An I/O Virtualization Framework With I/O-Related Memory Contention Control for Real-Time Systems

Niccolò Borgioli, *Student Member, IEEE*, Matteo Zini, Daniel Casini, *Member, IEEE*, Giorgiomaria Cicero, Alessandro Biondi, *Member, IEEE*, and Giorgio Buttazzo, *Fellow, IEEE*

***Abstract*—Modern applications are often characterized by a tight interaction with I/O devices. At the same time, many application domains are also facing a shift toward an integrated approach where multiple applications with mixed levels of safety and security need to co-exist on top of a shared hardware platform, which is typically managed by a hypervisor. This gives rise to the need for a predictable mechanism allowing multiple virtual machines to share I/O devices, while at the same time controlling contention delays when they access global memory. To deal with these shortcomings, this article proposes an I/O virtualization framework providing support for controlling the I/O-related memory contention by leveraging the ARM QoS-400 regulators. Extensive experiments are performed to compare the proposed solution with the Xen hypervisor, showing improvements up to 8× when controlling the I/O-related memory contention.**

***Index Terms*—Edge computing, I/O virtualization, memory-contention, real-time systems.**

I. Introduction

MANY application domains (e.g., avionics, automotive, etc.) that traditionally adopted dedicated hardware control units for implementing specific functions are now moving toward an *integrated* approach where multiple, logically distinct subsystems can co-exist on the same hardware. This trend shift has been dictated by the need to contain the growth of physical electronic control units, combined with the availability of powerful embedded platforms equipped with multiple heterogeneous cores and even with GPU-based and FPGA-based hardware accelerators, which offer an unmissable opportunity to reduce the size, weight, power, and cost required by each component (often called SWaP-C [1]).

On the other hand, such a paradigm shift gave rise to several new challenges due to the usage of virtual machines (VMs) to execute applications in isolation, using a fraction of the available hardware resources.

Manuscript received 5 August 2022; accepted 5 August 2022. Date of publication 29 August 2022; date of current version 24 October 2022. This work was supported in part by Huawei and the Italian Ministry of University and Research (MIUR), through the SPHERE Project Funded Within the PRIN-2017 Framework under Grant 93008800505. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor A. K. Coskun. *(Corresponding author: Niccolò Borgioli.)*

Niccolò Borgioli, Matteo Zini, and Giorgiomaria Cicero are with the TeCIP Institute, Scuola Superiore Sant'Anna, 56127 Pisa, Italy (e-mail: niccolo.borgioli@santannapisa.it).

Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo are with the TeCIP Institute, Scuola Superiore Sant'Anna, 56127 Pisa, Italy, and also with the Department of Excellence in Robotics and AI, Scuola Superiore Sant'Anna, 56127 Pisa, Italy.

Digital Object Identifier 10.1109/TCAD.2022.3202434

Hypervisor technology is a standard solution for running multiple VMs (also called domains) on the same hardware platform while providing safety, security, and predictability features employing a vast range of isolation techniques designed over the decades [2], [3], [4], [5], [6]. Providing such a virtual platform requires dealing with several nontrivial phenomena, e.g., CPU and memory interference, which are often tackled by implementing mechanisms for CPU and memory-bandwidth reservation [2], or cache coloring [6].

Furthermore, additional issues arise from the tight interaction of modern applications with I/O devices, e.g., in autonomous driving [7] or robotics [8], which calls for methods to virtualize shared I/O devices transparently.

The management of I/O devices is also crucial for reducing the delays due to memory contention when cores and I/O devices simultaneously access a globally shared memory (usually, a DDR memory). To limit the lateness that the memory traffic might introduce during I/O operations, some platforms (e.g., the Xilinx Zynq Ultrascale+ MPSoC [9]) are equipped with hardware regulators capable of controlling the number of memory transactions allowed from each device in a given time interval [10], [11], thus introducing a reservation mechanism to control the I/O-related memory traffic.

Such regulators offer a new opportunity to deal with the memory contention generated by I/O devices and, in the context of a virtualized device, design and implement a comprehensive I/O virtualization mechanism capable of controlling the impact of the traffic generated by I/O devices as a whole.

By leveraging such peripherals provided by modern embedded platforms, a predictable I/O virtualization mechanism should be able to: 1) virtualize I/O devices and make them available to VMs in a seamless and fully transparent manner; 2) control the impact of I/O devices on the memory contention; and 3) provide guarantees of I/O performance in terms of longest blackout delay and bandwidth (i.e., using a bounded-delay [12] (α, Δ) model) for I/O devices, both in transmission and reception.

This Article: This article proposes a robust I/O virtualization framework for mixed-criticality real-time systems that allows controlling I/O-related memory interference by means of ARM QoS-400 regulators. It has been implemented upon CLARE-Hypervisor as a software solution in order to be compatible with any type of I/O device. It comprises both a device-independent part, which includes the I/O virtualization scheduler and the related data structures and a device-dependent part. For the latter part, we considered the virtualization of the gigabit Ethernet MAC (GEM) controller of the Xilinx Ultrascale+ MPSoC [9] as a relevant example of a complex I/O device to virtualize. Finally, it reports the results of an extensive experimental study that has been performed to evaluate the proposed solution by comparing with the Xen hypervisor [13], and reporting on different I/O performance metrics based on a bounded-delay model [12].

II. Background

Hardware Platform: The Xilinx Zynq UltraScale+ MPSoC is a heterogeneous platform equipped with FPGA-based programmable logic (PL) and a processing system (PS). The PS comprises a Cortex R5 dual-core processor (named RPU) and a quad-core ARM Cortex A53 processor (named APU). The RPU has a single-level private cache for each core, while the APU has a two-level cache, where the first one is private to each core and separate for data and instructions, and the second level is shared among all the cores. The various components in the PS (e.g., cores and I/O devices) and the PL are interconnected by means of the ARM advanced microcontroller bus architecture advanced eXtensible interface (AMBA AXI) interface [14]. Cores, I/O devices, and the PL can store and retrieve data by interacting with a globally shared off-chip DDR memory.

Particularly interesting to us, the Ultrascale+ MPSoC provides several ARM CoreLink QoS-400 Regulators [10]. These regulators are designed to limit over time the number of AXI transactions (and, hence, the memory traffic) issued by the device they are connected to.

The QoS-400 supports three different types of regulations. The *outstanding transaction regulation*, which permits the user to specify the maximum number of requests the AXI master can simultaneously issue, the *transaction latency regulation*, which, by means of a feedback loop, exploits the QoS value included to every AXI request to attempt to achieve the desired requests' latency, and the *transaction rate regulation*. Most relevant to this work, the *transaction rate regulation* feature acts in accordance to a variant of the TSPEC specifications (RFC 2215) and can be controlled with three parameters, separately available for read and write requests: r (*average rate*), which indicates the average allowed transactions per clock cycle; b (*burstiness allowance*), which represents an additional budget of transactions that can be consumed by the device at a rate higher than r; and p (*peak rate*), which is the maximum rate of transactions per clock cycle. These values are expressed as a 12-bit fraction of the desired number of transactions per clock cycle.

CLARE-Hypervisor: CLARE-Hypervisor [15] is a bare-metal type-1 hypervisor. It integrates cutting-edge mechanisms to host safe, secure, and time-predictable VMs that can execute in isolation upon the same hardware platform. CLARE-Hypervisor has been designed for modern heterogeneous platforms, such as FPGA-based SoCs, to better exploit and control their computational resources. CLARE-Hypervisor follows a fully static approach with off-line configurations and optimization to allocate the onboard resources to the VMs. Physical devices are statically assigned to a single domain, while physical CPUs may host multiple virtual CPUs with real-time scheduling policies (e.g., fixed-priority scheduling). The hypervisor mitigates the interdomain interferences by statically assigning partitions of the last-level cache (LLC) (*via* cache-coloring) to the physical cores, partitioning the DRAM following a bank-aware approach, and implementing a memory-bandwidth reservation budgeting the memory transactions issued by the virtual CPUs.

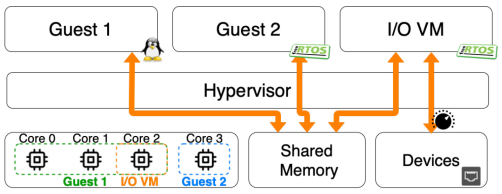

Fig. 1. High-level overview of the considered platform.

III. Design and Implementation

The I/O virtualization mechanisms presented in this article consider devices that need to be shared among VMs. The ones that are for the exclusive use of a VM can be accessed in pass-through and treated as simpler cases of the shared ones (details are provided in the following). Similarly to other proposals (e.g., hypervisors such as Xen [13], or other mechanisms proposed in scientific papers [16], [17], [18], [19], [20]), in our architecture (shown in Fig. 1) all the shared physical devices are exclusively assigned to a single VM by the hypervisor, called I/O-management VM (I/O VM in short), following a pass-through approach. The I/O VM implements a software-defined sharing mechanism using shared-memory buffers and an I/O scheduler to dispatch the accesses to each I/O device. In our implementation, the I/O VM is powered by FreeRTOS and can support the interaction with both FreeRTOS-based and Linux-based VMs. The I/O VM can be either assigned to a dedicated core or to a core shared with another VM. In the latter case, it is recommendable to assign the highest priority to the I/O VM to guarantee low-latency I/O.

A. Threat Model

The threat scenario considered in this work assumes that any VM (excluding the I/O VM) is untrusted. This means that an attacker could exploit a compromised guest VM to attack the other guest VMs. The attacks considered in this scenario are packet sniffing, packet manipulation, and I/O-related denial of service (DoS). In general, in a similar architecture, the packet sniffing and manipulation could happen in three specific areas: 1) the device memory; 2) the private memory of a guest; and 3) the memory shared between a guest and the I/O VM. In our architecture, thanks to the isolation capabilities of the hypervisor, area 1) can only be accessed by the I/O VM, area 2) by the interested guest VM only, and area 3) by the I/O VM and the interested VM only. Packet sniffing and manipulation are, hence, mitigated by design. An I/O-related DoS attack can be performed by generating I/O traffic from a malicious guest VM or an external data source connected to the system. They both have an impact on: 1) the dispatching of the I/O data of other guests, possibly introducing large I/O-related scheduling delays; 2) the on-chip memory traffic due to memory transactions issued by peripheral I/O devices; and 3) the processor workload, due to the interrupts raised by I/O devices. Protection mechanisms against these threats are highlighted later in this article when presenting the features of the proposed I/O virtualization mechanism.

B. Design Principles

Next, we describe our design principles and how they are addressed in our solution.

1) Design for Safety and Robustness to Security Attacks: I/O VM: Our I/O virtualization mechanism is designed to follow the best practices for safety and security. For example, the usage of an I/O-management VM allows avoiding implementing I/O management features *inside* the hypervisor. Indeed, implementing I/O-management inside the hypervisor increases its codebase (since drivers need to be ported within the hypervisor), which is not a good practice for safety and security, as the hypervisor's attack surface increases and significantly complicates certification.

Wait-Free Queues: The I/O virtualization mechanism uses wait-free ring buffers designed for robustness in a mixed-criticality environment so that if the producer corrupts the content of the shared memory, the code of the consumer will never crash, and *vice versa*. This reduces the extent of packet sniffing and manipulation attacks (see Section III-A).

I/O-Related Memory Traffic Regulation: QoS regulators are used to mitigate adverse effects due to excessive I/O-related memory contention and an excessive number of I/O-related interrupts, which need to be processed by both the hypervisor and the guest VM. This also allows bounding the effects of security attacks, where an attacker may take control of an I/O device and flood the system with a storm of I/O data, thus potentially compromising the system's performance or causing a DoS, which can be instead forbidden thanks to the QoS regulation on I/O devices. This helps address DoS attacks on the on-chip memory buses due to memory transactions issued by I/O devices on the platform and attacks on the processor workload due to interrupts raised by I/O devices.

Round-Robin I/O Scheduling: As discussed in detail in the following, the I/O VM implements a *round-robin* inter-VM and interdevice arbitration. This choice makes the I/O virtualization mechanism fair, predictable, and robust to security attacks. Note that round-robin is a predictable policy (also used in locking protocols [21]) for which worst-case bounds can be provided (see [16]). Its fairness also provides important features for security: a misbehaving VM continuously sending I/O data due to a security breach would never exacerbate the worst-case timing performance of other VMs because it will never get more than its assigned fair share given by round-robin, as long as other VMs and devices need to communicate. This, together with the memory-bandwidth regulation capabilities provided by CLARE-Hypervisor, mitigates DoS attacks from a malicious guest VM, which are discussed in Section III-A. Nevertheless, our solution can be easily extended to support other schedulers, e.g., priority-based ones.

2) Measurable Real-Time Performance Using the (α, Δ) Model: The I/O virtualization mechanism is designed to allow measuring the real-time performance of the I/O using a bounded-delay model [12] (α, Δ). The bounded-delay model, originally proposed in 2001 by Mok *et al.* [12] (α, Δ), is a well-known metric for real-time systems to measure the performance of mechanisms such as CPU-time resource reservation. In this article, we inherit this metric from the context of virtual CPUs and we use it (for the first time, to the best of our knowledge) in the context of I/O virtualization, designing our approach to provide measurable real-time performance using the (α, Δ) model, where the parameters are reinterpreted to fit I/O performance for each I/O peripheral as follows.

1) α refers to the bandwidth (in the long run) with which I/O data can be transmitted or received.
2) Δ refers to the maximum time-span elapsed between the reception/transmission of two data unit (e.g., network packets) while the system is continuously receiving/transmitting data.

The I/O-related traffic regulation must allow the designer to control α and Δ to meet the performance requirements.

3) Extensibility to Other Devices and Platforms: While an I/O virtualization mechanism inevitably needs to cope with several hardware-dependent aspects, we designed our solution in a modular manner, dividing its components into a *device-independent* part and a *device-dependent* part. This makes the approach more general and easily portable for other I/O devices and systems. As a relevant example of a complex I/O device to virtualize, we focus on the GEM provided by the Xilinx Ultrascale+ to present the device-dependent part and carefully evaluate our solution.

Our approach can be extended to support also other peripherals and platforms. Extensibility to other hardware platforms is favored by the fact that our virtualization approach is purely software-based and it does not rely on special hardware features. Other design alternatives might involve the usage of *hardware-assisted virtualization*. For example, Intel VT-d [22] or SR-IOV [23]. These solutions may indeed provide benefits to the performance of the I/O virtualization mechanism. However, most of them are device-specific and cannot be generalized to all I/O devices (e.g., a legacy CAN bus). Furthermore, SR-IOV is not available for the AXI-based peripherals considered in this article, while VT-d is a technology by Intel that is not available in the Ultrascale+ because it is an ARM-based embedded platform. Differently, our software-based mechanism provides a device-independent layer that can be generalized to any type of I/O device.

4) Full Transparency to the Final User: The I/O virtualization mechanism must allow the user to interact with the device from a guest VM in a fully transparent way, exactly as if the VM was granted direct and dedicated access to the physical device. This is ensured through the I/O virtualization driver implemented in each guest OS.

5) Regulation of the Impact of the I/O on Performance: The performance of the I/O virtualization mechanism without using QoS regulation should be no worse than existing I/O virtualization solutions from other hypervisors, e.g., Xen, and should improve the performance of a target VM or device when QoS regulators are used to reduce the I/O-related interference.

C. Device-Independent Features

Next, we start discussing the device-independent features of our I/O virtualization mechanism.

The I/O VM Scheduler: The proposed I/O virtualization solution can manage I/O traffic from/to multiple VMs and devices with the *I/O scheduler*. As shown in Fig. 2, the scheduler provides two queues of requests for each pair of VM and I/O devices: 1) one is for data flowing from the VM to the device and 2) the other for data flowing from the device to the VM. Each queue is managed in a first-in-first-out (FIFO) fashion. The access to the I/O is orchestrated using a round-robin algorithm based on the concept of *data quantum*, which denotes the minimum amount of data that can be meaningfully exchanged (e.g., one frame in the case of Ethernet). The round-robin scheduler handles both received and transmitted packets.

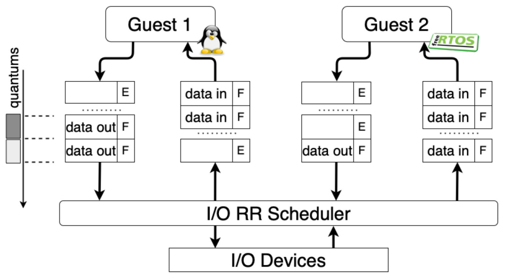

Fig. 2. Wait-free queues used to share data among guest VMs and the I/O scheduler, which runs in the I/O VM.

Wait-Free Queues: The data sharing between I/O and guest VMs is implemented through shared-memory, wait-free [24] ring buffers of N elements, each of size M bytes. Each of these elements is composed of a field of size $M-1$ bytes for the I/O data unit and a one-byte flag. This flag is used to identify if a buffer field is full or empty and allows to perform atomic write and read operations. Thanks to these flags, there is no need to share head and tail counters between the two VMs; instead, the producer (or consumer) can store privately only the head (or tail). The addresses of the shared memory regions are retrieved at runtime by the VMs using dedicated hypercalls (that we implemented on purpose in CLARE-Hypervisor), which also return the size M of the hosted ring buffers and the number N of elements. Wait-free ring buffers allow two VMs (a producer and a consumer) to exchange data without experiencing lock-related delays, as there are no race conditions on state variables [24].

The canonical `push` and `pop` operations can be performed on these buffers. If the producer incurs in an overflow, `push` returns `FALSE` and the data quantum is discarded. If this happens in a Guest VM (i.e., during the transmission phase) this event is notified to the upper layers which will eventually take care of retry to retransmit the data quantum again. If this happens in the I/O VM (reception phase), the data quantum is lost. Otherwise, `push` returns `TRUE`.

If the consumer incurs in an under-run, i.e., if there are no elements in the queue, the `pop` returns `FALSE`. If this happens in a Guest VM (in the reception IRQ handler) then there are no more received data to handle and the IRQ terminates. If this happens in the I/O VM (in the transmission phase of the scheduler) this means that there are no more data to transmit for the given guest and the scheduler skips to the reception phase for that guest. Otherwise, `pop` returns `TRUE`.

A data memory barrier for store operations is needed both in `push` and `pop`, to avoid the out-of-order execution of loads/stores that may bring the element in an inconsistent state from other cores.

Shared-Memory Configuration: For each guest-I/0 device couple, and for each needed direction (input and output), a memory region shared with the I/O VM is reserved to host the wait-free ring buffers. CLARE-Hypervisor handles the assignment of these shared memory regions and implements access control to guarantee that each of them can be accessed only by the authorized VMs.

Each of these shared memory regions is *uncached* to preserve cache-related isolation of VMs, which is provided by CLARE-Hypervisor through cache-coloring, without wasting a cache color for each VM-to-VM communication. Note that the number of cache colors for partitioning a shared cache level, e.g., L2, may be very low on many platforms. For instance, on the Xilinx Ultrascale+ they are at most 16 (and only 4 if L1 caches are not partitioned), and each color reserved for inter-VM communication would imply a corresponding physical memory area that only stores the inter-VM buffers. The latter may be much smaller than the area itself, hence, originating a consistent memory waste. Furthermore, note that the bandwidth of I/O peripherals is typically lower than the one available to processors to perform uncached memory accesses so that keeping the inter-VM buffers uncached does not introduce a bottleneck. For instance, on a Xilinx Ultrascale+, the uncached memory access bandwidth is more than 2 GB/s, which is larger than the bandwidth of high-performance I/O peripherals such as the GEM (1 Gb/s).

I/O Regulators: The I/O virtualization mechanism also allows to set the QoS regulation values of I/O devices by providing functions to change the values of all the three regulation parameters, i.e., the average rate, the peak rate, and the burstiness allowance (discussed in Section II). Technically speaking, these functions modify the value of the `ar_r`, `ar_p`, `ar_b` and `aw_r`, `aw_p`, `aw_b` registers of the QoS-400, which are used to regulate the average rate, peak rate, and burstiness, and enable the regulation by setting the value of the related bit in the `qos_cntl` register. Therefore, our virtualization mechanism is able to control the average number of transactions per time unit emitted by each I/O device and, consequently, its communication speed. This allows controlling I/O-related interference, both in terms of I/O-related memory contention and in the number of I/O-related interruptions that are generated (which then need to be managed both at the hypervisor and at the VM level, thus, possibly causing considerable delays). In this way, VMs can be properly isolated also at the level of the I/O traffic generated by the devices they use.

Furthermore, the I/O virtualization mechanism allows specifying the size, expressed in a number of words, of every AMBA AXI transaction. For example, in the case of the GEM Ethernet of the Ultrascale+, its value can be modified by the user by writing in the `amba_burst_length` field of the `dma_config` register of the GEM module. The value chosen as burst size indicates the maximum number of fixed-length words that can be merged into a single transaction. This parameter is important for regulation purposes: indeed, lower values of the burst length allow for a fine-grained regulation of AXI transactions (and, hence, memory traffic). The word size is architecturally defined. Thus, to convert any regulation value to the corresponding allowed bandwidth in bytes per second, the fraction expressed by the content of the registers must be multiplied by the clock frequency, the burst size, and the word-size values.

IV. Ethernet-Specific Features

As a representative case of a complex device that can be virtualized by the proposed I/O virtualization mechanism, we consider the GEM. The required Ethernet-specific features are spread between: 1) the Ethernet (physical) driver, used by the I/O VM to access the actual GEM device and 2) the I/O virtualization driver, used by any Guest VM to transparently interact with the GEM device through the I/O virtualization stack. Fig. 3 shows the two drivers interact with each other to allow any guest to transmit and receive Ethernet frames.

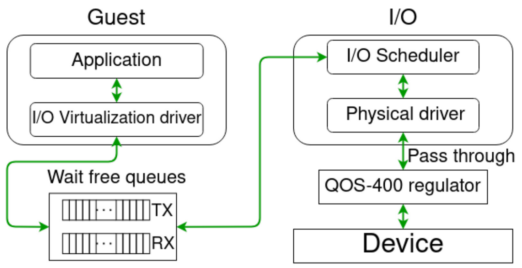

Fig. 3. Overview of the I/O virtualization software stack.

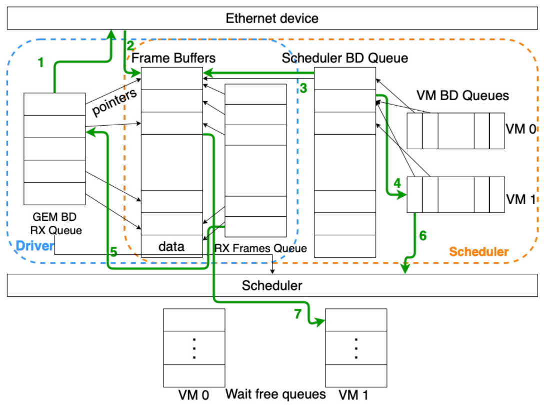

Fig. 4. Data structures used to receive an Ethernet frame.

A. Ethernet Driver

Each guest VM has a unique MAC address assigned at boot time by the I/O VM, which allows to perform packet routing on the data link layer. To allow granular control on the network traffic generated and received by the VMs, we have implemented a bare-metal driver to interact with the GEM controller [9] of the Ultrascale+ MPSoC. Unfortunately, to date, Xilinx does not provide a complete Ethernet driver allowing the degree of granularity required to implement the virtualization mechanism. The implementation of the driver required to properly validate our I/O virtualization framework costed us six person-months of work. This driver auto negotiates the link speed (gigabit, megabit, etc.) based on the physical link properties.

Packets transmission and reception are handled by the driver by means of GEM *frame buffers* and *Buffer Descriptors* (G-BDs). Frame buffers are arrays where the actual Ethernet frame received or to be transmitted is stored. G-BDs are data structures defined by the GEM driver that are used to describe the frame buffers used to store packets received and to transmit by storing the frame buffer *address*, the frame *size*, the G-BD status *flag* (i.e., either free, preprocess, post-process, or hardware), and another flag that marks the last descriptor in the queue. G-BDs are organized in two queues (one for transmission and one for reception) to handle multiple frames. To distinguish between G-BDs for transmission and reception, these are defined as *G-BD TX* and *G-BD RX*, respectively. On top of these data structures, shared by the driver and the device, the I/O scheduler adds further data structures to handle packets reception and transmission of the different guests. The I/O scheduler handles one packet at a time.

Reception: To handle and dispatch packets among the Guest VMs, the I/O scheduler leverages three additional data structures: the *RX Frames queue*, the *Scheduler Buffer Descriptor (S-BD) queue*, and the VM Buffer Descriptor (VM-BD) queues. The RX Frames queue is used to store the status of the frame buffers' queue holding a flag denoting whether the related frame buffer is in use or not. The S-BD queue includes a pointer to a frame buffer and a counter of the number of VMs that frame needs to be delivered to (useful for *multicast* or *broadcast* packets). Finally, a VM-BD queues is provided for each Guest VM sharing the device. Each VM-BD is a pointer to an S-BD, which allows the RX ISR to easily dispatch the received frame to the correct Guest VM without the need to copy it. Thanks to this structure, the scheduler can retrieve the next packet to be dispatched to a given Guest VM without scanning the S-BD. This, in combination with the counter in the S-BD, allows to easily handle multicast frames without introducing additional copies of the received data.

Since both the S-BD and VM-BD queues need to be manipulated by the driver's RX ISR and by the scheduler, these have been implemented as wait-free queues where the ISR acts as producer and the scheduler as a consumer. The flow of a received frame from the Ethernet device to the shared memory queue consists of the following steps, which are also shown in Fig. 4 with the same numbering.

1) The Ethernet device obtains the first G-BD associated with a free frame buffer.
2) The Ethernet device writes the packet received into the obtained frame buffer. When a packet write completes an RX interrupt is raised.
3) The RX ISR makes a free S-BD point the just copied frame buffer. Then, depending on how many VMs are interested to that packet (e.g., multicast communication), the counter is set accordingly.
4) Still in the ISR, a pointer to the just created S-BD is inserted in the VM-DB queue of each VM that will receive that frame.
5) Then, the ISR recycles the G-BD RX used by assigning to it a new free Frame Buffer obtained from the RX Frames Queue.
6) When the scheduler dispatches a VM RX packet, it extracts the next VM-BD from that VM queue.
7) The frame pointed by that buffer descriptor is copied in the wait free queue of the respective VM.

With this design, the S-BD queue avoids dangerous race conditions between the RX ISR and the scheduler by leaving the management of the G-BD RX to the ISR only.

Transmission: Similarly to the reception case, the I/O scheduler leverages the *TX Frames queue* to manage packet transmissions.

This queue holds the status of the TX frame buffer's queue by storing a flag that specifies if the related frame buffer is in use. The flow of a frame transmitted from a Guest VM shared-memory queue through the Ethernet device is described by the following steps, which are also shown in Fig. 5 with the same numbering.

1) The scheduler checks, in round-robin order, whether the current Guest VM has a new packet to transmit.
2) If there is a packet to transmit, the scheduler gets the index of the first free Frame Buffer from the TX Frames Queue. Otherwise, the scheduler considers the next wait-free queue in round-robin order.
3) Then, the scheduler pops the content of the shared memory element in the acquired Frame Buffer.

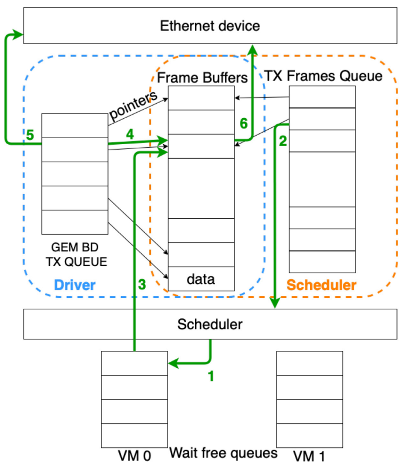

Fig. 5. Data structures used to transmit an Ethernet frame.

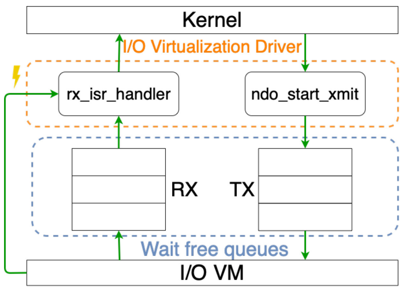

Fig. 6. I/O virtualization driver for Linux.

4) The driver allocates a new G-BD TX pointing to that Frame Buffer and triggers the device to transmit.
5) The Ethernet device gets the next buffer descriptor TX that needs to be handled.
6) The Ethernet device transmits the frame buffer pointed by that BD and raises a TX ISR, which recycles the used G-BD TX and frame buffer.

B. I/O Virtualization Driver

On the guest side, an I/O virtualization driver has been designed and developed to handle frames transmission and reception through the shared memory buffers introduced in the previous section. The implementation of this driver strictly depends on the target operating system: the proposed I/O virtualization software stack supports Linux and FreeRTOS. Nevertheless, our implementation is highly portable to other configurations since the two OSes are representative of reference rich and bare-metal OSes, respectively, and among the most used ones for real-time applications.

In ***Linux***, the driver is implemented as an *etherdevice* in a *kernel module*. The virtual device is recognized by Linux as a regular Ethernet interface, thus enabling the OS to send and receive packets as if it were interacting with a physical Ethernet device. In this way, the I/O virtualization mechanism is *fully transparent* to the user. When the module is loaded, it registers on the kernel the `net_device_ops` structure that defines four callback functions required by the Linux kernel to interact with the driver: `ndo_init`, `ndo_open`, `ndo_close`, and `ndo_start_xmit`. In addition to these functions, it registers the `rx_isr_callback` used to manage the RX interrupts received from the I/O VM. Fig. 6 illustrates the role of such two main functions.

The `ndo_init` function is called on interface registration (when the module is loaded) and allocates the private space required by the driver to hold all the required data structures. Then the `ndo_open` function is called when the interface is opened (e.g., when `ifconfig eth0 up` is executed). This function first retrieves (through a dedicated hypercall) the shared memory regions used to communicate with the I/O VM. Then, such regions are mapped to virtual addresses and marked as noncacheable. To complete the driver configuration, this function retrieves the MAC address assigned to the VM by performing a pop operation from the RX buffer. Finally, it configures the `rx_isr_handler` ISR handler to manage RX notification interrupts coming from the I/O VM and enables packet transmission requests from the kernel. The `ndo_close` is called when the interface is closed (e.g., when `ifconfig eth0 down` is executed), freeing all the resources allocated by `ndo_open`.

In Linux, Ethernet frames are described by means of *socket buffers*. These structures contain the frames to be transmitted plus some additional information used by the kernel to manage packets. Whenever the Linux kernel needs to transmit a new frame through this interface, the `ndo_start_xmit` callback function is invoked by passing as an argument a socket buffer that contains the data to be transmitted. This function performs the following steps: 1) it notifies the kernel that the device is busy, temporarily disabling the callback mechanism to avoid nested callback invocations; 2) it pushes the frame into the TX shared buffer; 3) it frees the frame descriptor received from the kernel; and finally, 4) it re-enables callback invocations.

On the reception side, whenever the I/O VM pushes a frame into the RX shared buffer, it also raises an interrupt for the target guest VM. The `rx_isr_handler` manages this interrupt as follows: 1) it gets the size of the next frame and allocates a socket buffer to contain that frame; 2) it pops the frame from the RX shared buffer by copying it into the socket buffer; 3) it submits the socket buffer to the network layer of the kernel; and 4) it marks the interrupt as handled.

To further improve the driver performance, we have also implemented a new API (NAPI), an interface widely used in the Linux kernel to reduce the overhead due to the interrupt context switch by disabling interrupts and polling for new incoming packets if the RX traffic is above a given threshold. When NAPI is active, whenever the driver receives an RX notification interrupt (handled by `rx_isr_handler`), instead of popping and forwarding to upper layers (steps 1–3) only one frame from the shared buffer, it keeps performing these steps until the buffer is empty (or no more space to allocate socket buffers is available).

In ***FreeRTOS***, the I/O virtualization driver is implemented similarly to the Linux driver. In FreeRTOS, new Ethernet interface drivers should be defined as *portings*. These interfaces should provide two main functions which are used by the FreeRTOS kernel to interact with it: `xNetworkInterfaceInitialise` and `xNetworkInterfaceOutput`. The first one is invoked by the kernel to initialize the driver interface. This function sets up all the data structures required to manage the wait-free queues. Moreover, it configures an ISR handler to manage the RX notification interrupts raised by the I/O VM. This ISR performs the same steps explained in the previous section but notifies the received packet to the TCP/IP stack of the FreeRTOS kernel using the `xSendEventStructToIPTask` function. Whenever the TCP/IP stack requires a new packet to be transmitted, the `xNetworkInterfaceOutput` callback function is invoked. This function receives as arguments a descriptor of the frame to be transmitted and a flag that specifies whether it needs to wait for the transmission to complete or not before returning. Then, this function performs the same steps explained for transmission in the previous section.

Why Not Using Virtio? One of the most famous solutions to implement the I/O virtualization driver on Linux is using Virtio [25]. Therefore, one may argue about why not use virtio instead of devising a new solution. Virtio is a widely used standard open interface that allows different guest VMs to share simple I/O devices. Concerning Virtio-net (the variant dedicated to networking), the proposed solution requires implementing a more efficient shared memory communication mechanism. Virtio requires the support of *Virtqueues*. Virtqueues are descriptors that point to a buffer, which can be used to store an incoming/outgoing frame. Our solution instead can operate without requiring additional data structures, but only by adding a flag for each element in the wait-free queue. This allows to reduce the communication overhead while maintaining the same wait-free properties. Conversely, the solution proposed in this work leverages hardware interrupts shared between a guest and the I/O VM and thus avoids the overhead caused by the intervention of the hypervisor. Furthermore, virtio does not support I/O-related interference control through QoS regulation, which is a key feature of our approach. Nevertheless, we do not exclude providing support for virtio in future work to foster interoperability.

V. Experimental Evaluation

This section reports an extensive experimental study that has been performed to evaluate: 1) the performance of our solution with respect to the I/O virtualization mechanism available in the popular Xen hypervisor; 2) the impact of QoS regulation on I/O virtualization performance, using the QoS-400 regulators [10]; and 3) the resilience of the proposed mechanism to DoS attacks based on the flooding of I/O data *via* the network.

A. Experimental Setup

The considered setup is composed of three VMs: one I/O VM and two Guest VMs, respectively, powered by FreeRTOS and Linux. The same setup is applied on both Xen and CLARE-Hypervisor, with the only difference being the fact that the I/O VM is used only for our solution with CLARE-Hypervisor. This setup uses the four Cortex-A53 cores provided by the Ultrascale+ SoC on the *ZCU102* board by Xilinx. The first two cores (0 and 1) are exclusively assigned to the Linux Guest VM; core 2 is shared by the Linux Guest and the I/O VM; core 3 is assigned solely to the FreeRTOS Guest. Core 2 hosts two virtual CPUs belonging to two different VMs: the hypervisor is configured to schedule the VMs with a fixed-priority scheduler where the I/O VM has higher priority, thus guaranteeing that the I/O operations are not interfered with by the Linux VM. Fig. 1 shows the described architecture.

In all the experiments, the sizes of the wait-free buffers were set to be large enough so that they never become full. In the experiments where QoS regulation is considered, we varied the average rate configuration parameter of the QoS-400 since it is the only one responsible for the long-term behavior of the device when it is subject to a heavy traffic load [11]. Therefore, different QoS configurations correspond to a different average number of transactions per time unit emitted by the Ethernet device. Ten relevant configurations were tested, where each one halves the traffic rate compared to the previous one. Excluding the first configuration value where the regulator is turned off, in the other nine ones the average rate value of the regulator is, respectively, set to 0×100, 0×80, 0×40, 0×20, 0×10, 0×8, 0×4, 0×2, and 0×1, which, as discussed in Section III-C, correspond to 15.5, 7.7, 3.9, 1.9, 0.97, 0.48, 0.24, 0.12, and 0.06 million transactions per second. The burst size was set to *one word per transaction*, since a lower value permits a more fine-grained control of the amount of generated traffic in terms of bytes allowed to transit by the QoS-400. The presence of additional memory traffic was considered, leveraging memory accesses from the CPU cores and the general-purpose DMA located in the low-power domain of the Ultrascale+ MPSoC [9] (named LPD-DMA). Both were exploited to create additional interference to the Ethernet device. In particular, the LPD-DMA was chosen since the interconnects traversed by its memory requests are in part shared with the ones traversed by the Ethernet's requests.

B. Measuring Bandwidth and Latency

We start providing the results of an experimental evaluation we performed to compare our solution with the state-of-the-art Xen hypervisor [13]. We selected Xen as a reference for comparison since it is a widely adopted hypervisor in many domains. Moreover, the embedded platform used in this work officially supports Xen and provides a rich documentation.

Comparing With Xen: In this comparison, the Ethernet bandwidth and latency of the Linux Guest VM were monitored for the following five configurations.

C1	without network traffic interference from the other guest VM
C2	10 packets/s traffic from the other guest VM
C3	100 packets/s traffic from the other guest VM
C4	100 packets/s traffic from the other guest VM and DMA interference
C5	100 packets/s of traffic from the other guest VM and DMA interference + DMA QoS regulation (only applied to our solution)

The DMA QoS regulator (considered in C5) can be applied to our solution only since it is not supported by Xen.

We considered two metrics: 1) TCP and 2) UDP bandwidth, measured by `iPerf3` (i.e., a tool commonly used to measure network performances), and ICMP latency, using the `ping` command. Each run of a bandwidth test was performed for 50 s, while each run of the latency test corresponds to 100 ping measurements. Fig. 7 reports the result of some relevant configurations, where the interfering LPD-DMA was regulated with the looser regulation (i.e., 0×8) that allows obtaining good Ethernet performance to avoid penalizing the LPD-DMA excessively. Reported values refer to the transmission phase, but analogous measurements were obtained for the reception. It is worth observing how the proposed solution performs better than Xen, even without using QoS regulation and without interfering memory traffic (C1), with improvements up to 60% and 14% in terms of bandwidth and latency, respectively. This is attributed to the lightweight implementation and the wait-free design choices we performed in our virtualization solution. Moreover, when the other guest VM generates traffic

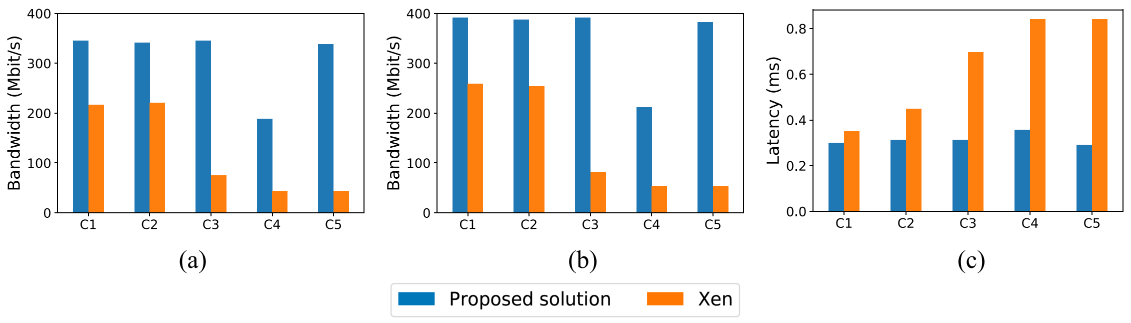

Fig. 7. Comparison of the proposed solution with Xen using TCP and UDP Ethernet communication. (a) TCP. (b) UDP. (c) Latency.

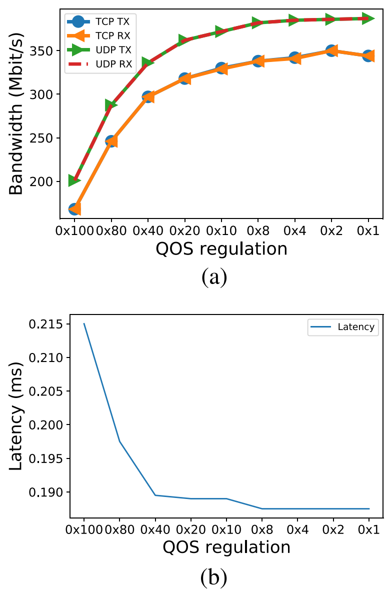

Fig. 8. Bandwidth and latency of the proposed solution in the presence of DMA traffic and varying DMA QoS regulators. (a) iPerf3 bandwidth. (b) Ping latency.

(C2 and C3), our solution is resilient to this interference thanks to the round-robin scheduling policy used by the I/O VM. For example, when the other Guest VMs are configured to generate a considerable amount of Ethernet traffic (C3), in the Xen configuration the bandwidth drops by 66% and the latency doubles, while in the proposed solution both metrics remain constant. When also the LPD-DMA disturbance is active on the platform (C4), both Xen and our solution have a drop in bandwidth and an increase in latency. However, thanks to the QoS-regulation (C5), our solution can reduce the effect of DMA disturbance on system performance, with improvements up to 8× compared to Xen.

Varying the QoS Regulation: Fig. 8 shows how bandwidth and latency vary with the QoS regulation of the interfering LPD-DMA device. Xen-related measurements are not reported in these plots as Xen does not support QoS regulation for I/O. This regulation can help mitigate the Ethernet performance drop due to LPD-DMA traffic. By limiting the LPD-DMA traffic to 0.48 million transactions per second (0×8 regulation), we can note that the proposed solution recovers similar performance as observed without DMA disturbances (C3).

Comparing an ICMP Ping: To further explore the performance of our solution, we also compared it against a classical software architecture without a hypervisor. In

TABLE I
COMPARISON OF MINIMUM, MEAN, AND AVERAGE TIME TO PING A PC CONNECTED ON THE LOCAL NETWORK (IN μS)

Scenario	Min	Mean	Max
(i) No virtualization	69	128	311
(ii) With I/O virtualization	128	184	251
(iii) With I/O virtualization and low traffic	135	186	299
(iv) With I/O virtualization and high traffic	163	187	312

particular, we measured the time taken to complete an ICMP ping from Linux to a remote PC over the local network. The test was performed under four different configurations: by running the ping from Linux: 1) without any hypervisor; 2) with hypervisor and our I/O virtualization framework; 3) same as 2) but with some interfering traffic (10 packets/s, simultaneously produced by the FreeRTOS guest VM); and 4) same as 2) but with higher interfering traffic (100 packets/s). The experiment was repeated 1 000 times for each case: the results are reported in Table I and show that both the interfering traffic and the I/O virtualization introduce delays in the order of a few hundreds of μs. Note that when the FreeRTOS VM is also producing Ethernet traffic, the delay suffered in the average case by the Linux VM is almost negligible and in the order of noise for this kind of measurement. It is worth noting that when increasing the amount of Ethernet traffic generated by the guest VMs, the latency experienced by each of the VM does not increase significantly. This is attributed to the fact that, in these conditions, the round-robin scheduling allows giving reasonable fairness guarantees to the guest VMs.

C. Measuring I/O Performance With the α/Δ Model

We report a set of experiments we devised to empirically quantify the I/O performance according to an α/Δ model.

Different memory load configurations were tested: `no-traffic`, in which the two guest VMs are not executing any memory-intensive task, and no other I/O device is enabled; `CPU-traffic`, in which the two guest VMs are both continuously executing the memory-bandwidth intensive task included in the Isolbench [26] benchmarks on all the cores at their disposal; `CPU-DMA-traffic`, in which, in addition to the Isolbench executions already introduced in the *CPU-memory-traffic* configuration, also the LPD-DMA was configured to continuously move data between two memory buffers, thus increasing the I/O-related memory contention.

Isolbench [26] is a Linux-based benchmarks suite designed to measure the memory bandwidth and latency of a system.

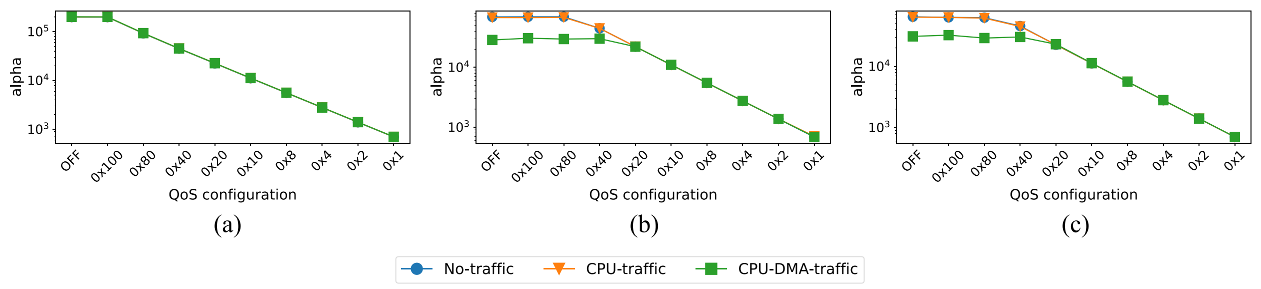

Fig. 9. Measurements of the α parameter (in packets per second) obtained by running the I/O virtualization framework under different configurations when the QoS regulation is varied. (a) IOVM_RX. (b) IOVM_TX. (c) GUEST_LINUX_RX.

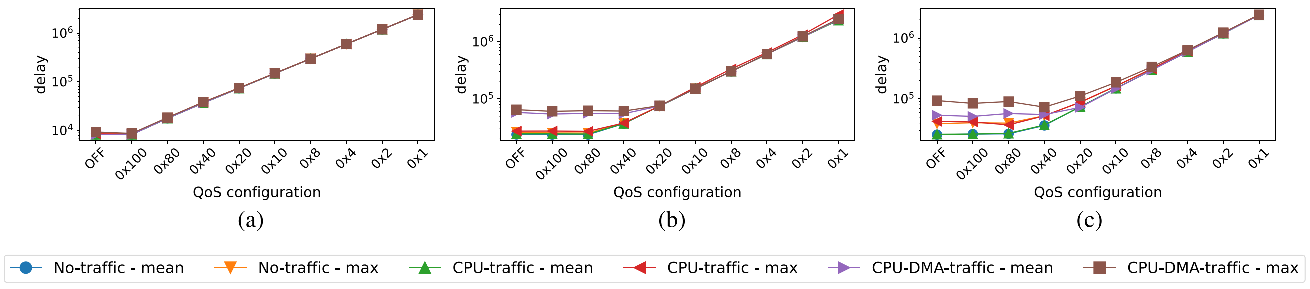

Fig. 10. Measurements of the Δ parameter (in nanoseconds) obtained by running the I/O virtualization framework under different configurations when the QoS regulation is varied. (a) IOVM_RX. (b) IOVM_TX. (c) GUEST_LINUX_RX.

For our purposes, the memory-bandwidth intensive task was partially modified to constantly generate heavy memory traffic and to permit its execution also on the FreeRTOS guest. The load produced in the `CPU-traffic` and `CPU-DMA-traffic` configurations was measured thanks to the AXI Performance Monitors available on the platform [9].

The Ethernet traffic for the experiments was generated by directly connecting the ZCU102 board to a personal computer (hereinafter referred to as *"test PC"*) through a Gigabit Ethernet link. On the test PC, the traffic was generated using the `packETH` tool. Transmission (TX) and Reception (RX) performance were measured independently. The tests were carried out under the following network traffic conditions: `RX`, in which traffic is generated from the test PC to the board; `TX`, in which traffic is generated by one of the guest VMs on the board to the test PC. We generated the maximum amount of traffic that can be managed without losing packets.

A wide set of experiments were performed, collecting several different performance metrics.

1) `IOVM_RX` and `IOVM_TX`, which measure the time elapsed between two consecutive receptions or transmissions interrupts handled by the I/O VM,[1] respectively.
2) `IOVM_RX_SCHED` and `IOVM_TX_SCHED`, which measures the time elapsed between two consecutive received packet dispatches to a given Guest VM by the I/O VM or two consecutive transmissions performed by the I/O VM for a given Guest VM's frame (pop from TX shared buffer), respectively.
3) `GUEST_LINUX_RX` and `GUEST_LINUX_TX`, which measures the time elapsed between two consecutive reception interrupts handled by the Linux Guest VM or two consecutive successful insertions of a Linux Guest VM's frame into the TX shared buffer, respectively.
4) `GUEST_FREERTOS_RX` and `GUEST_FREERTOS_TX`, defined as in the previous point but considering the FreeRTOS domain.

All the described metrics were gathered under the ten different QoS memory-bandwidth regulations of the Ethernet device and in the three memory load configurations.

Overall, a total of 240 different experiments were carried out considering all the possible setups. All the tests were performed by sending 10 000 consecutive packets. The data obtained by every run was aggregated to obtain the maximum and average values for the configuration under analysis.

Results: Next, we discuss some relevant configurations we selected from the extensive set of experiments we performed to explore the performance of the proposed I/O virtualization framework as a function of the GEM Ethernet QoS regulation. In this way, we aim at building the foundations for future research that will enable the design-space exploration of complex platform-aware systems that may leverage these measurements to achieve design-level goals, such as response times, minimum throughput, maximum I/O latency, and bandwidth. These measures will constitute the basic building block to infer empirical models able to predict the behavior of a virtualized system under memory-bandwidth regulation.

Figs. 9 and 10 report several results where the α and Δ parameters were experimentally evaluated. With stringent regulations for the Ethernet device both α and Δ degrade, because that implies reducing the memory bandwidth with a direct impact on the time required to copy a packet from/to the device memory.

With the same regulation and configuration, transmission and reception exhibit different behaviors. In particular, transmission is more resilient to the effects of the QoS regulation with respect to the reception. When transmitting, for observing a reduction in the performance, the memory bandwidth shall be below 3.9 million transactions/s (0×40 configuration of the QoS-400). When receiving, limiting the bandwidth

[1]The RX (resp., TX) interrupt is raised when the hardware concludes the reception (resp., transmission) of a complete frame.

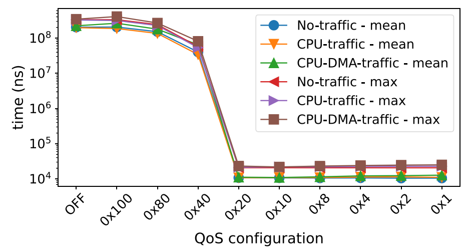

Fig. 11. I/O VM processing time under DoS attack (ns).

to 7.7 million transaction/s (0×80) already degrades the performance. The designer may configure different QoS regulations for the read and write operations of a device to achieve the same transmission and reception performance.

Moreover, also the variation of memory traffic and the presence of an additional device (in our experiments the DMA) degrades the I/O performance. This can be observed in Figs. 9(b) and 10(b), both α and Δ, respectively, decrease and increase with the presence of such disturbances.

However, when the effects of QoS regulation start affecting the performance of this setup (e.g., regulation 0×40 in transmission), a similar degradation is obtained in the setup without additional traffic. Furthermore, by constraining the memory traffic with certain configurations of the QoS-400 (0×20 in transmission), the performance degradation of both α and Δ when also adding the LPD-DMA traffic becomes comparable to the cases in which the LPD-DMA traffic is not present.

D. Resiliency to DoS Attacks

Next, we consider the scenario of a DoS attack, where an attacker continuously sends network packets to the platform, and we show that the proposed solution allows mitigating its effects by varying the QoS value of the regulator of the GEM. To this end, we measured the I/O VM processing, which spans from the reception of an interrupt for a packet and its successful insertion into the Guest VM's queue and includes the I/O scheduler computation time due to the round-robin arbitration and the time required for the execution of the reception ISRs. In particular, the RX ISRs raised by the Ethernet device preempt the execution of the I/O scheduling task of the I/O VM. For this reason, the I/O scheduler processing time depends on the amount of RX interrupts received.

The attacker traffic was generated using the `packETH` tool. Fig. 11 shows the positive effect of QoS regulation, which reduces the speed of the Ethernet device writes and, thus, the frequency of RX interrupts. This experiment was performed using the memory load configurations defined in Section V-C.

Since the rate of packets is high, the number of these interrupts deeply impacts the execution time of the I/O scheduler and, in some cases, it might even prevent the scheduler from being executed. Fig. 11 shows that for up to 3.9 million of transactions per second (regulation 0×40 of the QoS-400 GEM Ethernet regulator) the DoS attack strongly affects the performance. However, when the regulation is set to 1.9 millions of transactions per second (0×20) or lower, this interference is almost negligible and does not impact the latency introduced by the I/O scheduler to dispatch packets.

These results demonstrate that the regulation employed by the QoS-400 plays a key role in guaranteeing the desired performance of the I/O scheduler in the presence of high incoming traffic. This also allows us to protect the system from a denial-of-service attack, which otherwise would completely stall the I/O scheduler preventing the dispatch of any packet.

VI. Related Work

Several prior works targeted I/O virtualized systems. Pérez *et al.* [17] considered the XTratum hypervisor [18], comparing different I/O strategies. Kim *et al.* [27] proposed a priority boosting mechanism for virtual CPUs. Other works targeted the Quest-V separation kernel [19], [20], [28]: a summary of such works is reported by West *et al.* [29].

Targeting the Linux kernel, Kim *et al.* [30] presented a mechanism to dynamically prioritize I/O operations, so as to improve I/O throughput and latency for data-intensive applications, while Craciunas *et al.* [31] designed a system calls scheduler to implement an I/O resource manager.

Several works have been done to guarantee predictable I/O [32], [33], [34] in the context of BlueVisor [32], where I/O virtualization is implemented through custom hardware components prototyped in FPGA. Custom hardware mechanisms to handle I/O have been previously proposed by Pellizzoni and Caccamo [35]. Another work presented a framework to limit the traffic introduced in the I/O bus by I/O peripherals [36]. Schwaricke *et al.* [37] leveraged virtio to implement an interdomain communication mechanism, where a DMA handles inter-VM data transfers, but without targeting I/O virtualization. As in our proposal, [37] uses the QoS-400 regulators, but with a different objective, i.e., regulating the memory interference generated by DMA-based inter-VM data transfers. Richter *et al.* [38] implemented a scheduling mechanism for mitigating the interference generated by malicious VMs using the SRIO-V technology (not present in the Ultrascale+), and without considering real-time requirements and I/O-related memory contention. Pu *et al.* [39] presented a thorough experimental comparison of average-case performance metrics of different solutions to run CPU-bound and I/O-bound workloads in different VMs for cloud computing. Casini *et al.* [16] proposed a latency analysis of I/O virtualized systems, but without providing implementation and not considering I/O regulators. Other researches considered scheduling algorithms to improve I/O with coprocessing units [40]. Li *et al.* [41], [42] proposed a framework (called VATC) to improve the performance of the Xen network virtualization, by introducing networking priorities and a rate-limiting mechanism to avoid starvation for low-priority domains. However, [41], [42] do not support the memory-traffic regulation of I/O devices. The relevance of I/O-related memory contention in the schedulability analysis has been highlighted by Kim *et al.* [43], [44]. The usage of the QoS-400 regulators has been considered very recently by Sohal *et al.* [45], who proposed a framework to predict the timing behavior of tasks by analyzing their memory demand and by Serrano-Cases *et al.* [46], who studied different QoS options provided by the Xilinx Ultrascale+ MPSoC. Zini *et al.* [11] presented an extensive study of I/O-related memory contention and its regulation using the QoS-400 regulators, but without considering I/O virtualization. Table II positions our paper with respect to a selection of the most closely related research, classifying each paper according to: the consideration of (*RT*) real-time constraints or (*Section*) security features, the consideration of I/O virtualization (*I/O virt.*), the nature of this article [(*Prac./Th.*), i.e., practical or theoretical], and the usage of QoS regulators for virtualized I/O devices (*I/O reg.*).

TABLE II
POSITION WITH RESPECT TO A SELECTION OF RELATED RESEARCH

Paper	RT	Sec.	I/O virt.	Prac./Th.	I/O reg.
[41,42]	YES	NO	SW	Prac.	NO
[37]	YES	NO	NO	Prac./Th.	NO
[29]	YES	YES	HW/SW	Prac./Th.	NO
[38]	NO	YES	SRIOV	Prac.	NO
[47,48]	YES	NO	FPGA	Prac.	NO
[17]	YES	NO	SW	Prac.	NO
[11,46]	YES	NO	NO	Prac.	YES
[35]	YES	NO	NO	Prac./Th.	NO
[49]	YES	NO	SW	Th.	NO
This Work	**YES**	**YES**	***SW***	***Prac.***	***YES***

Overall, to the best of our knowledge, no previous work provided an I/O virtualization mechanism capable of controlling the I/O-related memory contention.

VII. CONCLUSION

This article proposed an I/O virtualization mechanism capable of controlling the I/O-related memory contention produced by virtualized devices, implementing it on a Xilinx Ultrascale+ MPSoC using the QoS-400 regulators. Then, we showed how the GEM can be virtualized using our framework. Finally, we reported the results of an extensive evaluation we performed to compare with the Xen hypervisor, which showed improvements up to 8×, and to derive the (α, Δ) parameters. A key research direction for future work consists in using profiled (α, Δ) parameters and the mechanisms developed in this article as a key building block to derive automatic strategies to configure the regulation of the memory traffic to meet both I/O and CPU-time performance. Other directions for future research include the evaluation of different scheduling strategies for the I/O VM, the use of hardware-assisted virtualization [47], and the design of mechanisms to simultaneously handle I/O devices and hardware accelerators [4], [50].

REFERENCES

[1] G. Heiser, "Virtualizing embedded systems—Why bother?" in *Proc. 48th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2011, pp. 901–905.

[2] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2013, pp. 55–64.

[3] F. Restuccia, A. Biondi, M. Marinoni, and G. Buttazzo, "Safely preventing unbounded delays during bus transactions in FPGA-based SoC," in *Proc. IEEE 28th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2020, pp. 129–137.

[4] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses," in *Proc. 21st Int. Conf. Real-Time Netw. Syst. (RTNS)*, 2017, pp. 48–57.

[5] A. Biondi *et al.*, "SPHERE: A multi-SoC architecture for next-generation cyber-physical systems based on heterogeneous platforms," *IEEE Access*, vol. 9, pp. 75446–75459, 2021.

[6] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, 2018, pp. 1651–1657.

[7] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, "Co-optimizing performance and memory footprint via integrated CPU/GPU memory management, an implementation on autonomous driving platform," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2020, pp. 310–323.

[8] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *Proc. 31st Euromicro Conf. Real-Time Syst. (ECRTS)*, 2019, pp. 1–23.

[9] *Zynq UltraScale+ Device—Technical Reference Manual*, Xilinx, San Jose, CA, USA, 2020.

[10] *ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service—Supplement to ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual*, ARM, Cambridge, U.K., 2016.

[11] M. Zini, G. Cicero, D. Casini, and A. Biondi, "Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms," *Softw. Pract. Exp.*, vol. 52, no. 5, pp. 1095–1113, 2021.

[12] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proc. 7th IEEE Real-Time Technol. Appl. Symp.*, 2001, pp. 75–84.

[13] P. Barham *et al.*, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, 2003.

[14] *AMBA AXI and ACE Protocol Specification*, ARM, Cambridge, U.K., 2020.

[15] "Clare software stack." 2022. [Online]. Available: https://accelerat.eu/clare

[16] D. Casini, A. Biondi, G. Cicero, and G. Buttazzo, "Latency analysis of I/O virtualization techniques in hypervisor-based real-time systems," in *Proc. 27th IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2021, pp. 306–319.

[17] H. Pérez, J. J. Gutiérrez, S. Peiro, and A. Crespo, "Distributed architecture for developing mixed-criticality systems in multi-core platforms," *J. Syst. Softw.*, vol. 123, pp. 145–159, Jan. 2017.

[18] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach," in *Proc. Eu. Dependable Comput. Conf.*, Apr. 2010, pp. 67–72.

[19] M. Danish, Y. Li, and R. West, "Virtual-CPU scheduling in the quest operating system," in *Proc. 17th IEEE Real-Time Embedded Technol. Appl. Symp.*, 2011, pp. 169–179.

[20] E. Missimer, K. Missimer, and R. West, "Mixed-criticality scheduling with I/O," in *Proc. 28th Euromicro Conf. Real-Time Syst. (ECRTS)*, Jul. 2016, pp. 120–130.

[21] B. B. Brandenburg, "Multiprocessor real-time locking protocols," in *Handbook of Real-Time Computing*. Singapore: Springer, 2020, pp. 1–99.

[22] D. Abramson *et al.*, "Intel virtualization technology for directed I/O," *Intel Technol. J.*, vol. 10, no. 3, pp. 179–192, 2006.

[23] "PCI-SIG. SR-IOV Website." 2022. [Online]. Available: http://pcisig.com/

[24] M. Torquati, "Single-producer/single-consumer queues on shared cache multi-core systems," 2010, *arXiv:1012.1824*.

[25] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008.

[26] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2016, pp. 1–12.

[27] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for I/O performance," in *Proc. VEE*, 2009, pp. 101–110.

[28] Y. Li, R. West, Z. Cheng, and E. Missimer, "Predictable communication and migration in the Quest-V separation kernel," in *Proc. IEEE Real-Time Syst. Symp.*, 2014, pp. 272–283.

[29] R. West, Y. Li, E. Missimer, and M. Danish, "A virtualized separation kernel for mixed-criticality systems," *ACM Trans. Comput. Syst.*, vol. 34, no. 3, p. 8, Jun. 2016.

[30] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O path: A holistic approach for application performance," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, 2017, pp. 345–358.

[31] S. S. Craciunas, C. M. Kirsch, and H. Röck, "I/O resource management through system call scheduling," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 44–54, Jul. 2008.

[32] Z. Jiang, N. C. Audsley, and P. Dong, "BlueVisor: A scalable real-time hardware hypervisor for many-core embedded systems," in *Proc. Real-Time Embedded Technol. Appl. Symp.*, Apr. 2018, pp. 75–84.

[33] Z. Jiang, N. Audsley, and P. Dong, "BlueIO: A scalable real-time hardware I/O virtualization system for many-core embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 3, p. 19, 2019.

[34] Z. Jiang, K. Yang, Y. Ma, N. Fisher, N. C. Audsley, and Z. Dong, "I/O-GUARD: Hardware/software co-design for I/O virtualization with guaranteed real-time performance," in *Proc. 58th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, 2021, pp. 1159–1164.

[35] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on WCET analysis of real-time embedded systems," *IEEE Trans. Comput.*, vol. 59, no. 3, pp. 400–415, Mar. 2010.

[36] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha, "Real-time I/O management system with cots peripherals," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 45–58, Jan. 2013.

[37] G. Schwäricke *et al.*, "A real-time virtio-based framework for predictable inter-VM communication," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, 2021, pp. 27–40.

[38] A. Richter, C. Herber, S. Wallentowitz, T. Wild, and A. Herkersdorf, "A hardware/software approach for mitigating performance interference effects in virtualized environments using SR-IoV," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, 2015, pp. 950–957.

[39] X. Pu *et al.*, "Who is your neighbor: Net I/O performance interference in virtualized clouds," *IEEE Trans. Services Comput.*, vol. 6, no. 3, pp. 314–329, Jul.–Sep. 2013.

[40] S. Zhao, Z. Jiang, X. Dai, I. Bate, I. Habli, and W. Chang, "Timing-accurate general-purpose I/O for multi- and many-core systems: Scheduling and hardware support," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

[41] C. Li, S. Xi, C. Lu, C. D. Gill, and R. Guerin, "Prioritizing soft real-time network traffic in virtualized hosts based on Xen," in *Proc. 21st IEEE Real-Time Embedded Technol. Appl. Symp.*, 2015, pp. 145–156.

[42] C. Li, S. Xi, C. Lu, R. Guérin, and C. D. Gill, "Virtualization-aware traffic control for soft real-time network traffic on Xen," *IEEE/ACM Trans. Netw.*, vol. 30, no. 1, pp. 257–270, Feb. 2022.

[43] N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter, "Supporting I/O and IPC via fine-grained os isolation for mixed-criticality real-time tasks," in *Proc. 21st Int. Conf. Real-Time Netw. Syst. (RTNS)*, 2018, pp. 191–201.

[44] N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter, "Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks," *Real-Time Syst.*, vol. 56, no. 4, pp. 191–201, 2020.

[45] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "E-warp: A system-wide framework for memory bandwidth profiling and management," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, 2020, pp. 345–357.

[46] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, "Leveraging hardware QoS to control contention in the Xilinx Zynq UltraScale+ MPSoC," in *Proc. 33rd Euromicro Conf. Real-Time Syst. (ECRTS)*, 2021, pp. 1–26.

[47] Z. Jiang and N. Audsley, "VCDC: The virtualized complicated device controller," in *Proc. 29th Euromicro Conf. Real-Time Syst. (ECRTS)*, 2017, pp. 1–21.

[48] Z. Jiang, N. Audsley, P. Dong, N. Guan, X. Dai, and L. Wei, "MCS-IOV: Real-time I/O virtualization for mixed-criticality systems," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, 2019, pp. 326–338.

[49] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *Proc. 26th IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2020, pp. 239–252.

[50] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "AXI HyperConnect: A predictable, hypervisor-level AXI interconnect for hardware accelerators in FPGA SoC," in *Proc. 57th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Sep. 2020, pp. 1–6.

Niccolò Borgioli (Student Member, IEEE) received the master's degree (*cum laude*) in embedded computing systems engineering jointly offered by the Scuola Superiore Sant'Anna, Pisa, Italy, and the University of Pisa, Pisa, in 2020. He is currently pursuing the Ph.D. degree with the Real-Time Systems (ReTiS) Laboratory, Scuola Superiore Sant'Anna of Pisa.

His research interests include the design and implementation of real-time operating systems and hypervisors, cyber-physical systems, machine learning, and cybersecurity.

Matteo Zini received the master's degree (*cum laude*) in embedded computing systems engineering jointly offered by the Scuola Superiore Sant'Anna of Pisa, Pisa, Italy, and the University of Pisa, Pisa, in 2020. He is currently pursuing the Ph.D. degree with the Real-Time Systems (ReTiS) Laboratory, Scuola Superiore Sant'Anna of Pisa.

His research interests include the design and implementation of real-time operating systems and hypervisors, cyber-physical systems, and analysis of the effects of memory interference on schedulability.

Daniel Casini (Member, IEEE) received the master's degree (*cum laude*) in embedded computing systems engineering jointly offered by the Scuola Superiore Sant'Anna of Pisa, Pisa, Italy, and the University of Pisa, Pisa, in 2016, and the Ph.D. degree (with Hons.) in computer engineering from the Scuola Superiore Sant'Anna of Pisa in 2020.

He is an Assistant Professor with the Real-Time Systems (ReTiS) Laboratory, Scuola Superiore Sant'Anna of Pisa. In 2019, he has been visiting scholar with the Max Planck Institute for Software Systems, Saarbrücken, Germany. His research interests include software predictability in multiprocessor systems, schedulability analysis, synchronization protocols, and the design and implementation of real-time operating systems and hypervisors.

Giorgiomaria Cicero received the master's degree (*cum laude*) in embedded computing systems engineering jointly offered by the Scuola Superiore Sant'Anna of Pisa, Pisa, Italy, and University of Pisa, Pisa, in 2017.

He is Senior Research Fellow with the Real-Time Systems (ReTiS) Laboratory, Scuola Superiore Sant'Anna of Pisa, and a CEO ad a Co-Founder with Accelerat Srl, Ghezzano, Italy, a spin-off company of Scuola Superiore Sant'Anna focused on software solutions for safe, secure, and time-predictable cyber-physical systems. He has been visiting trainee with the European Space Agency (ESTEC, Netherlands), Noordwijk, The Netherlands. His research interests include software predictability in multiprocessor systems and heterogeneous platforms, system-level cyber-security hardening techniques, and design and implementation of real-time operating systems and hypervisors.

Alessandro Biondi (Member, IEEE) received the master's degree (*cum laude*) in computer engineering from the University of Pisa, Italy, in 2013, within the excellence program, and the Ph.D. degree in computer engineering from the Scuola Superiore Sant'Anna, Pisa, Italy, in 2017, under the supervision of Prof. G. Buttazzo and Prof. M. Di Natale.

He is an Associate Professor with the Real-Time Systems (ReTiS) Laboratory, Scuola Superiore Sant'Anna. In 2016, he has been visiting scholar with the Max Planck Institute for Software Systems, Saarbrücken, Germany. His research interests include design and implementation of real-time operating systems and hypervisors, schedulability analysis, cyber-physical systems, synchronization protocols, and safe and secure machine learning.

Dr. Biondi was recipient of six Best Paper Awards, one Outstanding Paper Award, the ACM SIGBED Early Career Award in 2019, and the EDAA Dissertation Award in 2017.

Giorgio Buttazzo (Fellow, IEEE) received the graduated degree in electronic engineering from the University of Pisa, Pisa, Italy, the M.S. degree in computer science from the University of Pennsylvania, Philadelphia, PA, USA, in 1987, and the Ph.D. degree in computer engineering from the Scuola Superiore Sant'Anna of Pisa, Pisa, in 1991.

He is a Full Professor of computer engineering with the Scuola Superiore Sant'Anna of Pisa. He has authored 7 books on real-time systems and more than 300 papers in the field of real-time systems, robotics, and neural networks.

Dr. Buttazzo receiving 13 Best Paper Awards. He has been Editor-in-Chief of *Real-Time Systems*, and an Associate Editor of the *ACM Transactions on Cyber-Physical Systems*.

Open Access funding provided by 'Scuola Superiore "S.Anna" di Studi Universitari e di Perfezionamento' within the CRUI-CARE Agreement