electronics

MDPI

Article

Interrupt Latency Accurate Measurement in Multiprocessing Embedded Systems by Means of a Dedicated Circuit

Sara Alonso *, Leire Muguira, José Ignacio Garate, Carlos Cuadrado and Unai Bidarte

Department of Electronics Technology, University of the Basque Country (UPV/EHU), 48013 Bilbao, Spain; leire.muguira@ehu.eus (L.M.); joseignacio.garate@ehu.eus (J.I.G.); carlos.cuadrado@ehu.eus (C.C.); unai.bidarte@ehu.eus (U.B.)

* Correspondence: sara.alonso@ehu.eus

Abstract: Modern multiprocessing embedded applications require, in many cases, two different environments on the same platform: one that meets real-time requirements and another one with a general purpose operating system. Although several technologies can be used, two of the most popular are virtualization based on hypervisors and asymmetric multiprocessing software. However, using these tools introduces latency, which must be measured to verify compliance with real-time requirements. With the aim of facilitating these measurements, this work provides a hardware tool that is more precise and easier to use than other existing software solutions. The paper also studies the interrupt latency generated by different hypervisors and asymmetric multiprocessing frameworks in a Zynq UltraScale+ platform. This research work facilitates the accurate study of the temporal response of multiprocessor embedded systems, which allows for evaluating their suitability for applications with real-time requirements.

Keywords: virtualization; hypervisor; Xen; Jailhouse; OpenAMP; latency; real time; interrupt latency measurement

Citation: Alonso, S.; Muguira, L.; Garate, J.I.; Cuadrado, C.; Bidarte, U. Interrupt Latency Accurate Measurement in Multiprocessing Embedded Systems by Means of a Dedicated Circuit. *Electronics* **2024**, *13*, 1626. https://doi.org/10.3390/electronics13091626

Academic Editor: Alexander Barkalov

Received: 12 March 2024
Revised: 20 April 2024
Accepted: 21 April 2024
Published: 24 April 2024

1. Introduction

Real-time embedded systems play an essential role in today's industry. Many applications require the simultaneous use of two or more environments. It is common to combine a general purpose environment with a time-critical environment. This work focuses specifically on these scenarios.

Modern platforms have a heterogeneous processing system with some specific cores to process hard real-time applications—a Real-Time Processing Unit (RPU)—and some cores to process other applications—an Application Processing Unit (APU). Moreover, some software tools make it possible to run Real-Time Operating Systems (RTOSs) and General-Purpose Operating Systems (GPOSs) at the same time in homogeneous or heterogeneous processors.

Hypervisors are one of those tools. They virtualize the hardware resources so that different isolated environments can use them simultaneously. Virtualization is a widespread technique in computing and cloud applications due to its benefits: saving resources, increasing security, running several isolated environments with different Operating Systems (OSs), etc. However, its use in embedded systems has been considered only in recent years [1].

Other popular tools with a similar purpose are Asymmetric Multiprocessing (AMP) multicore frameworks, such as OpenAMP [2], that allow for running an RTOS on the RPU while a GPOS—e.g., the Linux OS—runs on the APU. These frameworks do not virtualize the CPU or isolate the environments as hypervisors do, but they allow for taking advantage of all the resources of the platform by using both processing units to run different environments simultaneously and to facilitate communication between them.

However, all this software may increase the latencies of the system, since they constitute extra layers of software. Measuring those latencies is essential for applications with hard real-time constraints, but it is a challenging task. Many studies in the literature analyze them in computing or cloud environments, and some propose tools, such as Cyclictest [3] or Thread-Metric [4], to facilitate the measuring process. In recent years, researchers have begun to conduct some timing analysis in multiprocessing embedded systems. Nevertheless, this is a largely unexplored field, especially for hardware-made measurements.

System-on-Chip (SoC) devices that have an FPGA usually have an embedded logic analyzer to make time measurements by hardware more accurately than by software. However, the logic analyzers are limited by the finite number of samples they can show—although they can be configured to only log changes so more samples can be caught, but in that case, a counter should be added to the signals—and, on the other hand, the communication protocols and the analysis software are not specifically developed for studying these latencies. Even with the maximum sample data depth, the process has to be repeated to obtain a significant number of measurements. Moreover, increasing the sample data depth in the logic analyzer increases the memory resources. In addition, the results must be processed manually, as the exported data are not easy to use. For this reason, in this paper, a new hardware circuit has been designed to facilitate the task of measuring latencies and managing this data in software.

A study of interrupt latencies introduced by different hypervisors and an AMP framework is also carried out in this paper, since this is one of the most critical parameters in hard real-time embedded systems, as the system's response time depends significantly on it [5]. In most scenarios, two environments are set—one running a GPOS and the other running real-time software—and there are also some control scenarios with only the hard real-time environment to compare the influence of hypervisors and AMP frameworks with them. The latencies produced in the real-time environment are studied to evaluate its real-time behavior. These measurements have been implemented on the Zynq UltraScale+ MPSoC ZCU102 platform using the hardware circuit proposed in this paper.

This paper is organized as follows. Section 2 explains basic concepts about the different techniques to run two distinct environments on a single device and the latencies in virtualized environments. Next, Section 3 shows the state of the art of timing analysis in multiprocessing embedded systems and some techniques designed for it. Section 4 presents the proposed circuit to measure interrupt latencies. The last Section 5 explains the methodology followed in this work to perform the timing analysis and summarizes the results. Finally, Section 6 concludes the paper.

2. Background

2.1. Multiprocessor System on Chip (MPSoC)

SoC technology is widely used due to its flexibility and reliability. Some SoCs have a Field Programmable Gate Array (FPGA) to reconfigure the hardware design and adapt the system to the constraints of each moment. Some modern platforms also have multiple processing units, thus increasing the reliability and throughput and decreasing the cost. These systems are called Multiprocessor Systems on Chips (MPSoCs). In these systems, the Programmable Logic (PL) encompasses the hardware components, including the FPGA and other circuits. On the other hand, the Processing System (PS) handles the software aspects, which are executed by the processors and their associated peripherals. Both sides, PL and PS, are connected and can share data at high speed—the recommended frequency for the AXI interface that connects the PS and PL is up to 200 MHz.

From a software perspective, multiprocessors can be Symmetric Multiprocessors (SMPs) if a single OS runs on all the cores and the work is divided between them, or they can be AMPs if each core has different applications or OSs. In the first case, all the processors can communicate with each other, and it is usually implemented when an application needs more CPU power to manage its workload. In the other case, processors contain a master–

slave relationship, and it is generally implemented when different CPU architectures or OSs are needed for a specific application.

Each processor can have multiple cores—i.e., multicore systems. From a hardware perspective, multicore designs can be homogeneous if the architecture of all the cores is the same or heterogeneous or if there are cores with different architectures. In heterogeneous multicore systems, both AMP and SMP architectures can be employed, whereas in homogeneous multicore systems, an SMP is the only viable option.

MPSoCs are widely used in applications that require two environments with different OSs. Avionics, autonomous driving, and the medical field are examples where more than one OS may be needed. For example, the RETINA project [6] provides time-critical, predictable, and reliable communication for automotive applications where two environments with different criticality levels are executed. Embedded 5G edge computing systems, computation-intensive AI, or Electronic Control Units (ECUs) are other examples [7].

2.2. Virtualization

Virtualization creates a representation of hardware resources to let different environments share them. It saves physical resources and energy consumption, and it makes the management of the applications easier.

There are three virtualization levels [8]:

- Full or Hardware Virtualization Machine (HVM): It makes it possible to run an OS inside a virtual machine, since the hardware architectures have the needed support for virtualization.
- Paravirtualization (PV): Some of the privileged instructions of the OS kernel are replaced by calls to the hypervisor.
- Static partitioning or core virtualization: It is a combination of the previous ones.

Virtualization is based on a hypervisor. ARM supports four exception levels (Figure 1), thus determining the processor's privilege and execution state. A hypervisor is a piece of software running at Exception Level 2 (EL2), which is used as the interface between the OS and the hardware. It allows for running multiple Virtual Machines (VMs) with different OSs and applications and distributes physical resources between them. A fundamental characteristic of hypervisors is the isolation between the guests, which increases the security of the system. The most common hypervisors in MPSoCs are type 1 and type 2. The aforementioned types constitute a classification based on where the hypervisors are executed: while the first runs directly on hardware, the second runs on a host OS.

Figure 1. Exception Levels (ELs) on ARMv8 architectures.

2.3. Xen Hypervisor

Xen is a widespread and well-documented open source hypervisor. It runs directly on hardware—i.e., it is a type 1 hypervisor—and combines full virtualization and paravirtualization to achieve the best performance.

Xen guests or VMs are called domains. Each domain runs an isolated OS or application. Dom0 is the first domain started by the hypervisor, and it runs a Linux OS. This domain can access the hardware resources and can manage other domains with XenControlTools. The other domains are called DomU, and they do not have privileges—i.e., they cannot

directly access the hardware resources. They can run any OS, but they can only be created while Dom0 is running. In I/O operations, Xen uses grant tables to share or transfer memory pages between domains. It marks these I/O requests by the event channels using a technique called hypercall. When the domain is scheduled, it checks the event channels and delivers the pending events by calling the corresponding interrupt handler [9].

Each domain has one or more virtual CPUs (vCPUs) assigned, and the guest's OS sees each vCPU as a single physical CPU. Each core can create eight vCPUs. Xen's Credit Scheduler synchronizes all the vCPUs with a fair share algorithm based on proportional scheduling [9]. Xen usually allocates just one vCPU to each domain, thus containing the information related to scheduling and event channels. Figure 2 [10] shows the architecture of Xen.

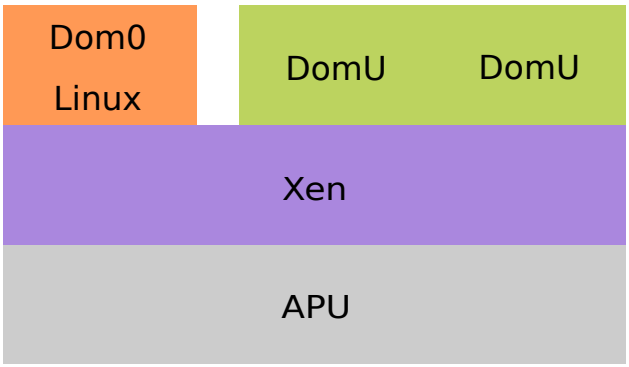

Figure 2. The architecture of the Xen hypervisor. Adapted with permission from Ref. [10]. 2023, Sara Alonso (979-8-3503-0385-8/23/$31.00 ©2023 IEEE).

2.4. Jailhouse Hypervisor

Jailhouse is a partitioning—i.e., it combines HVM and PV—open source hypervisor based on Linux, and it is a type 1 hypervisor. It allows for running bare metal applications or adapted OSs besides Linux.

Jailhouse is lightweight, as it aims for simplicity rather than feature richness. Compared to other hypervisors, it does not support the overcommitment of resources. For example, it does not offer vCPUs to guest systems. Thus, it does not need a scheduler to manage the resources and only virtualizes the software resources required. Instead of emulating the hardware resources as other hypervisors do, Jailhouse divides hardware into isolated compartments called "cells", so they are fully dedicated to guests called "inmates", which can control the resources assigned to the cell. The hypervisor manages the cells and ensures the isolation between them. There is a Linux root cell, which is the first started by the hypervisor. This cell has a kernel module and some tools to create, run, stop, and destroy other cells. Figure 3 shows the architecture of Jailhouse.

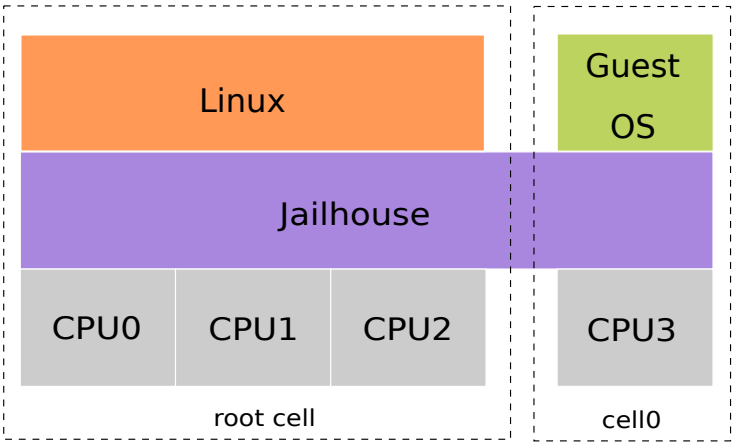

Figure 3. The architecture of the Jailhouse hypervisor.

2.5. OpenAMP

OpenAMP is not a hypervisor, as it does not virtualize the CPU or isolate the different VMs. It just uses virtualization for some memory addresses. However, it is interesting to measure its latencies too, because it is also used for running multiple environments, and likewise an extra software may introduce latencies.

Typically, hypervisors are used for running different OSs on homogeneous cores. OpenAMP, instead, is used for running different OSs on heterogeneous cores. It is an open source AMP multicore framework, as it uses an AMP software architecture. The most typical scenario in AMP systems is having Linux on the APU and an RTOS or a bare metal application running on the RPU.

When independent software stacks run on the processor cores, it is helpful to standardize how the environments interact. OpenAMP is a solution for doing so. It defines mechanisms to manage the life cycle of the remote processor—load/start/stop—and message passing between cores. It also allows for configuring the environments, sharing resources between environments, and porting any OS on top of a standardized abstraction layer.

OpenAMP can handle interrupts, access devices, and manage the memory using the Libmetal library. This library also provides Application Programming Interfaces (APIs) for synchronization primitives. There are three fundamental modules in the OpenAMP architecture which use Libmetal and are implemented in the upstream Linux kernel [11]:

- VirtIO: A virtualized communication standard for network and disk device drivers. It is an abstraction layer over devices in a paravirtualized hypervisor that manages the shared memory for OS interactions.
- Remoteproc: Allows a Linux master to manage remote processors—this allows the LCM of the slave processors. It allocates system resources and creates VirtIO devices.
- RPMsg: Provides Interprocess Communication (IPC) between master and remote processors.

To startup remote processors, first, OpenAMP assumes the master is already running, and the remote processor is waiting or powered down. Then, the OpenAMP master loads the remote processor firmware into the memory location. Finally, the OpenAMP master starts the remote processor, waits for it, and establishes a communication channel with it (Figure 4 [10]).

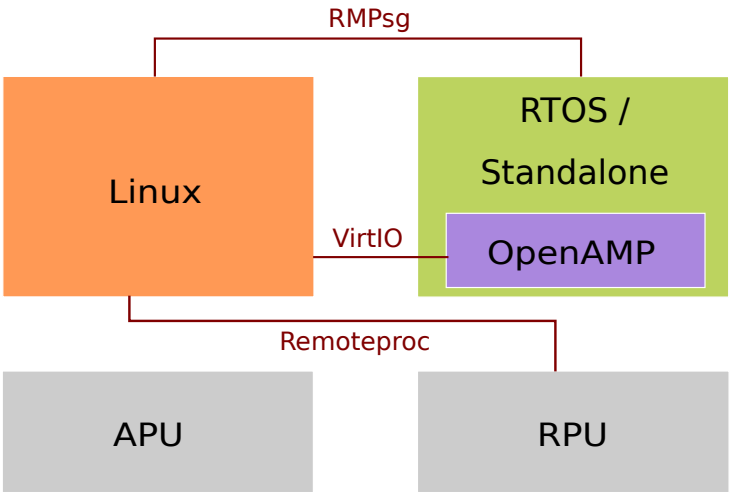

Figure 4. The architecture of the OpenAMP framework Adapted with permission from Ref. [10]. 2023, Sara Alonso (979-8-3503-0385-8/23/$31.00 ©2023 IEEE).

OpenAMP can be complementary to hypervisors. For instance, Cinque et al. [1] proposed using Xen and OpenAMP combined to share RPU resources between VMs that run over APUs—i.e., they used OpenAMP to use the RPU while multiple VMs were running on the APU thanks to a hypervisor.

To summarize, OpenAMP provides a software framework for developers to enable MPSoC LCMs, loads firmware across a multiprocessor system, and establishes communication between the processors.

2.6. Latencies in Embedded Systems

Hard real-time systems must meet specific timing requirements. They have to produce the expected result in a specific time constraint, coordinate independent clocks, and operate synchronously [12]. To evaluate the predictability of the system, the latency—i.e., the time between an event and the response to that event—and the jitter—i.e., the latency variation between iterations—must be studied. Usually, applications aim to achieve low and deterministic latency and bounded jitter. Latencies are further described in the following:

- Latencies in OSs: The most significant latency of an OS is the kernel latency [13]. In Figure 5, a task is running when an interrupt happens, thus indicating an external event has occurred. The task goes to standby mode, and the interrupt is handled. Then the scheduler is called, and it decides which task runs next—it can be the previous one or a different one depending on the priority of each task. When the scheduler finishes, the task runs.
  In Figure 5, the interrupt latency (1) refers to the duration from the generation of an interrupt to its handling, i.e., the first Interrupt Service Routine (ISR)'s instruction is executed. The handler duration (2) is the time spent in the ISR. The scheduler latency (3) is the time between a stimulus, which indicates if an event has happened, and the kernel scheduler being able to schedule the thread that is waiting for the stimulus to happen. The scheduler duration (4) is the time spent, inside the scheduler, to decide what thread to run and switch the context to it.
- Latencies in MPSoCs: Interrupt latency is one of the main latencies in MPSoC technology, as explained later. It depends on different factors such as interrupt controller implementation, low-level software architecture, OSs, middleware stacks, peripheral-specific interrupt handling requirements, the priority of the interrupt, and interrupt handler implementation [14]. In MPSoC technology, the PL and PS are communicated using AXI buses, and data exchange transactions may also increase latency [15]. Network congestion can also create some latency [16].
- Latencies with hypervisors and AMP frameworks: On the one hand, a hypervisor increases the OS's kernel latency—especially interrupt latency—and the network latency. It virtualizes the physical resources so that all the guests can share them, and this increases the latency, which stands out in the case of the network. It also affects the shared memory, since this is also virtualized [17]. Regarding the interrupt latency, the physical interrupt controller is used by the hypervisor, and the OSs of the VMs cannot access it. Instead, the hypervisor emulates a virtual interrupt controller for each VM—i.e., the interrupt source of the vCPU becomes the virtual interrupt controller [18]. Thus, hypervisors affect latency due to their mechanism to manage interrupts. On the other hand, AMP frameworks also affect the interrupt latency, as both processors need to communicate to manage interrupts. The IPC is affected because it is carried out by virtualized shared memory. It can also influence when the remote processor uses the network, as it has to ask for access to the APU.
- Interrupt latency: After identifying latencies in OSs, MPSoCs, hypervisors, and AMP frameworks, the interrupt latency comes up as one of the main latencies in embedded systems. In this work, latency is defined as the time difference between the interrupt triggering and its handling. In a virtualized system, Pavic and Dzapo [5] define it with the following Equation (1):

$$t = t_{irq} + t_{hyp} + t_{os} + t_{user} \quad (1)$$

  where t is the total latency of the interrupt, t_{irq} is the time required by the hardware to process the interrupt source and call the ISR, t_{hyp} is the latency introduced by the

hypervisor, t_{os} is the delay introduced by the OS's internal mechanisms, and t_{user} is the time for a task to be executed in response to the interrupt after it is handled. The parameter t_{hyp} can be very important, as it depends on the state and utilization of all the VMs and CPUs [10]. 979-8-3503-0385-8/23/$31.00 ©2023 IEEE.

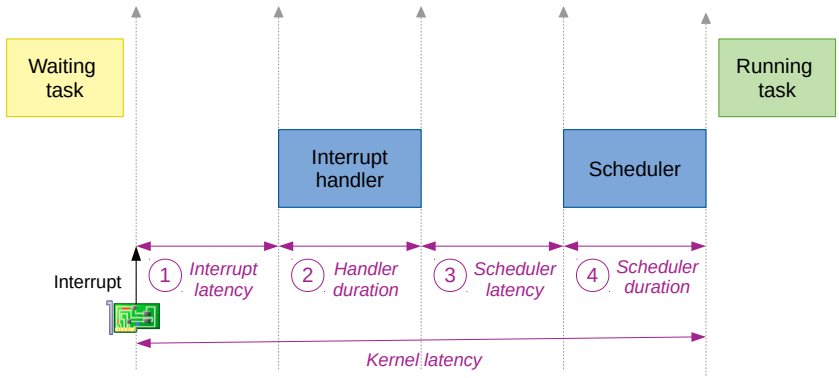

Figure 5. Kernel latency components in OSs Adapted with permission from Ref. [13]. 2023, Sara Alonso (979-8-3503-0385-8/23/$31.00 ©2023 IEEE).

3. Related Work

In the literature, there are many timing analyses of virtualized environments in the cloud or PC. For example, Abeni et al. [19] researched the latencies introduced by the Xen and KVM hypervisors, and they provided some guidelines for configuring the VMs to reduce those latencies; Tafa et al. [20] evaluated the CPU consumption, memory utilization, and transfer time in five hypervisors, and Queiroz et al. [21] compared general purpose hypervisors for hard real-time applications by measuring the latencies introduced by different hypervisors using Cyclictest—a used program to measure the period of time between when a timer expires and the kernel executes the thread set by that timer. Additionally, interrupt latency has also been studied in OSs. For instance, Macauley et al. [22] studied the speed with which the processor can respond to interrupts for members of the Intel 8086 family. Moreover, some tools for timing analysis on virtualized environments have also been proposed, such as the one designed by Xu et al. [23]. This is a cache-aware compositional analysis technique used to ensure timing compliance on a multicore virtualized platform. However, it was not designed for embedded systems. Latencies caused by OSs have been extensively investigated in embedded systems. However, in recent years, the latencies caused by virtualization in embedded systems have also begun to be studied.

Table 1 summarizes some of the most relevant works that have performed a timing analysis of an OS or a virtualized environment in embedded systems, including a brief description of the works, the hardware platform, the SW environment where the analysis was carried out, and the tool used for the analysis. Some make a timing analysis of virtualized environments [8,24,25], others analyze the interrupt latency on an OS [26,27], and others analyze the interrupt latency on virtualized environments [28–33]. Finally, some propose new tools for analyzing latencies in embedded systems [34–38].

The table shows that not many studies provide a timing analysis on virtualized environments on embedded systems. It also states that all of them use software tools for measuring latencies. Moreover, all the works that provide a tool for timing analysis are for OSs and do not specify if they could work on virtualized systems.

The RTOS also provides a tool to analyze the performance of the OS called ThreadMetric. Among the tests provided by this tool, there is one to evaluate interrupt handling. However, this is also a software tool, and it considers only software interrupts.

Thus, this paper presents a hardware circuit to measure latencies, which increases the accuracy of the measurement and provides a timing analysis of the interrupt latency in virtualized environments on a Zynq UltraScale+ platform.

Table 1. State of the art on timing analysis in embedded systems.

Description	HW Environment	SW Environment	Tool
Timing analysis on virtualized environments			
Alonso et al. [24] analyze the influence of Xen on the network connection delay and the network bandwidth.	Zynq UltraScale+ MPSoC ZCU102	Xen hypervisor with Linux guests	ping and iperf
Beckert et al. [25] provide a Worst-Case Response Time (WCRT) analysis of a sporadic server-based budget scheduling with a hypervisor.	ARM9 based LPC3250@ 200MHz	μC/OS-MMU hypervisor modified with μC/OS-II guests	PyCPA framework and Python
Sebouh et al. [8] evaluate the performance overhead introduced by different hypervisors.	banana-pi board (ARM)	Xen and Jailhouse hypervisors with Linux and Cpuburn-a8 application	Processor's internal counter
Timing analysis of interrupts on OSs			
Stangaciu et al. [26] propose an extension for FreeRTOS to guarantee the absence of task execution jitter. They also present a detailed analysis of this extension, including an analysis of interrupt latency and jitter.	EFM32_G890 _STK board	FreeRTOS	zlgLogic and Keil uVision
Liu et al. [27] propose RTLinux-THIN, a hybrid OS based on two-level hardware interrupts, and analyze and model the worst-case real-time interrupt latency for a Real-Time Application Interface (RTAI); they identify the key component for its optimization.	Platform based on Intel PXA270 processor	μC = OS-II and ARM Linux combination	mplayer, Bonnie and iperf
Timing analysis of interrupts on virtualized environments			
Alonso et al. [30] compare the influence of Xen and OpenAMP in PL-to-PS and PS-to-PL interrupts.	Zynq UltraScale+ MPSoC ZCU102	Xen hypervisor and OpenAMP with bare metal and FreeRTOS guests	Hardware ILA
Alonso et al. [28] compare the influence of Xen and OpenAMP in a PL-to-PS interrupt.	Zynq UltraScale+ MPSoC ZCU102	Xen hypervisor and OpenAMP with bare metal guests	Hardware ILA
Alonso et al. [29] analyze the influence of OpenAMP in the latencies of a PL-to-PS interrupt.	Zynq UltraScale+ MPSoC ZCU102	OpenAMP with bare metal and FreeRTOS guests	Hardware ILA
Klingensmith et al. [31] present Hermes, a hypervisor that enables standalone applications to coexist with RTOSs and other less time-critical software, on a single CPU and measure the interrupt latency.	ARM-Cortex-M CPUs	Hermes hypervisor with FreeRTOS guests	Performance counters
Garcia et al. [32] present work-in-progress results of hardware-based hypervisor implementation and study the performance of interrupt virtualization.	Xilinx ML505 board	Hardware hypervisor with AIC_IMR and HyperIMR guests	ISIM simulator and Chip-Scope
Sá et al. [33] port a hypervisor to RISC-V, which enables the interrupts, and evaluate their latency	Zynq UltraScale+ MPSoC ZCU104	Bao hypervisor with standalone guest	Timer

Table 1. *Cont.*

Description	HW Environment	SW Environment	Tool
	Tools for timing analysis		
Adam et al. [34] perform real-time measurements of Linux kernels with the PREEMPT_RT patch with new real-time software modules designed by the authors.	Raspberry Pi and BeagleBoard	Linux OS	Self-tool
Strnadel et al. [35] present a novel hybrid timing analysis technique and show its practical applicability in the area of Worst-Case Execution Time (WCET) analysis.	MSP430	FreeRTOS	Self-tool
Schliecker et al. [36] present a novel analytical approach to provide the WCRT for real-time tasks in multiprocessor systems with shared resources.	Multicore ECUs	RTOS based on the OSEK/VDX	Self-tool
Brylow et al. [37] present the Zilog Architecture Resource Bounding Infrastructure (ZARBI), a tool for deadline analysis of interrupt-driven Z86-based software, and make a deadline analysis of handling an interrupt.	Z86-based microcontroller	Bare metal	ZARBI (self-tool)
Liu et al. [38] propose a method to measure the interrupt response time.	W2 chip	Linux with real-time pre-emption patch	Timer (self-method)

4. Design of a Hardware Circuit to Measure Latency Accurately

As a main contribution of this work, a new hardware tool has been designed to measure the interrupt latency in SoC devices with an FPGA: the *Latencies IP*. This tool has been designed in VHDL and is capable of recording a large number of latency measurements, which can later be recovered and analyzed. The module has been designed to measure the latency introduced by hypervisors without interfering with the software structure of the system. While its primary purpose is interrupt latency measurement, the Latencies IP also boasts versatility, thus enabling other types of timing analysis. The aforementioned Latencies IP has been specifically designed by the authors from scratch for the task; nonetheless, the AXI Lite Slave interface comes from open source under an AMPA license.

Figure 6 shows the internal architecture of the Latencies IP, and Table 2 [10] describes all its signals and ports.

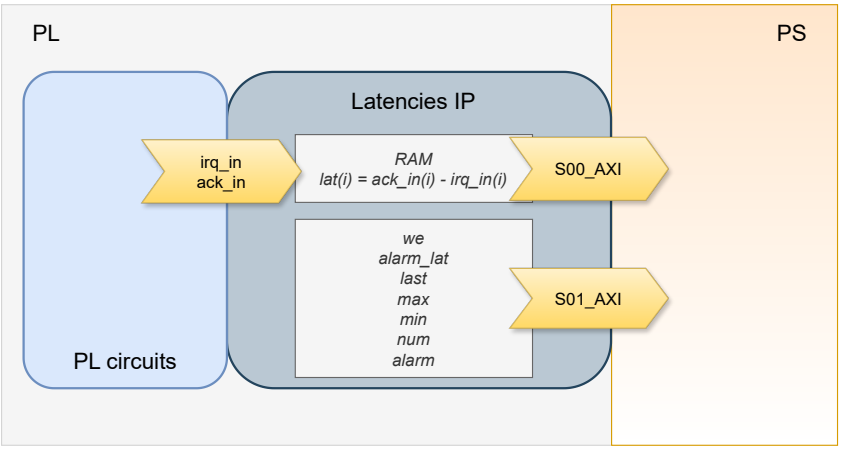

Figure 6. Interconnection and data of the Latencies IP.

Table 2. Ports and internal data of the Latencies IP core [10] *(979-8-3503-0385-8/23/$31.00 ©2023 IEEE).*

Port Signals	**In/Out**	**Description**	**Bits**
axi_s00	In	AXI slave port to read the RAM from PS by AXI.	32
axi_s01	In	AXI slave port to read other internal data from PS by AXI.	32
irq_in	In	The beginning point of the latency the user wants to measure—i.e., a signal that defines when the interrupt is set.	1
ack_in	In	The ending point of the latency the user wants to measure—i.e., a signal that defines when the interrupt is handled.	1
Internal Data	**In/Out**	**Description**	**Bits**
RAM	Out	A RAM memory which saves the measured latencies.	512 × 16
we	In	It enables the measurements.	1
alarm_lat	In	A value that the latencies must not exceed.	16
last	Out	The last value of the processed latencies.	16
max	Out	The maximum value of the latencies processed until that moment.	16
min	Out	The minimum value of the latencies processed until that moment.	16
num	Out	It indicates how many measurements have been made until that moment.	16
alarm	Out	It is activated when the calculated latency exceeds the alarm_lat value.	1

The block has two AXI Lite Slave interfaces with 32-bit data buses and two input ports, `irq_in` and `ack_in`. Two different buses are used for the measurements saved in RAM and the other data to make it easier for the user to manage them. In the IP core, the latency of each interrupt is calculated following Equation (2) and stored in a RAM memory. This RAM is connected to the S00_AXI interface so that the microcontroller can read all the data using AXI. Thus, the PS can manage the obtained data easily from software.

$$latency = ack\_in - irq\_in \tag{2}$$

Using the data stored in the RAM, the Latencies IP also calculates some output information saved in `last`, `max`, `min`, and `num` signals. It also sets the `alarm` signal when the measured latency exceeds a specific value. These signals are stored in registers of the S01_AXI interface, so the PS can also read them through AXI. The IP also has two input registers that can be written using AXI from the PS: `we`, which enables making the measurements, and `alarm_lat`, which specifies the value that activates the `alarm` signal when the measured latency is bigger.

The Integrated Logic Analyzer (ILA)—i.e., a logic analyzer core that can be used to monitor the internal signals of a design—can also produce the measurements that this block makes. However, manipulating the data exported from the ILA is complex and slow, and the number of measured interrupts is very limited. The IP block presented in this paper facilitates the process significantly. In addition, it provides some extra data when the maximum value or the alarm indicating that a latency determined by the user has been exceeded—i.e., it is possible to know if every interrupt is handled before a deadline. This tool is also an advance when compared with other tools that make the measurements by software, as hardware measurements are usually more accurate.

On the other hand, the Latencies IP differs from other software tools that measure interrupt latency, such as Cyclictest, because those tools focus on interrupts generated by PS timers, while the IP proposed in this work focuses on interrupts that pass through the PL–PS interface. In many applications, the real-time part—e.g., Time Sensitive Networking (TSN), where coprocessing circuits in the PL side are needed for time-critical processing—is implemented on the FPGA and uses interrupts and AXI signals to communicate with the processor. The Latencies IP allows for the evaluation of the processor's ability to handle these PL interrupts in real time.

Moreover, with the Latencies IP, the user can precisely choose the measurement of the interrupt latency and know more precisely what it is being measured. Therefore, the IP provides transparency in the measurements and precision of one clock cycle of the PL side, while Cyclictest's precision depends on the core's frequency—Cyclictest assumes a timer resolution of less than one microsecond [39].

Additionally, Cyclictest is designed to be used on Linux environments, and running it on other OSs requires additional effort from the user. The Latencies IP, instead, as it is installed on PL, supports any OS or software.

5. Timing Analysis of Interrupt Latency by Means of a Dedicated Circuit

A set of latency measurements have been performed using the Latencies IP in different scenarios. The Zynq UltraScale+ MPSoC ZCU102 evaluation board has been used for these experiments. This platform has been selected due to its popularity in embedded systems and its suitability to run hypervisors and AMP frameworks, as it is multicore and multiprocessor.

The IP and the interrupt latency measurement methodology proposed in the paper can be easily reused in SoC FPGA technologies (Xilinx Zynq, Intel FPGA Arria 10, or Intel FPGA Cyclone V devices) [40], where the processing system is based on ARM microprocessors, which is one of the most promising digital technologies for implementing smart controllers.

The selected platform has two major processing units in the PS: the APU with four Cortex-A53 cores running at 1.2 GHz frequency and the RPU with two Cortex-R5 cores running at 500 MHz frequency. It also has an FPGA in the PL part, which works at 100 MHz in our design.

5.1. Hardware Design

Several applications use interrupts to communicate between the PL and PS and combine hard real-time environments with soft real-time ones. In the automotive world, ECUs or autonomous driving are some examples [41,42]. Another example is audio or video stream data systems that usually work in real-time with an FIR filter in the PL [43].

There are also some applications in which data are collected from a sensor and processed in real-time, for instance, a pulse oximeter [44,45].

Based on the previous applications, a design was proposed, in which random data are periodically generated, and the processor reads it through AXI to process or use it. Although no specific use is given in this work, this data could be used for a cryptography application that needs to create random keys periodically.

The main blocks of this design are the PS and a random data generator IP core, which generates random data and stores it in a FIFO. The design follows this process:

1. A Triple Timer Counter (TTC) in the PS generates a periodic interrupt (`ps_pl_irq_ttc1_0`). The frequency between interrupts allows the system to process data in real time.
2. A circuit in the PL (`PL Random Data Generator`) handles the interrupt and produces random data, which is stored in a FIFO. The circuit generates another interrupt (`pl_ps_irq`) when the data are ready to be read.
3. The PL interrupt is handled in the PS. In the ISR, the data from the FIFO in the PL circuit (`axi_rvalid`) are read using AXI.

In this process, two latencies have been measured with the Latencies IP:

- T1—Latency in PS-to-PL interrupt: It is the latency from the moment the interrupt is generated in the PS to its handling in the PL. The time between the generation of the TTC interrupt in the PS and the generation of the interrupt that indicates that the data are ready in the PL was measured. For this measurement, the Latencies IP was connected with `irq_in` connected to `ps_pl_irq_ttc1_0`, and `ack_in` was connected to `pl_ps_irq`.
- T2—Latency in PL-to-PS interrupt: It is the time from the moment the interrupt is generated in the PL to its handling in the PS. The interrupt indicates that the data are ready in the PL. For this measurement, the Latencies IP was connected with `irq_in` connected to `pl_ps_irq`, and `ack_in` was connected to `axi_rvalid`—this signal indicates that the AXI-read transaction to read the data of the FIFO in the ISR was completed.

Figure 7 shows a diagram of the hardware design with the two Latencies IPs connected to make the measurements. Table 3 summarizes the actions that occurred in each measurement.

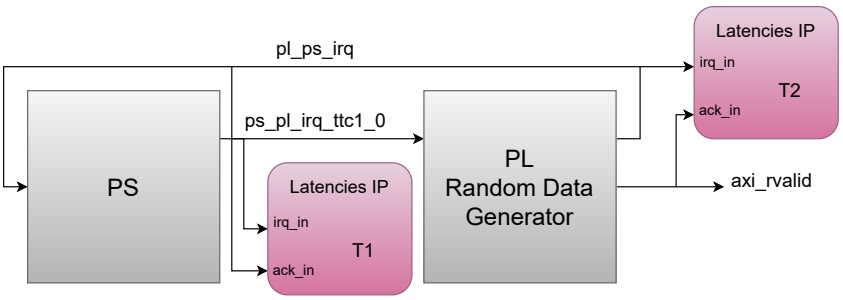

Figure 7. Connections of the Latencies IPs in the design.

Table 3. Actions included in each measurement.

Measurement	In/Actions
T1—PS-to-PL interrupt	The random data generator detects the new interrupt. The random data generator generates and stores in an FIFO the random data. The random data generator generates an interrupt indicating the data are ready to be read.
T2—PL-to-PS interrupt	The scheduler stops the running task and gives the control to the PL interrupts' ISR. The processor reads the data from the random data generator using AXI.

5.2. Software Scenarios

The experiment was carried out in eight different scenarios, so the influence of virtualization on the interrupt latency could be perceived:

1. **Standalone A53:** The measurements are made on an A53 core directly, without any OS.
2. **Xen:** The measurements are made on a Xen's domain without an OS, while Linux is running on the Dom0. The hypervisor runs on the A53 cores. Four vCPUs pinning to one physical core are used for Dom0, as less is not recommended [46], and one vCPU is pinning to another physical core for the other domain. In this implementation, Xen uses the default scheduler, i.e., the Credit Scheduler. This scenario is the same as the first but includes the Xen hypervisor.
3. **Xen stress:** Dom0 is stressed while the measurements are taken in another domain. This scenario is the same as the second, but the system is stressed.
4. **Jailhouse:** The measurements are made on a Jailhouse's cell without an OS, while Linux is running on the root cell. One different A53 core is used for each cell. This scenario is the same as the first but includes the Jailhouse hypervisor.
5. **Jailhouse stress:** The root cell is stressed while the measurements are taken in another cell. This scenario is the same as the fourth, but the system is stressed.
6. **Standalone R5:** The measurements are directly made on an R5 core directly, without any OS.
7. **OpenAMP:** The measurements are made on an R5 core with OpenAMP without an OS, while Linux is running on the four A53 cores. This scenario is the same as the sixth but includes OpenAMP.
8. **OpenAMP stress:** The APU running Linux is stressed while the measurements are taken in the RPU. This scenario is the same as the seventh, but the system is stressed.

In the scenarios with a hypervisor, the VM, where the measurements are done, accesses the hardware resources in passthrough mode to achieve lower latency.

In the stressed scenarios, the stress-ng [47] tool was used to exercise various physical subsystems and to test the performance under extreme conditions. Four stressors of all types were executed in parallel (*stress-ng -all 4*). The *-all* option includes the stressors of the vm and interrupt classes, which are the ones that can affect these scenarios the most. Repeating the measurements specifically using these stressors instead of the generic *-all* option yielded the same results.

It is worth mentioning that to compile the application programs, a common and null optimization level (-O0) was established for all cases. This decision stems from the fact that while compiler optimization levels may potentially contribute to performance enhancements, they can also introduce distortions that favor certain types of code over others, thus complicating code tracking and monitoring. Furthermore, the level of optimization does not carry a direct relationship with the object of study, since the proposed tool must measure latency, regardless of the level of optimization required when compiling.

5.3. Results

A total of 400 measurements were measured with the Latencies IP, as 385 is the value of the minimum number of samples to obtain statistically reliable results, which are obtained for a 95% confidence level—error range of 5%—assuming the population size is infinite. Equation (3) [48] was used to calculate the number of samples needed.

$$n = \frac{z^2 p(1-p)}{e^2} \tag{3}$$

where n is the sample size, z is a value obtained from the confidence level—for a confidence level of 95%, z is 1.96—p is the proportion—as this information is unknown, a 0.5 value was used—and e is the error range.

Table 4 summarizes the measurements' average, median, maximum, jitter, and deviation values. The jitter is the difference between the average value and the maximum or

minimum value—the greatest value is chosen. The deviation, instead, is calculated using Equation (4).

$$dev = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{(n-1)}} \quad (4)$$

where x_i is the value of each measurement, $\bar{x}$ is the mean value of the measurements, and n is the sample size—400 in our test.

Table 4. PL–PS interrupt latencies' measurements (ns).

	T2				
	Avg.	**Median**	**Max.**	**Jitter**	**Dev.**
Standalone A53	638.15	640.00	650.00	11.86	7.30
Xen	2319.86	2310.00	2760.00	440.14	79.20
Xen stress	2317.57	2290.00	3330.00	1012.43	74.56
Jailhouse	2084.00	2080.00	2120.00	36.00	9.32
Jailhouse stress	2078.39	2070.00	2130.00	51.61	12.99
Standalone R5	746.17	750.00	760.00	16.17	7.16
OpenAMP	2228.84	2210.00	2750.00	288.84	188.73
OpenAMP stress	2233.57	2200.00	2770.00	536.43	207.94

In the proposed application, T1 is a constant value, and there is no jitter, since it is the time that the random data generator needs to do a certain job, which is always the same. Thus, the latencies of the interrupts from PS to PL were low and deterministic. Furthermore, they are not affected by stress or the different software used to run two environments. But it might not be like that in other applications, for example, if the PL side circuit uses data from sources with variable response times. Therefore, the T2 values were analyzed, as they show the impact of the hypervisor or AMP framework, and they are the latencies that limit the real-time requirements.

Figure 8 shows graphically the statistical average and maximum data collected in Table 4 regarding the scenarios running on APU. Figure 9 shows the statistical average and maximum data regarding the scenarios running on RPU.

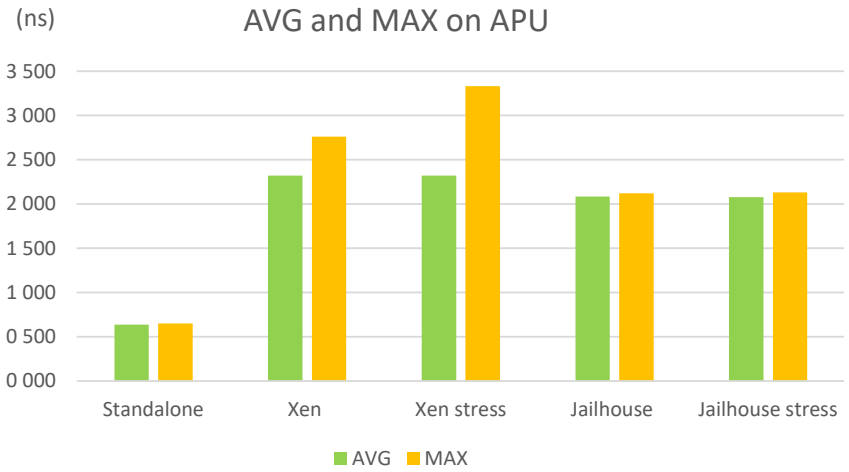

Figure 8. Comparison between the PL-to-PS latencies in each scenario running on APU.

Before analyzing the results obtained in the different setups, it is necessary to highlight the contribution of this paper in terms of the circuits, the software, and the methodology to carry out accurate latency measurements in multiprocessing embedded systems. Latencies

IPs are able to register on the PL side a large number of latency measurements without interfering in the execution of the software of the processing systems.

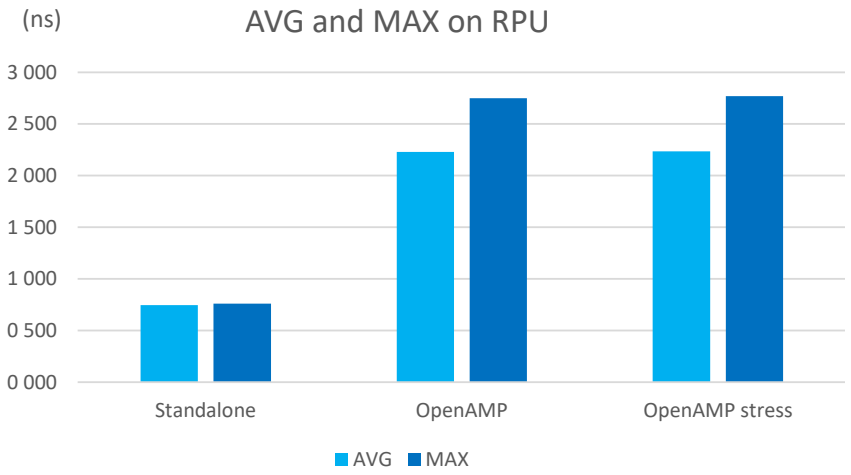

Figure 9. Comparison between the PL-to-PS latencies in each scenario running on RPU.

There are different options to access the data measured by the Latencies IP. In this case, it was decided to access them through software. As they were stored in a memory communicated using AXI, the data were easily readable from the PS. Therefore, a software program was designed to read this data using AXI, save them in an array, and, when finished, print them directly on the serial terminal. Once all the needed data were obtained, they were processed on the PC to calculate the average, median, maximum, jitter, and deviation values, and the results were graphed.

Next, we proceeded to analyze the results of the measurements. Xen, Jailhouse, and OpenAMP affected the T2 latency. As expected, Jailhouse was deterministic and did not introduce as much latency as Xen. This is because it is a partitioning hypervisor and does not need to virtualize the hardware. Moreover, the stress does not have any effect on it. The cell containing the application that handles the PL interrupt runs in one core, while the stress is applied in a different one, so the stress does not affect the execution of the application.

In contrast to this, the Xen hypervisor was less deterministic and had the highest latencies. The maximum latency was slightly higher than the average. This did not occur with Jailhouse, which always achieved similar values and very small jitter. This makes the Jailhouse hypervisor more suitable for real-time applications. In Xen's case, the stress did not affect the average and median values, but it did affect the maximum value. Some spikes appeared, which could be critical in hard real-time applications, since the worst case must be taken into account. In this scenario, the stress was applied to the Dom0 vCPUs running on one physical core, while the application that managed the PL interrupt ran on another vCPU running on a different core. Therefore, the application is isolated and should not be affected by stress, but as has been proved, it was slightly affected.

In the case of OpenAMP, the average latency it introduced was similar to the one introduced by Xen, and the maximum latency was quite bigger than this value, so this tool is also less suitable than Jailhouse for hard real-time constraints. The stress almost did not affect latency, since it was applied to A53 cores, while the application that handled the PL interrupt ran on an R5 core.

It is also noticeable that the latency was higher in the R5 cores. This is because the A53 cores work at a frequency of 1.2 GHz and the R5 cores at 500 MHz. If both cores worked at the same frequency, the results of the standalone A53 case would double those of the standalone R5 case. This is expected, because the R5s are real-time cores and usually have lower latencies.

It is worth mentioning that there are some techniques to reduce interrupt latency, such as the one presented in [14] to move the ISR to the On-Chip Memory (OCM) or some

hypervisor configuration options. However, these options are left out of the study, since the main objective of this work was to present the Latencies IP and its use in some scenarios with multiple OSs.

The comparison between the measurements made by the Latencies IP or other tools to measure interrupt latency is complicated, since each tool approaches the measurement in different ways. For example, to make Cyclictest and Latencies IP comparable, Cyclictest software should access memory space addressed on the PL side, so that the PS–PL AXI interface is activated, to achieve measurements comparable to those of the interrupts crossing the PL–PS interface, which could be an interesting future work. Regarding the precision of both tools, Cyclictest uses a high-resolution timer, while the Latencies IP has a precision of one clock cycle, which is 10 ns in our design.

In terms of resources, while Latencies IP uses few hardware resources on the PL side, Cyclictest uses PS side resources such as counters, timers, and memory space. To register 400 samples, Latencies IP only requires 94 flip-flops, 36 LUTs, and 0.5 BRAM in the Zynq UltraScale+ XCZU9EG.

6. Conclusions

The implementation of two environments that run two different OSs is becoming increasingly widespread due to the security and real-time requirements of modern applications. Some popular techniques to achieve this are hypervisor-based virtualization and AMP frameworks. However, these tools introduce some latencies that must be characterized for real-time applications.

The Latencies IP has been designed for this purpose, which is installed in the FPGA and is able to measure any latency defined by the user, such as the latency of interrupts that pass from the FPGA to the processing system and vice versa. This IP provides transparency in the measurements it performs, and its precision is one FPGA clock cycle. Moreover, the Latencies IP allows for the evaluation of the processor's ability to handle interrupts generated on the PL side, which is usual when hard real-time coprocessing circuits must communicate with the processing side. This kind of latency measurement is not done with tools in the literature like Cyclictest. In addition, it supports the use of any OS on the processor.

In addition, a comparison of the interrupt latency caused by two hypervisors—Xen and Jailhouse—and one AMP framework—OpenAMP—was carried out using the proposed IP. The Latencies IP saves a large number of measurements in an FPGA memory, which is communicated through AXI. These measurements were managed in the processor and printed. Then, the latencies were processed to obtain statistical data and compare the scenarios.

It was observed how stress affected the aforementioned software. In this study, it has been concluded that Jailhouse is the most suitable for hard real-time applications, as its maximum latency is limited. It was close to the average value, which implies that the jitter was very small, even in stress scenarios. In the case of Xen and OpenAMP however, the average latencies were higher and had a higher jitter, which could compromise compliance with hard real-time constraints. Additionally, in Xen's case, stress emphasized this condition.

The latency measurements were carried out using a hardware circuit and software for the PS side and a PC specifically designed for this purpose, which represents an advance compared to other existing tools.

In future work, using the proposed tool, more measurements will be performed with other hypervisors and different OSs. Among these scenarios, it is especially interesting to include two in which Linux runs directly in the PS, in one case in the general purpose processor and the other in the real-time processor, so that they can be compared to those carried out in this work. We also propose to expand the capabilities of the hardware module to obtain temporal analysis in more complex multiprocessing applications. Furthermore, the research is limited to theoretical analysis, and a case study could complete the work.

It would be interesting to use the Latencies IP in a specific application and compare the results with those obtained using other measurement tools.

Author Contributions: The overall execution of the research has been led by S.A.; Conceptualization, S.A., L.M. and U.B.; Methodology, S.A.; Software, S.A., J.I.G. and C.C.; Validation, S.A.; Formal Analysis, S.A., L.M., U.B., J.I.G. and C.C.; Investigation, S.A.; Writing—Original Draft Preparation, S.A.; Writing—Review and Editing, S.A., L.M., U.B., J.I.G. and C.C.; Supervision, L.M. and U.B.; Project Administration, S.A., L.M. and U.B.; Funding Acquisition, S.A., L.M., U.B., J.I.G. and C.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the Basque Government within the fund for research groups of the Basque University System IT1440-22, KK-2023/00015 and ZL-2023/00023 and by the Ministerio de Ciencia e Innovación of Spain through the Centro para el Desarrollo Tecnológico Industrial (CDTI) within the project IDI-20230111; these last two projects have been financed through the Fondo Europeo de Desarrollo Regional 2021–2027 (FEDER funds).

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author/s.

Acknowledgments: The authors would like to thank Ralf Ramsauer for his support, his kindness, and accessibility.

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

References

1. Cinque, M.; Tommasi, G.D.; Dubbioso, S.; Ottaviano, D. Virtualizing Real-Time Processing Units in Multi-Processor Systems-on-Chip. In Proceedings of the IEEE 6th International Forum on Research and Technology for Society and Industry (RTSI), Naples, Italy, 6–9 September 2021.
2. Projects, G. The OpenAMP Project. Available online: https://www.openampproject.org/ (accessed on 19 April 2022).
3. Foundation, T.L. Cyclictest. Available online: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start (accessed on 9 April 2024).
4. Lamie, W.; Carbone, J. Measure Your RTOS's Real-Time Performance. 2007. Available online: https://www.embedded.com/measure-your-rtoss-real-time-performance/ (accessed on 9 April 2024).
5. Pavic, I.; Dzapo, H. Virtualization in multicore real-time embedded systems for improvement of interrupt latency. In Proceedings of the 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 21–25 May 2018; pp. 1405–1410. [CrossRef]
6. Eurostars. RETINA. Available online: https://www.hipeac.net/network/projects/6856/retina/ (accessed on 9 June 2022).
7. Wulf, C.; Willing, M.; Göhringer, D. A Survey on Hypervisor-based Virtualization of Embedded Reconfigurable Systems. In Proceedings of the International Conference of Field-Programmable Logic and Applications (FPL), Dresden, Germany, 30 August–3 September 2021. [CrossRef]
8. Toumassian, S.; Werner, R.; Sikora, A. Performance measurements for hypervisors on embedded ARM processors. In Proceedings of the 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Jaipur, India, 21–24 September 2016; pp. 851–855. [CrossRef]
9. Zeng, L.; Wang, Y.; Feng, D.; Kent, K. XCollOpts: A Novel Improvement of Network Virtualization in Xen for I/O-Latency Sensitive Applications on Multicores. *IEEE Trans. Netw. Serv. Manag.* **2015**, *12*, 163–175. [CrossRef]
10. Alonso, S.; Lázaro, J.; Jiménez, J.; Muguira, L.; Bidarte, U. Timing requirements on multi-processing and reconfigurable embedded systems with multiple environments. In Proceedings of the 2023 38th Conference on Design of Circuits and Integrated Systems (DCIS), Málaga, Spain, 15–17 November 2023; pp. 1–6. [CrossRef]
11. Xilinx. Libmetal and OpenAMP User Guide (UG1186). 2020. Available online: https://docs.amd.com/r/2022.2-English/ug1186-zynq-openamp-gsg (accessed on 13 March 2024).
12. Intel. Real-Time Systems Overview and Examples. Available online: https://www.intel.com/content/www/us/en/robotics/real-time-systems.html (accessed on 9 March 2023).
13. EmbeddedSystem. Latency. Available online: https://hugh712.gitbooks.io/embeddedsystem/content/latency.html (accessed on 1 March 2023).
14. Xilinx. Zynq-7000 AP SoC—RealTime—InterruptLatency Reference Design and Demo Tech Tip. 2023. Available online: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842218/Zynq-7000+AP+SoC+-+RealTime+-+InterruptLatency+Reference+Design+and+Demo+Tech+Tip (accessed on 13 March 2023).
15. Marjanovic, J. Exploring the PS-PL AXI Interfaces on Zynq UltraScale+ MPSoC. 2021. Available online: https://j-marjanovic.io/exploring-the-ps-pl-axi-interfaces-on-zynq-ultrascale-mpsoc.html (accessed on 13 March 2023).

16. Bhulania, P.; R. Tripathy, M.; Khan, A. High-Throughput and Low-Latency Reconfigurable Routing Topology for Fast AI MPSoC Architecture. In *Applications of Artificial Intelligence and Machine Learning*; Choudhary, A., Agrawal, A.P., Logeswaran, R., Unhelkar, B., Eds.; Springer: Singapore, 2021; pp. 643–653.
17. Casini, D.; Biondi, A.; Cicero, G.; Buttazzo, G. Latency Analysis of I/O Virtualization Techniques in Hypervisor-Based Real-Time Systems. In Proceedings of the 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS), Nashville, TN, USA, 18–21 May 2021; pp. 306–319. [CrossRef]
18. Martins, J.; Pinto, S. Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems. In Proceedings of the 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS), San Antonio, TX, USA, 9–12 May 2023; pp. 40–53. [CrossRef]
19. Abeni, L.; Faggioli, D. Using Xen and KVM as real-time hypervisors. *J. Syst. Archit.* **2020**, *106*, 101709. [CrossRef]
20. Tafa, I.; Beqiri, E.; Paci, H.; Kajo, E.; Xhuvani, A. The Evaluation of Transfer Time, CPU Consumption and Memory Utilization in XEN-PV, XEN-HVM, OpenVZ, KVM-FV and KVM-PV Hypervisors Using FTP and HTTP Approaches. In Proceedings of the 2011 Third International Conference on Intelligent Networking and Collaborative Systems, Fukuoka, Japan, 30 November–2 December 2011; pp. 502–507. [CrossRef]
21. Queiroz, R.; Cruz, T.; Simoes, P. Testing the limits of general-purpose hypervisors for real-time control systems. *Microprocess. Microsyst.* **2023**, *99*, 104848. [CrossRef]
22. Macauley, M.W. Interrupt latency in systems based on Intel 80×86 processors. *Microprocess. Microsyst.* **1998**, *22*, 121–126. [CrossRef]
23. Xu, M.; Phan, L.T.X.; Sokolsky, O.; Xi, S.; Lu, C.; Gill, C.; Lee, I. Cache-aware compositional analysis of real-time multicore virtualization platforms. *Real-Time Syst. Vol.* **2015**, *51*, 675–723. [CrossRef]
24. Alonso, S.; Lázaro, J.; Jiménez, J.; Muguira, L.; Largacha, A. Analysing the interference of Xen hypervisor in the network speed. In Proceedings of the 2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS), Segovia, Spain, 18–20 November 2020. [CrossRef]
25. Beckert, M.; Ernst, R. Response Time Analysis for Sporadic Server Based Budget Scheduling in Real Time Virtualization Environments. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*, 161. [CrossRef]
26. Stangaciu, C.; Micea, M.; Cretu, V. An Analysis of a Hard Real-Time Execution Environment Extension for FreeRTOS. *Adv. Electr. Comput. Eng.* **2015**, *15*, 79–86. [CrossRef]
27. Liu, M.; Liu, D.; Wang, Y.; Wang, M.; Shao, Z. On Improving Real-Time Interrupt Latencies of Hybrid Operating Systems with Two-Level Hardware Interrupts. *IEEE Trans. Comput.* **2011**, *60*, 978–991. [CrossRef]
28. Alonso, S.; Lázaro, J.; Jiménez, J.; Bidarte, U.; Muguira, L. Evaluating Latency in Multiprocessing Embedded Systems for the Smart Grid. *Energies* **2021**, *14*, 3322. [CrossRef]
29. Alonso, S.; Lázaro, J.; Jiménez, J.; Muguira, L.; Bidarte, U. Evaluating the OpenAMP framework in real-time embedded SoC platforms. In Proceedings of the 2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS), Vila do Conde, Portugal, 24–26 November 2021. [CrossRef]
30. Alonso, S.; Lázaro, J.; Jiménez, J.; Muguira, L.; Bidarte, U. The influence of virtualization on real-time systems' interrupts in embedded SoC platforms. In Proceedings of the 2022 XXXVII Conference on Design of Circuits and Integrated Systems (DCIS), Pamplona, Spain, 16–18 November 2022. [CrossRef]
31. Klingensmith, N.; Banerjee, S. Using virtualized task isolation to improve responsiveness in mobile and IoT software. In Proceedings of the IoTDI '19: Proceedings of the International Conference on Internet of Things Design and Implementation, Montreal, QC, Canada, 15–18 April 2019; pp. 160–171. [CrossRef]
32. Garcia, P.; Gomes, T.; Salgado, F.; Monteiro, J.; Tavares, A. Towards hardware embedded virtualization technology: Architectural enhancements to an ARM SoC. *ACM SIGBED* **2014**, *11*, 45–47. [CrossRef]
33. Sá, B.; Martins, J.; Pinto, S. A First Look at RISC-V Virtualization From an Embedded Systems Perspective. *IEEE Trans. Comput.* **2021**, *71*, 2177–2190.
34. Adam, G.K. Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers. *Computers* **2021**, *10*, 64. [CrossRef]
35. Strnadel, J.; Rajnoha, P. Reflecting RTOS Model During WCET Timing Analysis: MSP430/Freertos Case Study. *Acta Electrotech. Inform.* **2012**, *12*, 17–29. [CrossRef]
36. Schliecker, S.; Negrean, M.; Ernst, R. Response Time Analysis on Multicore ECUs With Shared Resources. *IEEE Trans. Ind. Inform.* **2009**, *5*, 402–413. [CrossRef]
37. Brylow, D.; Palsberg, J. Deadline analysis of interrupt-driven software. *IEEE Trans. Softw. Eng.* **2004**, *30*, 634–655. [CrossRef]
38. Liu, Z.; Shi, Y.; Zhang, G.; Hu, B.; Ye, F.; Zhou, H. A Novel Testing Method for Interrupt Response Time. In Proceedings of the 2021 11th International Workshop on Computer Science and Engineering (WCSE 2021), Shanghai, China, 19–21 June 2021. [CrossRef]
39. Chris Simmonds. Mastering Embedded Linux Programming. Available online: https://www.oreilly.com/library/view/mastering-embedded-linux/9781787283282/e19a424b-9507-4186-a54f-430a53b62ad9.xhtml (accessed on 18 April 2024).
40. Monmasson, E.; Hilairet, M.; Spagnuolo, G.; Cirstea, M.N. System-on-Chip FPGA Devices for Complex Electrical Energy Systems Control. *IEEE Ind. Electron. Mag.* **2022**, *16*, 53–64. [CrossRef]

41. Chishiro, H.; Suito, K.; Ito, T.; Maeda, S.; Azumi, T.; Funaoka, K.; Kato, S. Towards Heterogeneous Computing Platforms for Autonomous Driving. In Proceedings of the 2019 IEEE International Conference on Embedded Software and Systems (ICESS), Las Vegas, NV, USA, 2–3 June 2019; pp. 1–8. [CrossRef]
42. Fernández, J.P.; Vargas, M.A.; García, J.M.V.; Carrillo, J.A.C.; Aguilar, J.J.C. Low-Cost FPGA-Based Electronic Control Unit for Vehicle Control Systems. *Sensors* **2019**, *19*, 1834. [CrossRef] [PubMed]
43. Xilinx. FIR Compiler (PG149). 2022. Available online: https://docs.amd.com/r/en-US/pg149-fir-compiler/FIR-Compiler-LogiCORE-IP-Product-Guide (accessed on 2 April 2024).
44. Castillo-Secilla, J.; Olivares, J.; Palomares, J. Design of a Wireless Pulse Oximeter using a Mesh ZigBee Sensor Network. In Proceedings of the International Conference on Biomedical Electronics and Devices, Rome, Italy, 26–29 January 2011; pp. 401–404.
45. Stojanovic, R.; Karadaglic, D. Design of an oximeter based on LED-LED configuration and FPGA technology. *Sensors* **2013**, *13*, 574–586. [CrossRef] [PubMed]
46. Tuning Xen for Performance. Available online: https://wiki.xenproject.org/wiki/Tuning_Xen_for_Performance (accessed on 27 February 2024).
47. King, C.I. stress-ng (Stress Next Generation). Available online: https://github.com/ColinIanKing/stress-ng (accessed on 6 March 2024).
48. Abebe, H. Determination of Sample Size and Errors. In *Promoting Statistical Practice and Collaboration in Developing Countries*; CRC: Boca Raton, FL, USA, 2022; pp. 321–338. [CrossRef]