electronics

MDPI

Article

Performance Assessment of Linux Kernels with PREEMPT_RT on ARM-Based Embedded Devices

George K. Adam [1,*], Nikos Petrellis [2] and Lambros T. Doulos [3]

[1] CSLab Computer Systems Laboratory, Department of Digital Systems, University of Thessaly, 41500 Larisa, Greece
[2] Department of Electrical and Computer Engineering, University of Peloponnese, 26334 Patra, Greece; npetrellis@uop.gr
[3] School of Applied Arts, Hellenic Open University, 26335 Patra, Greece; ldoulos@mail.ntua.gr
* Correspondence: gadam@uth.gr; Tel.: +30-241-0684-596

Abstract: This work investigates the real-time performance of Linux kernels and distributions with a PREEMPT_RT real-time patch on ARM-based embedded devices. Experimental measurements, which are mainly based on heuristic methods, provide novel insights into Linux real-time performance on ARM-based embedded devices (e.g., BeagleBoard and RaspberryPi). Evaluations of the Linux real-time performance are based on specific real-time software measurement modules, developed for this purpose, and the use of a standard benchmark tool, cyclictest. Software modules were designed upon the introduction of a new response task model, an innovative aspect of this work. Measurements include the latency of response tasks at user and kernel space, the response on the execution of periodic tasks, the maximum sustained frequency and general latency performance metrics. The results show that in such systems the PREEMPT_RT patch provides more improved real-time performance than the default Linux kernels. The latencies and particularly the worst-case latencies are reduced with real-time support, thus making such devices running Linux with PREEMPT_RT more appropriate for use in time-sensitive embedded control systems and applications. Furthermore, the proposed performance measurements approach and evaluation methodology could be applied and deployed on other Linux-based real-time platforms.

Keywords: real-time operating systems; PREEMPT_RT; response latency; performance measurements; embedded devices

Citation: Adam, G.K.; Petrellis, N.; Doulos, L.T. Performance Assessment of Linux Kernels with PREEMPT_RT on ARM-Based Embedded Devices. *Electronics* **2021**, *10*, 1331. https://doi.org/10.3390/electronics10111331

Academic Editor: Juan M. Corchado

Received: 2 April 2021
Accepted: 30 May 2021
Published: 1 June 2021

1. Introduction

The standard Linux kernel initially was designed as a time-sharing system without taking time-determinism strictly into account. However, over the years, many of the improvements designed and developed by the PREEMPT_RT project (documentation is maintained on the Linux Foundation Wiki [1]) are now part of the mainline Linux kernel according to the Linutronix Co. [2]. In the past, approaches that have been considered in providing a Linux kernel with real-time capabilities, either improve the Linux kernel itself so that it provides bounded latencies for real-time applications (e.g., the PREEMPT_RT project) or add a layer below the Linux kernel (co-kernel approach) that handles all the real-time requirements separately (e.g., RTLinux, RTAI and Xenomai) [3–5]. Regarding latency issues, although RTAI and Xenomai offer lower latency than PREEMPT_RT, Linutronix tests show that the differences are not so significant, especially in real-world scenarios. Other approaches use specific CPU architectures like the ARM with a programmable real-time unit subsystem (PRU) [6], or combinations of GPU and FPGA solutions [7].

There is a growing tendency in the use of Linux in the domain of embedded systems for real-time control applications. The results of an Aspencore survey [8] indicate that

open-source operating systems like Linux and FreeRTOS continue to dominate, while other platforms are declining. Embedded systems are primarily used in real-world applications. Real-time embedded systems are employed by a wide variety of applications ranging from simple consumer electronics and home appliances to military weapons and space systems [9]. A VDC Research study [10] suggests that the fast growth of IoT (Internet of Things) is accelerating the move towards open-source Linux in embedded market share. Its open-source license, very good performance and ease of adaptation in various hardware systems, utilizing at the same time the multicore and high-frequency architecture of such devices, has placed a considerable interest in developing control applications based on such systems [11–14]. Additionally, the increasing requirements of real-time applications and the need to reduce development costs and time to market led to an increase in the interest for employing COTS (commercial off-the-shelf) hardware and software components in real-time domains, for example, commercially available embedded microcontrollers (by BeagleBoard, NXP Semiconductors, Texas Instruments, Qualcomm, Intel, etc.) [15–17]. However, despite the continuous increase in the utilization of COTS-based components, their reliable performance in real-time systems still remains under further research, which is an objective of this work too.

This research work investigates the Linux kernel real-time capabilities with a PREEMPT_RT patch in handling real-time tasks and operations at user and kernel space. For this purpose, it explores a variety of popular Linux kernels and distributions (Debian, Ubuntu, Arch Linux) running in ARM-based embedded platforms, such as Raspberry Pi (a Raspberry Pi3, referred to from now on as RPi3) and BeagleBoard microcontroller (a BeagleBone AI-Artificial Intelligence, referred to from now on as BBAI). The choice of ARM-based microcontrollers is because they are being extensively used for embedded low-powered control applications, due to their computing power in regard to their price, their ease of use with peripheral equipment and low-energy consumption [18,19]. Research shows that such systems are capable of supporting adequate timing for most measurement purposes [20–23]. Therefore, such ARM-based embedded devices seem to be appropriate multicore platforms for hosting a real-time system controller, so it was encouraging to install and test Linux kernels with the PREEMPT_RT patch on such platforms. Although the Linux kernel distributions for RPis and BBAIs do not currently 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 kernel performance with the real-time preemption patch. For the purposes of this research, specific software modules were developed and applied together with the cyclictest standard benchmark tool [24] to investigate and evaluate the real-time performance of Linux kernels patched with PREEMPT_RT. Virtual platforms such as the GEM5 simulator [25–27] and the QEMU emulator [28–30] could have also been used for simulating such ARM multicore architectures, albeit the QEMU emulator is not guaranteeing timing (cycle) accuracy [31]. There are no free cycle-accurate simulators available for recent ARM cores. Therefore, as the best solution, it was decided to use an ARM-based development board for Linux OS with PREEMPT_RT performance evaluation. In any case, in future work, we intend to use the QUEMU emulator to evaluate OS real-time performance on such embedded platforms.

In real-time systems, low latencies do ensure quicker response times, but the most important is to be deterministic. The PREEMPT_RT patch improves the Linux kernel itself by providing bounded latencies and predictability. This is an outcome of this research too. The experimental measurements do provide some evidence that a latency value of about 150 μs, as an upper bound, could be an acceptable safety margin for real-time embedded systems based on such devices and connected to various kinds of actuators, which require guaranteed response times below this threshold value.

This research work contributes on providing new experimental results on real-time performance and latency metrics for Linux kernels patched with PREEMPT_RT, running on such embedded devices (RPi3 and BBAI). A new response task model is introduced upon which novel software real-time measurement modules were designed. Particular effort was placed on measuring the throughput time delay of response and periodic tasks' execution at user and kernel space. The measurements include the maximum sustained frequency, the response latency of user and kernel tasks and general latency performance metrics using the cyclictest benchmark.

Some of the key features and novel contributions of this research work are the following: (1) provides latency measurements based on specific software real-time measurement modules, designed and developed upon the introduction of a new response task model and the use of cyclictest standard benchmark, (2) reveals novel insights in Linux real-time performance on ARM-based development platforms (BeagleBoard and Raspberry Pi), based on a comparative evaluation of real-time latency measurements at kernels with and without real-time support and (3) presents a measurements approach and evaluation methodology potentially applicable to other Linux kernels and distributions on such ARM-based embedded devices.

This paper is structured as follows: Section 2 describes previous related work and a brief discussion; Section 3 presents some of the background information and terminology needed to understand real-time systems; Section 4 presents the methodology and the performance measurements objectives aimed to be achieved; Section 5 describes the software modules developed for this purpose; Section 6 presents the experimental platform used as the test bed for the evaluation measurements performed; Section 7 discusses the experimentations carried out, presents, respectively, the experimental results of the response and periodic tasks execution outcomes at user and kernel space, the performance measurements results using the cyclictest benchmark and a brief discussion analysis on the results of the experimental research and the evaluation of Linux real-time functionality with the PREEMPT_RT patch; Section 8 provides a summary of the research outcomes and draws conclusions.

2. Related Work

In the case of real-time systems, their performance is analyzed by many different approaches depending on the nature of the applications and other factors [32–35]. The techniques and tools used depend on the aspects of performance targeted, most commonly schedulability issues in real-time systems [36–41]. In Linux, latency issues, for example, interrupt latency and scheduling latency, typically are investigated and measured with benchmark tools, for example, the cyclictest benchmark. Cyclictest is usually used in scheduling latency measurements in Linux and its principal real-time variant, the PREEMPT_RT patch [42,43]. Nonetheless, latency tracing tools are also being applied [44,45]. Other works use and apply new benchmarks or test modules to investigate such performance metrics and provide comparisons of different operating systems with real-time capabilities [46,47].

In this research work, a combination of software test modules, developed particularly for latency performance measurements, together with cyclictest standard benchmark tool, is the approach followed in the investigation of real-time latency issues in Linux kernels patched with PREEMPT_RT. Regarding latency measurements, the work of Brown and Martin [48] inspired this work. They compare the performance of Linux kernels with real-time support such as Xenomai and the PREEMPT_RT patch (2.6.33.7-rt29), 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.

Performance evaluation of different kernel versions with real-time support has been presented in many cases, but primarily on an x86 platform. In the work of Litayem and Saoud [49], 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. [50], 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 [51], a comparison of scheduling latency in Linux, PREEMPT_RT and LITMUS RT [52] 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. This ongoing research shows that the Linux PREEMPT_RT competes with the tested commercial RTOSs [53]. Lately, some investigation studies have appeared concerning measurements of latency on Raspbian Linux (version of Debian) with real-time patch PREEMPT_RT vs. the default Raspbian [54–59]. However, measurements are performed only with Raspbian Linux and the cyclictest benchmark. Xenomai has also been used to provide hard real-time support to user space control applications [60,61]. An interesting performance analysis research work is conducted by Delgado et al. in [62] where the authors present an open-source EtherCAT Master implemented on an embedded board using a dual-kernel approach with the latest versions of Xenomai and embedded Linux.

This research is different from these works with respect to the multiple platforms and the variety of Linux kernels and distributions upon which it is executed. Several studies show that such Linux-based systems continue to gain more popularity and play an increasing role in the embedded systems real-time control field [63–65] and today's Internet of Things applications [66,67]. Nevertheless, there is no sufficient research work in the evaluation of the real-time performance of RPi's and BBAI's Linux kernels patched with PREEMPT_RT. This research work comes to add to this empirical knowledge of latency measurements and evaluation of real-time execution efficiency in such platforms having Linux kernels patched with PREEMPT_RT.

3. Background: Real-Time Approaches and Terminology

In this section, we introduce the basic background behind our real-time performance measurements and analysis framework.

3.1. Real-Time Operating System

Systems are referred to as real-time when their correct behavior depends not only on the operations they perform being logically correct but also on the time at which they are performed [63]. Within a real-time system, each real-time task must complete its work before a deadline.

A real-time operating system is an operating system intended to serve real-time applications which have well-defined fixed timing constraints and require timely responses. Real-time operating systems are designed to run applications with very precise timing, keeping the amount of error in the timing of a task over subsequent iterations of a program or loop, between acceptable limits [68]. Guaranteeing real-time performance requires the use of efficient scheduling policies or algorithms. A scheduling algorithm, among other features, assigns a priority to each task and defines how tasks are processed by the operating system. A scheduling algorithm for a real-time system ensures that each real-time task will always meet its deadlines.

A real-time operating system among other features has a deterministic timing behavior and preemption capabilities. Deterministic timing behavior ensures time deterministically bounds on task scheduling, interrupts response latency and random latency or jitter. Jitter is the delay variation between minimum and maximum response time. In this way, real-time operating systems provide the required determinism needed by a real-time application in order to be scheduled and executed in time. Lower latencies contribute substantially to this direction. That means efficient and low latency interrupt handling, where

a higher priority task can preempt a lower priority task. Such a feature is essential particularly in hard real-time control systems, where the response to real events, triggered by sensors, is critical and requires on-time control and response by some actuators [69,70]. Processing must be done within the defined timing constraints or the system will fail. Therefore, the goal of a hard real-time system is to ensure that all deadlines are met. On the other hand, soft real-time systems do not always guarantee that will meet a deadline and a critical real-time task will complete on time.

A real-time operating system ensures priority inheritance. Priority inheritance enables a higher priority task, which is awaiting for a lock (mutex) held by a low priority task, to wake up more quickly and continue execution, since the low priority task inherits its priority and shortens its execution time. In Linux, since kernel version 2.6.18, mutexes support priority inheritance.

3.2. Real-Time Approaches in Linux

Linux has been developed as a general-purpose operating system. The Linux kernel is a low-latency preemptible kernel by default, capable of satisfying soft real-time requirements. Preemption at the kernel level is a necessity in order to consider real-time Linux at any level. Towards this goal, several approaches have been proposed that introduce actual real-time capabilities in the kernel. Among these, is the PREEMPT_RT patch, capable of minimizing both operating system overheads and latencies by directly modifying the existent kernel code. According to Reghenzani et al. [53], the real-time approaches alternative to the kernel based on the PREEMPT_RT patch are cokernel and single kernel approaches. A key feature of cokernel approaches is that a second kernel is dedicated to the management of the real-time applications. This additional kernel is working as a layer between the hardware and the general-purpose Linux kernel. This layer handles all the real-time requirements. The most common open-source cokernel approaches are RTLinux, Xenomai and RTAI.

The PREEMPT_RT patch of the Linux kernel follows a single kernel approach that aims to improve the Linux kernel itself by providing bounded latencies and predictability. The PREEMPT_RT patch enables the kernel to be interrupted while executing a system call, in order to service a higher-priority task. The major goal of the PREEMPT_RT patch is to increase the degree of kernel code preemption towards a fully preemptible kernel (PREEMPT_RT_FULL). This preemption level allows the real-time tasks to preempt the kernel everywhere, even in critical sections. However, some regions are still non-preemptible, like the top half of interrupt handlers and the critical regions protected by raw spinlocks (raw_spinlock_t). All the sleeping mutexes have been replaced with rt_mutex type that implements priority inheritance.

3.3. Latency Measurements Approaches—Techniques

Real-time measurements are taken based on the performance of real-time tasks generated according to the response and periodic task models. In Linux environments, a task is synonymous with a thread. Therefore, each task is scheduled as a thread, with real-time SCHED_FIFO policy and high priority. Thus, a higher priority task could block (preempt) and temporarily suspend the execution of lower priority tasks (preemptive scheduling). Although a higher priority task would not be delayed by lower priority tasks, however, it may rarely have to wait within uninterruptible sections of execution. Ideally, each thread could be given dedicated resources such as CPU and memory. Indeed, for all real-time tasks current and future memory allocations are locked to prevent the page out of memory, which improves the determinism. However, processor affinity was intentionally not set, although each real-time task could easily be assigned to run in a different core to reduce as much as possible the intercore interferences (interference delay time). Therefore, tasks are allowed to migrate among all CPU cores (migration overheads). Thus, the CPU is shared by multiple real-time tasks. Overall, the above may introduce some additional overheads to the total response latency.

4. Methodology

A primary goal of this research is to measure the real-time responses of Linux kernels and distributions in ARM-based embedded systems and platforms with the real-time preemption patch PREEMPT_RT. For this purpose, specific measurement algorithms were developed and implemented as C threaded software modules. These modules enable the evaluation of the latency occurring under real-time tasks execution, under idle and load conditions, at the Linux kernel distributions with real-time support and the default ones. The measurements platform is based upon a master-slave schema, in which the slave devices (RPi3 and BBAI) under test are connected to and communicate with a Raspberry Pi3 (master device) that performs the actual measurements.

The steps of the research process involved the design of the response and periodic algorithms and models, upon which experimental tests were specified to run specific real-time measurement tasks. The measurements obtained were validated with an oscilloscope and compared to cyclictest benchmark results for verification purposes. Finally, an analysis of the outcomes was performed (Figure 1).

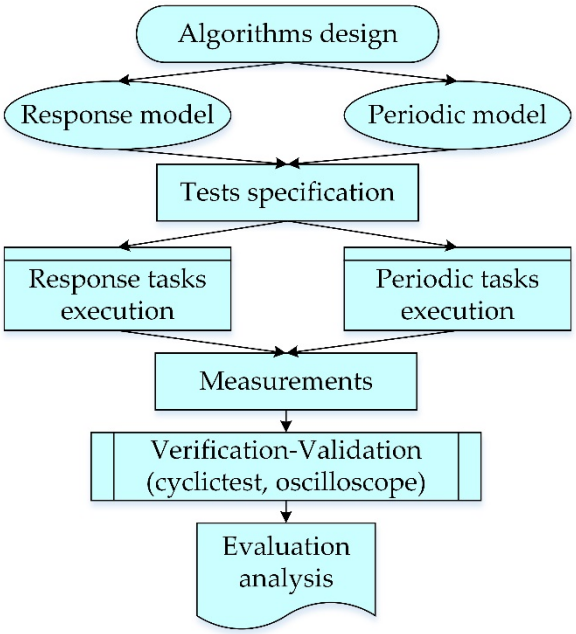

Figure 1. Research process steps.

4.1. Design Approach

According to Davis and Burns [68], the overwhelming majority of the research in multiprocessor real-time scheduling focuses on two simple task models: the periodic task model and the sporadic task model. 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 adopt a slightly different approach. This is partially based on the periodic task model and the introduction of a response task model. A job is one unit of work carried out by a single thread. The tasks that make up a real-time job are implemented based on two task models: the response and the periodic task model. In the periodic task model, each invocation of a task arrives strictly periodically, separated by a fixed time interval. In the 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, a maximum (or worst case) response latency $wcrl_i$ and a 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 worst-case response latency is a typical metric of the determinism of a real-time task since most of the real-time applications require upper-bounded response times. The worst-case response latency may be computed using various methods or by simple measurement programs. An average value of the response latency $wcrl_{avg}$ for a number of runs (*n* iterations) could be calculated using the following equation:

$$wcrl_{avg} = \frac{\sum_n^i (t_{run} - t_{arr})}{n}, \quad (1)$$

where t_{arr} is the task's arrival time and t_{run} is the task's run time.

The response latency includes: the interrupt latency t_{int}, that is, the time it takes for the interrupt to appear (indicated as latency); the interrupt service time t_{sv} (indicated as service time), the task scheduling latency t_{sc}, that is the time it takes for the scheduler to run (indicated as latency) and the task scheduling duration time t_{scd}, as shown in Figure 2a. All of these constitute the kernel's latency. In addition, some extra time is needed for the running task to produce results. Therefore, the total response latency is defined as the overall time t_{all} elapsed between a timer interrupt occurring at the master device (master triggers output), and the moment the corresponding awaiting user-space task in the slave device is switched to a running state producing results (enables GPIO output) (GPIO, General Purpose Input/Output). In the periodic task model, each task runs for a specific time interval time, semi-period t_{sm}, based on an internal timer. The actual time elapsed for a whole period (t_{pd}) is the timer period (Figure 2b).

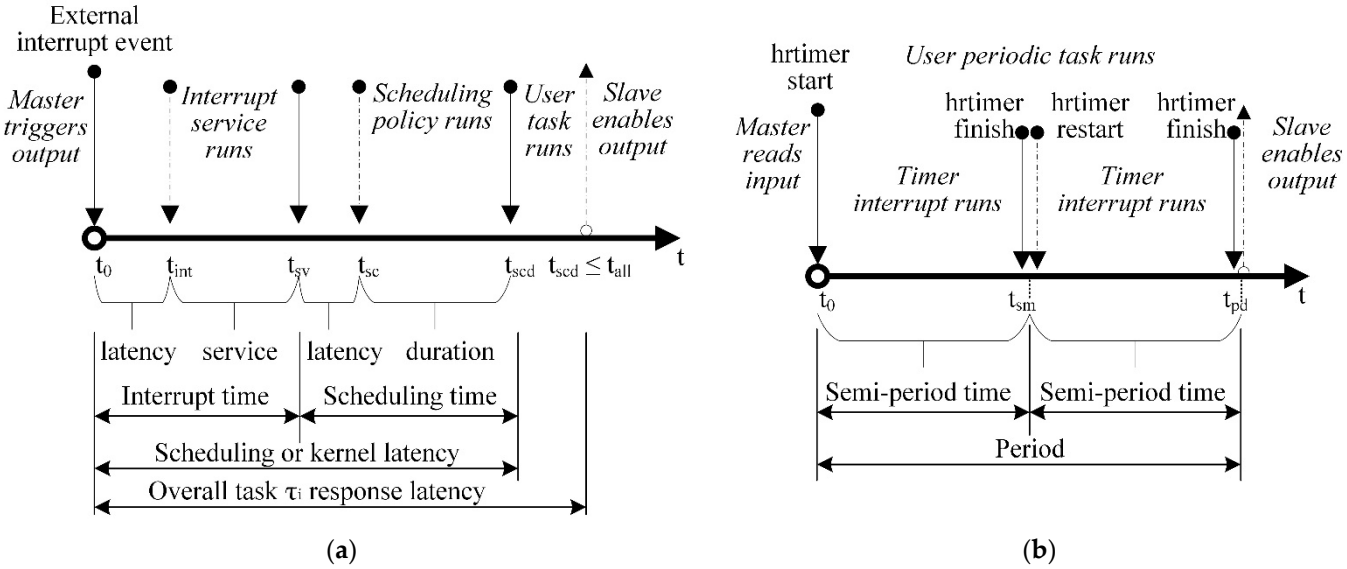

Figure 2. Response and periodic tasks: (**a**) overall response task latency; (**b**) periodic task time interval.

The algorithms developed implement the above task models as the measurement software modules designed in a modular mode. Within each module, each task executes the main measurements loop based on timing data acquired from the device under test and outputs the results. The measurements include the following:

- The maximum sustained interrupt frequency or frequency limits-stimuli, that is, estimate the optimum value for the time interval (t_{int}) between the generated interrupts, that the system can handle efficiently.
- The response latency (including worst-case latency, $wcrl_i$), that is, estimate the time elapsed from a GPIO input level change (IRQ, Interrupt Request) till the response change of a GPIO output, at user and kernel space.

- In response tasks, measure the time elapsed (total latency, t_{all}) until the (slave) device under test responds.
- In periodic tasks, measure whether the slave device responds at proper time periods (t_{pd}) and produces timer interrupts at exact time intervals according to specified frequencies.
- General performance measurements and particularly latency measurements using the cyclictest benchmark tool.

4.2. Measurements Approach

The measurements software was designed as threaded modules in C. These modules perform the measurements under two modes, at user and kernel space, of the response and periodic real-time tasks. Some of the underlying principles that govern the design methodology involve requirements, such as the running tasks to be scheduled as threads, with real-time SCHED_FIFO policy and high priority (99). Since in PREEMPT_RT patched Linux all interrupt handlers are switched to threaded (schedulable) interrupts, this feature adds to its real-time throughput performance. This is because threads can block while usual interrupt handlers cannot. Real-time application development with PREEMPT_RT requires the POSIX real-time API (e.g., the pthread library) which is part of the standard C library. Response and periodic tasks are passed as argument-functions to POSIX threads creation function calls. From a scheduling point of view, it makes no difference between the initial thread of a process, for example, executing the main() function and all additional threads created dynamically. Although real-time application performance is not just a matter of using a specific scheduling policy, on any system, all user-space processes are scheduled with real-time scheduling class SCHED_FIFO and high priorities. That aims to ensure timely execution of the tasks and decreased execution times and latencies. SCHED_FIFO policy allows multiple tasks with the same priority to be scheduled in order with respect to the time at which they were enqueued into the ready queue. Scheduling policy, attributes and priorities were also set per thread upon their creation with POSIX thread scheduling policy functions calls. Threads must voluntarily yield the processor for other threads, so the SCHED_FIFO scheduling policy is used. The above approach of the real-time software design is shown in Figure 3.

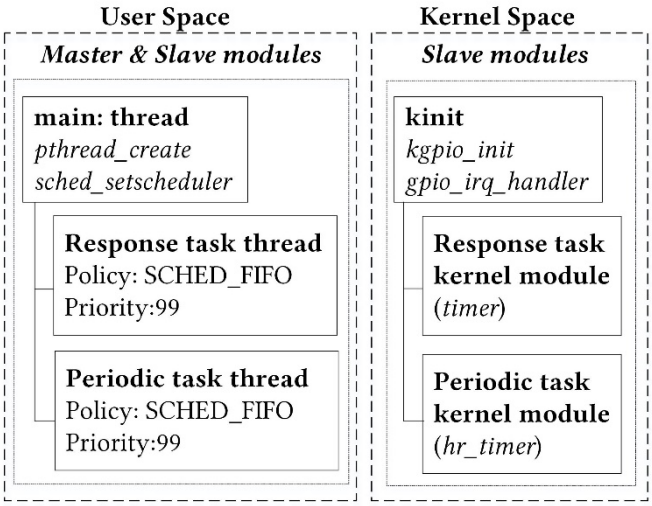

Figure 3. Measurement modules real-time design infrastructure.

Processor affinity was intentionally not set, although each real-time task could easily be assigned to run in a different core to reduce as much as possible the intercore interferences. Such a case, having more than one thread per core, may lead the performance to be somewhat unpredictable due to potential locks. However, this is not valid, due to the fact

that the amount of threads generated is very small, just a main process with one thread. Even with two threads of the same priority, the average thread switching latency in Linux with PREEMPT_RT is negligible, as shown by the work of Fayyad-Kazan et al. [71]. Therefore, tasks are allowed to migrate among all ARM CPU cores. Overall, the above may introduce some additional overheads to the scheduling operations that affect scheduling latencies. For this reason, they are also taken into consideration in total response latency.

Precise timing and reliable latency performance metrics and throughput evaluation of PREEMPT_RT patched Linux kernel require accurate timing source, for example, the system timer in RPi3, part of the Graphical Processing Unit (GPU). The execution time of the software modules depends on the processors' cores' clock frequency, which is variable. For example, the hardware 1 MHz system timer on the RPi3 is a dedicated timer that runs independently from the processor. The system clock on the RPi3 has 1 μs resolution accuracy. For this reason, in the master software modules that implement the performance measurements, the system call clock_gettime() (defined in POSIX timers implementation) is used for measuring time, with the highest possible resolution, and the clock id is set to CLOCK_MONOTONIC. The execution of this function in RPi3 goes down to the BCM2711 driver to get the time values from the system timer registers. This function gives results with nanosecond resolution and requires less than a microsecond to execute. Therefore, for time benchmarking purposes at such resolution, this is sufficient. The actual time measuring adds almost no overhead since it is in the order of a few tens of nanoseconds. The same negligible overhead is observed by Garre et al. [72] where they conclude that the time difference is below 100 nsecs.

Real-time performance metrics were additionally investigated using the cyclictest benchmark, which is part of the rt-tests repository maintained by Linux kernel developers. The benchmark was used to investigate further and conclude on the response latency of the devices under test. All the results obtained by running the cyclictest benchmark were later on compared to those obtained by the software modules. An oscilloscope was used to directly measure the output voltages, triggering times and frequencies of the signals on slave device's GPIOs, for validation purposes. System tracing tools could have also been used for some of the measurements, to investigate and analyze kernel latencies and performance execution issues. However, intentionally, it was decided that the development of specific algorithms and software modules would be dedicated to the specific measurements of response and periodic tasks at user and kernel space. These modules take into consideration specific critical real-time requirements, for example, high priorities, locks of memory pages and high-resolution timers.

4.3. Response Task Measurements

4.3.1. User Space Measurements

The master RPi3 device performs the measurements. The slave devices were tested with and without PREEMPT_RT-patched Linux kernels. The master device runs at specific time intervals a task τ_i that triggers a GPIO input at the slave device (external interrupt event), which the slave is polling in an infinite loop. Each running task τ_i consists of two subtasks, loops of "1 s" and "0 s". The master toggles its output value between 0 and 1 (loops of "0 s" and "1 s") and begins to measure the slave's response delay and perform relative measurement metrics. The number of iterations is multiplied with each specific task execution time t_i defines the total duration of the execution. The slave device, upon reading the change of the input state, sets its output accordingly (on a rising edge sets its output line, while on a falling edge clears its output line). Then, the master device repeats the loop for a number of cycles (100 K to 1 M) for sufficient measurements to be collected for analysis. The average duration of each sample cycle was found to be at about a few tens of microseconds, as it is documented further on. Measurements are performed on both edges of the trigger signals (rising and falling). For each task τ_i, execution time t_i is derived as the sum of the response latency times, that is t_{lat1} for loop of "1 s" and t_{lat0} for

loop of "0 s", plus the time interval t_{irv} in-between the generated subtasks (interrupts). The above is illustrated in Figure 4.

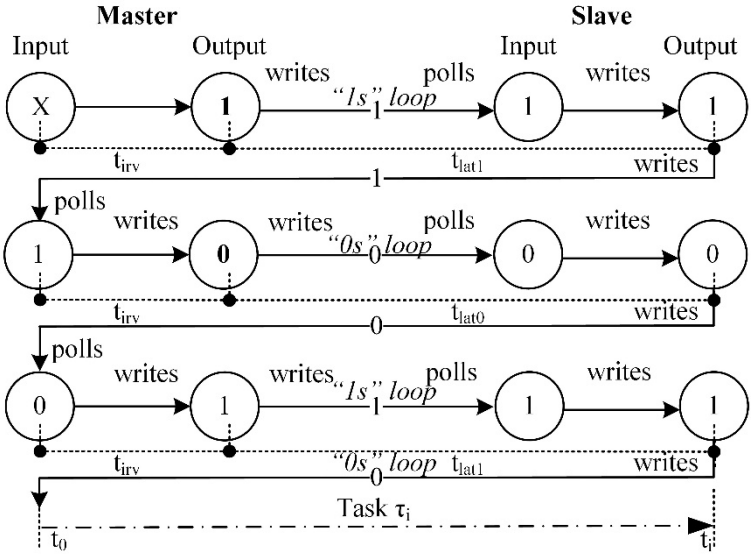

Figure 4. Master-slave inputs-outputs cycles of "1 s" and "0 s".

Response latency (responsiveness) is one of the substantial measurements the experimental tests aim to investigate. This is defined as the time from when the master device stimulates an output GPIO pin (master module) until the time the slave device handles the incoming interrupt event at its input GPIO pin and sets its output GPIO pin accordingly (slave module). The user tasks are scheduled as soon as the interrupt handler returns. Since with PREEMPT_RT (CONFIG_PREEMPT_RT_FULL) virtually all kernel code can be involuntarily preempted at any time. When a process becomes runnable, there is no more need to wait for kernel code (typically a system call) to return before running the scheduler (spinlocks are replaced by real-time mutexes, sleeping spinlocks). So, when an interrupt comes while the task is executing a system call, there is no need to finish this system call before another task can be scheduled.

During the experiments, the master device toggles a GPIO pin defined as an output repeatedly (external interrupt event), with the stimulus time set at 10 ms. At the end of each measurement cycle, it estimates the mean, minimum and maximum response latency of the slave device. The slave device was tested continuously with cycles of 1 million (1M) interrupts for each task loop of "1 s" and "0 s". The average running time of 1M samples cycle for each loop on both the devices under test was about 180 min. The average duration of each sample cycle was about 120 μs. This is the average time it takes the level of the input control signal to raise to 3.3 V or fall to 0 V (accordingly for "1 s" and "0 s"). The tests were executed continuously for several hours with cycles of 1 million (1M) interrupts for each task loop of "1 s" and "0 s", to obtain a sufficient number of measurements and extract reliable values. Statistical computations were performed at the end of the measurements.

The experimental software measurements in user space show that an additional delay influences in some cases the maximum values of response latency. This is due to the time required by the master software module to execute the measurements. Thus, there is a slight delay of a few microseconds between the activating event and the instant output when the task starts executing. This is acceptable, since there is always a delay between occurrence and completion of an event, as long as this delay does not exceed a maximum value, specified by the timing requirements of the real-time application. In our case, this minor delay does not affect the task's response times and does not impose a lower bound on the deadlines that can be supported by the system. However, in determining whether

a system can ensure the required output in a timely manner, it is appropriate to take into account any additional time required by a task to produce output. In a real-time system where the interaction with the external world has to be within predictable and acceptable working limits, the overall response delay is an important factor that is always being considered.

Both kernels were tested under load too. For this purpose, a custom script was implemented, running with high priority, in the slave device under test. The custom script uses rt-tests hackbench to generate synthetic workload by simply copying SD card contents to/dev/null, to nothing. That is about 8.1 GB and takes about 7 min (423 s) at 19.0 MB/s.

4.3.2. Kernel Space Measurements

At kernel space, the master device performs the measurements in a similar way to the user space experimentations. However, in this case, the slaves' software control module is based on a kernel module. The master device once again at specific time intervals triggers a GPIO input at the slave device, which the slave is polling in an infinite loop. The slave device, upon reading the change of the input state, sets its output accordingly, based on a kernel module developed for this purpose and inserted in the Linux kernel. This module uses an interrupt handler function (only the top-half) to service the input change and sets the output accordingly. Then, the master device again repeats the loop and performs the same measurements for a number of cycles (100 K to 1 M). The overall tests were executed repeatedly for a few hours.

4.3.3. Measurements Validation

Software modules provide consistent and reliable results based on experimental iterations. In order to validate further the results obtained by the software modules, the measurements taken internally are compared to those measured externally with an oscilloscope. In particular, the slaves' response delay is measured as the time between the rising edges of the incoming and the outgoing signals at the slave device. Their properties such as time intervals and frequencies were analyzed over time, in order to validate the latencies. The probes were attached in bare metal to the master's and slave's GPIO outputs. In this way, when the master triggers its output, the time delay until the slave enables its output is measured directly by the oscilloscope.

4.4. Periodic Task Measurements

Periodic task measurements aim to verify that the slave device responds at proper periods and at the same time to investigate the state in which the slave device cannot react properly.

4.4.1. User Space Measurements

The slave device at a specific periodic rate, based on an internal timer generates a periodic square wave that toggles periodically the value of an output configured pin. The master device that performs the measurements polls an input that connects to the slave's output in an infinite loop. Once its state is changed (rising edge of the first interrupt), it begins to count the time until it is changed again (falling edge of the second interrupt). Therefore, measurements are performed on both edges. In this way, on every even timer interrupt at the slave device, the master device measures the actual semi-period between both edges.

4.4.2. Kernel Space Measurements

The experimental setup and layout of the devices are 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 slaves' control software is a kernel module that uses

an internal high-resolution timer to produce the periodic interrupts. The slave device at a specific periodic rate, based on an internal high-resolution timer, toggles the value of an output pin periodically. The master device polls an input that corresponds to the slave's output, in an infinite loop. Once its state is changed (rising edge), it begins to count the time until it is changed again (falling edge). In other words, measurements are performed again on both edges. In this way, on every even timer interrupt at the slave device, the master device measures the actual semi-period between both edges. This experiment is to verify that the slave device responds at proper periods and at the same time to investigate the state in which the slave device cannot react properly.

4.5. Cyclictest Measurements

The real-time performance of Linux kernels with PREEMPT_RT support running in ARM-based devices was additionally investigated using the cyclictest standard benchmark. This benchmark measures the time from the occurred event (e.g., an interrupt) to the start of real work. It provides a mechanism to measure the latency of the processor for a number of times defined by the user. For this purpose, it creates a number of threads (one per core) that repeatedly checks in a loop how much time the processor takes to respond during a period of time. On each cycle, the actual time is determined, the maximum difference in expected versus actual time is calculated and various other statistics are collected.

5. Measurements Software

The experimental tests run for a long duration to evaluate the latency occurring in real-time tasks execution, under idle and load conditions, at the Linux kernel distributions with real-time support (patched with PREEMPT_RT) and the default ones. Measurements include the throughput time delay of response and periodic task execution at user and kernel space. The results of the experiments performed are reproducible. The experimental software modules are available as an open-source project at GitHub https://github.com/gadam2018/RPi-BeagleBone (accessed on 12 February 2021) [73]. The same evaluation modules and performance measurement methodology could be applied to other Linux-based systems and platforms.

5.1. Response Task Measurement Algorithm

In user space measurements, the software module in the master device performs the overall control of execution and metrics measurements. The same master module is applied for kernel space measurements. This module triggers the slave device at specific and random time intervals (e.g., 1 ms to 10 ms) in a loop for a number of iterations (e.g., 100 K to 1 M) and measures the time elapsed (latency) until the slave device under test responds. 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 measurements, 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. For real-time critical interrupts, top-half interrupt handlers are preferred as they are started by the CPU as soon as interrupts are enabled. On the other hand, a bottom-half starts after all pending top-halves have completed their execution. In the PREEMPT_RT patch, this is accomplished by forcing bottom half processing to take place in kernel threads, which can be scheduled such that they do not delay high-priority real-time tasks.

Both the default Linux kernels and the ones with the PREEMPT_RT patch were tested under normal and load conditions with a high priority load test script. In real-time patched Linux kernels the modules run with low latency delays and low signal jitter (pe-

riod irregularity), despite the fact that the multicore processor was under load. The kernels with the PREEMPT_RT patch were capable to maintain low latencies despite the increased load.

C implementations use the kernel's sysfs interface, although they require calls into the kernel to change GPIO states and these require costly context switches. The basic functionality of these modules is shown in Figure 5 and as pseudocode in Appendix A (Algorithm A1). The master response module performs initializations, sets the scheduling policy, the events to poll and triggers the device under test (writes GPIO output, gets clock time and polls input). In user space, the slave software polls the input and writes accordingly the output, while in kernel space, the slave software (as a kernel module) services the interrupt by getting the input value and setting the output. Once the desired number of loops is reached, the master module performs metrics calculations and outputs the results.

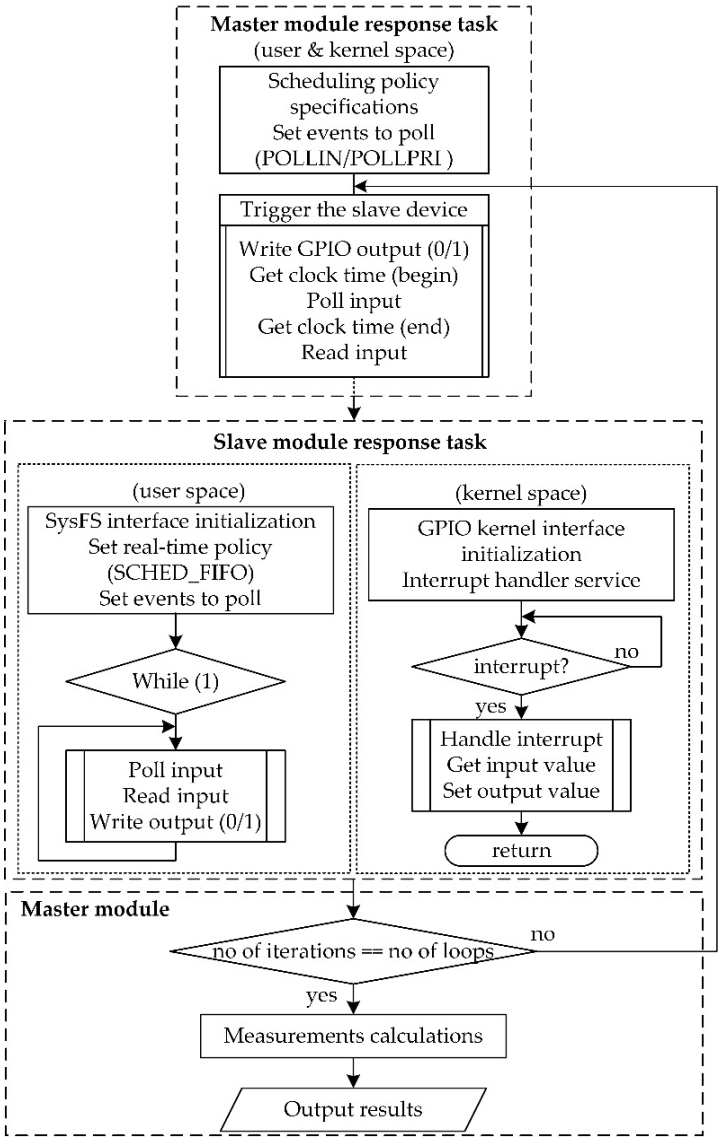

Figure 5. User and kernel space response task measurement algorithm.

5.2. Periodic Task Measurement Algorithm

In the master device, the control software monitors whether the slave device under test responds in proper periods. In particular, whether it produces timer interrupts at exact time intervals. The same master module is applied for kernel space measurements. In user space measurements, the control software in the slave device toggles the value (0, 1) of an output pin, at specific time intervals, based on an internal timer (e.g., with a period of 30,000 μs and decreasing values). In the master device, the control software performs the measurements of the actual time elapsed (timer period). In kernel space measurements, the control software in the slave is inserted into the slave's kernel as a kernel module. The difference with the slave's response task module at kernel space is that this module uses an internal high-resolution timer. A high-resolution timer is usually a requirement in real-time systems when a task needs to occur more frequently than the 1 millisecond resolution offered with Linux.

Both the default Linux kernels and the ones with the PREEMPT_RT patch were tested under load conditions. The results show that despite the stress load, the kernels with the PREEMPT_RT patch produce timer interrupts at exact time intervals.

At the master device, this periodic task module is similar to the master's response module at user and kernel space. A substantial difference is that this module does not produce any triggering output but reads the input for the interrupts occurred and measures the time interval in between (half period).

At the slave device, in user space, this periodic task module is similar to the slave's response module at user space. The only difference is in the thread's function structure. It uses a high-resolution timer to produce timer-based interrupts. In kernel space, it is similar to the slave's response kernel module at kernel space, but with the difference that it uses an internal high-resolution timer (linux/hrtimer.h). The basic functionality of these modules is shown in Figure 6 and as pseudocode in Appendix A (Algorithm A2). The master periodic module performs initializations, sets the scheduling policy, the events to poll and reads the device under test (reads GPIO input, gets clock time and polls input). In user space, the slave software reads the timer until the time interval is elapsed and writes the output. In kernel space, the slave software (as a kernel module) in periodic mode starts the high-resolution timer, services the interrupt and returns. Once again, the master module performs metrics calculations and outputs the results.

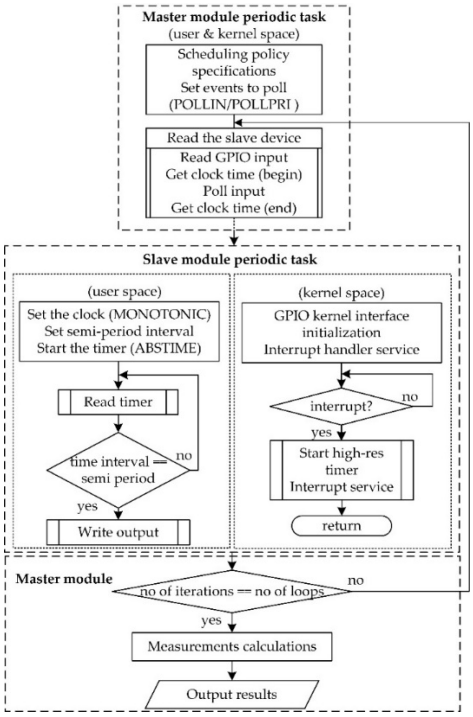

Figure 6. User and kernel space periodic task measurement algorithm.

6. Experimental Platform

A master-slave schema was applied, in which the slave devices (RPi3 and BBAI) under test were connected to and communicating with a Raspberry Pi3 (master device) that performed the actual measurements. The experimental setup is shown in Figure 7.

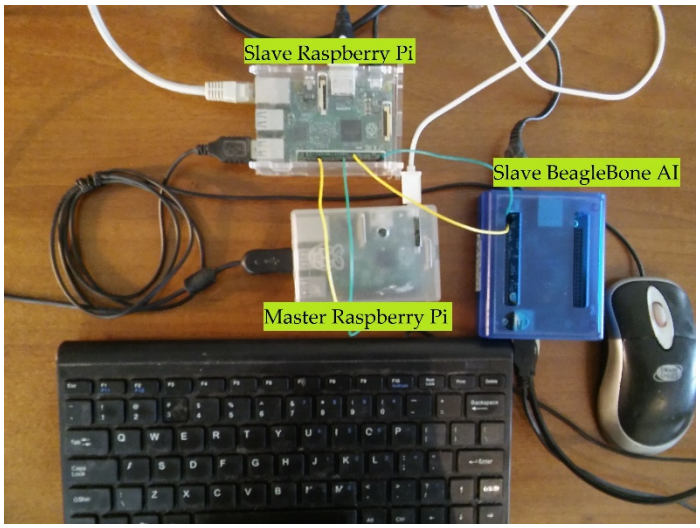

Figure 7. The system hardware experimental setup: RPi3 to RPi3 and RPi3 to BBAI.

In the majority of the experiments, the stimulus time (the time interval between the generated interrupts) was set at an optimal value of 10 ms and below for testing and sensitivity analysis purposes. The basic hardware components of the experimental test system include an RPi3 that connects in a master-slave schema to another RPi3 and BBAI, having bidirectional communication. There are a number of GNU/Linux distributions such as Debian, Ubuntu and Arch Linux running in ARM-based platforms as the RPi3 and BBAI. The experimental tests carried out were based on the communication between these devices. The devices were connected through GPIOs in a master-slave schema, as illustrated in Figure 8. For RPi3, GPIO27 (pin 13) in the slave device was defined as input and connected to GPIO17 (pin 11) defined as output in the master device. For BBAI, GPIO_76 (P9, pin 15) was defined as input and GPIO_209 (pin 17) as output. The communication was bidirectional, so the same connection was applied in reverse from the slave devices to the master.

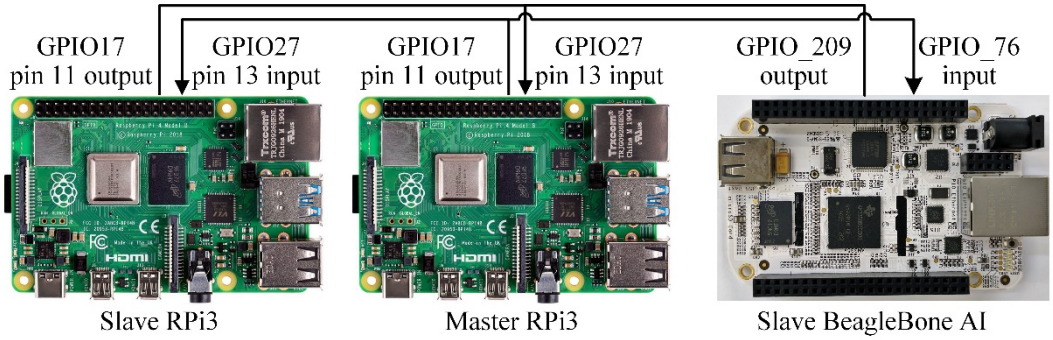

Figure 8. Schematic diagram of system connections: RPi3 to RPi3 and RPi3 to BBAI.

The RPi3 is a low-cost, low-power, portable and stand-alone single-board computer, which is being extensively used for embedded applications. The RPi3 has integrated an SoC based on Broadcom BCM2837, which includes an ARM Cortex-A53 quad-core processor running at 1200 MHz, and other chips on board supporting interface circuitry with

real-time peripherals (e.g., sensors and actuators). The CPU supports ARMv8-A architecture and is capable of supporting 32-bit and 64-bit instruction sets. Although, primarily, it is designed to function as a general processing computer, it shares many characteristics with an embedded system [74]. The BeagleBone AI fills the gap between small single board computers and more powerful industrial computers. The development board is based upon a Sitara AM5729 SoC from Texas Instruments having a dual ARM Cortex-A15 processor, which supports ARMv7-A architecture, running at 1 GHz up to 2.5 GHz, with 1 GB RAM and 16 GB eMMC on-board flash storage.

Both the default Linux kernels and with real-time support (patched with PREEMPT_RT) were tested. For this purpose, different kernel configurations (on microSD cards) were installed and configured on the slave devices under test. A PicoScope 3206 A oscilloscope was used to visualize and measure the latencies occurring at the slave devices under test. For validation purposes, these latency measurements were compared to the results of the experimental software measurements.

7. Results and Discussion

7.1. Estimation of Maximum Sustained Frequency

A number of tests were executed with variable frequency values to estimate an optimum value for the time interval between the generated interrupts (stimulus time at the master device). Indicative results for some of these tests, for example, for 1 million (1 M) interrupts, show that the slave devices with PREEMPT_RT can handle all the generated interrupts if the time interval in between is above 10 ms (GPIO toggle frequency greater than 10 kHz). Therefore, the safe maximum sustained frequency under which the system does not miss any of the generated interrupts would be appropriate if the time interval is set above 10 ms. For the majority of the experiments, the stimulus time was set at this optimal value of 10 ms and below for testing and sensitivity analysis purposes. That means that we could toggle the pin, for example, with a low frequency of 1000 Hz, for a period of several hours (1 M interrupts) and above and get reliable responses. It was also observed that for intervals less than 1 ms, the slave device under test could not always respond efficiently, since in some cases long delays (above 1 ms) begin to appear.

7.2. Periodic Task User Space Measurements

Measurements were performed for a variable number of time samplings starting at 10,000 and decreasing, with a semi-period at 15,000 μs (30,000 μs period) and decreasing. The results obtained show that the slave devices generate the timer interrupts at exact time intervals, both at PREEMPT_RT-patched Linux kernels and without real-time support. In other words, the master device measures the same semi-period length as the one produced by the slaves' internal timer. However, it was also observed that as the semi-period gets smaller and at the same time the quantity of samples increases, the slave devices cannot respond satisfactorily. For example, when the number of samples is increased above 10,000 with a semi-period less than 2000 μs, then longer delays start to appear.

The fully preempted kernels and without PREEMPT_RT support were also tested under load by using the same aforementioned custom script, running with a high priority set on 99, on the slave devices under test. The results show that the custom load makes almost no difference to the performance since the timer interrupts were measured to be at exact time intervals with minor variations.

7.3. Periodic Task Kernel Space Measurements

Measurements were performed again for the same quantity of samples starting at 10,000 and decreasing, with a semi-period at 15,000 μs (30,000 μs period) and decreasing. In general, it was observed that the slave devices under test produce the timer interrupts

at exact time intervals since the master device measures the same length of the semi-period. Furthermore, it was also noticed that as the semi-period gets smaller and the quantity of samples increases, the slave devices again cannot respond satisfactorily.

7.4. Latency Measurements with Cyclictest

Cyclictest options, such as the behavior of each thread, the priority, the number of loops, the thread base interval and the same priority for all threads, were tuned to match the required measurements. Cyclictest measurements were performed for all Linux kernels and distributions with and without the PREEMPT_RT patch, using the same execution run parameters, for example, with locked current and future process memory allocations and standard SMP option for equal priority across all threads (e.g., #cyclictest –l 500,000 *-n* –t 1 *–p* 99 –i 400 --smp). A representative sample output of the performance measured with this benchmark (e.g., for 500,000 loops) for kernel version 4.19.67-2 under a Debian distribution of Linux OS running on Raspberry Pi variants (used by many of the approaches that examine RPi's real-time properties with cyclictest) shows an average latency of about 10 μs, while the worst-case latency reached about 83 μs, as shown in Table 1.

The results show the latency statistics in microseconds for each core in the slave device. Avg represents the average latency being measured on the system, while Max represents the maximum latency detected on the system.

Further on, cyclictest measurements were executed using mlockall (additional option –m) to lock current and future process's memory allocations, in order to ensure reliable and fast response in time. Data automatically generated were collected for plotting a chart of latency data using the gnuplot utility. The result of each execution is a histogram of latencies, where the *x*-axis represents the measured latency delay (in μs) and the *y*-axis the absolute frequency of the corresponding value. An indicative outcome of the experiments is compared and depicted in Figure 9. Results at instant measurements show that for a small number of samples (e.g., 0,5 M–1 M) there are no particular differences.

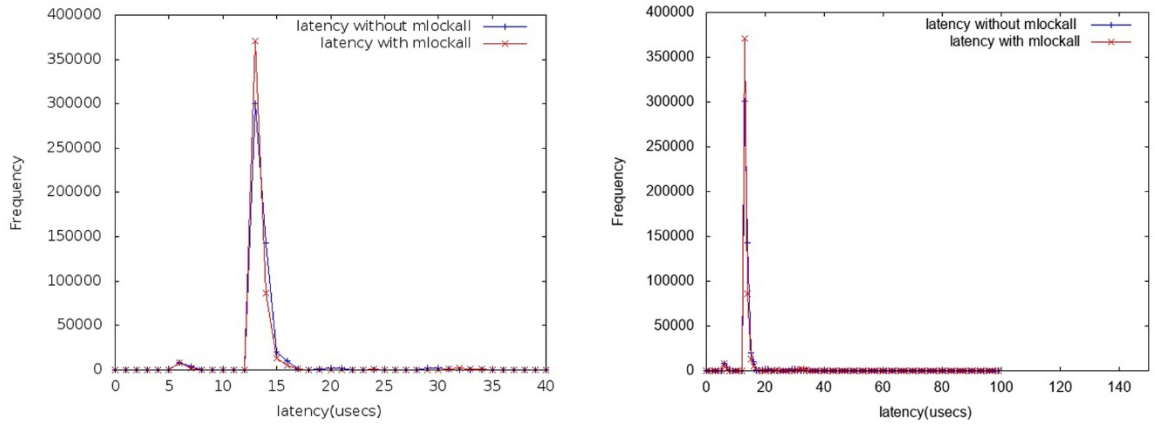

Figure 9. Cyclictest latency with and without mlockall: (**a**) latency normal distribution; (**b**) latency maximum.

Table 1. Cyclictest latency of Debian Linux kernel version 4.19.67-2 with the PREEMPT_RT patch.

Thread	Priority	Thread Base Interval (μs)	Number of Loops	Time (μs)			Max Difference
				Min	Actual	Avg	
T: 0 (1153)	99	400	500,000	6	6	7	72
T: 1 (1154)	99	900	24,298	11	16	15	74
T: 2 (1155)	99	1400	142,859	6	10	8	76
T: 3 (1156)	99	1900	105,264	6	16	10	83

The histogram has a normal distribution centered on 12 μs to 13 μs. The maximum latency observed with PREEMPT_RT support is quite low at 83 μs, with a considerable amount of samples below 50 μs.

The results obtained for all kernel versions and Linux distributions are summarized and examined further in Table 3. Nevertheless, they all reconfirm that the Linux kernels with the PREEMPT_RT patch can achieve sufficiently lower real-time responses, within acceptable limits, well suited in many cases of real-time applications. In the majority of cyclictest measurements, the minimum latency is below 50 μs and the maximum below 100 μs. This indicates that Linux kernels with real-time support provide better responses with lower latencies than the default kernels.

7.5. Latency Measurements

As examined earlier in the related work section, most of the approaches on measuring Linux OS real-time performance and particularly latency are based on x86 CPU architectures and the use of benchmark tools like cyclictest. There are a few approaches based on ARM architectures, and particularly Raspberry Pi platforms, with PREEMPT_RT patched kernel (under a Debian Linux version), however all of them are using only cyclictest standard benchmark for latency measurements. A summary of the results achieved compared to our approach is provided in Table 2.

Table 2. Cyclictest latency results comparison for Raspberry Pi with Linux kernels with PREEMPT_RT.

	Hardware (Raspberry Pi)	**Real-Time Kernel (Debian Version)**	**Cyclictest Latency (μs) (Min, Avg, Max)**
Our approach	RPi3 Model B 64-bit ARM Cortex-A53 quad core, 1200 MHz	4.4.16-rt17-v7+	<50, 53, 80
Molloy [20]	RPi2 Model B 32-bit ARM Cortex-A7 quad core, 900 MHz	3.18.16-rt13-v7+	9, 12, 98
EMLID [54]	RPi Model B+ 32-bit ARM1176JZFS, 700 MHz	3.18.7-rt1-v7+	12, 27, 77
Durr [55]	RPi Model B 32-bit ARM1176JZFS, 700 MHz	4.4.9-rt17-v7+	23, 37, 156
Benway [56]	RPi Model B+ 32-bit ARM1176JZFS, 700 MHz	4.4.9-rt17-v7+	20, 36, 105
Balci [57]	RPi3 Model B 64-bit ARM Cortex-A53 quad core, 1200 MHz	4.9.47-rt37-v7+	<50, <50, 91
Autostatic [58]	RPi3 Model B 64-bit ARM Cortex-A53 quad core, 1200 MHz	4.9.33-rt23-v7+	-, 40–70, 75–100
Riva [59]	RPi3 Model B 64-bit ARM Cortex-A53 quad core, 1200 MHz	4.14.27-rt21-v7+	-, 50–150, 147

Although the above cyclictest latency results of Linux kernels with the PREEMPT_RT are based only on Raspberry Pi, they do reconfirm that our measurements approach provides valid results, approximately within the same range (min: <50 μs, max: 150 μs).

Regarding latency measurements with specific software, very few works, like the work of Brown and Martin [48], proceed into the development of certain software measurement modules particularly for ARM-based Linux platforms patched with PREEMPT_RT. Table 3 provides a summary of the results obtained by this work compared to their results for Ubuntu Linux kernels with PREEMPT_RT. The intention is to reconfirm the results obtained with PREEMPT_RT rather than providing a fair comparison since the kernel versions are significantly different.

Table 3. Software modules latency results for both approaches under Ubuntu Linux kernels with PREEMPT_RT.

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	30.000	49, 147	50, 67
	4.14.74-rt44-v7	jitter = 0	90% of the latencies << 147	95% of the latencies << 67
BeagleBone AI 32-bit ARM Cortex-A15 1000 MHz (our platform approach)	Ubuntu, kernel	30.000	48, 142	46, 72
	4.14.74-rt44-v7	jitter = 0	90% of the latencies << 142	95% of the latencies << 72
BeagleBoard C4, OMAP3520 SoC, 32-bit ARM Cortex-A8, 720 MHz (Brown and Martin [48])	Ubuntu Lucid Linux, kernel	7.071	157 (max) for 95% of the time	43 (max) for 95% of the time
	2.6.33.7-rt29	jitter = 0	796 (max) for 100% of the time	336 (max) for 100% of the time

In RPi3 with the PREEMPT_RT patched kernel, the minimum latency is measured to be below 50 µs, both at user and kernel spaces. In user space, 90% of the latencies fall below the maximum of 140 µs, while in kernel space, 95% of the latencies fall below the maximum of 67 µs. In BeagleBone AI, again the minimum latency is measured to be below 50 µs, both at user and kernel spaces. In user space, 90% of the latencies fall below the maximum of 142 µs, while in kernel space, 95% of the latencies fall below the maximum of 72 µs. 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 10 illustrates the maximum and minimum response latencies and averages at user and kernel space for both approaches and kernels with PREEMPT_RT in all devices (Raspberry Pi3, BeagleBone AI and BeagleBoard C4).

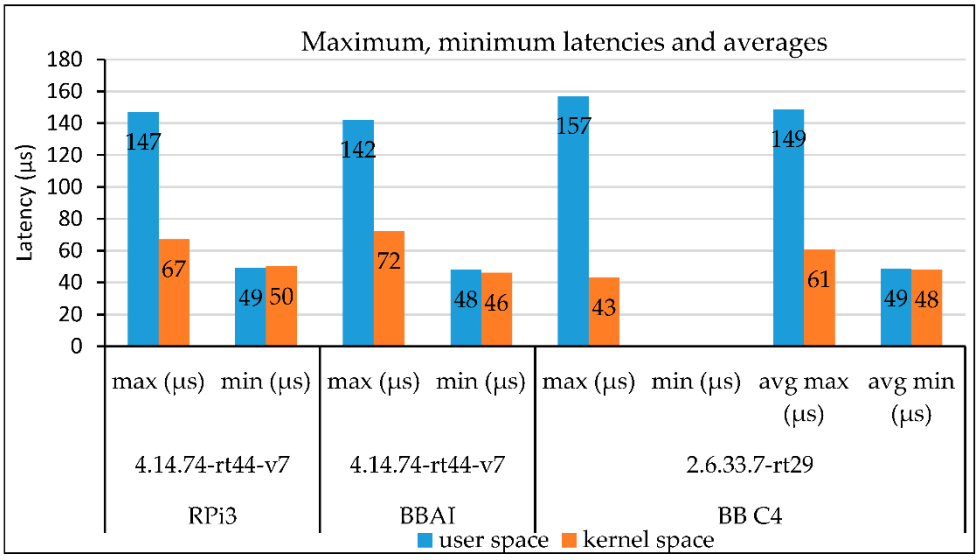

Figure 10. Maximum, minimum latencies and averages for preempted kernels at user and kernel space in all devices.

Both approaches use similar software modules for measurements and the communication structure of the devices. However, the hardware development platforms and Linux kernel versions are different. On the other hand, they are both ARM-based CPU architectures running among other versions of Linux, both Ubuntu too. Nevertheless, the results on the real-time performance with PREEMPT_RT are quite close. In all experimental test

platforms, the real-time patched Linux kernels produce exact time periods (jitter is zero) based on their internal timer.

7.6. Overall Response Latency Results

The measurements software runs on a master RPi3 that connects to and communicates with the slave devices (Raspberry Pi3 and BeagleBone AI) under test. Measurements include: the throughput time delay or response latency of response tasks execution at user and kernel space, the response time at specific periodic rates of periodic tasks execution at user and kernel space, the maximum sustained frequency and general latency performance metrics using the cyclictest benchmark. All experimental software measurements are validated with an oscilloscope.

A summary of the response latency results for Raspberry Pi3 and BeagleBone AI running Linux kernels with the PREEMPT_RT patch is provided in Table 4. Information on variance and standard deviations is also provided. Overall, latency results, and particularly maximum values on both the slave devices with preemption support, are lower. In the Linux kernels patched with PREEMPT_RT, the oscilloscope measurements reconfirm the results produced with the software measurements modules. Cyclictest results reconfirm that Linux kernels with the PREEMPT_RT patch maintain much lower latencies than the default Linux kernels.

Table 4. Overall response latency results with and without the PREEMPT_RT patch.

Raspberry Pi3/Beagle-Bone AI	Linux Kernel Version	Samples	Space	PREEMPT_RT	Latency (µs)			
					Software Modules		Oscilloscope Min/Max/Avg	Cyclictest Min/Max
					Min/Max (wcrl)	Stdev/Variation		
RPi3	Ubuntu Mate 4.14.74-rt44-v7	1 M	user	yes	49/147	65/4261	49/128/105	<50/61
				no	53/360	33/1137	51/370/109	<50/305
			kernel	yes	50/67	20/417	42/56/50	-
				no	51/81	15/233	44/93/70	-
BBAI	Ubuntu 4.14.74-rt44-v7	1 M	user	yes	48/142	63/4122	49/105/98	<51/57
				no	52/290	11/197	51/354/109	<51/280
			kernel	yes	46/72	21/453	47/67/60	-
				no	50/73	24/578	51/67/55	-
RPi3	Arch Linux 4.19.10-1-ARCH	1 M	user	yes	42/122	67/2336	44/129/98	<50/67
				no	54/330	32/990	51/350/111	<50/310
			kernel	yes	51/56	20/411	40/51/51	-
				no	51/79	17/788	43/89/65	-
BBAI	Arch Linux 4.19.10-1-ARCH	1 M	user	yes	52/114	57/965	50/124/93	<50/72
				no	54/341	29/622	53/371/111	<50/355
			kernel	yes	55/65	20/298	50/71/68	-
				no	57/88	15/239	50/87/73	-
RPi3	Debian (Buster) 4.19.67-2	1 M	user	yes	40/90	79/6336	41/98/95	<50/80
				no	53/343	34/1190	53/310/101	<50/290
			kernel	yes	48/53	19/383	42/50/49	-
				no	48/60	13/2988	44/76/59	-
BBAI	Debian 4.19.67-2	1 M	user	yes	47/91	68/3450	49/94/90	<50/69
				no	54/310	29/978	54/351/112	<50/369
			kernel	yes	51/65	24/531	50/69/62	-
				no	55/77	12/165	51/76/71	-

7.6.1. Results at User Space

The majority of the Linux kernels' measurements with PREEMPT_RT-patched kernel, have shown the minimum response latency to be below 50 μs. The maximum worst-case response latency (wcrl), which indicates the longest time it takes the slave device to respond to an event, reached about 118 μs as an average value (for RPi3 and BBAI). The majority of the latencies (about 90%) on the RPi3 and BBAI with preemption support are quite below this maximum. However, on some occasions, maximum latency exceeded one millisecond. In general, maximal latencies do not often cross that value with the PREEMPT_RT patched kernel. The same measurements without PREEMPT_RT support (default Linux kernels) have shown the minimum response latency to be about the same and below 55 μs, however the maximum at about 329 μs. This maximum observed latency is significantly higher than the one observed under the PREEMPT_RT-patched Linux kernels. Figure 11 illustrates the maximum worst-case response latencies and how measurements are spread out from the average (stdev, as a stacked area) at user space for both kernels (default and preempted) and devices (RPi3 and BBAI).

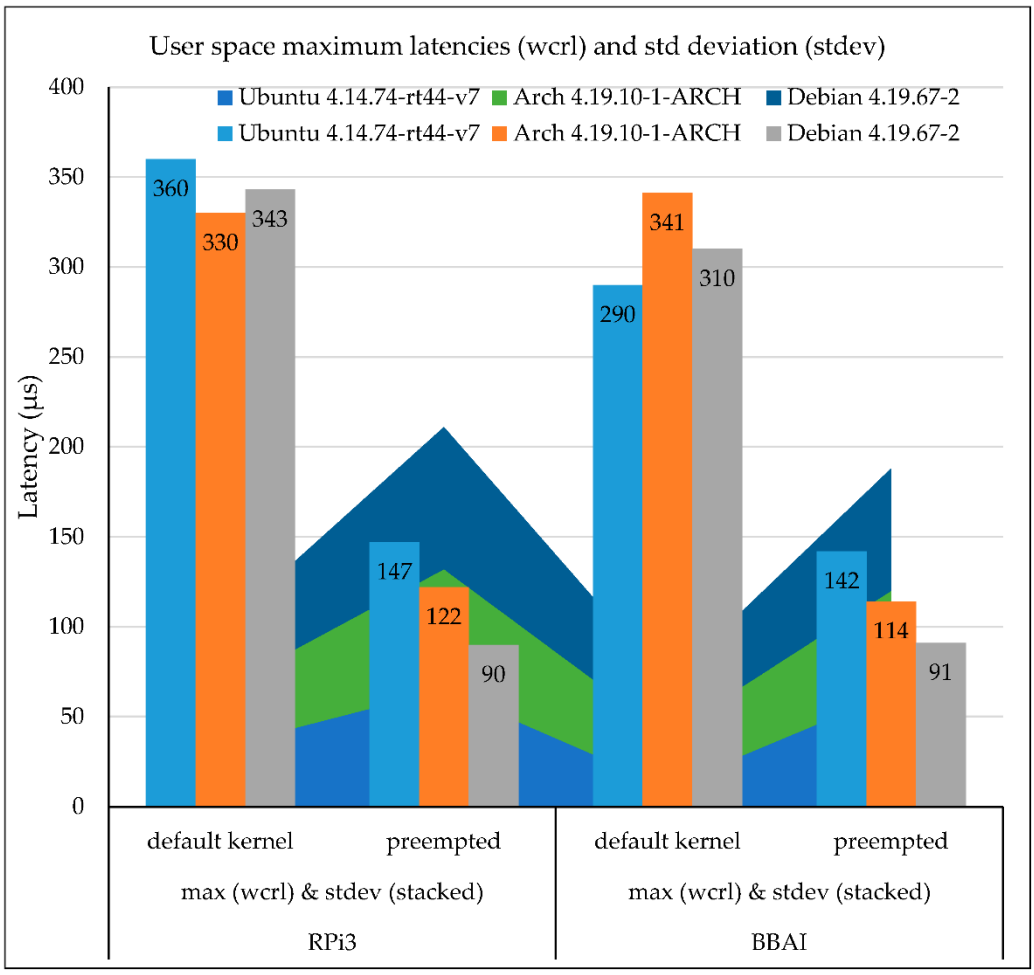

Figure 11. Maximum worst-case latencies and standard deviation for both default and preempted kernels at user space.

For real-time systems with strict timing constraints, this worst-case latency needs to be considered. The experimental results indicate that a value of about 150 μs, as an upper bound, could be an acceptable safety margin for such low frequencies in most real-time systems running in such a cooperative way, as long as the frequency time step value is higher.

7.6.2. Results at Kernel Space

Measurements with PREEMPT_RT-patched Linux kernels have shown the minimum response latency to be again below 50 μs with a maximum of 72 μs. A considerable amount of latencies (about 95%) are below this maximum. Interrupt service time measured at the slave devices was found to be very small (about 2–5 μs). The same measurements without PREEMPT_RT support have shown the minimum response latency to be about the same and below 57 μs with a maximum of 88 μs.

8. Conclusions

This research work presents the experimental evaluation of Linux real-time performance, and particularly latency issues, in kernels and distributions with the real-time variant PREEMPT_RT on Raspberry Pi3 Model B and BeagleBone AI ARM-based microcontrollers. The choice is based on the fact that such microcontroller units have sufficient processing power, are low cost, quite flexible and used extensively in various embedded control applications. Currently, there is limited research investigating the real-time performance of such ARM-based embedded platforms running Linux patched with PREEMPT_RT. Lately, a few studies have appeared to tackle such issues, although measurements are based primarily only on the cyclictest benchmark.

Experimental measurements provide novel insights on Linux real-time performance on ARM-based devices. The experimental results show that Linux kernels with the PREEMPT_RT patch provide better guarantees of hard real-time performance than the default ones. The majority of latencies on kernels with real-time support are considerably lower compared to those in the default kernels and are below 50 μs. The average maximum observed latency of 118 μs is still significantly lower than the one observed under the default Linux kernels. In real-time embedded devices running in such a master-slave mode (such as RPi and BBAI), a response latency value of about 150 μs, as an upper bound, could be an acceptable safety margin. Overall, the latencies and particularly the maximums are reduced in kernels with real-time support, thus making Linux with PREEMPT_RT more suitable for use in time-sensitive embedded control systems, as this experimental research provides evidence.

Real-time performance evaluations are based on the development of new specific real-time software measurement modules designed upon the introduction of a new response task model, an innovative aspect of this work. Some of the key features and contributions of this research work are the following:

- Provides latency measurements based on specific software real-time measurement modules, designed upon the introduction of a new response task model;
- Reveals novel insights on Linux real-time performance on ARM-based development platforms (BeagleBoard and Raspberry Pi), based on a comparative evaluation of real-time latency measurements at kernels with and without real-time support;
- Presents a measurements approach and evaluation methodology potentially applicable in other Linux kernels and distributions on such ARM-based embedded devices.

In our future work, we will extend our measurements methodology and evaluation analysis and provide further comparisons with other real-time OS such as Xenomai, etc.

Author Contributions: Conceptualization, G.K.A.; methodology, G.K.A.; software, G.K.A.; validation, G.K.A., N.P. and L.T.D.; formal analysis, G.K.A.; data curation, G.K.A.; writing—original draft preparation, G.K.A.; writing—review and editing, G.K.A., N.P. and L.T.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The experimental software modules are available as an open-source project at GitHub https://github.com/gadam2018/RPi-BeagleBone (accessed on 12 February 2021) [73].

Acknowledgments: The authors would like to thank the Computer Systems Laboratory (CSLab, https://cslab.ds.uth.gr/) (accessed on 15 January 2021) [75] for the technical support and the resources provided for this experimental research.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Algorithm A1 User and kernel space response task measurement algorithm

```
Master response task module at 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
Slave response task module at 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
Slave response task module at 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
```

Algorithm A2 User and kernel space periodic task measurement algorithm

```
Slave periodic task module at 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
Slave periodic task module at 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
```

References

1. The Linux Foundation: Real Time Linux. Available online: https://wiki.linuxfoundation.org/realtime/start (accessed on 4 December 2020).
2. Linutronix Linux for Industry: Real-Time. Available online: https://linutronix.de/en/ (accessed on 14 June 2020).
3. Regnier, P.; Lima, G.; Barreto, L. Evaluation of interrupt handling timeliness in real-time Linux operating systems. *SIGOPS Oper. Syst. Rev.* **2008**, *42*, 52–63, doi:10.1145/1453775.1453787.
4. Dellinger, M.; Garyali, P.; Ravindran, B. ChronOS Linux: A best-effort real-time multiprocessor Linux kernel. In *Proceedings of the 48th Design Automation Conference (DAC '11), San Diego, California, USA, 5–10 June 2011*; ACM: New York, NY, USA, 2011; 474–479, doi:10.1145/2024724.2024836.
5. Sousa, P.B.; Pereira, N.; Tovar, E. Enhancing the real-time capabilities of the Linux kernel. *SIGBED Rev.* **2012**, *9*, 45–48, doi:10.1145/2452537.2452546.
6. Texas Instruments: Programmable Real-Time Unit Subsystem and Industrial Communication SubSystem (PRU-ICSS). Available online: http://processors.wiki.ti.com/index.php/PRU-ICSS / (accessed on 14 January 2021).
7. Open Source Summit Tokyo Japan: The Many Approaches to Real-Time and Safety-Critical Linux. Available online: http://events17.linuxfoundation.org/sites/events/files/slides/talk_10.pdf (accessed on 12 November 2020).
8. AspenCore Electronics Industry Media: Embedded Markets Study -Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments. Available online: https://m.eet.com/media/1246048/2017-embedded-market-study.pdf (accessed on 23 November 2020).
9. 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, doi:10.1145/3291387.
10. Walbrecht, A. Growing, IoT Next Big Win. *Full Circle Mag.* **2015**, *96*, 9.
11. Lee, I.; Leung, J.Y.-T.; Son, S.H. *Handbook of Real-Time and Embedded Systems*, 1st ed.; Chapman & Hall/CRC: New York, NY, USA, 2007, doi:10.1201/9781420011746.
12. Kopetz, H. *Real-Time Systems—Design Principles for Distributed Embedded Applications*, 2nd ed.; Springer Publishing Company: Boston, MA, USA, 2011. Available online: https://link.springer.com/book/10.1007/978-1-4419-8237-7 (accessed on 11 March 2019).
13. Adam, G.K. DALI LED Driver Control System for Lighting Operations Based on Raspberry Pi and Kernel Modules. *Electronics* **2019**, *8*, 1021, doi:10.3390/electronics8091021.
14. 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, doi:10.3390/electronics8010033.
15. Radojkovic, P.; Girbal, S.; Grasset, A.; Quinones, E.; Yehia, S.; Cazorla, F.J. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.* **2012**, *8*, 25, doi:10.1145/2086696.2086713.
16. 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; 178–189, doi:10.1145/3302505.3310080.
17. Bundalo, Z.; Bundalo, D. *Embedded Systems Based on Open Source Platforms. Introduction to Data Science and Machine Learning*; IntechOpen: London, UK, 2019, doi:10.5772/intechopen.85806.
18. Nayyar, A.; Puri, V. A Review of Beaglebone Smart Board's-A Linux/Android Powered Low Cost Development Platform Based on ARM Technology. In Proceedings of the 9th International Conference on Future Generation Communication and Networking (FGCN '15), Jeju, Korea, 25–28 November 2015; pp. 55–63, doi:10.1109/FGCN.2015.23.
19. Ardito, L.; Torchiano, M. Creating and evaluating a software power model for linux single board computers. In *Proceedings of the 6th International Workshop on Green and Sustainable Software (GREENS '18), Gothenburg, Sweden, 27 May 2018*; ACM: New York, NY, USA, 2018; pp. 1–8, doi:10.1145/3194078.3194079.

20. Molloy, D. *Exploring Raspberry Pi Interfacing to the Real World with Embedded Linux*; John Wiley & Sons, Inc.: Indianapolis, IN, USA, 2016.
21. Yan, Y.; Gokul, G.; Dantu, K.; Ko, S.Y.; Ziarek, L.; Vitek, J. Can Android Run on Time? *ACM Trans. Embed. Comput. Syst.* **2019**, *17*, 1–26, doi:10.1145/3289257.
22. 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, doi:10.3390/app10093228.
23. 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, doi:10.3390/computers9040097.
24. The Linux Foundation: Cyclictest. Available online: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start (accessed on 11 October 2020).
25. Binkert, N.; Beckmann, B.M.; Black, G.; Reinhardt, S.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.; Krishna, T.; Sardashti, S.; et al. The gem5 simulator. *SIGARCH Comput. Archit. News* **2011**, *39*, 1–7.
26. Soto-Camacho, R.; Vergara-Limon, S.; Vargas-Treviño, M.A.D.; Paic, G.; López-Gómez, J.; Vargas-Treviño, M.; Gutierrez-Gutierrez, J.; Martínez-Solis, F.; Patiño-Salazar, M.E.; Velázquez-Aguilar, V.M. A Current Monitor System in High-Voltage Applications in a Range from Picoamps to Microamps. *Electronics* **2021**, *10*, 164, doi:10.3390/electronics10020164.
27. Bick, K.; Nguyen, D.T.; Lee, H.-J.; Kim, H. Fast and Accurate Memory Simulation by Integrating DRAMSim2 into McSimA+. *Electronics* **2018**, *7*, 152, doi:10.3390/electronics7080152.
28. Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the USENIX Annual Technical Conference, Anaheim, CA, USA, 10–15 April 2005; pp. 41–46.
29. Diaz, E.; Mateos, R.; Bueno, E.J.; Nieto, R. Enabling Parallelized-QEMU for Hardware/Software Co-Simulation Virtual Platforms. *Electronics* **2021**, *10*, 759, doi:10.3390/electronics10060759.
30. Kirova, V.; Karpov, K.; Siemens, E.; Zander, I.; Vasylenko, O.; Kachan, D.; Maksymov, S. Impact of Modern Virtualization Methods on Timing Precision and Performance of High-Speed Applications. *Future Internet* **2019**, *11*, 179, doi:10.3390/fi11080179.
31. Bismarck, J.L.; Morales, F. Evaluating Gem5 and QEMU Virtual Platforms for ARM Multicore Architectures. Master's Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 25 September 2016.
32. Halang, W.A.; Gumzej, R.; Colnaric, M.; Druzovec, M. Measuring the performance of real-time systems. *Springer Real-Time Systems* **2000**, *18*, 59–68, doi:10.1023/A:1008102611034.
33. 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, doi:10.1109/MM.2009.56.
34. 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.
35. Marieska, M.D.; Kistijantoro, A.I.; Subair, M. Analysis and benchmarking performance of real time patch Linux and Xenomai in serving a real time application. In *Proceedings of the IEEE Electrical Engineering and Informatics (ICEEI'11), Bandung, Indonesia, 17–19 July 2011*; IEEE Press: Piscataway, NJ, USA, 2011; pp. 1–6, doi:10.1109/ICEEI.2011.6021563.
36. Bini, E.; Buttazzo, G.C. Measuring the performance of schedulability tests. *Springer Real-Time Syst.* **2005**, *30*, 129–154, doi:10.1007/s11241-005-0507-9.
37. Gardner, K.; Harchol-Balter, M.; Hyytia, E.; Righter, R. Scheduling for efficiency and fairness in systems with redundancy. *Perform. Eval.* **2017**, *116*, 1–25, doi:10.1016/j.peva.2017.07.001.
38. Slanina, Z.; Srovnal, V. Embedded Systems Scheduling Monitoring. In *Proceedings of the Third International Conference on Systems (ICONS'08), Cancun, Mexico, 13–18 April 2008*; IEEE Press: Piscataway, NJ, USA, 2008; pp. 124–127, doi:10.1109/ICONS.2008.47.
39. Bertogna, M.; Cirinei, M.; Lipari, G. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Trans. Parallel Distrib. Syst.* **2009**, *20*, 553–566, doi:10.1109/TPDS.2008.129.
40. 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; doi:10.4271/2014-01-0200.
41. Mejia-Alvarez, P.; Moncada-Madero, D.; Aydin, H.; Diaz-Ramirez, A. Evaluation framework for energy-aware multiprocessor scheduling in real-Time systems. *J. Syst. Archit.* **2019**, *98*, 388–402, doi:10.1016/j.sysarc.2019.01.018.
42. Koolwal, K. Investigating latency effects of the linux real-time preemption patches (PREEMPT_RT) on AMD's GEODE LX platform. In *Proceedings of the 11th Real-Time Linux Workshop (OSADL'09), Dresden, Germany, 28–30 September 2009*; OSADL: Heidelberg, Germany, 2009; pp. 131–146.
43. Emde, C. Long-term monitoring of apparent latency in PREEMPT_RT Linux real-time systems. In *Proceedings of the 12th Real-Time Linux Workshop (OSADL'10), Nairobi, Kenya, 25–27 October 2010*; OSADL: Heidelberg, Germany, 2010; pp. 1–6.
44. Beamonte, R.; Giraldeau, F.; Dagenais, M. High performance tracing tools for multicore linux hard real-time systems. In *Proceedings of the 14th Real-Time Linux Workshop (OSADL'12), Chapel Hill, Virginia, USA, 18–20 October 2012*; OSADL: Heidelberg, Germany, 2012; pp. 1–7.
45. Beamonte, R.; Dagenais, M.R. Linux Low-Latency Tracing for Multicore Hard Real-Time Systems. *Hindawi Adv. Comput. Eng.* **2015**, *8*, doi:10.1155/2015/261094.

46. Barbalace, A.; Luchetta, A.; Manduchi, G.; Moro, M.; Soppelsa, A.; Taliercio, C. Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application. *IEEE Trans. Nucl. Sci.* **2008**, *55*, 435–439, doi:10.1109/TNS.2007.905231.
47. 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*; IFAC-PapersOnLine: Geneva, Switzerland, 2016; Volume 49, pp. 401–406, doi:10.1016/j.ifacol.2016.12.080.
48. 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: Heidelberg, Germany, 2010; pp. 1–17.
49. 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, doi:10.5120/2202-2796.
50. 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, doi:10.5176/2251-3043_3.1.244.
51. 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.
52. Calandrino, J.; Leontyev, H.; Block, A.; Devi, U.C.; Anderson, J. LITMUS: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06), Rio de Janeiro, Brazil, 5–8 December 2006*; IEEE Press: Piscataway, NJ, USA, 2006; pp. 111–123, doi:10.1109/RTSS.2006.27.
53. Reghenzani, F.; Massari, G.; Fornaciari, W. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Comput. Surv.* **2019**, *52*, 36, doi:10.1145/3297714.
54. EMLID Raspberry Pi Real-Time Kernel. Available online: https://emlid.com/raspberry-pi-real-time-kernel/ (accessed on 7 November 2020).
55. Raspberry Pi Going Realtime with RT Preempt. Available online: http://www.frank-durr.de/?p=203 (accessed on 3 October 2020).
56. Real-Time on Raspberry Pi. Available online: https://www.distek.com/blog/part-2-real-time-on-raspberry-pi/ (accessed on 4 December 2020).
57. 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 20 October 2020).
58. AUTOSTATIC RPi3 and the Real Time Kernel. Available online: https://autostatic.com/2017/06/27/rpi-3-and-the-real-time-kernel/ (accessed on 25 October 2020).
59. 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 24 October 2020).
60. Delgado, R.; You, B.-J.; Choi, B. Real-time Control Architecture Based on Xenomai Using ROS Packages for a Service Robot. *J. Syst. Softw.* **2019**, *151*, 8–19. 10.1016/j.jss.2019.01.052.
61. Delgado, R.; Park, J.; Choi, B.W. Open Embedded Real-time Controllers for Industrial Distributed Control Systems. *Electronics* **2019**, *8*, 223, doi:10.3390/electronics8020223.
62. Delgado, R.; Hong, C.; Shin, W.; Choi, B. Implementation and performance analysis of an etherCAT master on the latest real-time embedded linux. *Int. J. Appl. Eng. Res.* **2015**, *10*, 44603–44609.
63. Princy, S.E.; Nigel, K.G.J. Implementation of cloud server for real time data storage using Raspberry Pi. In *Proceedings of the Online International Conference on Green Engineering and Technologies (IC-GET'15), Coimbatore, India, 27 November 2015*; IEEE: Piscataway, NJ, USA, 2015; pp. 1–4, doi:10.1109/GET.2015.7453790.
64. 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, doi:10.1016/j.procs.2017.12.207.
65. Guravaiah, K.; Thivyavignesh, R.G.; Velusamy, R.L. Vehicle monitoring using internet of things. In *Proceedings of the 1st International Conference on Internet of Things and Machine Learning (IML'17), Liverpool, UK, 17–18 October 2017*; ACM Press: New York, NY, USA, 2017; 1–7, doi:10.1145/3109761.3109785.
66. Kruger, C.P.; Hancke, G.P. Benchmarking Internet of Things devices. In *Proceedings of the 12th IEEE International Conference on Industrial Informatics (INDIN'14), Porto Alegre, Brazil, 27–30 July 2014*; IEEE Press: Piscataway, NJ, USA, 2014; pp. 611–616, doi:10.1109/INDIN.2014.6945583.
67. 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; 64-69, doi:10.1145/3059009.3059028.
68. Davis, R.I.; Burns, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.* **2011**, *43*, 44, doi:10.1145/1978802.1978814.
69. National Instruments: What Is a Real-Time Operating System (RTOS)? Available online: http://www.ni.com/white-paper/3938/en/ (accessed on 12 November 2020).
70. Chakraborty, S.; Eberspacher, J. *Advances in Real-Time Systems*; Springer: Berlin/Heidelberg, Germany, 2012. https://link.springer.com/book/10.1007/978-3-642-24349-3 (accessed on 16 September 2018).

71. Fayyad-Kazan, H.; Perneel, L.; Timmerman, M. Linux PREEMPT-RT v2.6.33 versus v3.6.6: Better or worse for real-time applications? *ACM SIGBED Rev.* **2014**, *11*, 26–31, doi:10.1145/2597457.2597460.
72. Garre, C.; Mundo, D.; Gubitosa, M.; Toso, A. Real-Time and Real-Fast Performance of General-Purpose and Real-Time Operating Systems in Multithreaded Physical Simulation of Complex Mechanical Systems. *Hindawi Math. Probl. Eng.* **2014**, 14, doi:10.1155/2014/945850.
73. RPi-BeagleBone. Available online: https://github.com/gadam2018/RPi-BeagleBone (accessed on 12 February 2021).
74. Holt, A.; Huang, C.-Y. *Embedded Operating Systems A Practical Approach*; Springer International Publishing: Cham, Switzerland, 2014; doi:10.1007/978-3-319-72977-0.
75. Computer Systems Laboratory. Available online: https://cslab.ds.uth.gr/ (accessed on 15 January 2021).