Real-time performance assessment using fast interrupt request on a standard Linux kernel

Rayanne Souza¹ | Miguel Freitas¹ | Marcelo Jimenez¹ | João Magalhães¹ | Alan C. Kubrusly¹ | Noemi Rodriguez²

¹Center for Telecommunication Studies, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil
²Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil

Summary
This article presents the use of ARM’s fast interrupt request (FIQ) to accomplish better jitter performance on real-time drivers without using patches for real-time extensions on the native Linux kernel code. Writing an FIQ interrupt handler is challenging due to the lack of Linux kernel support and the need to avoid page faults exception during its execution. We investigate and evaluate a mechanism that employs static mapping for peripherals and changes on the Linux kernel code to allow the FIQ interrupt handler to be written in the C language. Furthermore, the FIQ performance was evaluated by comparing it with a timer Interrupt Request on Linux PREEMPT-RT in full CONFIG_PREEMPT_RT mode. Both were applied on a Linux driver for data acquisition of a pipeline inspection gauge system. Results show that the FIQ approach was able to reduce in 97.49% the interrupt jitter and, as a result, allowed an increase in the data acquisition frequency from 1024 Hz to 2048 Hz, showing that the FIQ approach can be considered for real-time applications without resorting to real-time extensions.

KEYWORDS
embedded software, FIQ interrupt, jitter, Linux kernel, pipeline inspection gauge, real-time driver

1 INTRODUCTION

Real-time applications are widely employed by several industrial sectors, such as robotics,¹ aerospace,² and automotive,³ using real-time operating systems (RTOSs). The timing correctness of systems is traditionally guaranteed by the worst-case execution time (WCET) analysis. The purpose of this analysis is to determine if the system meets its real-time constraints, that is, short response time of a task, short execution time of the interruption, and among others. Usually, the WCET analysis is mandatory for hard-real time systems,⁴ however, other real-time applications can be less demanding regarding accomplishments of the real-time constraints, and their performance can be evaluated according to their latency and jitter.⁵ Jitter caused by both hardware (caches and pipelines) and software effects (disabled interrupts and mechanisms for synchronizing processes) is typically in the order of microseconds, which can be long enough to affect a real-time application. For instance, in real-time control systems, feedback control loops are implemented as execution of periodic tasks. The impact of jitter on such application is typically analyzed to ensure control system's stability.⁶-⁸ Latency is defined as the time elapsed between the occurrence of an external event and the correct reaction to that
event, whereas jitter is a random deviation from the ideal timing of the event. Therefore, jitter can be measured by the standard deviation of the latency. Ensuring low jitter and low latency is crucial for real-time measurements systems. Since, in our real-time application, jitter is a critical parameter, we evaluated the performance of our system in terms of it.

In the Oil & Gas industry, a pipeline inspection gauge (PIG) is commonly used to identify defects in pipes. PIGs are devices which are inserted into, and travel throughout the length of a pipeline, driven by the product flow. They usually carry a set of sensors and a real-time data acquisition system can be used to perform the characterization of pipeline defects while travelling along it. The resulting interrupt latency and jitter affects the acquisition system in such a way that the larger the jitter or latency, the lower the resolution and the accuracy of the pipe characterization that the sensing tool can provide. Therefore, jitter and the latency become critical parameters when the system needs to ensure reliability and quality in a high-frequency data acquisition scenario, such as monitoring systems that are in constant motion, like PIGs. For example, a 200-microsecond jitter for an acquisition frequency of 2048 Hz (equivalent to a 488-microsecond acquisition period) has a severe impact on the performance of the system, since 40.98% of the time is wasted by the jitter. This low performance makes it impractical to use acquisition frequencies at this order or higher. In this article, we investigate the use of ARM’s fast interrupt request (FIQ) feature as a mechanism to allow real-time applications in an industrial-related system. The data acquisition system under study here is a geometric feeler PIG that uses a real-time driver running on a standard Linux kernel version 4.9.66. In this system, it is currently impossible to ensure low response time and high reliability for an acquisition rate of 2048 Hz with its 130 multiplexed sensors. Without FIQ, the system could only reliably operate up to 1024 Hz, because the jitter is too high (over 130 microseconds). Hence, the interrupt jitter is the most critical parameter to be examined.

2 | LINUX KERNEL IN EMBEDDED SYSTEMS

The use of the Linux OS in embedded systems is motivated by its free distribution and its hardware portability. It has compatibility with a wide range of architectures, including the popular ARM, PowerPC, SPARC, x86, and MIPS, and offers a vast number of libraries and drivers with source code available for modification. This amount of implemented code can reduce development efforts, time, and cost, making Linux a strong alternative to the development of specialized approaches for the embedded systems community. However, Linux is not originally designed to support real-time tasks. To reduce the overall latency and make certain classes of real-time applications feasible on Linux, kernel version 2.6 introduced the possibility of preempting the kernel code itself, besides the usual user-space preemption, by enabling the CONFIG_PREEMPT option. However, the preemptible kernel still contains long critical sections that may cause long interrupt latencies. In order to allow their usage in real-time applications with lower latency, the embedded systems community usually develops support features or applies patch sets, such as PREEMPT_RT, Xenomai, and RTAI. The PREEMPT_RT patchset adds preemption to most of the kernel critical section mechanisms, like spinlocks, while deferring interrupt code execution to the so-called “Interrupt Request (IRQ) thread” that runs in process context. Only interrupt handlers that specifically ask to be executed in hardware-interrupt context will remain preemptible, therefore effectively reducing the true critical sections to a minimum. With most of the kernel code running as threads and processes, the RT-aware scheduler is then able to arbitrate the priorities based on real-time requirements. Real-time drivers, however, may still require a further patch set such as Xenomai, which adds a dual kernel configuration to avoid the latency due to the last bits of nonpreemptible interruption code and the scheduler itself.

Similar to Xenomai, the RTAI patch is also a dual-kernel approach where a nanokernel hardware abstraction layer is installed between the hardware and the OS to add hard real-time support to Linux. The RTAI has a slightly different layout from Xenomai in what concerns interrupt reception. In the latter, those are managed by the nanokernel, whereas in the former by its core. If an event of receiving an interrupt is associated with a real-time application, the RTAI core handles it immediately. When an interrupt is not interesting to RTAI, it is sent to the Linux kernel using the nanokernel. That structure offers better performance than Xenomai since it avoids the latency introduced by the nanokernel. However, Xenomai offers support for a larger amount of architecture. Both of these dual-kernel approaches require the real-time driver to be written using a specific API, different from the Linux standard one.

Lelli et al discussed another approach that also aims at offering real-time support for the Linux kernel: the real-time scheduler SCHED_DEADLINE. This feature is available in the baseline Linux kernel since kernel version 3.14. On its scheduling policy, it is mandatory to specify the period of the task execution and the amount of CPU time needed to execute
a task every period, that is, the runtime. With these parameters, the scheduler ensures that each task has its execution limited by the runtime in every period. Its algorithm computes a scheduling deadline every time the task wakes up. The scheduling deadline parameter corresponds to a time limit beyond which one cannot go over to accomplish a task. It is used to sort the ready queue in such a way that the task with the highest priority is the one which has a deadline closest to the current time, meaning that this mechanism dynamically assigns priorities to tasks. By sizing these three parameters properly, each task can meet the timing constraints of the real-time application.

The coexistence of a general purpose OS and a real-time OS on multi-core platforms is a recent trend in real-time solutions. It is possible through the use of virtual machine monitors, that manages shared computational resources to provide an isolated virtual environment. This approach allows executing less critical tasks on a general purpose OS, such as a Linux guest, benefitting from the availability of rich libraries and drives. The critical tasks are executed by a dedicated core running a real-time OS to guarantee timing constraints of the real-time application. Also, within the context of multiple OSs, the real-time task can be handled by a single core running a bare-metal implementation in place of the RTOS. The bare-metal approach runs directly on the physical hardware platform and, consequently, allows direct access to the hardware resources. Under the bare-metal concept, there are no system calls and the application can perform I/O accesses in a timely, deterministic manner. However, this approach requires the ability to code in a very low-level, which makes the code development hard and time-consuming for complex tasks. In contrast, the RTOS offers an interface to simplify the application code with little overhead imposed by the system calls.

Real-time performance has been previously assessed by several authors. Brown and Martin evaluated three cases, namely the performance of a baseline Linux kernel, the performance of the same kernel with the PREEMPT_RT patches, and the performance of this kernel with the Xenomai patches, by measuring the latency interrupt and the jitter of a periodic task. Qualitative tests were performed using an Atmel AVR microcontroller as platform. For the Linux kernel space, they implemented a periodic task employing hrtimers to toggle a GPIO pin. In their work, jitter was measured for two categories of real-time, specifically the 95% hard real-time and the 100% hard real-time, where “the real-time requirements should be met at least 95%, and 100%, respectively, of the time”. They concluded that in the kernel space, the native Linux code has better performance for the 95% hard real-time category and Xenomai is the best for the 100% hard real-time category. Arm et al analyzed the real-time capabilities for the RTAI Linux real-time extension, which is also a dual kernel approach, on LinuxCNC. In industrial applications, the LinuxCNC is offered to operate Computer Numerical Control machines. It is available as packages for the Ubuntu and Debian distributions. The authors used the RTAI interrupt latency as one of the evaluation parameters. The AMD Athlon XP 3000+ processor was employed as a hardware platform. Interrupt latency and jitter are also relevant issues in RTOS. In order to minimize the interrupt latency, Zhang et al have developed a method that protects critical sections without disabling interrupts on the SmartOSEK RTOS. While this approach reduces interrupt latency, it also increases the execution time for system services. Barbalace et al compared Xenomai, RTAI, and the baseline Linux kernel (v 2.6) in terms of interrupt latency and jitter. A test program was integrated to a Linux module and was run on a Motorola MVME5500 single-board computer. The test's results showed that the maximum latency, 73.2 microseconds, was obtained with Xenomai and the minimum, 71.8 microseconds, with RTAI. The baseline Linux produced the maximum jitter value, 0.40 microseconds, while the RTAI reported the minimum, 0.15 microseconds. The RTAI real-time extension outperformed Xenomai and the Linux in both cases. The authors stated that Xenomai is slightly less performing than RTAI due to its layered approach, which introduces some overhead in interrupt management.

Unlike the previously mentioned works, in this work real-time performance with native Linux code has been achieved without applying patches for real-time extensions. FIQ is the highest priority interrupt request in an ARM CPU and, while restricted to this specific architecture, it is technically very similar to the nonmaskable interrupt signal available in other architectures. The Linux kernel neither uses nor disables FIQ interrupts in its system services, consequently, an FIQ event can preempt any running Linux code. In our experiments, the FIQ interrupt allowed for real-time driver latencies as good as, or better than the more complex dual-kernel approaches. However, the lack of Linux kernel support comes at a price and some restrictions have to be enforced to use this mechanism.

In the next sections, we present the FIQ interrupt feature, the challenges of using this interrupt in the Linux kernel, two strategies to allow the FIQ usage and the resulting evaluation of the FIQ performance, on a standard Linux kernel, over the IRQ, on Linux PREEMPT-RT, in a PIG’s data acquisition system.
3 | FAST INTERRUPT REQUEST

There are two types of hardware interrupts in an ARM CPU: the FIQ for low interrupt latency and the IRQ for general purpose. A hardware interrupt occurs whenever a device sends an electrical signal to a specific interrupt line. The AT91SAM9G25 processor has one line for the IRQ and another one for the FIQ, denoted as nIRQ and nFIQ, respectively. The FIQ has a higher priority than the IRQ. Therefore, when the nFIQ line is activated by some device, the processor serves this interrupt even if the highest priority IRQ handler is running. When the processor responds to an FIQ interrupt, it jumps into a processor mode called FIQ mode and disables the FIQ and IRQ interrupt requests. The processor mode defines access rights to the system resources. For example, the FIQ mode allows read/write access to the Current Program Status Register (CPSR) (privileged mode) and provides a set of associated registers known as banked registers that are multiple copies of a register at the same address. In addition to the priority level difference, the FIQ mode has seven registers banked into the system (r8-r14) while the IRQ mode has two banked registers (r13-r14). A large number of banked registers in the FIQ mode reduces the number of cycles on entry and exit to the interrupt service routine (ISR) code, since the processor needs to save fewer registers on the stack, decreasing the number of push and pop operations to the ISR. The last entry of the interrupt vector table is associated with the FIQ interrupt and has the address 0x1C. Being the last entry means that the ISR can be placed directly at this address, avoiding the need for a branch instruction and allowing a faster service of the FIQ interrupt.

When the core responds to an FIQ interrupt, the processor mode changes to FIQ mode and the core executes the following steps: (i) saves the CPSR value into SPSR_FIQ (saved processor status register); (ii) saves the PC value into r14_FIQ (link register); (iii) sets the PC register to 0x1C address (FIQ exception vector address); and (iv) runs the interrupt handler. When the core exits an FIQ handler, it performs the following two steps: (i) updates the PC with r14_FIQ minus four (offset value of the FIQ interrupt) and (ii) moves back the SPSR_FIQ to the CPSR. This operation automatically changes back the processor mode to the previous one.

3.1 | FIQ in the Linux kernel

The Linux kernel provides some functions to facilitate FIQ manipulation. They can be seen in the file /arch/arm/fiq.h. Listing 1 shows the four steps needed to initialize the FIQ interrupt, which can be summarized as: (i) request of the FIQ interrupt by calling the function claim_fiq (line 7); (ii) registering the stack used by the FIQ to save the registers of the interrupted process (line 12); (iii) registering the interrupt handler through the function set_fiq_handler (line 14); and (iv) enabling the FIQ interrupt (line 15). The function set_fiq_handler copies the handler code directly to the FIQ vector (at 0x1C address). The ARM architecture has designed the FIQ interrupt vector as the last entry in the vector table to increase the execution speed by removing the need for a branch and associated delays.

When compiling the application, option "config FIQ" in file /arch/arm/Kconfig must be set to "y" (yes). Because it is currently not possible to modify this value via a "make menuconfig" option, we needed to apply a patch for this modification.

```
static u8 fiqstack[4096];
static struct fiq_handler fh= {.name = "kpig_fiq"};
static int fiq_init( void ){
    struct pt_regs fiqregs;
    if (claim_fiq(&fh)){
        pr_crit( "Error in claim_fiq \n");
        return -EBUSY;
    }
    fiqregs.ARM_sp = (u32)&fiqstack[sizeof(fiqstack)];
    set_fiq_regs (&fiqregs);
    set_fiq_handler((void *)fiqhandler , 8);
    enable_fiq(ID_FIQ)
    return 0;
}
```

Listing 1: The FIQ init function
3.1.1 FIQ handler in the C language

The GCC compiler provides the directive `__attribute__((interrupt("FIQ")))` to facilitate the writing of the FIQ interrupt handler in C. This attribute indicates to the compiler how the prologue and epilogue of an FIQ interrupt routine should be handled. Because the FIQ mode has seven banked registers (r8-r14), the compiler automatically manages the storage of values from registers r0 to r7. Both the restoration of these registers and the return address are managed by the compiler as well.

The function `set_fiq_handler` takes as parameters a pointer to the handler routine and the handler routine size. Using `memcpy`, it copies the handler routine to a reserved area of approximately five hundred bytes. Listing 2 illustrates the definition of an FIQ interrupt handler in C using an assembly wrapper.

```
1 u32 fiqhandler_asm[2];
2 fiqhandler_asm[0] = 0xE51FF004; /* ldr pc, [pc, #-4] */
3 fiqhandler_asm[1] = (u32) fiqhandler_c;
4 set_fiq_handler ((void*)fiqhandler_asm, 8);
```

Listing 2: Assembly wrapper for the FIQ interrupt handler in C

In this strategy, the ISR consists of a jump to the C handler as shown in Listing 3. Notice that this frees us from the need to specify the size of the C interrupt handler, at the price of delaying its execution because of the execution of an extra jump instruction.

```
1 0x001C: ldr pc, [pc, #-4]
2 0x0020: .word fiqhandler_c
```

Listing 3: FIQ interrupt handler form

In Linux, writing an FIQ handler in C is not as simple as writing an IRQ handler. Most kernel APIs internally depend on scheduler functions and spinlocks to manage concurrent accesses, which are strictly forbidden in FIQ code because the FIQ code needs to be fast and nonblocking. Typically, it is necessary to either write or rewrite drivers to have the same functionalities that the API offers without blocking operations. In addition, we must also ensure that all resources accessed from the FIQ handler are already mapped into the process memory in order to avoid page faults, since the CPU does not support handling page faults in FIQ mode. In fact, most kernel developers assert that FIQ can only be implemented by a code that is statically linked to the kernel and executed early in the boot process. In the next section, we discuss the strategies to avoid page faults occurrences when the FIQ handler is running from a loadable kernel module (LKM).

4 AVOIDING PAGE FAULTS IN FIQ HANDLER

Writing the FIQ handler into a Linux LKM is convenient, since it is not necessary to reboot the system to insert and remove the module during the testing stage. The `vmalloc` function allocates the LKM on the kernel Module Space area by applying the Demand Paging technique, which consists in delaying the page frame allocation until a process requires access to a page that is not currently mapped in memory. Access to a nonmapped memory region causes a page fault exception that must be handled by the kernel, which then loads the requested memory page and resumes the execution transparently. Therefore, any code residing in kernel Module Space may produce unpredictable page faults at the time it gets executed or accessed. When a page fault occurs in an FIQ handler, the OS gets a kernel Panic due to the kernel's incapacity to interrupt the FIQ in order to handle this exception. This is further complicated by the fact that page mapping is a per-process structure, so even if the requested data are already loaded into the physical memory, that does not mean
that all processes have a valid page mapping for it. When dealing with FIQ interruptions, one is never able to predict which process will be running at the moment the FIQ interrupt takes place.

The kernel Panic occurrence crashes the entire system. It is unacceptable that a system presents such a behavior. For this reason, the FIQ handler cannot require access to a memory area allocated by the \texttt{vmalloc} function. Hence, the usage of the \texttt{vmalloc} function by an FIQ handler is not allowed. Over the following subsections, we discuss two strategies to avoid the occurrence of a page fault inside the FIQ handler, namely, allocating all memory demanded by the LKM using \texttt{kmalloc} and creating a static mapping of the physical address for peripherals. The complete patch set of the Linux kernel can be found our repository.\textsuperscript{35}

\section{Changing the LKMs allocation for use \texttt{kmalloc}}

All memory demanded by the LKM must be allocated using \texttt{kmalloc} because this function allocates contiguous pages and returns a logical address, which is mapped directly to a physical address,\textsuperscript{36} meaning that there is no need to update pages on demand. Therefore, we modified the original kernel allocation function that loads the LKM into the kernel Module Space area using \texttt{vmalloc} to use the \texttt{kmalloc} function. As a consequence, we also had to modify the kernel function that frees memory so as to use \texttt{kfree}.

\section{Static mapping for peripherals}

In embedded software, it is usual to deal with peripherals inside the interrupt handlers. In our data acquisition system, the FIQ handler needs to access peripherals, such as a Serial Peripheral Interface (SPI), a Timer Counter (TC), and a Direct Memory Access (DMA) controller to perform writing and reading operations. The function \texttt{ioremap} does the dynamic mapping of a given physical address and returns the virtual address base that allows both read and write accesses. The memory area returned by this function corresponds to the same memory area returned by the \texttt{vmalloc} function.\textsuperscript{36} Therefore, there is no guarantee that the use of this virtual address will not result in a page fault during the FIQ execution. The static mapping technique solves this problem since it statically defines and associates a virtual address and the physical address of the peripheral. The kernel performs this type of mapping during the boot phase, ensuring that the virtual address is already mapped for any process that will ever be created during the system execution. Whenever a physical address is statically mapped to a virtual address, the function \texttt{ioremap} returns the virtual address already defined, instead of adopting the dynamic mapping.

\subsection{Choosing the virtual address}

Once mapped, the virtual address for the static mapping cannot be mapped to another physical address. Hence, one needs to choose a memory area that is known to be available and not used by the system. Therefore, we chose the virtual address area destined to the Instruction Tightly-Coupled Memory (ITCM) feature to define the virtual address for the static mapping, since it has fixed virtual addresses and this processor feature was not used by our system. Using this memory area for the peripheral mapping is convenient because it ensures that such addresses will not be used by the kernel for any other functions.

\subsection{Creating static mapping in Linux kernel}

The Linux kernel defines memory mappings through an array of \texttt{map_desc} structs. Hence, we defined a \texttt{map_desc} termed \texttt{at91_iomap_desc} according to the peripherals accessed by the FIQ handler (i.e., SPI, TC, and DMA). We then created a static mapping function \texttt{at91_map_io} that calls \texttt{iotable_init} with the \texttt{map_desc} struct to map through the \texttt{create_mapping} call. It was also necessary to modify the initialization of the \texttt{DT_MACHINE_START} macro by adding a static mapping function \texttt{at91_map_io}, since this macro loads the information about the kernel initialization on the host board (available on the \texttt{at91sam9.c} file of the 4.9.66 kernel).
At the booting stage, the kernel performs the following steps for static mapping:37

- In the directory /arch/arm/kernel/setup.c, the setup_arch() routine calls paging_init;
- paging_init executes the devicemaps_init function;
- devicemaps_init does the static mapping by calling mdesc->map_io(). This call uses the at91_iomap_desc registered from DT_MACHINE_START macro;

Once the static mapping is performed early in the boot process, any kernel code is able to use it by requesting its virtual address from ioremap as usual. The Linux ioremap is smart enough to check if a given physical address is already statically mapped before performing the dynamic allocation. Essentially, all necessary changes on the Linux kernel code to allow the FIQ usage can be summarized as follows:

1. Enabling the FIQ feature by modifying the Kconfig file located in arch/arm/.
2. Modifying the LKM allocation method to use the kmalloc function instead of the vmalloc.
3. Adding a static mapping function to the map_desc struct of the target board.

5 | LIMITATIONS OF THE FIQ IN THE PROPOSED APPROACH

The proposed approach was designed for systems that have one real-time task. Thus, some limitations may arise when applying the proposed approach to solve different real-time problems. These limitations are discussed below and summarized in Table 1.

5.1 | Single FIQ source

Usually, the FIQ feature has a single high-priority interrupt source connected to the nFIQ line because it was designed to quickly serve a specific critical task. With a single source, it is not necessary to determine the source of the interrupt.38 Therefore, the ISR can be executed directly. In our system, we have precisely one interrupt source associated with the FIQ line. However, some applications may need to deal with multiple sources and, consequently, require a mechanism to identify the interrupt source and establish a priority, which increases the FIQ response time. Further studies would be necessary to determine whether the use of FIQ could be advantageous in these scenarios.

5.2 | Incompatibility with Linux kernel synchronization primitives

Because the FIQ handler should execute as fast as possible, no blocking operation can occur within the handler. To use any Linux kernel API functions inside the handler, the programmer must make sure that these functions do not

| Limitation                                      | Description                                                      |
|------------------------------------------------|------------------------------------------------------------------|
| Single FIQ source                              | The proposed design was evaluated considering only one interrupt source. |
| Incompatibility with Linux kernel synchronization primitives | Linux kernel synchronization primitives are strictly forbidden in FIQ code. |
| Unavailability of the ITCM feature             | The virtual address area reserved for ITCM is used to create the static mapping for peripherals. |
| Size of the FIQ module                         | The kernel modules are allocated with kmalloc instead of vmalloc. Hence, the size of the FIQ module depends on the amount of memory that kmalloc can allocate. |
contain spinlocks or schedulers. Whenever a data structure needs to be shared between FIQ handlers and other code, the programmer can resort to a data structure that allows unidirectional communication between the real-time process and the normal process [for instance, a lock-free first in first out (FIFO)] or disabling the FIQ as a mean of synchronization. The latter is a simpler solution but it risks of increasing the response time of real-time task if competition is frequent. The use of the former technique is usually restricted to cases when there is only one real-time process accessing the data structure. In our application, the FIQ interrupt accesses and modifies a FIFO structure that is shared with normal code. If we had used some kernel synchronization primitive to protect this access, we would have risked blocking the FIQ handler. To allow the real-time process to share this data structure, we could use a lock-free queue or temporarily disable the FIQ interrupt within the normal code whenever it needs to access the shared structure.

5.3 | Unavailability of the ITCM feature

In Section 3.1, we loaded a jump instruction on the FIQ vector table entry to jump where the FIQ handler code is located. We could have considered using the ITCM feature to locate our FIQ handler rather than using the ITCM mapping area to map our peripherals. That would probably improve the time determinism of the FIQ handler by avoiding caches and DRAM latencies. However, in our application, using the FIQ interrupt without adding other approaches that tend to reduce time, was enough to meet the applications' timing constraints. Hence, the usage of ITCM mapping area to meet our need for a fixed virtual address is much more attractive than using the ITCM feature to increase speed performance.

5.4 | Size of the FIQ module

In Section 4.1, the kernel modules are allocated with kmalloc instead of vmalloc. By adopting this strategy, every LKM will reside in a contiguous area of physical memory regardless of being an FIQ module. It can be hard to find a physically contiguous block of memory for large allocations because the physical memory space is much smaller than the virtual memory space. Thereby, the size of the FIQ module is limited by the physical memory available in the system. Because the FIQ handler is usually in the order of 100 B, the LKM module dedicated to the FIQ does not usually demand a large memory space and can be conveniently allocated by the kmalloc function. Note that the aforementioned LKM loading strategy may not be suitable for applications requiring the allocation of a large number of LKM or large memory space for any LKM. For these cases, it is more appropriate to allocate only the FIQ module using kmalloc and to use vmalloc for the other LKMs. Another alternative would be to try to get the contiguous area of physical memory for the LKM and if it fails, use the vmalloc function.

6 | EVALUATION OF THE FIQ APPROACH

The evaluation of the FIQ approach was divided into two parts. First, the interrupt jitter for the FIQ and IRQ approaches was measured to evaluate the jitter reduction when replacing the timer IRQ handler on Linux PREEMPT-RT with the FIQ handler on a standard Linux without applying any real-time patch. Then the performance of our data acquisition system was evaluated for both interrupts in order to quantify the performance improvements of the acquisition system when using the FIQ. The time effectively used for acquisition and the available time (AT) were used as metrics.

6.1 | Measuring the interrupt jitter

We have measured the FIQ and IRQ interrupt jitter using a TC inside their respective handlers at a frequency of 2048 Hz. We defined a vector with 6000 elements to store the counter values immediately after calling the FIQ and IRQ interrupts. The clock value assigned to the TC channel has the highest possible frequency, namely, MCK/2.\(^\text{39}\) For our system, this corresponds to 66.5 MHz. Thus, we have a high-precision timer to record the instant
Table 2: The standard deviation and the maximal jitter value in the IRQ and the FIQ interrupts

| Test load | Standard deviation IRQ (μs) | Standard deviation FIQ (μs) | Max IRQ (μs) | Max FIQ (μs) |
|-----------|-----------------------------|-----------------------------|--------------|--------------|
| CPU       | 38.54                       | 0.877                       | 136.60       | 3.42         |
| I/O       | 10.20                       | 0.42                        | 28.60        | 2.04         |
| MEMORY    | 7.194                       | 1.03                        | 59.23        | 2.84         |
| TIMER     | 27.61                       | 1.03                        | 95.38        | 2.88         |

Abbreviations: FIQ, fast interrupt request; IRQ, interrupt request.

in which the interrupt handler starts its execution. The precision is 15.03 nanoseconds, which is equal to the clock period.

First, a consecutive subtraction of each vector element is performed to get the number of clocks within each call of the interrupt handler. Next, these values are multiplied by the clock period in order to express its value in seconds. The jitter is calculated by subtraction between these values and their average. It is worth mentioning that this proposed methodology is unfit for measuring any constant latency, since only the elapsed time between handling two consecutive interrupts is recorded. Any supposedly constant delay between the hardware requesting the interrupt and execution of the handling code would be cancelled by the subtraction and, therefore, cannot be detected this way. That is, this methodology is only able to record jitter, not absolute latency.

6.1.1 Load test

In order to stress different code paths inside the kernel, exercising different exclusion and locking mechanisms, four intensive load tests were designed to measure the jitter by using the stress-ng tool.

1. The CPU stresser loads the CPU at 95% by executing the command `stress-ng -c 1 -l 95` while running the interrupt handler at 2048 Hz.
2. The I/O stresser creates four processes to stress the hardware I/O by executing the following command, `stress-ng –ionice-class realtime –ionice-level 0 -i 10`, which flushes the cached data to disk and sets the I/O scheduling class as realtime and the priority level as the highest.
3. The memory stresser calls the mmap/munmap for 900 KB and keeps on rewriting to the allocated memory by executing the following command `stress-ng –vm-bytes 900k –vm-keep -m 1`.
4. The timer stresser creates timer events at a rate of 10 MHz by executing the command `stress-ng –timer-freq 10000000 -T 1`.

Each test was performed five times. These tests were designed considering the possibility of dispute with the interrupts to require high-level attention of the processor in order to create concurrency with the interrupt handler. Figure 1 shows the distribution of the jitter in logarithmic scale for the worst result among the five executions of the four test loads. Note that the scale of the horizontal axis is different for the IRQ and FIQ plots. Table 2 summarizes the test results. The FIQ interrupt provided much lower jitter than the IRQ for all tests. However, values virtually equal to zero were expected for the execution of the load tests because the kernel code does not disable the FIQ interrupt and it has higher priority than the IRQs. These noneffectively zero jitter values are probably related to core hardware operations, which delay the FIQ interrupt response.

Running the IRQ and the FIQ interrupts at an acquisition frequency of 2048 Hz (ie, a period of 488 microseconds) showed that the worst case for the IRQ was the CPU stress test (Figure 1A), which presented 27.99% of wasted acquisition time, due to the jitter. For the FIQ this rate was only 0.70% (Figure 1B). This means that the FIQ interrupt allowed a 97.49% reduction of the interrupt jitter. Therefore, the FIQ usage provided a better performance than the IRQ by a drastic reduction of the wasted acquisition time. High jitter values for the IRQ interrupt were expected since the kernel disables interrupts in some parts of the code to keep the data structure’s consistency; for example, the scheduler code in the file `kernel/sched/core.c`. The rise of the interrupt latency exposed during the load test is related to the increase of code disabling interrupts. Consequently, it is expected that certain tasks may increase the latency when compared to idle values.
FIGURE 1  Jitter measuring for the IRQ, on Linux PREEMPT-RT, and the FIQ, on a standard Linux, concerning the four tests load, namely, CPU stress test (A and B), I/O stress test (C and D), memory stress test (E and F), and timer stress test (G and H). On the left column, the IRQ data are shown (A, C, E, and G). On the right column, the FIQ data are shown (B, D, F, and H).
### 6.2 Performance assessment of data acquisition of an instrumented PIG

In order to evaluate the data acquisition performance, it is convenient to define two quantities, namely the effective time (ET) and the AT. The latter is the overall AT to perform the acquisition, and the former is the portion of AT in which the system is effectively acquiring data. Thus,

\[ ET = AT - T_{\text{overhead}} \]  

where \( T_{\text{overhead}} \) is the overhead time. For a system with periodic data acquisition, we can define the AT as the acquisition period, that is, \( AT = 1/f \), where \( f \) is the acquisition frequency, in hertz. The relative effective time, \( T_{\text{Rel}} \), is defined as the ET to AT ratio. \( T_{\text{Rel}} \) quantifies how much the software takes advantage of the AT for the acquisition. Using Equation 1, the theoretical value for the \( T_{\text{Rel}} \) is given by

\[ T_{\text{Rel, Theo}} = \frac{AT - T_{\text{overhead}}}{AT} = 1 - \left( \frac{T_{\text{overhead}}}{f} \right) . \]  

Note that, ideally, in the absence of the overhead time, \( T_{\text{Rel}} \) equals to 1, which implies that the software takes advantage of 100% of AT to complete the data acquisition.

Experimentally, the relative effective time can be calculated through:

\[ T_{\text{Rel, Exp}} = \frac{f \times N}{f_S} . \]  

where \( N \) is the number of samples acquired at each acquisition, and \( f_S \) is the system sampling rate, in samples per second. Note that, as for the theoretical effective time, ideally, the product of the acquisition frequency by the number of samples should result in the sampling rate of the A/D. Nevertheless, jitter and latency negatively influence the time that the system is capable to spend acquiring data. Hence, it is expected that \( f \times N \leq f_S \) and, therefore, \( T_{\text{Rel, Exp}} \leq 1 \). In our system, we use an A/D converter with 12-bits precision and \( f_S = 480 \) k samples/s to convert the analog signal obtained from the sensor channels. Four acquisition frequencies were used, namely, 256 Hz, 512 Hz, 1024 Hz, and 2048 Hz. For each acquisition frequency, we have experimentally tried the maximum number of sensor channels (\( N \)) that we may use with the A/D converter without overrun errors, for both timer IRQ and FIQ interrupts. \( T_{\text{Rel, Exp}} \) is summarized in Table 3 for the FIQ and IRQ interrupts.

In order to calculate the theoretical value \( T_{\text{Rel, Theo}} \), we consider as part of the \( T_{\text{overhead}} \), the maximum jitter value shown in Table 2, that is, 136.60 microseconds and 3.42 microseconds for the IRQ and FIQ interrupts, respectively. The interrupt handler executes a code snippet needed for specific settings in our architecture before starting the acquisition. Hence, it was necessary to compute this fixed latency to add it to the \( T_{\text{overhead}} \). This latency was measured with the help of an oscilloscope. One oscilloscope channel was used to show when the interrupt handler was serviced by the processor and another channel was used to show when the A/D clock started to be serviced by the interruption code. In both cases, we control a GPIO pin to raise the measured level. The difference in time between the moment when the interrupt is serviced and the moment the A/D clock starts is the code latency. Considering the worst-case scenario, we measured 16.55 microseconds of latency.

Figure 2 shows the theoretical and experimental relative effective time, expressed as percentages, using solid lines and dot symbols, respectively. The linear fit of the experimental points is shown by the dotted lines. The coefficient of

| \( f \) (Hz) | AT (μs) | ET (IRQ) (μs) | N (IRQ) | \( T_{\text{Rel, Exp}} \) (IRQ) | ET (FIQ) (μs) | N (FIQ) | \( T_{\text{Rel, Exp}} \) (FIQ) |
|---|---|---|---|---|---|---|---|
| 256 | 3906.25 | 3729.17 | 1790 | 95.47% | 3822.92 | 1835 | 97.87% |
| 512 | 1953.12 | 1791.67 | 860 | 91.73% | 1897.92 | 911 | 97.17% |
| 1024 | 976.56 | 845.83 | 406 | 86.61% | 931.25 | 447 | 95.36% |
| 2048 | 488.28 | 354.17 | 170 | 72.53% | 447.92 | 215 | 91.73% |

Abbreviations: FIQ, fast interrupt request; IRQ, interrupt request.
determination of the linear fit is 0.95294 and 0.9804 for the FIQ and IRQ approaches, respectively, which indicates that the experimentally measured relative effective time is highly linear, as expected by the theoretical relationship of Equation 2. The linearity of $T_{\text{Rel}}$ decreases with the frequency increase because the jitter and latency code snippets are not influenced by the value of acquisition frequency.

Hardware operations, not assessed in this study, can result in additional overhead which was not considered in Equation 2. This reflects the difference between the theoretical slope value and the experimental slope value in Figure 2. In general, the FIQ approach is more efficient since the system is acquiring data more than 90% of the time at all the evaluated acquisition rates. Moreover, it allowed us to increase the acquisition frequency up to 2048 Hz, which was not possible using an IRQ. At this frequency, $T_{\text{Rel}}^{\text{Exp}}$ is excessively low with IRQ (72.53%), due to the high amount of wasted time due to jitter.

7 | CONCLUSION

A new mechanism that employs static mapping for peripherals and changes on the Linux kernel was proposed and evaluated in order to allow the FIQ interrupt handler to be written in the C language. Its performance was assessed by comparing it with a timer IRQ on Linux PREEMPT-RT in full CONFIG_PREEMPT_RT mode. The proposed approach is suitable for systems with exactly one real-time task and that do not use the ITCM area.

The results showed that the FIQ approach achieved a 97.49% reduction of the interrupt jitter when compared to the load test file transfer applied to a timer interrupt (IRQ). Thanks to the large jitter reduction, the system became highly efficient. For instance, at 1024 Hz, the system can use 86.61% of the AT for data acquisition using IRQ, while with the FIQ approach the utilization is up to 95.36%. This shows an increase of 9.17% in the use of acquisition time. Furthermore, it becomes feasible to set the acquisition frequency up to 2048 Hz since the system takes advantage of 91.73% of AT instead of just 72.53%. It is worth to emphasize that in a moving acquisition system such as a PIG, to be able to increase the acquisition frequency is an extremely welcome feature. The results indicate that the FIQ approach deserves to be considered for real-time applications since its efficiency was high enough to achieve real-time driver performance without the need to apply real-time extensions to the Linux kernel.

CONFLICT OF INTEREST

The authors declare no potential conflict of interest.

DATA ACCESSIBILITY

The data that support the findings of this study are openly available in kernel_4.9.66_fiq at https://bitbucket.org/rgsouza/kernel_4.9.66_fiq/src/master/, reference number [35].

ORCID

Alan C. Kubrusly

https://orcid.org/0000-0002-8633-4903
REFERENCES

1. Deng L, Zhao X, Qi C, Gao F. A real-time walking robot control system based on Linux RTAI. Paper presented at: Proceedings of the 2013 International Conference on Advanced Mechatronic Systems; 2013, 530-534. https://doi.org/10.1109/ICAMechS.2013.6681701

2. Marinoni M, Facchinetti T, Buttazzo G, Franchino G. An embedded real-time system for autonomous flight control. Paper presented at: Congress of ANIPLA on Methodologies for Emerging Technologies in Automation (ANIPLA 2006); 2006.

3. Hamayun MM, Spyridakis A, Raho DS. Towards hard real-time control and infotainment applications in automotive platforms. Paper presented at: Proceedings of the 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, 2014; Madrid, Spain, 39-44.

4. Kirner R. The WCET analysis tool CalcWcet167. Paper presented at: International Symposium on Leveraging Applications of Formal Methods, Verification and Validation; 2012, 158-172.

5. Altenber J. Using the realtime preemption patch on ARM CPUs. Paper presented at: Proceedings of the 11th Real-Time Linux Workshop; 2009, 229-236.

6. Hoang H, Buttazzo G. Reducing delay and jitter in software control systems. Paper presented at: Proceedings of the 15th International Conference on Real-Time and Network Systems; 2007.

7. Marti P, Fuertes JM, Fohler G, Ramamirtham K. Jitter compensation for real-time control systems. Paper presented at: Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420); 2001, 39-48. https://doi.org/10.1109/REAL.2001.990594.

8. Proctor F, Shackleford W. Real-time operating system timing jitter and its impact on motor control. Paper presented at: Proceedings of SPIE - The International Society for Optical Engineering; 2001, 4563. https://doi.org/10.1117/12.452653.

9. Moryc P, Černohorský J. Task jitter measurement under RTLinux operating system. Proceedings of the International MultiConference on Computer Science and Information Technology; 2007, 849-858.

10. Barbian A, Beller M. In-line inspection of high pressure transmission pipelines: state-of-the-art and future trends Paper presented at: Proceedings of 18th World Conference on Nondestructive Testing; 2012.

11. Pigg Products and Services Association. An Introduction to Pipeline Pigging. Oxford, UK: Gulf Professional Publishing; 1995.

12. Barbian A, Beller M, Hartmann S, Schneider U. High resolution ultrasonic in-line inspection: added value and special applications. Paper presented at: PTC 6th Pipeline Technology Conference; 2011.

13. Camerini C, Marinho C, Franzoi A, Ibajé S, Freitas M, Weid J. Feeler pig - a new tool for multi-size pipeline inspection. European Conference Federation for Non-Destructive Testing-ECNDT; 2006; Berlin, Germany.

14. Reghennazi F, Massari G, Fornaciari W. The real-time linux kernel: a survey on preempt_RT. ACM Comput Surv. 2019;52:1-36. https://doi.org/10.1145/3297714.

15. Yang J, Chen Y, Wang H, Wang B. A Linux kernel with fixed interrupt latency for embedded real-time system. Paper presented at: Proceedings of 2nd International Conference on Embedded Software and Systems (ICESS‘05); 2005, 8. https://doi.org/10.1109/JICESS.2005.3.

16. Real Time Linux. https://wiki.linuxfoundation.org/realtime/start. Accessed July 18, 2018.

17. Gerum P. Xenomai - implementing a RTOS emulation framework on GNU/Linux. https://www.xenomai.org/documentation/xenomai-2.0/pdf/xenomai.pdf; 2004. Accessed August 1, 2018.

18. Mantegazza P, Dozio L, Papacharalambous S. RTAI: real time application interface. Linux J. 2000;2000:10.

19. Dozio L, Mantegazza P. Real time distributed control systems using RTAI. Paper presented at: 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing; 2003, 11-18. https://doi.org/10.1109/ISORC.2003.1199229

20. Barbalace A, Luchetta A, Manduchi G, Moro M, Soppelsa A, Tallercio C. Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application. IEEE Trans Nucl Sci. 2008;55(1):435-439.

21. Lelli J, Scordino C, Abeni L, Faggioni D. Deadline scheduling in the Linux kernel. Softw Pract Exp. 2016;46:821-839. https://doi.org/10.1002/spe.2335.

22. Biondi A, Marinoni M, Buttazzo GC. Challenges in virtualizing safety-critical cyber-physical systems. Paper presented at: Proceedings of Embedded World Conference; 2018.

23. Lankes S, Wassen G. Bare-Metal execution of hard real-time tasks within a general-purpose operating system. Paper presented at: Proceedings of 15th International Workshop on Worst-Case Execution Time Analysis (WCET); 2015.

24. Brown JH, Martin B. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. Paper presented at: Proceedings of the 12th Real-Time Linux Workshop (RTLWS’12), 2010.

25. Arm J, Bradac Z, Kaczmarczyk V. Real-time capabilities of Linux RTAI. IFAC-PapersOnLine. 2016;49:401-406.

26. About LinuxCNC. http://linuxcnc.org/docs/html/getting-started/about-linuxcnc.html; Accessed July 18, 2018.

27. Zhang P, Li H, Gao Z. PIL: a method to improve interrupt latency in real-time kernels. Paper presented at: Proceedings of 2009 International Conference on Scalable Computing and Communications; 8th International Conference on Embedded Computing; 2009, 75-80. https://doi.org/10.1109/EmbeddedCom-ScalCom.2009.24.

28. PREEMPT_RT patch versions https://wiki.linuxfoundation.org/doku.php?id=realtime:preempt_rt_versions&do=. Accessed September 12, 2018.

29. Andrew NS, Dominic S, Wright C, John R. ARM system developer's guide. Designing and Optimizing System Software. San Francisco, CA: Elsevier Inc.; 2004.

30. LTD ARM. ARM generic interrupt controller architecture specification https://static.docs.arm.com/ihi0069/d/IHI0069D_gic_architecture_specification.pdf. Accessed June 25, 2018.

31. Olsen Karl. http://thread.gmane.org/gmane.linux.ports.arm.kernel/34562; 2007. Accessed June 7, 2017.
32. ARM. ARM developer suite developer guide http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/Cacbhjji.html. Accessed June 25, 2018.
33. STMicroelectronics. Real-Time Performance Using FIQ Interrupt Handling in SPEAr MPU. Appl. Note AN3129. 1st ed.; 2010. https://www.st.com/content/ccc/resource/technical/document/application_note/a4/a5/00/8b/7c/7c/c8/c8/CD00259791.pdf/files/CD00259791.pdf/jcr:content/translations/en.CD00259791.pdf.
34. Bovet D, Cesati M. Demand paging. Understanding the Linux Kernel: From I/O Ports to Process Management. Sebastopol, CA: O’Reilly Media, Inc.; 2005:385-388.
35. Rayanne S. Kernel_4.9.66_fiq. https://bitbucket.org/rgsouza/kernel_4.9.66_fiq/src/master/; 2019. commit: c0db1bfa5a7946cf8f2075b97d6f12018d2dc61e.
36. Kroah-Hartman G, Rubini A, Corbet J. Allocating memory. Linux Device Drivers: Where the Kernel Meets the Hardware. Sebastopol, CA: O’Reilly Media; 2005:213-234.
37. NXP Semiconductors. i.MX51 Board Initialization and Memory Mapping Using the Linux Target Image Builder (LTIB). Appl. Note AN3980. 1st ed. Austin, Texas: Multimedia Application Division Freescale Semiconductor, Inc.; 2010.
38. AMBA. HB example AMBA system technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0170a/DDI0170.pdf; 1999. Accessed August 05, 2019.
39. Microchip Technology. Atmel | SMART ARM-Based Embedded MPU. 2015. SAM9G25 Datasheet. San Jose, CA: Atmel Corporation. http://ww1.microchip.com/downloads/en/devicedoc/Atmel-11032-32-bit-ARM926E-J-S-Microcontroller-SAM9G25_Datasheet.pdf
40. Stress-ng, general commands manual. https://github.com/ColinIanKing/stress-ng. Accessed September 10, 2019.
41. Eisenhauer J. Regression through the origin. Teach Stat. 2003:25:76-80. https://doi.org/10.1111/1467-9639.00136.

How to cite this article: Souza R, Freitas M, Jimenez M, Magalhães J, Kubrusly AC, Rodriguez N. Real-time performance assessment using fast interrupt request on a standard Linux kernel. Engineering Reports. 2020;2:e12114. https://doi.org/10.1002/eng2.12114