computers

MDPI

Article

Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers

George K. Adam

CSLab Computer Systems Laboratory, Department of Digital Systems, University of Thessaly, 41500 Larisa, Greece; gadam@uth.gr; Tel.: +30-2410684596

Abstract: This research performs real-time measurements of Linux kernels with real-time support provided by the PREEMPT_RT patch on embedded development devices such as BeagleBoard and Raspberry Pi. The experimental measurements of the Linux real-time performance on these devices are based on real-time software modules developed specifically for the purposes of this research. Taking in consideration the constraints of the specific hardware platforms under investigation, new measurements software was developed. The measurement algorithms are designed upon response and periodic task models. Measurements investigate latencies of real-time applications at user and kernel space. An outcome of this research is that the proposed performance measurements approach and evaluation methodology could be applied and deployed on other Linux-based boards and platforms. Furthermore, the results demonstrate that the PREEMPT_RT patch overall improves the Linux kernel real-time performance compared to the standard one. The reduced worst-case latencies on such devices running Linux with real-time support could make them potentially more suitable for real-time applications as long as a latency value of about 160 µs, as an upper bound, is an acceptable safety margin.

Keywords: Linux kernel; real-time; operating systems; latency; performance measurements

Citation: Adam, G.K. Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers. *Computers* **2021**, *10*, 64. https://doi.org/10.3390/computers10050064

Academic Editor: Paolo Bellavista

Received: 26 April 2021
Accepted: 11 May 2021
Published: 16 May 2021

1. Introduction

The Linux kernel by standard successfully handles lightweight or soft real-time requirements. Nevertheless, it does not provide full assurance for hard timing deadlines required in safety-critical applications in industrial automation and control (e.g., robotics control, aerospace and air traffic control, vehicles control). Linux is a general purpose operating system that provides important features, such as process management, although not all of them have strict timing constraints, e.g., the scheduler can cause unbounded latencies which makes Linux not deterministic enough and cannot guarantee to meet the task deadlines. However, safety-critical systems must be safe at all times. On the other hand, PREEMPT_RT, a real-time preemption patch provided by Ingo Molnar and Thomas Gleixner is a popular patch for the Linux kernel that transforms Linux into a hard real-time operating system with deterministic and predictable behavior. This patch allows nearly all of the kernel code to be preempted by higher priority kernel threads, and reduces the maximum thread switching latency, although that depends on the system—that is, on a combination of hardware and software. By way of example, not all microprocessors have included a memory management unit (MMU), or it is not always enabled, even if it is present. Currently, documentation is maintained on the Linux Foundation Wiki [1]. In addition, many other kernel developers and real-time experts have contributed with significant contributions to the development of this patch too.

Open source operating systems such as Linux continue to evolve and have a significant impact in many embedded systems for control applications. In particular, embedded systems with real-time support are employed by a wide variety of applications ranging from simple consumer electronics and home appliances to military weapons and space

systems [2]. The fast growth of Industrial Internet of Things (IIoT) is accelerating the move towards open source Linux in embedded market share. The increasing requirements on the performance of real-time applications, and the need to reduce development costs and time, led to an increase in the interest for employing COTS (commercial off-the-shelf) hardware and software components in real-time applications [3–6]. However, their reliable real-time performance is still under investigation. This is an objective of this work research too. The real-time measurements are focused on real-time capabilities provided by PREEMPT_RT patch in handling real-time tasks and operations in user and kernel space. The experimental measurements platform is based upon ARM-based embedded devices, such as Raspberry Pi (a Raspberry Pi3 referred from now on as RPi3) and BeagleBoard microcontroller (a BeagleBone Black referred from now on as BBB), running in a master-slave mode. Raspberry Pi was designed as an educational and experimental board [7]. However, it has already made the leap into industry, e.g., with the Compute Module 3 (CM3) intended for industrial applications. Raspberry Pi applications are now part of industry 4.0 and the Internet of Things, e.g., the JanzTec emPC-A/RPI3+ Industrial Controller [8], and the Kunbus Revolution Pi [9].

The Linux kernel distributions for RPis and BBBs do not currently have any hard real-time support. Therefore, it is an issue under investigation in using Linux for hard real-time applications. However, it is possible to patch them with PREEMPT_RT, and hopefully in the future it will be provided as the de facto standard option in the mainline kernel for such microcontrollers. For the purposes of this research, specific software modules were developed and applied to investigate and evaluate the real-time performance of Linux kernels patched with PREEMPT_RT. Standard benchmark tools such as cyclictest [10] could have also been used. However, this benchmark is difficult to extend and does not combine different types of operations [11]. As a result, taking in consideration the specific hardware platforms under investigation and the aimed real-time applications with certain constraints and requirements (e.g., high priorities, locks of memory pages, high-resolution timers, and specific metrics measurements), it was decided as the most optimal approach to build our own new measurements software. The locking of memory pages is essential in order to avoid page faults and even thrashing. However, this is an issue which needs further investigation to ensure the most optimal memory usage. Currently, there is under investigation an interesting approach provided by Reuven and Wiseman [12], specifically for systems with very heavy memory usage, which propose thrashing minimization by splitting the processes into a number of bins, using Bin Packing approximation algorithms.

Development platforms such as Raspberry Pi and BeagleBone are being extensively used in IoT embedded applications, and even in Industrial IoT. Although their Linux kernel distributions do not have any hard real-time support, this is possible with the installation and configuration of the PREEMPT_RT patch. However, there is still no sufficient research work in the evaluation of the real-time performance of Linux kernels patched with PREEMPT_RT on such development platforms. This was one of the major motivations to investigate the real-time Linux kernels' behavior with the real-time preemption patch.

This work provides experimental results on real-time latency metrics for Linux kernels patched with PREEMPT_RT, on Raspberry Pi3 and BeagleBone Black development boards. Response and periodic task models were introduced, upon which novel software real-time measurement modules were designed. These modules take into consideration specific critical real-time requirements, e.g., high priorities, locks of memory pages, and high-resolution timers. In the majority of measurement cases, the worst-case maximum latency was decreased down to values in the order of a few tens of microseconds. One of the key findings is that a value of about 160 μs, as an upper bound, could be an acceptable safety margin for such low frequencies in many real-time systems running in a master-slave mode. Although that is a general outcome of several measurements under this specific master-slave schema, it provides some evidence that it could be valid for real-time embedded systems based on such devices and connected to various kinds of actuators, which require fast response times below this threshold value. Taking into account that

real-time capabilities of PREEMPT_RT patch and Linux mainline kernel continue to evolve, together with other constant improvements of ARM-based microcontrollers, both in terms of hardware and software, such systems can be another candidate for computing intensive applications in hard real-time applications. Some of the important aspects and outcomes of this research work are the following:

- Extends the measurements methodology presented in previous work by Brown and Martin [13], by introducing new sets of experiments with additional measurement metrics, applicable in a wider range of Linux kernels and distributions in ARM-based platforms.
- Implements latency measurements based on software real-time measurement modules, designed upon response and periodic task models.
- The same performance measurements approach and evaluation methodology could be applied and deployed on other Linux-based boards.

This paper is structured as follows: Section 2 describes previous related work; Section 3 presents the key components of the methodology followed; Section 4 describes the performance measurements algorithms and modules developed; Section 5 presents the setup of the experimentation platform; Sections 6 and 7 present the results of the experimental measurements of response and periodic tasks in user and kernel space; Section 8 presents a discussion analysis on the research findings; Section 9 provides a summary of the research results and draws conclusions.

2. Related Work

The real-time performance of operating systems and applications is analyzed with many different approaches [14–16]. The tools and methods used rely upon the performance metrics targeted, most commonly schedulability issues in real-time systems [17,18]. There is also a number of interesting schedulability analysis tools, e.g., RTDruid, TimeWiz, symTA/S, and chronVAL. Many different scheduling policies and algorithms exist, but not all of them are adequate for real-time tasks. Scheduling policies for real-time systems should ensure a number of factors, including first and foremost the timely response to critical events, low task switching and interrupt latency, low worst-case execution times, allowing for the preemption of any kind of task in the system, etc. In real-time operating systems, the methods and approaches used have to guarantee that certain deadlines are always met. The methods used to investigate predictability and timing characteristics of such systems typically measure scheduling jitter and interrupt latency with benchmark tools [19,20]. Tracing tools are also being used to identify latency issues [21,22]. Other approaches introduce the design and development of new benchmarks and software modules that investigate performance metrics of real-time operating systems [23,24].

The approach followed in this research work is based on software test modules, developed particularly for latency performance measurements in Linux kernels patched with PREEMPT_RT. A similar approach that inspired this research is presented in the work of Brown and Martin [13]. They developed a test system for evaluating the performance of two real-time tasks on Linux and Xenomai systems. They compare the performance of Linux kernels with real-time support such as Xenomai and the PREEMPT_RT patch, using C software modules to perform timing measurements of responsive and periodic tasks, with real-time characteristics, at user and kernel space. However, their evaluation is based only on a BeagleBoard microcontroller and Ubuntu Lucid Linux kernel configuration.

It is worth the effort to run a real-time kernel and evaluate its potential and performance benefits for applications. The advantages of using a real-time kernel are presented in many cases. However, performance evaluation of different kernel versions with real-time support has been presented primarily on Intel x86 platforms [25]. In the work of Litayem and Saoud [26], the authors evaluate the timing performance (latency) and throughput of PREEMPT_RT with different kernel versions, using cyclictest and unixbench. The platform is an x86 computer with CoreTM 2 Duo Intel CPU, running Ubuntu Linux 10.10. In the work of Fayyad-Kazan et al. [27], the authors present experimental measurements and

tests that benchmark RTOSs such as Linux with PREEMPT_RT (v3.6.6-rt17) against two commercial ones, QNX and Windows Embedded Compact 7. The tests were executed on an x86 platform (ATOM processor). In the work of Cerqueira and Brandenburg [20], a comparison of scheduling latency in Linux, PREEMPT_RT, and LITMUS RT is presented, based again on a 16-core Intel CPU platform. The majority of these works rely upon x86-based computer platforms with Ubuntu Linux.

The open source code accessibility and portability, the amount of implemented algorithms and libraries have made Linux with PREEMPT_RT a strong alternative to commercial RTOSs and specialized approaches, also in industrial environments [28]. Other research articles have recently focused on latency measurements of Raspbian Linux with real-time patch PREEMPT_RT vs. the standard Raspbian [29,30]. However, measurements are performed only with Raspbian Linux and the cyclictest benchmark.

The latest research shows that such Linux-based embedded systems play an important role in nearly every aspect of modern life, particularly in systems' real-time control [31–33]. However, there is still no sufficient research work on the evaluation of the real-time performance of Linux kernels patched with PREEMPT_RT on Raspberry Pi and BeagleBone Black development platforms. Their low cost, open source design, and ease of integration with various peripherals make these development platforms appropriate for research in various fields, particularly in embedded control systems, robotics, smart cities, sensors systems, and for fast experimentation and prototyping in manufacturing [34–36].

This research focuses on Linux latency measurements aiming to find out how the real-time patch affects its real-time performance. In contrast to many of the above works, the experimental platform includes multiple Linux kernels and distributions. This experimental work of the latency performance of the Linux kernels patched with PREEMPT_RT running adds to the knowledge and understanding of real-time execution behavior in such platforms.

3. Methodology

3.1. Objectives

One of the major goals of this research is to measure the real-time responses of Linux kernels and variants in ARM-based development platforms with the real-time preemption patch PREEMPT_RT. This goal is addressed by creating new software multithreaded modules in C, which implement the proposed measurement algorithms. These modules provide the ability to observe the execution state of multiple parameters including response latency during the real-time tasks execution.

3.2. Design Methodology

The research methodology is based upon two simple task models, the periodic task model and the sporadic task model [37]. In the periodic task model, the tasks of a job arrive strictly periodically, separated by a fixed time interval. In the sporadic task model, each task may arrive at any time once a minimum interarrival time has elapsed since the arrival of the previous task. This is because real-time tasks are usually activated in response to external events (e.g., upon sensor triggering) or by periodic timer expirations.

In this research, we introduce a response task model. In a periodic task model, each invocation of a task arrives strictly periodically, separated by a fixed time interval. In the proposed response task model, each task may arrive at any time upon the arrival of the previous task. Each task τ_i is characterized by: its execution time relative to a deadline t_i, maximum (or worst-case) response latency $wcrl_i$, and minimum interval time t_{irv}. A task's worst-case response latency $wcrl_i$ is defined as the overall time elapsed from the arrival of this task (timer interrupt) to the moment this task is switched to a running state producing results. The models' structure is described in the algorithms provided below in Section 4 and implemented as the measurement software modules. In the experiments, each task τ_i is scheduled using the highest real-time priority to eliminate the latency caused by scheduling jitter. Each module executes the measurements loop, based on timing data

acquired from the device under test, performs analysis of the measurements and outputs the results. The experiments with the software modules were executed multiple times to obtain the following measurements:

- The optimum sustained interrupt frequency, that is, the maximum frequency of the signal on the associated GPIO line (General Purpose Input/Output) that can handle efficiently running in continuous mode.
- The response latency, that is, the estimated time elapsed between GPIO input level change (IRQ trigger—interrupt request) and GPIO output level change.
- In response tasks, measure the total time elapsed until the device under test responds, while in periodic tasks, measure whether the slave device responds at proper time periods.

3.3. Measurements Software Design Considerations

Real-time multithreaded modules were executed under two modes, in user and kernel space. A thread is a basic unit of CPU utilization, which can be implemented in user space or in kernel space. These multithreaded applications perform the proposed response and periodic real-time tasks. Processes are scheduled under the real-time policy SCHED_FIFO, having a sched_priority value in the range of 1 (low) to 99 (high). This ensures a timely execution of the tasks and decreased execution times and latencies. SCHED_FIFO policy runs a task until it is preempted by a higher priority task. This may not contribute to the overall throughput, but will increase the determinism by allowing all kernel space to be preemptible as all interrupt handlers are switched to threaded interrupts. Scheduling policy, attributes and priorities were also set per thread upon their creation, with POSIX thread scheduling policy functions calls. The design approach is illustrated in Figure 1.

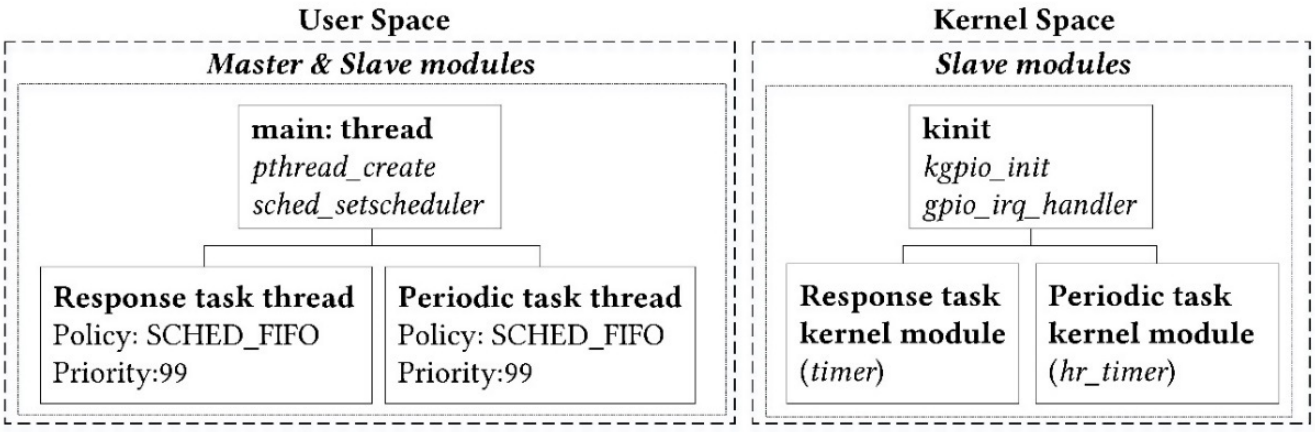


Figure 1. Software modules' real-time design scheme.

The software modules are designed in a master-slave mode and perform measurements in user and kernel space of response and periodic real-time tasks. The master software module controls the overall execution process and performs the actual measurements in user and kernel space. The slave modules run the actual tasks on the device under test and provide feedback to the master control modules. The measurements are passed as function arguments to threads function calls during their creation (pthread_create()). The running tasks are scheduled as threads, with real-time SCHED_FIFO scheduling policy (sched_setscheduler()) and high priority set to 99. All user-space processes are scheduled with real-time scheduling class SCHED_FIFO, and high priorities. From a scheduling point of view, it makes no difference between the initial thread of a process, e.g., executing the main() function, and all additional threads created dynamically. The slave modules in kernel space are implemented as kernel modules. The initialization function (kgpio_init) in the response task uses the GPIO kernel interface and an interrupt handler function

(gpio_irq_handler) to service the input changes. In the periodic task, the slave modules uses a high-resolution timer (hrtimer) to produce timer-based interrupts.

It is possible to avoid intercore interferences by setting the processor affinity. However, the intention is to investigate threads execution by having more than one thread per core, and threads are allowed to migrate among all cores in RPi3 ARM CPU. This will affect scheduling latencies due to potential locks, and will add to the total response latency.

4. Performance Measurements Modules

Usually, the worst-case execution time (WCET) analysis is mandatory for hard real-time system performance evaluation according to their latency. Using the software modules developed, experimental tests run for a long duration (about 1 h each test run) to evaluate the latency that occurred in real-time task execution, at the Linux kernels patched with PREEMPT_RT and the standard ones. Measurements were conducted at user and kernel space.

The software runs in a master-slave mode. A synopsis of the overall software control flow is presented in Figure 2.

The master software module performs initializations, sets the scheduling policy and events to poll, and triggers the device under test (writes GPIO output, gets clock time and polls input). In user space, the slave software in response mode polls the input and writes the output accordingly, while in periodic mode it reads the timer until the time interval elapsed and writes the output. In kernel space, the slave software (as a kernel module) in response mode services the interrupt by getting the input value and setting the output, while in periodic mode it services the interrupt starting the high-resolution timer and returns. Once the desired number of loops is reached, the master software module performs metrics calculations and outputs the results.

4.1. Response Task Modules

The response task modules, based on the measurements analysis presented earlier, invoke code that measures the responsiveness of the real-time applications in user and kernel space. The measurements software module in the master device performs the overall control of execution and metrics measurements at user and kernel space. This module triggers the slave device at specific and random time intervals in a loop for a number of iterations (1 M), and measures the time elapsed (latency) until the slave device under test responds. In user space, in the slave device, the software module responds to GPIO toggle frequency (e.g., 10 kHz) in an asynchronous manner by activating a GPIO output, as soon as the level of a GPIO input changes. In kernel space, in the slave device the software module is inserted into the slave's kernel as a loadable kernel module. This module uses an interrupt handler function (only the top-half) to service the input change.

Master and Slave Response Tasks in User and Kernel Space

The Linux kernel has a way to expose internal structures using SysFS, a virtual file system which exposes a common interface for kernel implementation details and internal structures. The software modules make use of the kernel's SysFS interface. The algorithms that describe the basic functionality of these modules are shown as pseudocode in Algorithms 1–3.

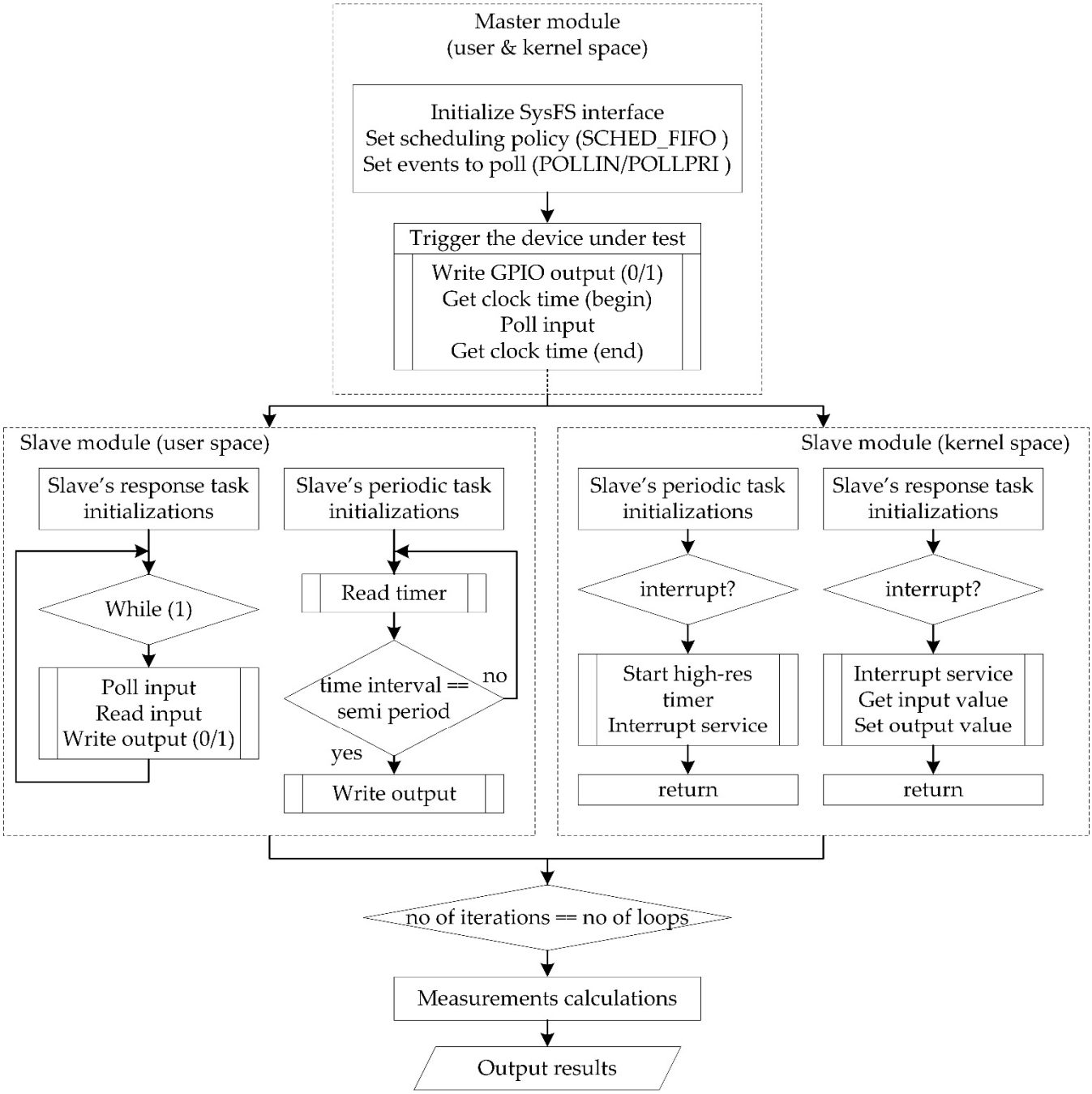


Figure 2. Software control flow.

Algorithm 1 Master response task in user and kernel space

```
scheduling is SCHED_FIFO at priority 99 ← set thread's scheduling algorithm to real-time
events is POLLIN or POLLPRI ← set the events to poll until there is data to read
loops ← set by command line argument
no_of_iterations is below or equal to loops
while no_of_iterations is below or equal to loops, do
        setting ← 1
        write fd_output setting ← set the value of output pin that triggers the slave
        clock_gettime begin_time
        poll fd_input for events ← await for interrupt infinitely
        clock_gettime end_time
        read fd_input ← read input once enabled by the slave
        setting ← 0
end
perform measurements
```

Algorithm 2 Slave response task in user space

```
scheduling is SCHED_FIFO at priority 99 ← set thread's scheduling algorithm to real-time
events is POLLPRI ← set the events to poll until there is data to read
while 1 do
read fd_input ← read input once enabled by the master
        poll fd_input for events ← await for interrupt infinitely
        write fd_output setting ← set the value of output pin accordingly (to 0 or 1)
end
```

Algorithm 3 Slave response task in kernel space

```
function kgpio_init ← uses the GPIO kernel interface
        gpio_request gpio_out ← request GPIO output
        gpio_direction output ← set up as output
        gpio_request gpio_in ← request GPIO input
        gpio_direction input ← set up as input
        gpio_to_irq irqNumber ← maps GPIO to IRQ number
        irq_request irq_handler ← request an interrupt line
end function kgpio_init
function gpio_irq_handler ← uses an interrupt handler function (only the top-half) to service the
input change
        gpio_get_value gpio_in ← gets GPIO input value
        gpio_set_value gpio_out to gpio_in ← sets GPIO output accordingly
        return IRQ_HANDLED ← interrupt serviced
end function gpio_irq_handler
```

4.2. Periodic Task Modules

The purpose of the periodic task modules is to periodically execute at a specific interval certain process interrupts. The master control software monitors whether the slave device under test responds at proper periods in user and kernel space measurements. The slave device responds to the interrupts by toggling the value (0, 1) of an output pin, at specific time intervals, based on an internal timer. In kernel space, the slave's software uses an internal high-resolution timer, which is inserted as a kernel module. Due to the fact that a periodic timer interrupt is not an appropriate solution for a real-time kernel, most of the existing real-time kernels provide high-resolution timers [38–41]. Since hard real-time systems usually have timing constraints in the micro seconds range, a high-resolution timer is usually a requirement when a task needs to occur more frequently than the 1 millisecond resolution provided under Linux.

Master and Slave Periodic Tasks in User and Kernel Space

Reliable latency performance measurements require accurate timing source. For this reason, the performance measurements software make use of the system call clock_gettime() with the highest possible resolution, and the clock is set to CLOCK_MONOTONIC. The master control software reads the slave's input for the corresponding interrupts and measures the time interval in between (half period). The slave's software module uses, again, a high-resolution timer to produce timer-based interrupts. The algorithms that describe the basic functionality of these modules are shown as pseudocode in Algorithms 4 and 5.

Algorithm 4 Slave periodic task in user space

```
timerfd_create is CLOCK_MONOTONIC ← set the clock to mark the timer's progress
timerfd_settime is ABSTIME ← start the timer
semi_period_interval ← set by command line argument
no_of_iterations is below or equal to semi_period_interval
while no_of_iterations is below or equal to semi_period_interval, do
        read timer_fd ← read the timer until the time interval is elapsed
        write fd_output setting ← set the value of output pin accordingly (to 0 or 1)
end
```

Algorithm 5 Slave periodic task in kernel space

```
function kgpio_init ← uses an internal high-resolution timer
        hr_timer_init high_res_timer
        hr_timer_set CLOCK_MONOTONIC
        hr_timer_mode HRTIMER_MODE_REL
        hr_timer_function timer_func
end function kgpio_init
function gpio_irq_handler ← the GPIO IRQ handler function
        hrtimer_start high_res_timer ← starts high-resolution timer
        return IRQ_HANDLED ← interrupt serviced
end function gpio_irq_handler
```

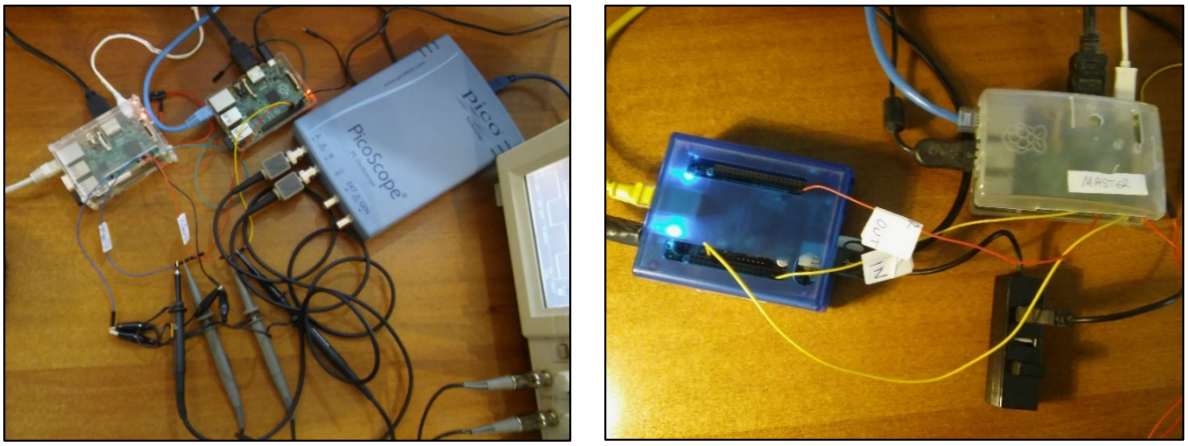
4.3. Issues Solved

Real-time metrics measurements depend upon how well software or benchmarking modules are written, as well as how well the kernel is configured. Comparing the performance of a real-time application running in different systems is a challenge, mainly because of the difficulty to isolate the various different factors that may affect performance. That usually implies the configuration of the kernel and adaptation of the source software to the native kernel of each system. Optimal decisions also have to be made on how to set various settings related, e.g., to memory management mode, system timers, peripheral devices configuration, etc., since they can make a huge difference on the latencies of a given system. During the experimental work, a few problems were encountered and solved, meaning that there are still issues to be considered and improved in real-time support with PREEMPT_RT. In some cases, long latencies were due to the use of timer functions on time measurements other than clock_gettime() or clock_nanosleep(). In another case, it was observed that during the experimental runs, after a few minutes the RPi run into instability and the system had to be restarted. In particular, the FIQ (Fast Interrupt reQuest) system implementation causes lock ups of the RPi when using threaded interrupts. A solution to this problem is proposed by the Open Source Automation Development Lab (OSADL) [42], which disables the IRQ while the FIQ spin lock is held, and indeed the kernel run stable. As the Linux Foundation points out [1], since the kernel of Raspberry Pi is not part of the mainline, there are some known limitations of PREEMPT_RT running on RPi platforms.

5. Experimental Setup

A Raspberry Pi3 (RPi3) is used as the master device in all measurement schemes. The RPi3 has integrated a System on Chip (SoC) based on Broadcom BCM2837, which features a 1.2 GHz 64-bit quad-core ARM Cortex-A53 (ARMv8) processor. The BeagleBone Black development board features a 1 GHz ARM Cortex-A8 (ARMv7) processor based on TI Sitara AM3358AZCZ100 SoC from Texas Instruments. Both the devices are low-cost and low-power single-board computers, commonly used as development platforms for various system applications, specifically for embedded systems.

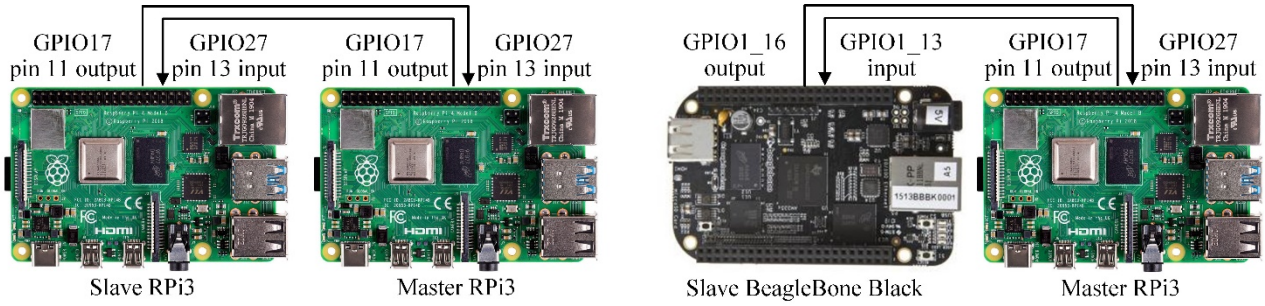
The slave devices (RPi3 and BBB) can communicate and transfer data to and from the master device using the standard GPIO interface. The master and slave arrangements are shown in Figure 3.



(**a**) (**b**)

Figure 3. The system experimental setup schemes: (**a**) RPi3 to RPi3; (**b**) RPi3 to BBB.

The master and slave devices are connected through GPIOs in a master-slave schema, as illustrated in Figure 4. For RPi3, GPIO27 (pin 13) in the slave device is defined as input and connected to GPIO17 (pin 11) defined as output in the master device. For BBB, GPIO1_13 (pin 11) is defined as input and GPIO1_16 (pin 15) as an output. The connections establish a pin-to-pin bidirectional communication, so the same connection is applied, respectively, in reverse directions from the slave devices to the master.



(**a**) (**b**)

Figure 4. Illustration of devices connections: (**a**) RPi3 to RPi3; (**b**) RPi3 to BBB.

Standard Linux kernel configurations and kernels with real-time support were installed and configured (on different microSD cards) on the slave devices under test. These include: Ubuntu Mate (4.14.74-rt44-v7), Arch Linux (4.19.10-1-ARCH), and Debian (4.19.67-2). The developed software measurement modules provide consistent and reliable results based on multiple experiments. These were visualized and validated with

an oscilloscope. In particular, the latency measurements obtained internally by the software modules are compared to those directly measured externally with an oscilloscope.

6. Response Task Measurements in User and Kernel Space

6.1. Response Task Measurements in User Space

The master control software at specific time intervals runs a task τ_i that triggers a GPIO input on the slave device, with loops of "0 s" and "1 s", which the slave is polling in an infinite loop. Then, it begins to measure the slave's response delay and accumulates relevant measurement metrics. The slave device, upon reading the change of the input state, sets its output accordingly (on a rising edge it sets its output line, while on a falling edge it clears its output line). Then, the master device repeats the loop for a number of cycles (1 million loops) for sufficient samples to be collected for analysis. The variation in the input signal level (values of 0 s and 1 s) provides a way to check that the devices under test read the input signals correctly, and respond appropriately and accordingly. Measurements are performed on both edges, rising and falling, of the trigger signals, as shown in Figure 5.

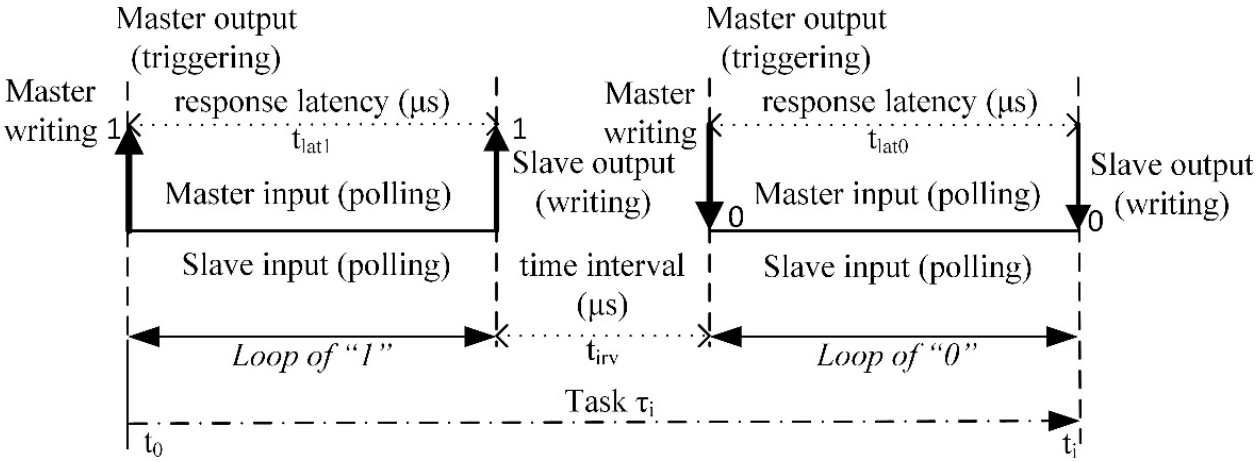


Figure 5. Measurements on both edges of the trigger signals.

Each running task τ_i runs two loops, and thus consists of two subtasks—loops of "1 s" and "0 s"—which are executed sequentially and alternatingly. The total execution time includes the execution of all the subtasks (that is, t_{lat1} for loop of "1 s" and t_{lat0} for loop of "0 s") times the amount of iterations, plus the time interval t_{irv} between the generated subtasks. In the experiments, the time interval t_{irv} in between was initially unset and random; however, later, for efficiency purposes, it was set to specific values within the range of 1 to 10 ms. This is because for lower time intervals it was observed that long delays sometimes appear on latency measurements. Although rare, such delays make it apparent that the devices could not react properly at such frequencies.

6.1.1. Estimation of Maximum Sustained Frequency

The time interval between two consecutive generated interrupts is estimated so that subtasks are properly initialized and executed. For this purpose, a number of tests have been executed with variable frequency values to determine the optimum value for the time interval between the generated interrupts at the master device. The results show that the slave devices with PREEMPT_RT can handle all the generated interrupts if the time interval in between is above 10 ms. This value was set for the majority of the experiments, and below, for testing and sensitivity analysis purposes. That means that we could toggle the state of a GPIO pin, e.g., with a low frequency, e.g., of 1000 Hz, with millions of interrupts (1 M), and get reliable responses.

6.1.2. Response Latency Measurements

The importance of measuring the response latency is unquestionable in real-time systems. The slave devices were tested continuously by circulating the loops of "1 s" and "0 s" for a million (1M) interrupts, with an average cycle duration of 120 µs, and an overall running time of about 3 h. At the end of each measurement cycle, the master control software processes the results and estimates the mean, minimum, and the maximum response latency, plus some statistics on variance and standard deviation.

6.2. Response Task Measurements in Kernel Space

In kernel space, experimentation is conducted in a similar way. The master control software initiates the triggering cycles at specific intervals, which the slave's software module is polling in an infinite loop, and responds once a change of the input state is detected. The slave's software in this case is a kernel module developed for this purpose and inserted in the kernel.

7. Periodic Task Measurements in User and Kernel Space

In periodic measurements, the master device measures the signal's length period produced by the slave's internal timer. The master software module checks at specific time periods the slave's output status in order to verify that the device responds at proper periods, and at the same time to investigate the state upon which the slave device cannot react properly.

In user space, the master device is polling the slave device in an infinite loop, until its GPIO input status is changed (rising edge of the first interrupt). On the other hand, the slave device toggles periodically the value of an output configured pin at a specific periodic rate, based on an internal timer. The master control software begins to count the time until its input status has changed again (falling edge of the second interrupt) (Figure 6).

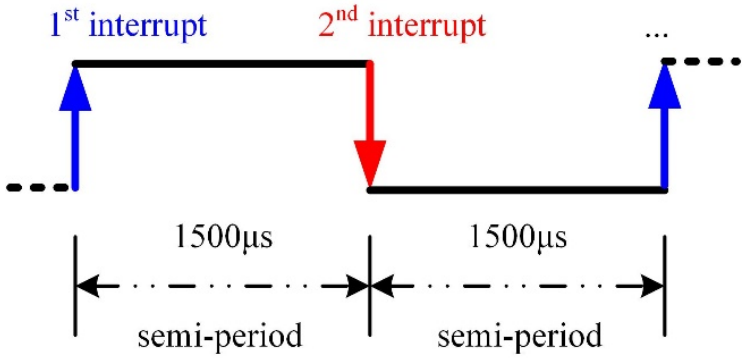


Figure 6. Measurements of the slave's timer interrupts.

In kernel space, the experimental setup and layout of the devices is the same as described earlier. The master device performs the measurements in a similar way to the user space experimentations. However, in this case, the slave's control software is a kernel module that uses an internal high-resolution timer to produce the periodic interrupts.

Measurements are performed on both edges of the triggering signals for a variable number of samplings starting at 10,000 and decreasing, with a semi-period at 15,000 µs (down to a 1500 µs period). The results show that the slave devices generate the timer interrupts at exact time intervals, both at standard Linux kernels and with real-time support.

8. Results and Discussion

Linux-based platforms on ARM-based devices such as the Raspberry Pi3 and Beagle-Bone Black are continuously gaining popularity in various standalone control applications as embedded systems. However, many of the approaches to measuring their real-time performance and particularly latency are still based on x86 CPU architectures and the use of

benchmark tools such as cyclictest. Regarding latency measurements, very few works, such as the work of Brown and Martin [13], proceed into the development of specific software measurement modules for such ARM-based devices. Their research inspired this work to extend the measurement metrics to a wider range of Linux kernels and distributions in such ARM-based embedded platforms. There are software structure similarities in both approaches; however, the hardware development platforms and Linux kernel versions are different. On the other hand, both hardware platforms are based on ARM CPU architectures running among other Linux distributions, Ubuntu too. Table 1 provides a summary of the results obtained in both approaches for Ubuntu Linux distributions and kernels with real-time support. Even though the results are very close, the intention is rather to reconfirm the results obtained with PREEMPT_RT, rather than providing a fair comparison, since the kernel versions are significantly different.

Table 1. Software module latency results on RPi3, BeagleBone Black and BeagleBoard C4.

Hardware	Linux OS with PREEMPT_RT	Periodic Tasks Period (µs)	Response Tasks Latency (µs)	
			User Space (min, max)	Kernel Space (min, max)
RaspberryPi3 Model B 64-bit ARM Cortex-A53 quad core, 1200 MHz (our platform approach)	Ubuntu Mate, kernel 4.14.74-rt44-v7	30.000 jitter = 0	49, 147 90% of the latencies <<147	50, 67 95% of the latencies <<67
BeagleBone Black 32-bit ARM Cortex-A8 1000 MHz (our platform approach)	Ubuntu, kernel 4.14.74-rt44-v7	30.000 jitter = 0	50, 160 90% of the latencies <<160	48, 76 95% of the latencies <<76
BeagleBoard C4, OMAP3520 SoC, 32-bit ARM Cortex-A8, 720 MHz (Brown and Martin [13])	Ubuntu Lucid Linux, kernel 2.6.33.7-rt29	7.071 jitter = 0	157 (max) for 95% of the time 796 (max) for 100% of the time	43 (max) for 95% of the time 336 (max) for 100% of the time

In RPi3 and BeagleBone Black with PREEMPT_RT patched kernels, the minimum latency is measured below 50 µs, both at user and kernel spaces. In user space, 90% of the latencies fall below the maximum of 147 µs and 160 µs, respectively, while in kernel space, 95% of the latencies fall below the maximum of 67 µs and 76 µs, respectively. In BeagleBoard C4, at user space, for 95% of the time the maximum latency does not exceed the value of 157 µs, while in kernel space, this value is lower at 43 µs. Figure 7 illustrates the above results for both approaches in all devices.

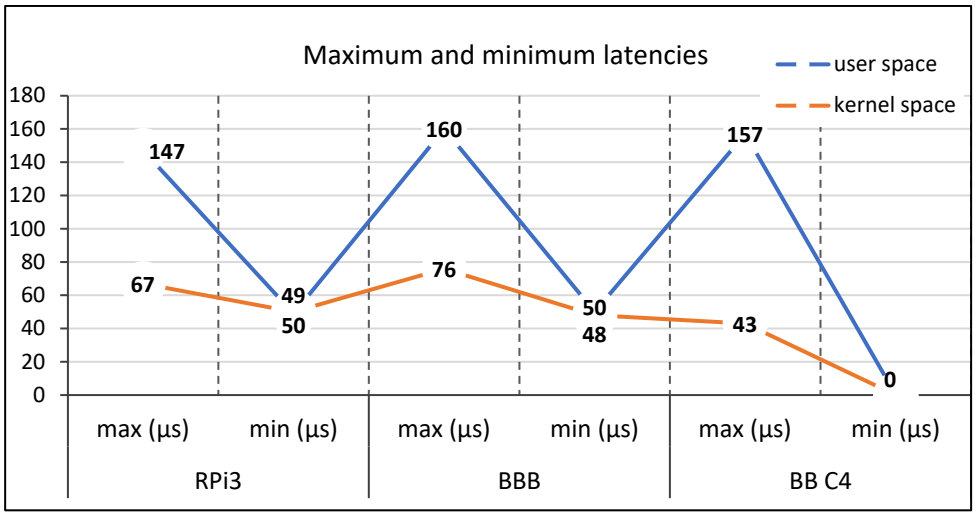


Figure 7. Latency comparison for both approaches and preempted kernels in user and kernel space on all devices (Raspberry Pi3, BeagleBone Black and BeagleBoard C4).

Response Latency Results

Table 2 presents a comparative summary of the response latency results for Raspberry Pi3 and BeagleBone Black running Linux kernels with PREEMPT_RT patch. Data on the most commonly used measures of spread that is variance (var) and standard deviation (stdev) are also given.

Table 2. Response latency results on RPi3 and BeagleBone Black.

RaspberryPi3/ BeagleBone Black	Linux Kernel Version	Samples	Space	PREEMPT_RT	Latency (µs)		
					Software Modules		Oscilloscope min/max/avg
					min/max (wcrl)	stdev/var	
RPi3	Ubuntu Mate 4.14.74-rt44-v7	1 M	user	yes	49/147	65/4261	49/128/105
				no	53/360	33/1137	51/370/109
			kernel	yes	50/67	20/417	42/56/50
				no	51/81	15/233	44/93/70
BBB	Ubuntu 4.14.74-rt44-v7	1 M	user	yes	50/160	69/4771	50/122/102
				no	54/380	14/208	53/385/120
			kernel	yes	48/76	23/566	49/60/59
				no	49/70	23/564	52/80/67
RPi3	Arch Linux 4.19.10-1-ARCH	1 M	user	yes	42/122	67/2336	44/129/98
				no	54/330	32/990	51/350/111
			kernel	yes	51/56	20/411	40/51/51
				no	51/79	17/788	43/89/65
BBB	Arch Linux 4.19.10-1-ARCH	1 M	user	yes	54/134	61/1025	50/139/101
				no	55/360	30/678	54/389/120
			kernel	yes	57/69	22/312	50/75/70
				no	56/86	16/243	51/98/76
RPi3	Debian (Buster) 4.19.67-2	1 M	user	yes	40/90	79/6336	41/98/95
				no	53/343	34/1190	53/310/101
			kernel	yes	48/53	19/383	42/50/49
				no	48/60	13/2988	44/76/59
BBB	Debian 4.19.67-2	1 M	user	yes	47/96	71/5085	49/104/101
				no	53/376	33/1089	55/360/121
			kernel	yes	51/67	25/644	49/70/65
				no	53/78	13/190	50/89/79

The specified kernel versions are different in order to investigate the spread of the variations in latency results. Linux kernels with real-time support maintain much lower latencies. The oscilloscope measurements reconfirm the results produced with the software measurement modules.

The majority of Linux kernels' measurements with PREEMPT_RT-patched kernel show the minimum response latency to be below 50 µs, both in user and kernel space. The maximum worst-case response latency (wcrl) reached 147 µs for RPi3 and 160 µs for BBB in user space, and 67 µs and 76 µs, respectively, in kernel space (average values). Most of the latencies are quite below this maximum (90% and 95%, respectively, for user space and kernel space). In general, it seems that maximal latencies do not often cross these values.

The measurements in standard Linux kernels show the minimum response latency to be about the same and below 55 µs, both in user and kernel space. However, the

maximum worst-case response latency reached 360 µs (RPi3) and 380 µs (BBB) in user space, and 160 µs and 86 µs, respectively, in kernel space. This maximum observed latency is significantly higher than the one observed under the PREEMPT_RT-patched Linux kernels. Figure 8 illustrates the worst-case response latencies in user space for both kernels (standard and preempted).

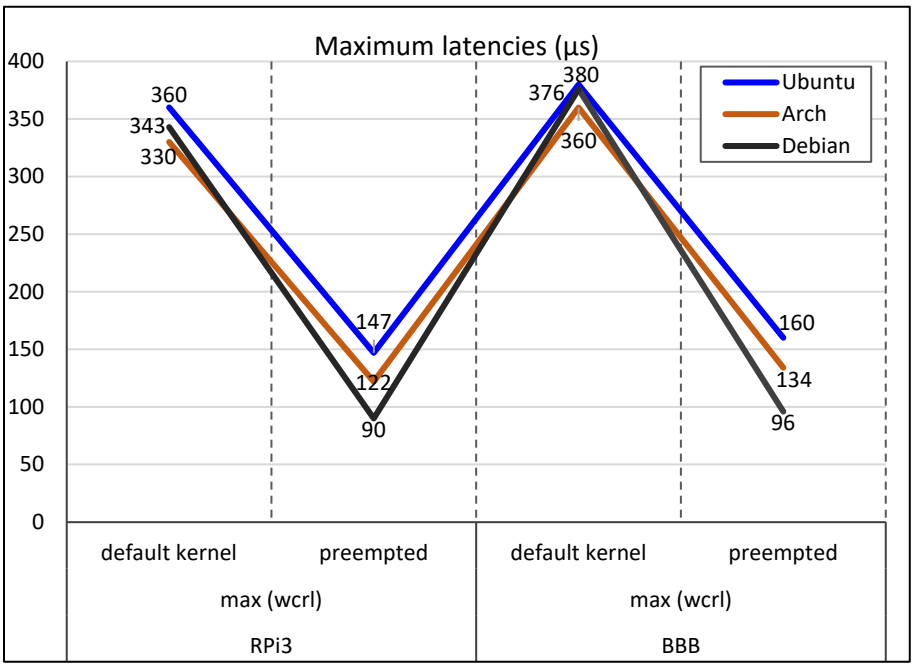


Figure 8. Worst-case latency (wcrl) comparison for both kernels (standard and preempted) in user space for Raspberry Pi3 and BeagleBone Black.

In real-time systems designed with tight timing constraints, these worst-case latency values must be taken into consideration.

9. Conclusions

This research work presents the experimental evaluations on the real-time performance of the PREEMPT_RT patch, and particularly latency metrics, in Linux kernels and distributions running on Raspberry Pi3 Model B and BeagleBone Black ARM-based development platforms. These devices have become a popular choice for a wide range of applications in many embedded systems, while being easy to use, flexible, and lower cost.

Challenges in recent real-time embedded systems, such as those found in cloud computing platforms using commercial-off-the-shelf technology, have prompted further research into their real-time behavior. However, currently, there is still limited research on investigating the real-time performance of such ARM-based architectural platforms running Linux patched with PREEMPT_RT.

This experimental work provides further insights into their real-time behavior. The performance measurement and evaluation approach is based upon the introduction of response and periodic task models implemented as new specific real-time software measurement modules. This could be applied and deployed on other Linux-based development boards and platforms too. Any device that supports a Linux kernel version, e.g., from release 4 (e.g., 4.4, 4.9, 4.14, 4.19) and later, and configured with the PREEMPT_RT patch, is an appropriate platform to deploy the developed measurement modules. These experimental software modules written in C are available as an open-source project at GitHub https://github.com/gadam2018/RPi-BeagleBone (accessed on 10 April 2021). There are further details provided about their installation and usage. The experimental results show that latencies on kernels with real-time support are considerably lower compared to those in the standard kernels and the majority falls below 50 µs. The average maximum observed

latency of 160 µs is still significantly lower than the one observed under the standard Linux kernels. As an outcome, Linux kernels patched with PREEMPT_RT on such devices have the ability to run in a deterministic way as long as a latency value of about 160 µs, as an upper bound, is an acceptable safety margin. Such results reconfirm the reliability of such COTS devices running Linux with real-time support and extend their life cycle for the running applications. In addition, such devices could further stimulate their use in the development of architectural frameworks and systems for reliable real-time control applications, as is the case presented here [43]. Initially, the preliminary results of this research were also utilized in the development of a real-time controller based on Raspberry Pi and kernel modules [44].

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The experimental software modules are available as an open-source project at GitHub https://github.com/gadam2018/RPi-BeagleBone (accessed on 10 April 2021).

Acknowledgments: The author would like to thank the Computer Systems Laboratory (CSLab, https://cslab.ds.uth.gr/, accessed on 25 April 2021) at the Department of Digital Systems, University of Thessaly, Greece, for the technical support and the resources provided for this experimental research.

Conflicts of Interest: The author declares no conflict of interest.

References

1. The Linux Foundation: Real Time Linux. Available online: https://wiki.linuxfoundation.org/realtime/start (accessed on 14 January 2021).
2. Sheikh, S.Z.; Pasha, M.A. Energy-Efficient Multicore Scheduling for Hard Real-Time Systems—A Survey. *ACM Trans. Embed. Comput. Syst.* **2019**, *17*, 26. [CrossRef]
3. Adam, G.K.; Petrellis, N.; Kontaxis, P.A.; Stylianos, T. COTS-Based Real-Time System Development: An Effective Application in Pump Motor Control. *Computers* **2020**, *9*, 97. [CrossRef]
4. Adam, G.K.; Kontaxis, P.A.; Doulos, L.T.; Madias, E.-N.D.; Bouroussis, C.A.; Topalis, F.V. Embedded Microcontroller with a CCD Camera as a Digital Lighting Control System. *Electronics* **2019**, *8*, 33. [CrossRef]
5. Mukherjee, A.; Mishra, T.; Chantem, T.; Fisher, N.; Gerdes, R. Optimized trusted execution for hard real-time applications on COTS processors. In Proceedings of the 27th International Conference on Real-Time Networks and Systems (RTNS '19), Toulouse, France, 6–8 November 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 50–60. [CrossRef]
6. Rahman, M.; Ismail, D.; Modekurthy, V.P.; Saifullah, A. Implementation of LPWAN over white spaces for practical deployment. In Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI '19), Montreal, QC, Canada, 15–18 April 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 178–189. [CrossRef]
7. ElAarag, H. Deeper learning in computer science education using raspberry pi. *J. Comput. Sci. Coll.* **2017**, *33*, 161–170. [CrossRef]
8. JanzTec Industrial Computing Architects. emPC-A/RPI3+: Embedded Computing System Based on Raspberry Pi 3 B+ Module. Available online: www.janztec.com/en/devices/embedded-computer/empc-arpi3/ (accessed on 2 December 2020).
9. Kunbus Industrial Communication. RevPi Connect. Available online: https://revolution.kunbus.com/revpi-connect/ (accessed on 4 December 2020).
10. The Linux Foundation: Cyclictest. Available online: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start (accessed on 15 November 2020).
11. Maggio, M.; Lelli, J.; Bini, E. Rt-Muse: Measuring real-time characteristics of execution platforms. *Springer Real-Time Syst.* **2017**, *53*, 857–885. [CrossRef]
12. Reuven, M.; Wiseman, Y. Medium-Term Scheduler as a Solution for the Thrashing Effect. *Comput. J.* **2006**, *49*, 297–309. [CrossRef]
13. Brown, J.; Martin, B. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. In Proceedings of the 12th Real-Time Linux Workshop (OSADL'10), Nairobi, Kenya, 25–27 October 2010; OSADL: Stuttgart, Germany, 2010; pp. 1–17.
14. Tan, S.L.; Nguyen, B.A.T. Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers. *IEEE Micro* **2009**, *99*, 1. [CrossRef]
15. Marieska, M.D.; Hariyanto, P.G.; Fauzan, M.F.; Kistijantoro, A.I.; Manaf, A. On performance of kernel based and embedded real-time operating system: Benchmarking and analysis. In Proceedings of the International Conference on Advanced Computer Science and Information Systems (ICACSIS'11), Jakarta, Indonesia, 17–18 December 2011; IEEE Press: Piscataway, NJ, USA, 2011; pp. 401–406.

16. Hambarde, P.; Varma, R.; Jha, S. The Survey of Real Time Operating System: RTOS. In Proceedings of the 2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies, Nagpur, India, 9–11 January 2014; IEEE Press: Piscataway, NJ, USA, 2014; pp. 34–39. [CrossRef]
17. Gardner, K.; Harchol-Balter, M.; Hyytia, E.; Righter, R. Scheduling for efficiency and fairness in systems with redundancy. *Perform. Eval.* **2017**, *116*, 1–25. [CrossRef]
18. Garre, C.; Mundo, D.; Gubitosa, M.; Toso, A. Performance comparison of real-time and general-purpose operating systems in parallel physical simulation with high computational cost. In Proceedings of the SAE World Congress & Exhibition, Detroit, MI, USA, 8–10 April 2014; SAE International: Warrendale, PA, USA, 2014. [CrossRef]
19. Bristot de Oliveira, D.; Casini, D.; Oliveira, R.; Cucinotta, T. Demystifying the Real-Time Linux Scheduling Latency. In Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020), Modena, Italy, 7–10 July 2020; Dagstuhl Publishing: Wadern, Germany, 2020. Available online: https://drops.dagstuhl.de/opus/volltexte/2020/12372/ (accessed on 12 October 2020).
20. Cerqueira, F.; Brandenburg, B. A Comparison of Scheduling Latency in Linux, PREEMPT_RT, and LITMUS RT. In Proceedings of the 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT'13), Paris, France, 9 July 2013; SYSGO: Mainz, Germany, 2013; pp. 19–29.
21. Gebai, M.; Dagenais, M.R. Survey and analysis of kernel and userspace tracers on Linux: Design, implementation, and overhead. *ACM Comput. Surv.* **2018**, *51*, 33. [CrossRef]
22. Beamonte, R.; Dagenais, M.R. Linux Low-Latency Tracing for Multicore Hard Real-Time Systems. *Hindawi Adv. Comput. Eng.* **2015**, 8. [CrossRef]
23. Vincze, D.; Kovacshazy, T. Benchmark Tool for the Characterization of the Real-Time Performance of Linux on System on a Chip Platforms for Measurement Systems. In Proceedings of the 21st IMEKO TC-4 International Symposium and 19th International Workshop on ADC Modelling and Testing, Budapest, Hungary, 7–9 September 2016; International Measurement Confederation IMEKO: Budapest, Hungary, 2016.
24. Arm, J.; Bradac, Z.; Kaczmarczyk, V. Real-time capabilities of Linux RTAI. In Proceedings of the14th IFAC Conference on Programmable Devices and Embedded Systems (PDES'16), Brno, Czech Republic, 5–7 October 2016; Volume 25, pp. 401–406. [CrossRef]
25. Delgado, R.; Choi, B. New Insights into the Real-Time Performance of a Multicore Processor. *IEEE Access* **2020**, *8*, 199–211. [CrossRef]
26. Litayem, N.; Saoud, S.B. Impact of the Linux Real-time Enhancements on the System Performances for Multi-core Intel Architectures. *Int. J. Comput. Appl.* **2011**, *17*, 17–23. [CrossRef]
27. Fayyad-Kazan, H.; Perneel, L.; Timmerman, M. Linux PREEMPT-RT vs. Commercial RTOSs: How Big is the Performance Gap. *GSTF J. Comput.* **2013**, *3*, 135–142. [CrossRef]
28. Reghenzani, F.; Massari, G.; Fornaciari, W. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Comput. Surv.* **2019**, *52*, 36. [CrossRef]
29. Latency of Raspberry Pi3 on Standard and Real-Time Linux 4.9 Kernel. Available online: https://metebalci.com/blog/latency-of-raspberry-pi-3-on-standard-and-real-time-linux-4.9-kernel/ (accessed on 19 November 2020).
30. Raspberry Pi: Preempt-RT vs. Standard Kernel 4.14.y. Available online: https://lemariva.com/blog/2018/02/raspberry-pi-rt-preempt-vs-standard-kernel-4-14-y (accessed on 14 October 2020).
31. Boltov, Y.; Skarga-Bandurova, I.; Kotsiuba, L.; Hrushka, M.; Krivoulya, G.; Siriak, R. Performance Evaluation of Real-Time System for Vision-Based Navigation of Small Autonomous Mobile Robots. In Proceedings of the 10th International Conference on Dependable Systems, Services and Technologies (DESSERT), Leeds, UK, 5–7 June 2019; IEEE Press: Piscataway, NJ, USA, 2019; pp. 218–222. [CrossRef]
32. Bokingkito, P.B.; Llantos, O.E. Design and Implementation of Real-Time Mobile-based Water Temperature Monitoring System. In Proceedings of the 4th Information Systems International Conference (ISICO'17), Bali, Indonesia, 6–8 November 2017; Elsevier Procedia Computer Science: Amsterdam, The Netherlands, 2017; Volume 124, pp. 698–705. [CrossRef]
33. Kurkovsky, S.; Williams, C. Raspberry Pi as a Platform for the Internet of Things Projects: Experiences and Lessons. In Proceedings of the 22nd Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '17), Bologna, Italy, 3–5 July 2017; ACM Press: New York, USA, 2017; pp. 64–69. [CrossRef]
34. Petrov, N.; Dobrilovic, D.; Kavalic, M.; Stanisavljev, S. Examples of Raspberry Pi usage in Internet of Things. In Proceedings of the International Conference on Applied Internet and Information Technologies, Bitola, Macedonia, 3–4 June 2016; FICT Press: Bitola, RN, Macedonia, 2016; pp. 112–119. [CrossRef]
35. Costa, D.G.; Duran-Faundez, C. Open-Source Electronics Platforms as Enabling Technologies for Smart Cities: Recent Developments and Perspectives. *Electronics* **2018**, *7*, 404. [CrossRef]
36. Kour, V.P.; Arora, S. Recent Developments of the Internet of Things in Agriculture: A Survey. *IEEE Access* **2020**, *8*, 129924–129957. [CrossRef]
37. Davis, R.I.; Burns, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.* **2011**, *43*, 44. [CrossRef]

38. Abeni, L.; Goel, A.; Krasic, C.; Snow, L.; Walpole, J. A measurement-based analysis of the real-time performance of linux. In Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02), San Jose, CA, USA, 25–27 September 2002; pp. 133–142. [CrossRef]
39. Gleixner, T.; Niehaus, D. Hrtimers and beyond: Transforming the Linux time subsystems. In Proceedings of the Linux Symposium, Ottawa, ON, Canada, 19–22 July 2006; pp. 333–346.
40. High Resolution Timers. Available online: https://elinux.org/High_Resolution_Timers (accessed on 25 April 2021).
41. Simmonds, C. *Mastering Embedded Linux Programming*, 2nd ed.; Packt Publishing: Birmingham, UK, 2017.
42. Raspberry Pi and Real-Time Linux. Available online: www.osadl.org/Single-View.111+M5c03315dc57.0.html (accessed on 25 April 2021).
43. Adam, G.K.; Petrellis, N.; Garani, G.; Stylianos, T. COTS-Based Architectural Framework for Reliable Real-Time Control Applications in Manufacturing. *Appl. Sci.* **2020**, *10*, 3228. [CrossRef]
44. Adam, G.K. DALI LED Driver Control System for Lighting Operations Based on Raspberry Pi and Kernel Modules. *Electronics* **2019**, *8*, 1021. [CrossRef]