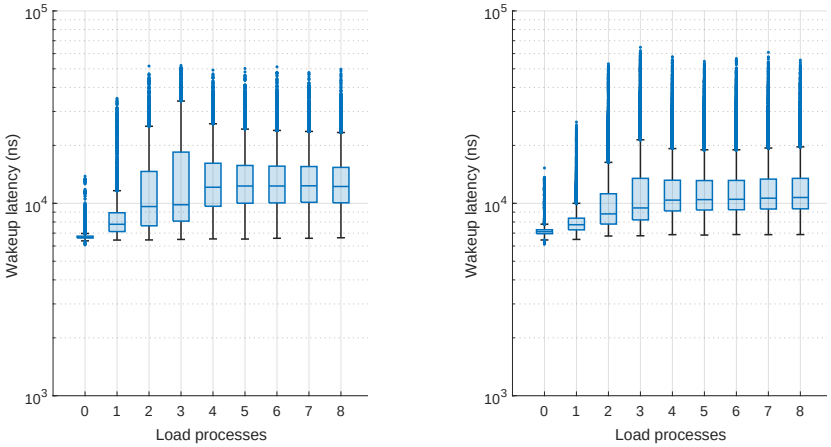
Figure 5.5: Wake-up latency under varying number of load processors (from 0 to 8) using the preemption with CPU isolation test series, where the main process runs under the real-time scheduler. Subfigures a, b, and c corresponds to three different kernel setups.

All setups seen in Figure 5.5 show consistently better real-time performance than the corresponding non-isolating setups seen in Figure 5.1 and 5.2. These benefits is offset by the relative cost of using core isolation for real-time processes, since an entire CPU-core must be set aside for the process. Interestingly, the non-preempting kernel setup seen in Figure 5.5a shows the best real-time performance, most likely due to the lower overhead in a non-preempting kernel setup.

Since the process is isolated on a core, scheduling and preemption can be assumed to have little actual impact on the main process wake-up latency. Instead, the impact seen from an increased number of load processes in all three subfigures in Figure 5.5 can most likely be attributed to the shared L2 cache of the hardware platform.

Further tests were applied to different kernel setups varying in timer frequency and mode. Figure 5.6 compares setups 3 and 5 under the same isolated-core condition, again with the only difference being the timer frequency (250 Hz versus 1000 Hz). As observed previously in Section 5.2, we again see a wider spread of outliers with the higher-frequency kernel. However, in this case, there is also a noticeable difference in worst-case latency. The 1000 Hz kernel shows slightly higher worst-case latency values. While the exact cause is unclear, the increased likelihood of timer interrupts coinciding with the awakening of the main process could explain the rise in outliers.

In addition, consistent with earlier findings, the higher frequency still results in a lower average wake-up latency.



a) Frequency 250 Hz. b) Frequency 1000 Hz.

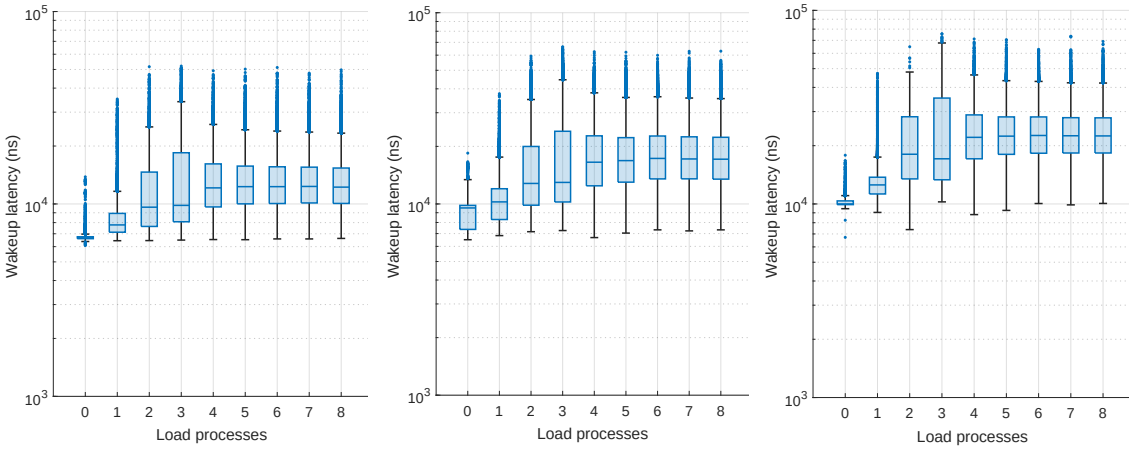
Figure 5.6: Wake-up latency under varying number of load processors (from 0 to 8) using the preemption with CPU isolation test series, where the main process runs under the real-time scheduler. Subfigures a and b corresponds to two different kernel setups.

Overall, timer frequency does not have a major impact on performance. However, it is interesting to note that under isolation, with only the main process pinned to the isolated core, there seems to be a trade-off. The 1000 Hz kernel yields lower average latency but slightly higher worst-case latency, while the 250 Hz kernel has more bounded worst-case latency but slightly higher average latency. The differences are however relatively small.

Finally, Figure 5.7 compares the effect of all three available timer modes under full preemption. It contrasts kernel setups 3, 4, and 6, which uses periodic, tickless idle, and full tickless modes, respectively. This comparison is most meaningful under CPU isolation, as full tickless mode requires isolated conditions to take effect.

The comparison between periodic mode (Figure 5.7a) and tickless idle mode (Figure 5.7b) aligns well with expectations. Unlike the result in Section 5.2, the tickless idle kernel here shows no exponential growth in worst-case latency over increased load processes. This is likely because the isolated core is excluded from the general scheduler, and thus no background load processes are scheduled on it. Still, tickless idle mode has both higher worst-case and average latency, likely due to delays introduced when waking the core from deep sleep.

Figure 5.7c shows results from running the core in full tickless mode, where timer interrupts are completely disabled on the isolated core. Surprisingly, this setup does not improve latency. On the contrary, both average and worst-case latencies are higher than for the kernel in tickless idle mode. This does not align with theory, which suggests that full tickless mode should reduce overhead by eliminating timer interrupts when only a single task is active on the core. The increased latency may indicate that the necessary conditions for running in full tickless mode were not fully met. Further tracing and analysis would be required to confirm this.



a) Periodic timer mode. b) Tickless idle mode. c) Full tickless mode.

Figure 5.7: Wake-up latency under varying number of load processors (from 0 to 8) using the preemption with CPU isolation test series, where the main process runs under the real-time scheduler. Subfigures a, b, and c corresponds to three different kernel setups.

6

Discussion

This chapter reflects on the results and discusses how they relate to the goals of the project, including the system's suitability for sensor control systems. Finally, potential directions for future work and ethical aspects are discussed.

6.1 Impact on Real-Time Perfomance

A multitude of figures are presented throughout Chapter 5 that can be compared to show the impact of the PREEMPT_RT patch on the real-time performance of Linux. The results generally align with the expected outcome of the tests, with the PREEMPT_RT providing enhanced real-time performance for as long as preemption is enabled by using the real-time FIFO scheduler. However, several other insights can also be gleaned by observing the data.

The results of individual tests in Chapter 5 are presented as box plots. The median value of the box plots can be interpreted as the working wake-up latency of a setup, an important metric for soft real-time systems. Meanwhile, upper outliers of the box plots can be used to roughly estimate of the worst-case wake-up latency of a setup, most important for hard real-time systems. The lower bounds of a box plots are also interesting, as they can be used to estimate the best-case wake-up latency for a system, which is useful for comparing overhead.

To clarify the numerical results, Appendix Tables A.1, A.2, and A.3 present absolute measurements of average latency, maximum latency, and jitter for the evaluated setups during two selected load conditions, namely with four and eight load processes. Each table shows the results from performing the three standard test series: without preemption (using the CFS scheduler), with preemption (using the real-time scheduler), and with CPU isolation, respectively.

6.2 Usability for Sensor Control Systems

Using the model of a sensor control system defined in Section 4.2 we can compare and contrast the results above from a sensor control system context. If we interpret the wake-up latency measurements discussed in Section 4.3.1 as analogous to the performance of the real-time process defined in Section 4.2, then the results presented in Chapter 5 should roughly correspond to the real-time performance of sensor control systems using identical setups.

It should be noted that there are many limitations to the model used to represent a sensor control systems in this project, mostly due to its simplicity. Since the real-time process is idealized as a process which only measures its own wake-up latency, the real-time process itself causes next to no load on the system, nor does the model take into account the potential presence of several competing real-time processes on the same sensor control system. Since there are no actual external sensors involved in the main process measurement, things such as I/O latency, which might be a more important real-time performance metric for actual sensor control systems, cannot be measured either. The model also does not take into account potential time constraints on the output of data.

It should be noted however that these idealizations were made in order to keep the model fairly general, which makes the results applicable to a wider variety of implementations of sensor control systems.

6.3 Future Work

The field of Linux as a real-time platform has seen a lot of progress recently, and there is much research left to be done, especially as improvements like PREEMPT_RT are being merged into the mainline kernel.

Recently, in the 6.6 version of the Linux kernel, a new default scheduler known as Earliest Eligible Virtual Deadline First (EEVDF) was deployed to replace the old default CFS scheduler [30]. The EEVDF scheduler aims to increase the responsiveness of the Linux platform by, among other things, allowing time sensitive tasks to be prioritized. It is unclear how exactly this new scheduler would affect the tests performed in this project, but it is likely that the results would differ.

Using a more complex setup that takes external sensors and I/O load and I/O latency into account could yield more relevant data. This would enable the measurement of externally triggered events rather than relying solely on internally generated ones, thus providing more accurate and representative data for sensor control systems.

Additionally, evaluating the impact of the PREEMPT_RT patch across a broader range of hardware platforms could be valuable. Real-time performance is highly dependent on underlying hardware characteristics such as CPU architecture, cache sizes, and other system-level factors. Testing a variety of systems, e.g., with multi-processors or a higher number of CPU cores could reveal hardware-dependent factors that influence the effectiveness of the PREEMPT_RT patch.

6.4 Ethical Aspects

While this project focuses specifically on sensor control systems, there is always the possibility that the data produced can be used in a more general sense to evaluate the possibility of using Linux for real-time applications. This is true even in cases where those uses may cause ethical or societal harm.

Since this project is done in cooperation with Saab AB, a company involved in the defense industry, it is important to note the potential military uses of this technology.

Sensor control applications are commonly used in surveillance technology such as radar systems. Such systems often serve primarily as a defensive measure in order to protect equipment, personnel or civilians from harm. While there is always the possibility of such technology being implemented into offensive military systems, even the use of such systems can often be justified in the name of defence.

While the main accountability as to how a system is used ultimately rests with the systems end user, some responsibility also rests with the developer and producer of such systems for providing the opportunity. Thus, it is important for producers of such technology to take measures in order to control who gains access to it.

7

Conclusion

In this project, we investigated the real-time capabilities of Linux using the PREEMPT_RT patch, which enhances preemptibility within the kernel. Our investigation focused on building multiple kernel setups, all derived from a scaled-down version of Linux 6.1.40, to evaluate the effects of different preemption settings and timer configurations. Three primary setups were compared: two using standard preemption levels available in unpatched Linux, and one using the highest preemption level enabled by the PREEMPT_RT patch.

To evaluate these setups, test software was developed to simulate realistic sensor control applications. The software measured wake-up latencies of a periodically awakened main process under increasing system load. Three test series were conducted: one using a real-time FIFO scheduler, one using the standard CFS scheduler, and one combining the FIFO scheduler with CPU isolation. Together, these tests enabled a comprehensive comparison of how the PREEMPT_RT patch, compared to unpatched Linux, affects real-time performance; how CPU isolation can further reduce latency and jitter; and how non-real-time (CFS scheduled) tasks are influenced by the same factors.

Given the results of this project, it can be concluded that the fully preemptible kernel configuration introduced by the PREEMPT_RT patch significantly improves the real-time performance of real-time scheduled tasks on Linux. Latency, both average (median) and maximum, as well as jitter, were consistently lower than in the unpatched kernel setups.

Interestingly, an exception to this trend was observed when CPU isolation was used. In that scenario, the unpatched kernel with no preemption support achieved slightly better latency and jitter results than the fully preemptible kernel. Likewise, for non-real-time tasks, the introduction of the PREEMPT_RT patch led to slightly increased latencies, likely due to an additional overhead introduced by the patch.

Regarding timer configurations, the choice of timer mode showed limited overall impact on real-time performance. However, the periodic timer mode provided the most consistent improvement among the tested options. In addition, some real-time performance gains are possible by turning up the frequency of the system clock, but with unchanged jitter.

While feasibility must ultimately be established on a per-implementation basis, the findings from this project demonstrate that Linux-based sensor control systems can, in many scenarios, benefit from the PREEMPT_RT patch.

Bibliography

[1] M. M. Madden, "Challenges using linux as a real-time operating system," NASA Center for Aerospace Information (CASI), Tech. Rep. NF1676L-30228, Jan. 2019.

[2] G. K. Adam, N. Petrellis, and L. T. Doulos, "Performance assessment of linux kernels with PREEMPT_RT on ARM-based embedded devices," in *Electronics*, vol. 10, Jun. 2021, p. 1331, doi: 10.3390/electronics10111331.

[3] N. Vun, H. F. Hor, and C. J. W., "Real-time enhancements for embedded linux," in *2008 14th IEEE Int. Conf. on Parallel and Distrib. Syst., Melbourne, VIC, Australia*, 2008, pp. 737–740, doi: 10.1109/ICPADS.2008.108.

[4] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: a survey on PREEMPT_RT," in *ACM Comp. Surv.*, vol. 52, no. 1, Jan. 2020, pp. 470–505, doi: 10.1145/3297714.

[5] K. Chalas, "Evaluation of real-time operating systems for FGC controls," *CERN*, Sep. 17, 2015.

[6] N. Litayem and S. B. Saoud, "Impact of the linux real-time enhancements on the system performances for multi-core intel architectures," in *Int. J. of Comput. Appl.*, vol. 17, no. 3, 2011, doi: 10.5120/2202-2796.

[7] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in linux, PREEMPT-RT, and LITMUS RT," in *Proc. of Int. Workshop on Operating Syst. Platforms for Embedded Real-Time Appl. (OSPERT'13)*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:14096981

[8] Y.-C. Tian, *Handbook of Real-Time Computing*, Y.-C. Tian and D. C. Levy, Eds. Springer Singapore, Jan. 2019.

[9] H. Kopetz, "The real-time environment," in *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. New York, NY, USA: Springer, 2011, ch. 1, pp. 1–28.

[10] P. A. Laplante, "Basic real-time concepts," in *Real-time systems design and analysis: an engineer's handbook*, 3rd ed. New York, NY, USA: IEEE Press, 1997, ch. 1, pp. 1–22.

[11] ——, "Hardware considerations," in *Real-time systems design and analysis: an engineer's handbook*, 3rd ed. New York, NY, USA: IEEE Press, 1997, ch. 2, pp. 23–72.

[12] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards achieving fairness in the Linux scheduler," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, p. 34–43, Jul. 2008. [Online]. Available: https://doi.org/10.1145/1400097.1400102

[13] P. Hambarde, R. Varma, and S. Jha, "The survey of real time operating system: RTOS," in *2014 Int. Conf. on Electron. Syst., Signal Process. and Comput. Techn.*, 2014, pp. 34–39, doi: 10.1109/ICESC.2014.15.

[14] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "Effects of clock resolution on the scheduling of interactive and soft real-time processes," in *Proc. of the 2003 ACM SIGMETRICS Int. Conf. on Meas. and Model. of Comput. Syst.*, ser. SIGMETRICS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 172–183. [Online]. Available: https://doi.org/10.1145/781027.781049

[15] S. Rostedt and D. V. Hart, "Internals of the RT patch," in *Proc. of the Linux Symp.*, vol. 2, Jun. 2007, pp. 161–172.

[16] G. Kroah-Hartman, *Linux kernel in a nutshell*. O'Reilly Media, Inc, 2006.

[17] The Linux Kernel Organization, "Kernel Parameters: The Linux Kernel Documentation," Accessed: Apr. 5, 2025. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html

[18] E. Barbieri, "What is real-time linux? part III," Canonical Ltd., Accessed: Apr. 5, 2025. [Online]. Available: https://ubuntu.com/blog/what-is-real-time-linux-part-iii

[19] ——, "Tuning a real-time kernel," Canonical Ltd., Accessed: Apr. 5, 2025. [Online]. Available: https://ubuntu.com/blog/real-time-kernel-tuning

[20] C. Simmonds, "Real-time programming," in *Mastering Embedded Linux Programming: Harness the Power of Linux to Create Versatile and Robust Embedded Solutions*, 1st ed. Packt Publishing, Limited, Birmingham, 2015, ch. 14, pp. 353–373.

[21] Enea, *Enea® Linux Real-Time Guide*. Enea Software AB, 2014.

[22] Kernel Documentation, "NO_HZ: Reducing Scheduling-Clock Ticks," Accessed: Apr. 11, 2025. [Online]. Available: https://docs.kernel.org/timers/no_hz.html

[23] J. Ognness, "A Checklist for Writing Linux Real-Time Applications," LINUTRONIX, Linux for Industry, Accessed: Mar. 27, 2025. [Online]. Available: https://ogness.net/ese2020/ese2020_johnogness_rtchecklist.pdf

[24] The Linux Foundation, "RCU Configuration for Real-Time Systems," Accessed: Apr. 11, 2025. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/rcu

[25] M. Bernhardt, "On pinning and isolating CPU cores," Accessed: Apr. 14, 2025. [Online]. Available: https://manuel.bernhardt.io/posts/2023-11-16-core-pinning/

[26] Real-time Ubuntu documentation, "How to tune IRQ affinity," Canonical Ltd., Accessed: Apr. 5, 2025. [Online]. Available: https://documentation.ubuntu.com/real-time/en/latest/how-to/tune-irq-affinity/

[27] The Linux Foundation, "Threaded interrupt handler," Accessed: Apr. 14, 2025. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/threadirq

[28] Xilinx, Feb. 2023, "ZCU102 Evaluation Board User Guide," AMD, Technical Information Portal. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd

[29] AMD, Mar. 2025, "Zynq^{TM} UltraScale+^{TM} MPSoC Data Sheet: Overview (DS8911)," AMD, Technical Information Portal. [Online]. Available: https://docs.amd.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview

[30] The kernel development community, "Eevdf scheduler," Accessed: May. 29, 2025. [Online]. Available: https://docs.kernel.org/scheduler/sched-eevdf.html

A

Appendix 1

All latency and jitter measurements presented in the tables are in microseconds.

Table A.1: Wake-Up Latency and Jitter without Preemption

Set-up	4 Background Loads			8 Background Loads		
	Avg. Latency	**Max. Latency**	**Jitter**	**Avg. Latency**	**Max. Latency**	**Jitter**
1	115.217	393.280	351.366	177.727	3,699.730	3,669.547
2	148.902	1,037.183	1,006.450	175.635	2,839.534	2,803.181
3	110.400	1,101.410	1,034.753	122.522	7,029.242	6,963.086

Table A.2: Wake-Up Latency and Jitter with Preemption

Set-up	4 Background Loads			8 Background Loads		
	Avg. Latency	**Max. Latency**	**Jitter**	**Avg. Latency**	**Max. Latency**	**Jitter**
1	54.155	358.016	345.525	51.261	365.007	354.256
2	49.509	345.634	332.363	49.311	265.196	254.935
3	37.418	125.042	116.621	37.439	105.480	97.279
4	37.494	131.393	122.782	37.531	108.721	100.300
5	27.950	121.322	113.341	28.142	113.581	106.051

Table A.3: Wake-Up Latency and Jitter with CPU-Isolation

Set-up	4 Background Loads			8 Background Loads		
	Avg. Latency	**Max. Latency**	**Jitter**	**Avg. Latency**	**Max. Latency**	**Jitter**
1	12.002	45.144	40.703	11.774	52.135	47.874
2	20.288	76.998	69.317	19.788	70.107	62.296
3	13.395	49.245	42.714	13.165	49.765	43.145
4	18.326	62.546	55.876	18.494	62.947	55.627
5	12.191	57.735	50.865	12.210	55.485	48.605
6	23.534	71.177	62.366	23.497	69.087	59.026