Latency Analysis of I/O Virtualization Techniques in Hypervisor-Based Real-Time Systems

Daniel Casini*†, Alessandro Biondi*†, Giorgiomaria Cicero*, and Giorgio Buttazzo*†
*TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
†Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

***Abstract*—Nowadays, hypervisors are the standard solution to integrate different domains into a shared hardware platform, while providing safety, security, and predictability. To this end, a hypervisor virtualizes the physical platform and orchestrates the access to each component. When the system needs to comply with certification requirements for safety-critical systems, virtualization latencies need to be analytically bounded for providing off-line guarantees. This paper presents a detailed modeling of three I/O virtualization techniques, providing analytical bounds for each of them under different metrics. Experimental results compare the bounds for a case study and quantify the contribution due to different sources of delay.**

I. Introduction

In the last decade, the problem of reducing size, weight, power, and cost in automotive components (called the SWAP-c [1] problem) gave rise to an increasing effort for integrating multiple applications in the same platform. This is particularly relevant for embedded and cyber-physical systems, where resources are typically scarce, and becomes crucial in the presence of safety-critical systems such as automotive and avionics. On the other hand, integrated applications must retain high predictability, especially when certification is mandatory. Nowadays, *hypervisors* are the standard solution to virtualize a shared hardware platform, allowing to integrate different applications (also called *domains*) while preserving predictability, safety, and security thanks to many isolation strategies developed over the years [2]–[5]. Such mechanisms include cache coloring [4], DRAM bank partitioning [6], memory bandwidth reservation [7], and temporal isolation algorithms [8]. In this way, different domains can safely run applications with different levels of criticality, potentially under different operating systems (OSes), while sharing the same hardware platform, thus greatly alleviating the SWAP-c.

With the current increasing interest toward embedding artificial intelligence in modern automotive systems, virtualization is also a key means to isolate feature-rich OSes (e.g., Linux) from real-time OSes [9]. Indeed, in modern autonomous-driving applications, both domains have to co-exist. For instance, a feature-rich OS may be essential to timely execute a deep neural network on a hardware-accelerator by providing drivers and software stacks that are not available for most of the other OSes. On the other hand, a real-time kernel may be in charge of executing safety-critical operations based on results produced by applications running in other domains.

In this context, it is of the utmost importance for the virtualized system to provide predictable mechanisms to access input/output (I/O) devices, which can be, in turn, virtualized. In safety-critical systems, I/O virtualization techniques must be accompanied by a proper timing analysis that provides latency bounds for off-line guarantees.

Contribution. This paper presents a fine-grained modeling and analysis of different I/O virtualization techniques. For each of them, response times and latencies due to input and output operations are analytically bounded. This provides the basis to drive application designers in setting different parameters (e.g., ISRs priorities) while guaranteeing specific worst-case I/O latency requirements. An experimental study reports on the performance of the bounds obtained with the different virtualization techniques for a case study based on the 2019 WATERS Industrial Challenge by Bosch [10].

II. Related Work

Prior work mostly targeted empirical evaluations or hardware-assisted I/O virtualization and management [26]–[32]. Two of the most widespread hardware-assisted I/O virtualization technologies are Intel VT-d [33], which allows to assign I/O devices to VMs while providing VM routing for device interrupts, and SR-IOV [34], which improves the management of PCIe devices. Jiang et al. [35] implemented in hardware different I/O management components [17, 18, 36, 37] in the context of BlueVisor [38]. Jiang and Audsley [17] proposed the Virtualized Complicated Device Controller (VCDC), a hardware component to perform I/O requests directly from virtual machines (VMs), bypassing the guest operating system (OS) with improved performance and predictability. Münch et al. [19, 20, 39] studied techniques for exploiting different hardware features to assist I/O virtualization, e.g., the I/O memory management unit (IOMMU) [19] and the I/O Memory Protection Unit (IOMPU) [20].

In the context of the Quest-V separation kernel, Danish et al. [11] proposed the Priority Inheritance Bandwidth-preserving Server (PIBS), where I/O-handling virtual CPUs inherit the priority of those that originally issued the I/O request. No bounds for I/O latencies are provided. Li et a. [13] proposed an approach to perform real-time communication in Quest-V. Worst-case bounds are provided only for the round-trip delay of the inter-domain communication mechanisms, and without providing formal proofs. Bounds for I/O delays are not provided. A method for migrating tasks among VM is also proposed. The mechanisms available in Quest-V are summarized by West et al. [14], where its support for Intel

TABLE I: Comparison of a selection of the related work.

Paper	Virtualization	Analytical I/O latencies	RT bounds	Proofs	Custom HW/SW	Context
Danish et al. [11], Missimer et al. [12]	SW	NO	YES	NO	PIBS	Quest-V
Li et al. [13]	SW	NO	YES	NO	Migration/Inter-VM comm.	Quest-V
West et al. [14]	HW/SW	NO	YES	NO	PIBS/Intel VT-x	Quest-V
Golchin et al. [15]	SW	YES	YES	NO	Tuned Pipes	Boomerang
Masrur et al. [16]	SW	NO	YES	NO	SEDF/PSEDF	Xen
Jiang et al. [17, 18]	HW	NO	NO	NO	Custom HW	BlueVisor/MC
Perez et al. [8]	SW	NO	NO	NO	NO	Xtratum
Munch et al. [19], [20]	HW	NO	NO	NO	SRIO-V/IOMMU - IOMPU	Avionics
Pellizzoni and Caccamo [21]–[23]	NO	NO	YES	YES	NO	Analysis
Bak et al. [24], Betti et al. [25]	HW	NO	YES	YES	Custom HW	HW Design
Kim et al. [26]	NO	NO	NO	NO	MC^2 Project	MC^2
This Paper	***SW***	***YES***	***YES***	***YES***	***No custom HW***	***Generic***

VT-x hardware virtualization is described. Missimer et al. [12] proposed a response-time analysis for sporadic servers and PIBS servers in the context of Quest-V and adaptive mixed-criticality scheduling, without formally proving the analysis.

Masrur et al. [16] analyzed the problem of guaranteeing real-time constraints in the Xen hypervisor [40], also proposing a new scheduler for reducing delay and jitter. Golchin et al. [15] implemented an I/O system comprising real-time task pipelines in the Boomerang hypervisor, extending the concept of "tuned pipes", previously introduced for USB devices [41], and empirically comparing it with a standalone Linux and the ACRN hypervisor [42]. Formulas are also provided to guarantee QoS in end-to-end latencies, without proofs. Perez et al. [8] empirically evaluated communication performance in the context of ARINC-like systems based on the XTratum hypervisor [43]. Ramsauer et al. [44] empirically measured the interrupt latency of the Jailhouse hypervisor on an Nvidia Jetson TK1. Beckert et al. [45] presented an approach for reducing interrupt latencies in hypervisors using TDMA.

Pellizzoni and Caccamo [21]–[23] presented an analysis to compute a worst-case execution time bound considering I/O peripherals. Pellizzoni et al. [46] also presented an analysis of the PCI bus. Bak et al. [24] proposed a framework to control the I/O traffic on I/O peripherals in a COTS-based embedded system, implemented in hardware on a FPGA board, and later extended by Betti et al. [25]. Tabish et al. [47, 48] proposed a real-time scratchpad-centric OS, with support for predictable I/O for tasks using a three-phase execution model [49]–[52].

A selection of the related work is compared in Table I, where each paper is classified according to: **(i)** virtualization type (software, hardware or none), **(ii)** the presence of analytical bounds to I/O latencies and general real-time bounds (e.g., to ensure schedulability), **(iii)** the presence of proofs in support of theoretical results, **(iv)** the need for custom hardware and/or software, and **(v)** the context of the paper.

Why this work? Overall, most prior work considered hardware-assisted mechanisms (which may not be available in most systems), the presence of *custom* scheduling mechanism and algorithms (e.g., PIBS servers), and empirical studies. Among the previous research providing analytical bounds, none of them targeted specific I/O virtualization latencies as those presented in this paper, and most of them presented the results without providing proofs. In contrast, this work provides a detailed modeling and a formally-proved analysis for three different I/O virtualization scenarios that can be achieved with *standard inter-domain communication mechanisms commonly available in most hypervisors*, without relying on specific hardware features or custom scheduling algorithms that may not be always available.

III. System Model

The computing platform considered in this paper consists of a set $\mathcal{P} = \{p_1, \ldots, p_m\}$ of m identical (physical) cores and a set $\mathcal{D} = \{d_1, \ldots, d_z\}$ of I/O devices. Each device is provided with a corresponding DMA engine that can perform simultaneous I/O operations. A bare-metal hypervisor is in charge of virtualizing the computational resources to different domains, each referred to as *virtual machine* (VM). Overall, the hypervisor handles a set $\mathcal{V} = \{v_1 \ldots, v_w\}$ of w VMs. We consider a partitioning hypervisor, i.e., each VM $v_i \in \mathcal{V}$ is statically assigned to a set of cores $\mathcal{C}(v_i)$ in an exclusive way, i.e., no other VM than v_i uses the cores in $\mathcal{C}(v_i)$.

Task Model. Each VM $v_i \in \mathcal{V}$ is in charge of managing a set Γ_i^{VM} of real-time tasks. Each task $\tau_j \in \Gamma_i^{\text{VM}}$ is a computational activity statically assigned to a VM $\mathcal{M}(\tau_j) = v_i \in \mathcal{V}$ and characterized by a worst-case execution time (WCET) C_j, a relative deadline D_j, and a fixed priority π_j.

Each task τ_j releases a potentially-infinite sequence of instances called *jobs*. Jobs may be activated by multiple types of stimuli, e.g., a periodic timer implemented within the guest operating system (OS) of a VM, or an external event (interrupt) caused by an I/O device. In both cases, an *event arrival curve* $\eta_j(\delta)$ is associated with each task τ_j to upper-bound the maximum number of release events of τ_j in any interval of length δ. For example, a periodic task τ_j with period T_j is modeled with an arrival curve $\eta_j(\delta) = \lceil \delta / T_j \rceil$. Each task is characterized by an *inter-job precedence constraint*, i.e., at most one job of each task can be pending at the same time, which can be enforced by setting a deadline $D_i \leq T_i$, where T_i is the minimum time elapsed between the release of two consecutive jobs of τ_i, and checking the schedulability of τ_i. A task τ_i is said to be schedulable if all jobs complete within D_i time units from their release. Each VM can access a set $\mathcal{H}(v_i) \subseteq \mathcal{D}$ of I/O devices. Similarly, the set of devices accessed by τ_j is referred to as $\mathcal{H}(\tau_j)$. For the sake of

conciseness in the notation, for each VM $v_i \in \mathcal{V}$, we assume that at most one task $\tau_j \in \Gamma_i^{\text{VM}}$ may access each I/O device $d_f \in \mathcal{H}(v_i)$. Such a task is denoted by $\mathcal{T}(v_i, d_f)$. We will discuss in Section VIII about how to relax this assumption. Tasks do not self-suspend and are executed by each VM according to a partitioned fixed-priority scheduling policy, i.e., each task τ_j is statically assigned to a core $\mathcal{C}(\tau_j) \in \mathcal{C}(v_i)$, where $v_i = \mathcal{M}(\tau_j)$. The set of tasks running on core p_k is denoted as Γ_k. The set $\mathsf{hep}_k(\tau_j)$ denotes all other tasks with a higher or equal priority than τ_j allocated on p_k. Similarly, set $\mathsf{lp}_k(\tau_j)$ denotes all tasks with a lower priority than τ_j.

ISR Model. The system also handles a set of interrupt-service routines (ISRs) S. Each ISR $\sigma_a \in S$ is characterized by a WCET C_a, a fixed-priority π_a, and releases a potentially-infinite sequence of instances. The maximum number of release events of σ_a in an interval of length δ is upper-bounded by $\eta_a(\delta)$. ISRs are always assigned to higher priorities than tasks. Each ISR $\sigma_a \in S$ executes on a specific core $p_k = \mathcal{C}(\sigma_a) \in \mathcal{P}$. The set of ISRs running on p_k is denoted by S_k. The hypervisor reacts to all interrupts raised by the I/O devices and is in charge of dispatching them to the target VMs, i.e., by issuing virtual interrupts to the VMs [53]. Hence, for each interrupt, there are two ISRs serving it: one at the hypervisor level (denoted as H-ISR) and one for the target VM (denoted as VM-ISR). To model this behavior, each ISR $\sigma_a \in S$ is associated with a type $u_a \in \{H, V\}$, where H-ISR have $u_a = H$ and VM-ISRs have $u_a = V$. Each VM-ISR σ_a is triggered by a corresponding H-ISR σ_b denoted by $\mathcal{F}(\sigma_a) = \sigma_b$. For each core $p_k \in \mathcal{P}$, VM-ISRs are included in the set S_k^{VM}, whereas H-ISRs are in the set S_k^{H}, such that $S_k^{\text{VM}} \cup S_k^{\text{H}} = S_k$. H-ISRs have always higher priorities than their VM-level counterparts. Sets $\mathsf{hep}_k(\sigma_a)$ and $\mathsf{lp}_k(\sigma_a)$ are defined as for the case of tasks by replacing τ_j with σ_a and Γ_k with S_k. Recent interrupt controllers provide direct interrupt routing, allowing to avoid the need for H-ISRs: they can be handled by the methods of this paper by considering VM-ISRs only. Tasks, ISRs, and device DMAs may experience memory contention when accessing main memory and shared caches. The corresponding delays are assumed to be already factored in the WCETs and in the I/O transfer delays (introduced later). They can be bounded using state-of-the-art methods, e.g., by considering the individual worst-case of each memory request [54]. The integration of other methods to account for DRAM and cache contention, e.g., [55]–[62], requires further details in the modeling of memory accesses, and it is left as a future work. Also, overheads due to the guest OS and the hypervisor are assumed to be already included in the timing parameters. Delays to transfer I/O data are not included in the WCETs and they are discussed later.

Non-Interruptible Sections. Tasks and ISRs may require to enforce mutual exclusion by temporarily masking interrupts and disabling preemptions. To model such behaviors, a system ceiling Π_k is provided with each core p_k. Informally, the system ceiling is a threshold on the priority that needs to be overtaken to enable preemption. This paper does not consider tasks accessing lock-protected shared resources, although it can be easily extended to support them. Hence, we model only three values for the ceiling: $\rho_k^{\text{LOW}} = \min\{\pi_j \mid \tau_j \in \Gamma_k\}$ (i.e., all tasks and ISRs can preempt), $\rho_k^{\text{VM}} = \max\{\pi_j \mid \sigma_j \in S_k^{\text{VM}}\}$ (i.e., only H-ISRs can preempt), and $\rho_k^{\text{H}} = \max\{\pi_j \mid \sigma_j \in S_k^{\text{H}}\}$ (i.e., preemption is disabled). The availability of three levels for the system ceiling allows to flexibly model many use cases commonly found in hypervisor-based systems, e.g., the masking of VM-level ISRs only or letting ISRs or hypercalls to disable interruptions to avoid race conditions.

Due to the priority ordering among tasks and VM-level and hypervisor-level ISRs, it follows that $\rho_k^{\text{H}} > \rho_k^{\text{VM}} > \rho_k^{\text{LOW}}$. Then, a task $\tau_j \in \Gamma_k$ or an ISR $\sigma_j \in S_k$ can preempt another task τ_c or ISR σ_c only if $\pi_j > max(\Pi_k, \pi_c)$. Hence, when $\Pi_k = \rho_k^{\text{LOW}}$, preemptions are allowed; when $\Pi_k = \rho_k^{\text{VM}}$, only H-ISRs can preempt; finally, when $\Pi_k = \rho_k^{\text{H}}$, preemption is forbidden. Tasks and ISRs may enter one or more non-interruptible regions (NIRs) by raising the value of Π_k to ρ_k^{VM} or ρ_k^{H}. When the region completes, Π_k is set to ρ_k^{LOW}. To enforce safety and security in the system, only H-ISRs can raise Π_k to ρ_k^{H}. Furthermore, they do never raise Π_k to ρ_k^{VM}, as $\forall \sigma_h \in S : u_h = H \Rightarrow \pi_h > \rho_k^{\text{VM}}$. For each task $\tau_j \in \Gamma_k$ or ISR $\sigma_j \in S_k$, ω_j bounds the length of the longest non-interruptible region executed with $\Pi_k = \rho_k^{\text{H}}$, in the case of H-ISRs, and with $\Pi_k = \rho_k^{\text{VM}}$, for VM-ISRs or tasks. These terms are then used next in the timing analysis to bound the priority-inversion blocking [63, 64].

Hypercalls. Hypercalls are software traps (i.e., exceptions) caused by tasks to invoke the hypervisor and are used to perform privileged operations that cannot be directly executed by the VM. Hypercalls are hence similar to hypervisor-level ISRs but, differently, they are synchronous with respect to the execution of the task requesting it (i.e., tasks wait until hypercalls complete). This allows modeling each hypercall as a part of the task execution subject to *priority elevation* by specifying only two parameters: the WCET of the hypercall and the priority at which it executes. In this work, we focus on hypercalls called for performing I/O operations and, to keep the system model simple, we introduce the necessary notation only when needed in the following. The duration of hypercalls is not included in the WCET of the requesting task.

Communication Model. Each task τ_j performs a set $\mathcal{R}_j$ of I/O requests. Any arbitrary y-th request $r_{j,y}^f \in \mathcal{R}_j$ (with $y \in [1, |\mathcal{R}_j|]$)) involving a task τ_j and a device d_f is characterized by a type $\Delta_{j,y}^f = \{I, O\}$ and a size $sz(r_{j,y}^f)$ expressed in bytes. Requests assigned to type $\Delta_{j,y}^f = I$ are *input operations*, meaning that the data flows from device d_f to task τ_j. Conversely, if $\Delta_{j,y}^f = O$, $r_{j,y}^f$ is an *output operation*, i.e, the data flows from τ_j to d_f. Each I/O device is provided with two I/O buffers (stored in memory): one for input and one for output data. Buffers are assumed to be large enough so that they never become full. To trigger an output operation, the corresponding data needs to be placed in the corresponding I/O buffer: then, the DMA associated with the I/O device is instructed to perform the copy from the I/O buffers to the device. Input operations involve the copy from the I/O device

to its I/O buffer.

Device Events. When data is received from the external environment, it is copied into the I/O buffers of the corresponding device $d_f \in \mathcal{D}$. Such data arrivals are modeled by associating a set $\mathcal{E}_f$ of *device events* with d_f. Device events $e^{\mathrm{f}}_{j,y} \in \mathcal{E}_f$ (originated from a device d_f for a task τ_j) are associated with a size $sz(e^{\mathrm{f}}_{j,y})$. They do not have a type (data arriving from the external environment only need to be copied into the I/O buffers). The arrival of device events is notified to the system with a VM-level ISR $\mathcal{I}(e^{\mathrm{f}}_{j,y}) = \sigma_v$.

Table of symbols. Table II summarizes the main symbols introduced in this paper.

TABLE II: Table of symbols

Symbol	Description
p_k	k-th physical core
v_i	i-th virtual machine
d_y	y-th I/O device
$\mathcal{P}$	set of the physical cores
$\mathcal{V}$	set of virtual machines
$\mathcal{D}$	set of I/O devices
τ_j	j-th task
σ_h	h-th ISR
C_j	j-th task WCET
π_j	j-th ISR/task deadline
$\eta_j(\delta)$	j-th task/ISR arrival curve
ω_j	length of the longest NIR
u_h	type of the h-th ISR (*H* or *V*)
Γ_k	set of tasks allocated to p_k
Γ_i^{VM}	set of tasks managed by v_i
S_k^{H}	set of hypervisor-level ISRs allocated to p_k
S_k^{VM}	set of VM-level ISRs allocated to p_k
S_k	set of ISRs allocated to p_k
$\mathcal{C}(v_i)$	set of cores exclusively assigned to v_i
$\mathcal{C}(\tau_j)/\mathcal{C}(\sigma_h)$	core where τ_j/σ_h is allocated to
$\mathcal{M}(\tau_j)$	virtual machine managing τ_j
$\mathcal{H}(\tau_j)/\mathcal{H}(v_i)$	set of devices assigned to τ_j/v_i
$\mathcal{T}(v_i, d_f)$	$\tau_j \in \Gamma_i^{\mathrm{VM}}$ accessing d_f
$\mathcal{F}(\sigma_a)$	hypervisor-level ISR associated with σ_a
$\mathcal{R}_j$	set of requests issued by τ_j
$r^f_{j,y}$	y-th request involving τ_j and d_f
$\Delta^f_{j,y}$	type of $r^f_{j,y}$ (*I* or *O*)
$sz(r^f_{j,y})$	number of bytes copied for $r^f_{j,y}$
$\mathcal{E}_f$	set of device events originated by d_f
$e^{\mathrm{f}}_{j,y}$	y-th device event involving τ_j and d_f
$\mathcal{I}(e^{\mathrm{f}}_{j,y})$	VM-level ISR associated with $e^{\mathrm{f}}_{j,y}$
$sz(e^{\mathrm{f}}_{j,y})$	number of bytes copied for $e^{\mathrm{f}}_{j,y}$

IV. Modeling I/O Virtualization Techniques

Clearly, many different I/O virtualization techniques may be devised. To keep the analysis framework as general as possible, without relying on custom mechanisms that might not be practically available in most hypervisors, we focus on three I/O management solutions that we deem particularly relevant, as they can be realized with standard inter-VM communication mechanisms available in almost all hypervisors. Thanks to its generality, the analysis framework can be extended for other techniques.

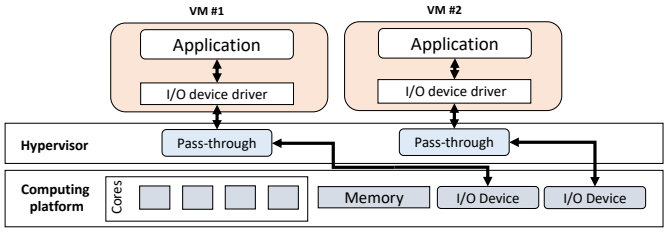

Fig. 1: Pass-through I/O virtualization.

A. Pass-Through I/O

When using *pass-through* I/O (see Figure 1), each VM is exclusively associated with one or more I/O devices. Clearly, this scenario has the disadvantage of forbidding the device sharing among different VMs. On the other hand, it is simple to implement and it may allow meeting *stringent latency requirements* since it avoids device contention among VMs. Under this setting, each I/O input buffer is provided with an integer value c_f denoting the number of bytes related to new data (e.g., produced by d_f but never consumed by $\tau_j \in \Gamma_i^{\mathrm{VM}}$). Pass-through I/O behaves according to the following rules:

PT1. When device d_f receives data from the external environment (i.e., a device event $e^{\mathrm{f}}_{j,y}$ occurs), it instructs its DMA engine to copy the data in the I/O buffer of d_f. Upon completion, it triggers a H-ISR σ_a (that in turn will trigger a VM-ISR $\sigma_b : \sigma_a = \mathcal{F}(\sigma_b)$) to increment c_f and notify the termination of the data transfer.

PT2. When a task $\tau_j \in \Gamma_i^{\mathrm{VM}}$ performs an *input* operation $r^f_{j,y}$, it tries to copy $sz(r^f_{j,y})$ bytes from the I/O buffer to the task-local memory of τ_j. The memory buffers whose content has been copied are released to the I/O device for future data. If $c_f \geq sz(r^f_{j,y})$ the copy succeeds and c_f is decremented of $sz(r^f_{j,y})$ units; otherwise, it fails.

PT3. When a task $\tau_j \in \Gamma_i^{\mathrm{VM}}$ performs an *output* operation $r^f_{j,y}$, it first copies data into the I/O buffers and then instructs the DMA engine of d_y to copy $sz(r^f_{j,y})$ bytes from the task-local memory of τ_j to the device. Afterwards, it continues executing without blocking. An interrupt notifies the completion of the DMA copy.

The DMA copies require $\lambda^{\mathrm{IN}}_{\mathrm{DMA}}$ and $\lambda^{\mathrm{OUT}}_{\mathrm{DMA}}$ time units for each byte written and read to/from the I/O buffers, respectively. The copies performed by the task require λ^{CP} time units per byte.

B. I/O Para-Virtualization with I/O VM

Often, it is required by multiple VMs to share one or more I/O devices and it is not possible to merely use pass-through I/O. To enable device sharing, we consider the case in which the actual interaction with the devices is performed by a dedicated VM named as *I/O management virtual machine* (I/O VM, e.g., as in [8, 43]), which hosts the execution of the device drivers — see Figure 2. We deem this case particularly relevant as it can be realized by using the standard inter-VM communication mechanisms commonly available in most hypervisors, and we name it I/O para-virtualization, as it

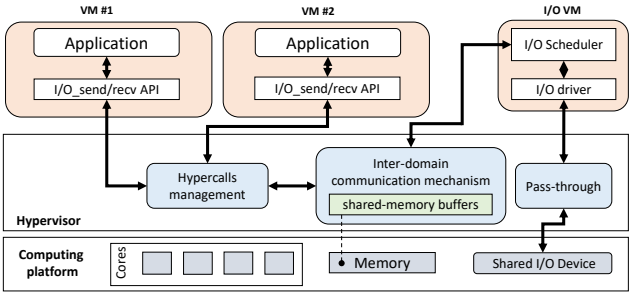

Fig. 2: I/O Para-Virtualization with I/O VM.

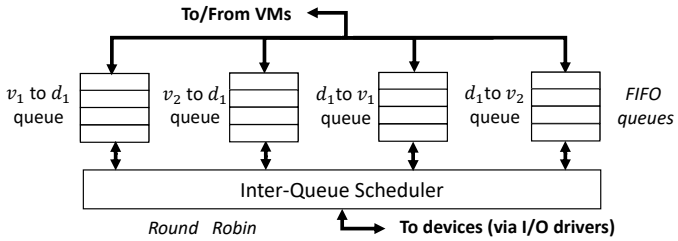

Fig. 3: I/O Scheduler with two VMs sharing an I/O device.

requires a hypervisor-specific API to be offered to VMs. Under this configuration, for each pair (v_i, d_f) with $d_f \in \mathcal{H}(v_i)$, two corresponding memory buffers are maintained in the hypervisor memory space: one for input operations and one for output operations. We assume the memory buffers to be large enough to guarantee that they never become full. Such buffers are used by an *inter-domain communication mechanism* to transfer I/O data between the I/O virtual machine v_{io} and the other VMs. The I/O VM is in charge of performing I/O requests on behalf of the other domains, and it is statically allocated to a single core $p_{io} \in \mathcal{P}$ in an exclusive manner. The I/O VM executes a single task τ_{io}, which implements an *I/O scheduler* (also called *I/O manager*), which enqueues requests from different VMs and dispatches them to each device according to a scheduling policy. Task τ_{io} can access all the devices via pass-through. Hence, determining the behavior of this setting requires defining both **(i)** how each VM interacts with the I/O VM by means of the communication mechanism provided by the hypervisor and, **(ii)** how I/O requests are managed by the I/O scheduler. We start discussing point (i).

Inter-domain communication mechanism. VMs $v_i \in \mathcal{V} \setminus \{v_{io}\}$ interact with I/O devices by means of the inter-domain communication mechanism implemented by the hypervisor. The interaction is regulated by the following rules:

C1. When a task $\tau_j \in \Gamma_i^{\text{VM}}$ performs an input operation $r_{j,y}^f$, it invokes a hypercall that tries to copy $sz(r_{j,y}^f)$ bytes from the input memory buffers stored in the hypervisor memory related to the pair (v_i, d_f) to the task-local memory of τ_j. Each input buffer is associated with a counter $c_{i,f}$ denoting the amount of bytes produced by device d_f (and made available by means of v_{io}) but not yet consumed by v_i. If $c_{i,f} \geq sz(r_{j,y}^f)$ the copy succeeds and $c_{i,f}$ is decremented by $sz(r_{j,y}^f)$; otherwise, it fails.

C2. When a task $\tau_j \in \Gamma_i^{\text{VM}}$ performs an output operation $r_{j,y}^f$, it invokes a hypercall to copy $sz(r_{j,y}^f)$ bytes from the task-local memory of τ_j to the hypervisor output memory buffer related to the pair (v_i, d_f). Contextually, a request $r_{j,y}^f$ with $\Delta_{j,y}^f = O$ is inserted in the output queue of the pair (v_i, d_f) in the I/O scheduler.

Copy operations (from VM buffers to the hypervisor buffers and vice versa) require at most λ^{CP} time units per byte.

I/O scheduler. The I/O scheduler is graphically illustrated in Figure 3. It interacts with two queues of requests provided by the inter-domain communication mechanism for each pair (v_i, d_f), with $d_f \in \mathcal{H}(v_i)$: a queue $Q_{i,f}^{\text{OUT}}$ for data flowing from v_i to d_f (output operations), and a queue $Q_{i,f}^{\text{IN}}$ for data flowing from d_f to v_i (input operations). In each queue, requests are managed in a *first-in first-out* (FIFO) order. An inter-queue dispatcher serves the request at the top of a queue selected with round-robin arbitration[1]. The I/O VM accesses devices according to the rules presented next.

V1. When device d_f receives data from the external environment (i.e., a device event $e_{j,y}^{\text{f}}$ occur), it instructs its DMA engine to copy the data into the I/O buffers of d_f. Upon completion, it triggers a H-ISR σ_a (which in turn triggers a VM-ISR) to increment c_f and notify the termination of the data transfer by inserting a request $r_{j,y}^f$ with $\Delta_{j,y}^f = I$ in the queue of the I/O scheduler.

V2. When the I/O VM serves an input request $r_{j,y}^f$, it copies $sz(r_{j,y}^f)$ bytes from the I/O buffers of d_f to the buffers related to the pair (v_i, d_f), with $\mathcal{M}(\tau_j) = v_i$. The counter c_f is decremented by $sz(r_{j,y}^f)$ units, incrementing $c_{i,f}$ by the same amount (see rule C1). The condition $c_f \geq sz(r_{j,y}^f)$ is guaranteed by rule V1.

V3. When the I/O VM serves an output operation $r_{j,y}^f$, it first copies the data from the hypervisor memory to the I/O buffers. Then, it instructs the DMA engine of d_f to copy $sz(r_{j,y}^f)$ bytes from the I/O buffers to the device. Afterwards, it continues executing without blocking. When the DMA copy terminates, an interrupt notifies completion.

Note that, being memory buffers related to I/O communications in the hypervisor memory, data need to be copied using hypercalls due to the memory-protection mechanisms of the hypervisor to ensure isolation among VMs. Furthermore, hypercalls introduce blocking times to latency-sensitive tasks that may not be tolerated (they cannot be preempted by the tasks). Next, we propose a variant of the I/O Para-Virtualization approach to overcome these issues.

C. I/O Para-Virtualization with I/O VM and Shared Buffers

This approach, illustrated in Figure 4, extends the previous one by storing the memory buffers related to each pair (v_i, d_f), with $d_f \in \mathcal{H}(v_i)$, in a portion of memory shared between v_i

[1]Note that FIFO and round-robin allow providing reasonable fairness guarantees for requests associated with each queue while being predictable policies. For example, they are largely used in locking protocols [65]–[67].

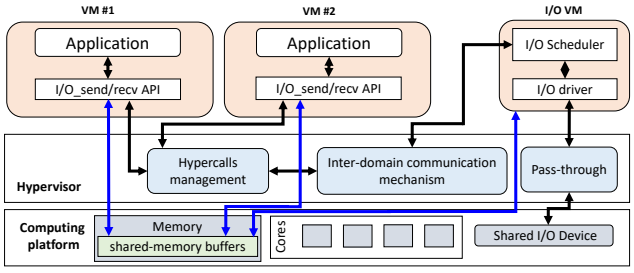

Fig. 4: I/O Para-Virtualization with I/O VM and shared buffers.

and the I/O VM v_{io}. Shared buffers can be implemented with wait-free queues [68]. In this way, the copies from the VM memory to the hypervisor memory mandated by rules C1 and C2 are not needed. Indeed, since the shared memory buffers are accessible by the VM that issues the I/O operations, the tasks can directly use such buffers to produce and consume the data while executing. Counters $c_{i,f}$ defined in rule C1 are also defined in this scenario. Based on these consideration, the following set of rules is defined to specify how an arbitrary VM $v_i \in \mathcal{V} \setminus \{v_{io}\}$ performs I/O operations:

CS1. During its execution, a task $\tau_j \in \Gamma_i^{\text{VM}}$ tries to acquire the buffers for each input operation it is willing to perform during execution. For each *input* operation $r_{j,y}^f$, if $c_{i,f} \geq sz(r_{j,y}^f)$ the access to the shared buffer is granted; otherwise, it fails. Then, the task is allowed to work by directly using such buffers. When the job completes or it explicitly releases one or more of such buffers, they return to the portion of shared memory to store future data. When the buffer related to a request $r_{j,y}^f$ is released, $c_{i,f}$ is decremented by $sz(r_{j,y}^f)$ units.

CS2. During its execution, a task $\tau_j \in \Gamma_i^{\text{VM}}$ acquires the buffers for each output operation it is willing to perform during execution. When a specific *output* operation $r_{j,y}^f$ is completed, it releases the corresponding shared buffer. Contextually, a request $r_{j,y}^h$ with $\Delta_{j,y}^h = O$ is inserted in the queue of the I/O scheduler.

Rules V1, V2, and V3, are left unaltered with the exception that each interaction with the hypervisor memory is replaced with an interaction with the shared-memory buffers.

D. Timing metrics studied in this work

We conclude the section by presenting the metrics used in the paper to evaluate the I/O virtualization scenarios.

*Definition 1 (**Response Time**):* The worst-case response time (WCRT) of a task τ_j is the longest time span elapsed between the release and the completion of one of its jobs, for each possible schedule in which a job of τ_j is executed.

*Definition 2 (**Data Delivery Latency**):* The Input (resp., Output) Data Delivery Latency (IDDL, resp., ODDL) is the longest time span elapsed between the time in which a device (resp., a VM) starts sending the data, and the time in which a target VM (resp., a device) is allowed to consume such data.

*Definition 3 (**Input Processing Latency**):* The Input Processing Latency (IPL) is the longest time span elapsed between the time in which a device starts sending the data, and the time in which a given consumer task completes the execution of a job using such data as input.

The main difference between IPL and IDDL consists in the fact that the former depends on the response time of the consumer task while the latter depends on I/O delays only.

V. Analyzing Pass-Through I/O

First, we start discussing the case of pass-through I/O. For each task $\tau_j \in \Gamma_k$, a response-time bound R_j is derived next.

Lemma 1: The response time of a task $\tau_j \in \Gamma_k$ is bounded by the least positive solution of the following equation:

$$R_j = \overline{C}_j + B_j + \sum_{\tau_h \in \mathsf{hep}_k(\tau_j)} \eta_h(R_j)\overline{C}_h + \sum_{\sigma_h \in S_k} \eta_h(R_j)C_h,$$

where $\overline{C}_j = C_j + \sum_{r_{j,y}^f \in \mathcal{R}_j} sz(r_{j,y}^f) \cdot \lambda^{\text{CP}}$, and $B_j = \max\{\omega_l \mid \tau_l \in \mathsf{lp}_k(\tau_j)\}$.

Proof: The lemma follows from standard response-time analysis provided that low-priority blocking and high-priority interference are bounded. First note that non-interruptible regions executed by a lower-priority task at ceiling level $\Pi_k = \rho_k^{\text{VM}}$ can block τ_j. Regions executed at $\Pi_k = \rho_k^{\text{H}}$ cannot block τ_j as they are due to H-ISRs, which have higher priorities than tasks. Blocking can be experienced only once: as soon the non-interruptible section completes, the blocking task is preempted. For each lower-priority task $\tau_l \in \mathsf{lp}_k(\tau_j)$, ω_l is a bound on the length of the longest non-interruptible region. Consequently, B_j bounds the priority-inversion blocking of τ_j. High-priority interference is given by **(i)** ISRs allocated on the same processor of τ_j (set S_k) and **(ii)** higher-priority tasks, both contributing with up to $\eta_h(R_j)$ instances, each one with a maximum duration of C_h and $\overline{C}_h$, respectively. The execution cost of each τ_j is bounded by its WCET C_j (due to inter-job precedence constraints, see Section III) and the time to copy data from/to the I/O buffers ($\sum_{r_{j,y}^f \in \mathcal{R}_j} sz(r_{j,y}^f) \cdot \lambda^{\text{CP}}$). ■

If $R_j \leq D_j$ then task τ_j is schedulable and R_j is said to be a valid response-time bound. Lemma 1 can be used under the assumption that all the involved parameters are known. This may not always the case for arrival curves. For example, VM-ISRs are triggered when the corresponding H-ISRs complete. As noted in prior works [69]–[72], when a computation is triggered by another one, the arrival curve of the successor needs to account for a release jitter, reflecting the activation delay due to the completion of the predecessor.

Given a VM-ISR $\sigma_v \in S : u_v = V$, its arrival curve can be derived from the one of the corresponding H-ISR $\sigma_h \in S : u_h = H \wedge \mathcal{F}(\sigma_v) = \sigma_h$ as $\eta_v(\delta) = \eta_h(\delta + R_h)$ [72]. It follows that bounding the response time of a task requires knowing the response time of the H-ISRs, needed to define the arrival curve of the corresponding VM-ISRs. Since H-ISRs always have a higher priority than VM-ISRs ones, this issue is solved by computing the response-time bounds in order of priority.

Differently from tasks, ISRs do not have inter-job precedence constraints. Hence, an ISR instance released at a time t_r can suffer self-interference from other instances of the same ISR released prior to time t_r, i.e., multiple instances of the same ISR can interfere with the one under analysis. This

issue is typically solved with extended response-time analysis techniques that holistically study such instances [73, 74]. Due to lack of space, Lemma 2 provides a simpler but conservative bound, leaving its refinement as future work.

Lemma 2: The response time of an ISR $\sigma_a \in S_k$ is bounded by the smallest solution of the equation:
$R_a = B_a + \sum_{\sigma_h \in \mathsf{hep}_k(\sigma_a) \cup \sigma_a} \eta_h(R_a) C_h$, where

$$B_a = \begin{cases} \max\{\omega_l \mid \sigma_l \in \mathsf{lp}_k(\sigma_a) \vee \tau_l \in \Gamma_k\} & \text{if } \sigma_a \in S_k^{\mathrm{VM}}, \\ \max\{\omega_l \mid \sigma_l \in \mathsf{lp}_k(\sigma_a) \cap S_k^{\mathrm{H}}\} & \text{if } \sigma_a \in S_k^{\mathrm{H}}. \end{cases} \tag{1}$$

Proof: Due to the priority assignment, tasks cannot interfere with σ_a. ISRs can be blocked from lower-priority ISRs or tasks raising the system ceiling. The blocking is bounded by B_a. If σ_a is a VM-ISR, B_a includes the length of the longest NIR of both lower-priority ISRs and tasks, which may raise the system ceiling Π_k to ρ_k^{VM} (first branch of Eq. (1)). Recall from Sec. III that VM-ISRs and tasks cannot raise Π_k to ρ_k^{H} and, if σ_a is a H-ISR, $\pi_a > \rho_k^{\mathrm{VM}}$. Hence, σ_a can be blocked only by other H-ISRs with a lower priority raising Π_k to ρ_k^{H} (second branch of Eq. (1)). The lemma follows by noting that ISRs $\sigma_h \in \mathsf{hep}_k(\sigma_a) \cup \sigma_a$ may interfere with at most $\eta_h(R_a)$ jobs executing for at most C_h. ∎

Input Data Delivery Latency. By rule PT1, a H-ISR σ_h is released to signal to the hypervisor the completion of the DMA transfer. However, the target VM is actually notified only after the completion of a corresponding VM-ISR σ_v, which is triggered by σ_h. This functional behavior creates a chain of three pipelined events: **(i)** the DMA copy, **(ii)** the execution of σ_h and, **(iii)** the execution of σ_v.

A simple way to bound the IDDL is to use the basic Compositional Performance Analysis approach (CPA) [73]:

$$L^{IDD}(e^{\mathrm{f}}_{j,y}) = sz(e^{\mathrm{f}}_{j,y}) \cdot \lambda^{\mathrm{IN}}_{\mathrm{DMA}} + R_h + R_v, \tag{2}$$

where $sz(e^{\mathrm{f}}_{j,y}) \cdot \lambda^{\mathrm{IN}}_{\mathrm{DMA}}$ bounds the time required by the DMA to copy the data due to device event $e^{\mathrm{f}}_{j,y}$, while R_h and R_v are WCRT bounds for the H-ISR σ_h and the VM-ISR σ_v. While this approach benefits of simplicity and it is general enough to flexibly handle the case in which σ_h and σ_v run in different cores, it may yield pessimistic bounds as it chains response times computed in independent worst-case scenarios [75, 76].

To improve the analysis precision, we holistically consider a chain ℓ_x consisting of an ordered sequence of ISRs where: **(i)** all the ISRs involved in the chain are allocated to the same core p_k, and **(ii)** there is a precedence constraint between the i-th ISR and the (i+1)-th ISR of the chain. Before focusing on the specific case in which $\ell_x = (\sigma_h, \sigma_v)$, Lemma 3 bounds the priority-inversion blocking for an arbitrary chain.

Lemma 3: Let $\ell_y = (\sigma_a, \ldots, \sigma_w)$ be a chain of ISRs where all ISRs are allocated to the same core p_k. Then, ℓ_y may be blocked at most once by at most B_a time units from Eq. (1).

Proof: As in Lemma 1, the first ISR σ_a of the chain can experience blocking due non-interruptible regions. After the i-th ISR σ_i of the chain completes, either **(i)** all the ISRs of the chain completed or **(ii)** the (i+1)-th ISR σ_{i+1} becomes ready to execute on p_k. In case (i), no additional blocking is suffered by ℓ_y. In case (ii), when σ_i completes, either σ_{i+1} (which runs on p_k, as σ_i) or a higher-priority ISR $\sigma_w \in \mathsf{hep}_k(\sigma_{i+1})$ starts executing on p_k. Hence, no ISR $\sigma_{lp} \in \mathsf{lp}_k(\sigma_{i+1})$ can block σ_{i+1}. The lemma follows by noting that no ISRs in the chain can be blocked except the first one (σ_a), and its blocking is bounded by B_a by Lemma 2. ∎

Building on Lemma 3, Lemma 4 provides a bound that holistically considers the delays incurred by a chain ℓ_y.

Lemma 4: Let $\ell_y = (\sigma_a, \ldots, \sigma_w)$ be a chain of ISRs allocated to core p_k, where the i-th ISR has a priority higher than or equal to the (i+1)-th ISR. Then, the latency incurred by ℓ_y is bounded by the smallest positive solution of: $R(\ell_y) = B_a + \sum_{\sigma_d \in \mathsf{hep}_k(\sigma_w) \cup \sigma_w} \eta_d(R(\ell_y)) C_d$, where B_a is defined as in Eq. (1).

Proof: By Lemma 2, the blocking time incurred by the first ISR σ_a of ℓ_y is B_a. By Lemma 3, B_a also bounds the blocking time of ℓ_y. By definition, $\ell_y = (\sigma_a, \ldots, \sigma_w)$ completes when its last ISR σ_w terminates. The lemma follows by noting that since ISRs in ℓ_y have decreasing priorities, they can be interfered only by ISRs with a priority higher than or equal to the one of σ_w, and all its predecessors already completed before σ_w starts executing as they have a higher or equal priority (i.e., the delay due to predecessors is already included in the bound). ∎

As H-ISRs have higher priorities than VM-ISRs, the assumptions of Lemma 4 hold. Hence, the IDDL is bounded by:

$$L^{IDD}(e^{\mathrm{f}}_{j,y}) = sz(e^{\mathrm{f}}_{j,y}) \cdot \lambda^{\mathrm{IN}}_{\mathrm{DMA}} + R(\ell_x), \tag{3}$$

where $\ell_x = (\sigma_h, \sigma_v)$, with $\sigma_v \in S_k^{\mathrm{VM}}$ and $\sigma_h \in S_k^{\mathrm{H}} \wedge \mathcal{F}(\sigma_v) = \sigma_h$. Using Eq. (3), arrival bursts of interfering ISRs and priority-inversion blocking are accounted only once in term $R(\ell_x)$, instead of accounting them twice, i.e., in terms R_h and R_v of Eq. (2). Also, priority-inversion blocking is considered only once thanks to Lemma 3. As a drawback, Eq. (2) requires σ_h and σ_v to be allocated in the same core, which is anyway a typical and reasonable assumption for most systems.

Output Data Delivery Latency. Similarly to the case of input data, by rule PT3 and Definition 2 the ODDL of an output request $r^f_{j,y}$ can be bounded as:

$$L^{ODD}(r^f_{j,y}) = sz(r^f_{j,y}) \cdot \lambda^{\mathrm{OUT}}_{\mathrm{DMA}} + R_h + R_v, \tag{4}$$

computing individual response-time bounds for the hypervisor-level ISR σ_h and the VM-level ISR σ_v, or

$$L^{ODD}(r^f_{j,y}) = sz(r^f_{j,y}) \cdot \lambda^{\mathrm{OUT}}_{\mathrm{DMA}} + R(\ell_x), \tag{5}$$

bounding $\ell_x = (\sigma_h, \sigma_v)$ by means of the results of Lemma 4.

Input Processing Latency. Differently from the IDDL, the IPL also depends on the response time of the task τ_j consuming the data. We distinguish between two different cases: **(i)** τ_j is triggered by the I/O event (i.e., by the completion of the corresponding VM-level ISR σ_v), and **(ii)** otherwise. In case (i), the arrival curve of τ_j needs to be derived from the one of σ_v as $\eta_j(\delta) = \eta_v(\delta + R_v)$, where R_v is a valid response-time bound for σ_v. Then, the IPL of a device event $e^{\mathrm{f}}_{j,y}$ can be derived by computing the individual response times as

$$L^{IP}_S(e^{\mathrm{f}}_{j,y}) = sz(e^{\mathrm{f}}_{j,y}) \cdot \lambda^{\mathrm{IN}}_{\mathrm{DMA}} + R_h + R_v + R_j, \tag{6}$$

with $\sigma_v \in S_k^{\text{VM}}$ and $\sigma_h \in S_k^{\text{H}} \wedge \mathcal{F}(\sigma_v) = \sigma_h$ or, leveraging Lemma 4 as done for the IDDL,

$$L_S^{IP}(e_{j,y}^{\text{f}}) = sz(e_{j,y}^{\text{f}}) \cdot \lambda_{\text{DMA}}^{\text{IN}} + R(\ell_q), \quad (7)$$

where $\ell_q = (\sigma_h, \sigma_v, \tau_j)$ is a hybrid chain composed of ISRs and tasks, all allocated to p_k. $R(\ell_q)$ is a holistic bound for ℓ_q, which can be derived as: $R(\ell_q) = B_h + \overline{C}_j + \sum_{\sigma_d \in S_k} \eta_d(R(\ell_q))C_d + \sum_{\tau_d \in \mathsf{hep}_k(\tau_j)} \eta_d(R(\ell_q))\overline{C}_d$, where $\overline{C}_j$ is defined as in Lemma 1 and B_h as in Eq. (1). The bound follows similarly to Lemma 4, noting that ℓ_q fulfills the assumptions of Lemma 3 and 4, and that the inter-job precedence constraint of τ_j guarantees that at most one instance of τ_j may contribute to $R(\ell_q)$ when the system is schedulable.

In case (ii), the release of τ_j is asynchronous with respect to the I/O event. When adopting arbitrary arrival curves, this case may lead to an unbounded IPL, as long as a maximum inter-arrival time between consecutive job release is not available. This is due to the fact that an arbitrary job of τ_j may complete an input operation ϵ units of time (with $\epsilon > 0$ arbitrary small) after new data becomes available, thus needing to wait the next job to sample the updated data [77]. When a maximum inter-release time is not available, it is not possible to provide a bound for the worst-case sampling delay.

Hence, we focus on *periodic* tasks to compute the IPL in the asynchronous case. When considering periodic tasks, the worst-case sampling delay is well defined and equal to the period T_j. Hence, the input processing latency is bounded as:

$$L_A^{IP}(e_{j,y}^{\text{f}}) = sz(e_{j,y}^{\text{f}}) \cdot \lambda_{\text{DMA}}^{\text{IN}} + R_h + R_v + T_j + R_j, \quad (8)$$

with $\sigma_v \in S_k^{\text{VM}}$ and $\sigma_h \in S_k^{\text{H}} \wedge \mathcal{F}(\sigma_v) = \sigma_h$, by noting that the data delivery chain ending up with the execution of σ_v may complete just after task τ_j samples the data, which in any case will re-occur after T_j time units because of the periodicity of τ_j. A refined IPL bound can be obtained by bounding the response time of chain $\ell_x = (\sigma_h, \sigma_v)$ with Lemma 4:

$$L_A^{IP}(e_{j,y}^{\text{f}}) = sz(e_{j,y}^{\text{f}}) \cdot \lambda_{\text{DMA}}^{\text{IN}} + R(\ell_x) + T_j + R_j. \quad (9)$$

VI. Analyzing I/O Para-Virtualization

The I/O Para-Virtualization scheme described in Section IV-B has its main focus on the concept of I/O VM.

As shown in Figure 3, there are two queues for each pair of (v_i, d_f): a queue $Q_{i,f}^{\text{OUT}}$ containing output operations $r_{j,y}^f$ (with $\tau_j \in \Gamma_i^{\text{VM}}, \Delta_{j,y}^f = O$), where the data flows from a VM v_i to a device d_f, and a queue $Q_{i,f}^{\text{IN}}$ storing input operations $r_{j,y}^f$ (with $\Delta_{j,y}^f = I$), where the data flows from a device to a VM. Each type of operation incurs in a different contribution to the total delay. Note that the copies specified by rules V1 and V2 have an overall cost of $\lambda^{\text{CP}} \cdot sz(r_{j,y}^f)$. By rule V3, the I/O VM instructs the DMA to copy the data from the I/O buffers to the device. In principle, this is a non-blocking operation and the I/O manager can proceed processing other requests while the DMA performs the copy. However, it is possible that the next operation to be served still targets the same device, causing a contention for the DMA. A safe solution for accounting such a contention cost in the bounds presented next requires including the time required by the DMA (i.e., $\lambda_{\text{DMA}}^{\text{OUT}} \cdot sz(r_{j,y}^f)$) in the delay contribution of output operations. In summary, the costs for performing input and output operations is bounded by $\lambda_{\text{REQ}}^{\text{IN}}(r_{j,y}^f) = \lambda^{\text{CP}} \cdot sz(r_{j,y}^f)$, and, $\lambda_{\text{REQ}}^{\text{OUT}}(r_{j,y}^f) = \lambda^{\text{CP}} \cdot sz(r_{j,y}^f) + \lambda_{\text{DMA}}^{\text{OUT}} \cdot sz(r_{j,y}^f)$, respectively.

A. Analyzing of the I/O Manager

To bound the delay incurred by a request $r_{j,y}^f$ under analysis, we first need to bound the number of requests enqueued in each queue in a time interval. We start focusing on queues $Q_{i,f}^{\text{OUT}}$ of output operations. The number of requests enqueued in $Q_{i,f}^{\text{OUT}}$ in an interval of length δ is bounded by $N_{i,f}^{\text{OUT}}(\delta) = \sum_{\tau_j \in \Gamma_i^{\text{VM}}} \eta_j(\delta + R_j)|\mathcal{R}_j^{\text{OUT}}(d_f)|$, where $\mathcal{R}_j^{\text{OUT}}(d_f) = \{r_{j,y}^f \in \mathcal{R}_j : \Delta_{j,y}^f = O, \tau_j \in \Gamma_i^{\text{VM}}\}$ is the set of output requests issued by τ_j to d_f, and R_j is a WCRT bound for τ_j.

Input operations are stimulated by the external environment and their occurrence depend on the activation pattern of data event arrivals from a device d_f, which are notified to the system with a VM-level ISR. Therefore, the number of requests enqueued in $Q_{i,f}^{\text{IN}}$ in an interval of time of length δ is bounded by $N_{i,f}^{\text{IN}}(\delta) = \sum_{e_{j,y}^{\text{f}} \in \mathcal{E}_f(v_i)} \eta_v(\delta + R_v)$, with $\mathcal{E}_f(v_i) = \{e_{j,y}^{\text{f}} \in \mathcal{E}_f : \tau_j \in \Gamma_i^{\text{VM}}\}$, where $\eta_v(\delta)$ is the arrival curve of the VM-ISR $\mathcal{I}(e_{j,y}^{\text{f}}) = \sigma_v$ that inserts in $Q_{i,f}^{\text{IN}}$ the data of $e_{j,y}^{\text{f}}$ and R_v is a response-time bound for σ_v.

By rules C2 and V1, the I/O manager is notified of new requests by the execution of task τ_j and the VM-level ISR σ_v: this is considered in $N_{i,f}^{\text{IN}}(\delta)$ and $N_{i,f}^{\text{OUT}}(\delta)$ by adding a jitter R_j and R_v, respectively, considering a larger time window.

Lemma 5 defines the multisets containing the delays due to I/O requests issued in an interval of time with length δ.

Lemma 5: The service delay of I/O operations initiated in a time window of length δ, and involving a VM $v_i \in \mathcal{V}$ and a device $d_f \in \mathcal{H}(v_i)$ are contained into the multisets[2]

$$\mathcal{SD}_{i,f}^{\text{OUT}}(\delta) = \biguplus_{r_{j,y}^f \in \mathcal{R}_j^{OUT}(d_f)} \{\lambda_{\text{REQ}}^{\text{OUT}}(r_{j,y}^f)\} \otimes \eta_j(\delta + R_j), \quad (10)$$

and

$$\mathcal{SD}_{i,f}^{\text{IN}}(\delta) = \biguplus_{e_{j,y}^{\text{f}} \in \mathcal{E}_f(v_i)} \{\lambda_{\text{REQ}}^{\text{IN}}(e_{j,y}^{\text{f}})\} \otimes \eta_v(\delta + R_v), \quad (11)$$

respectively, where $\tau_j = \mathcal{T}(v_i, d_f)$, and $\eta_v(\delta)$ is the arrival curve of $\mathcal{I}(e_{j,y}^{\text{f}}) = \sigma_v$.

Proof: For each VM $v_i \in \mathcal{V}$, at most one task $\tau_j = \mathcal{T}(v_i, d_f)$ can access $d_f \in \mathcal{D}$. Consequently, $\mathcal{R}_j^{\text{OUT}}(d_f)$ is the set of output requests issued from v_i to d_f. Each output operation $r_{j,y}^f$ may delay other operations up to $\lambda_{\text{REQ}}^{\text{OUT}}(r_{j,y}^f)$ units of time. Since they are issued by tasks, and each task can issue requests at any time during execution, each of them contributes up to $\eta_j(\delta + R_j)$ requests. Input operations are

[2]The operator $\uplus$ denotes the union of multisets, e.g., $\{1,1\} \uplus \{1,4\} = \{1,1,1,4\}$, and the product operator $\otimes$ multiplies the number of instances of each element in the multiset, e.g., $\{6,9\} \otimes 3 = \{6,6,6,9,9,9\}$.

triggered by devices by means of device events. The set $\mathcal{E}_f(v_i)$ includes all the events issued from a device $d_f \in \mathcal{D}$ to a task $\tau_j \in \Gamma_i^{\text{VM}}$. Each event $e_{j,y}^{\text{f}}$ delays other operations up to $\lambda_{\text{REQ}}^{\text{IN}}(e_{j,y}^{\text{f}})$ time units for each of the $\eta_v(\delta + R_v)$ event instances released in a time window of length δ. The lemma follows. ∎

We define the notation $\Sigma(x, M)$ to refer to the sum of the x largest elements in a multiset M. If x is greater than the number of elements in M, all elements are summed.

Theorem 1: The maximum delay incurred by a request r enqueued in a queue $Q_{i,f}^T$, where $T \in \{\text{IN}, \text{OUT}\}, v_i \in \mathcal{V}, d_f \in \mathcal{D}$, in a window of length δ is bounded by the smallest positive solution $D_{i,f}^{T*}$ of the following recursive equation:

$$D_{i,f}^T(\delta) = \sum_{v_x \in \mathcal{V}} \sum_{d_y \in \mathcal{D}} \big(\Sigma(N_{i,f}^T(\delta), \mathcal{SD}_{x,y}^{\text{IN}}(\delta)) + \Sigma(N_{i,f}^T(\delta), \mathcal{SD}_{x,y}^{\text{OUT}}(\delta))\big) + \sum_{\sigma_d \in S_{io}} \eta_d(\delta) C_d, \tag{12}$$

where $N_{i,f}^T(\delta)$ is a bound on the maximum number of requests stored in $Q_{i,f}^T$ in a window of length δ, and S_{io} is the set of ISRs allocated on p_{io}.

Proof: Due to FIFO intra-queue policy, the request r under analysis is processed after at most $N_{i,f}^T(\delta) - 1$ requests stored in its own queue $Q_{i,f}^T$. Due to inter-queue RR scheduling, each of the $N_{i,f}^T(\delta)$ requests (including r) can suffer interference from *at most one* request per other queue $Q_{x,y}^T \neq Q_{i,f}^T$. Hence, for each request in $Q_{i,f}^T$, at most $N_{i,f}^T(\delta)$ requests from each queue $Q_{x,y}^T \neq Q_{i,f}^T$ can interfere with r.

By Lemma 5, in any time window of length δ, the requests that can be enqueued in any queue $Q_{x,y}^T$ are included in multiset $\mathcal{SD}_{x,y}^T(\delta)$. The maximum inter-queue interference generated by the requests in each queue $Q_{x,y}^T$ is bounded by the duration of the longest $N_{i,f}^T(\delta)$ ones, that is, $\Sigma(N_{i,f}^T(\delta), \mathcal{SD}_{x,y}^T(\delta))$. Similarly, also the intra-queue interference suffered by r (i.e., in $Q_{i,f}^T$) is bounded by $\Sigma(N_{i,f}^T(\delta), \mathcal{SD}_{i,f}^T(\delta))$. Therefore, the total interference suffered by r due to other requests is given by the sum of the first two terms in the equation. Finally, the I/O manager is implemented by a task $\tau_{io} \in \Gamma_k$ and hence is subject to interference due to ISRs. This proves the theorem. ∎

B. Analyzing I/O Para-Virtualization

When using I/O Para-Virtualization, I/O tasks can execute two hypercalls to copy the data from the hypervisor memory to the VM memory and viceversa (rules C1 and C2, respectively). Both hypercalls are modeled with a WCET of $\lambda^{\text{CP}} \cdot sz(x)$, where x is the request served by the hypercall, and they are assigned to a priority π_{hc}. As discussed in Section III, hypercalls are assigned to higher priorities than tasks (they are software ISRs). Hence, when a task τ_j is synchronously executing a hypercall, it is subject to a priority elevation that may cause priority-inversion blocking to tasks and ISRs with a priority π_x such that $\pi_j < \pi_x < \pi_{hc}$.

Therefore, the WCRT bounds derived in Lemmas 1 and 2 (for tasks and ISRs, respectively) need to be updated to consider additional blocking due to priority inversion. To include the blocking due to hypercalls in Lemma 1, a new bound B_j^* needs to be defined as $B_j^* = \max\{B_j^{\text{HC}}, B_j\}$, where

$$B_j^{\text{HC}} = \max_{\tau_l \in \Gamma_k} \{sz(r_{l,y}^f) \cdot \lambda^{\text{CP}} \mid r_{l,y}^f \in \mathcal{R}_l \ \wedge \ \pi_l < \pi_j < \pi_{hc}\}, \tag{13}$$

and B_j is defined as in Eq. (1). As in the proof of Lemma 1, Eq. (13) holds since blocking can occur at most once for each job of τ_j. In the remainder of the proof of Lemma 1, the part leveraging rules PT2 and PT3 to bound the cost incurred for input and output operations still holds when using rules C1 and C2 in place of PT2 and PT3. No other changes to Lemma 1 are required. Similarly, Lemma 2 needs to consider additional priority-inversion blocking due to hypercalls. A new blocking bound is defined as $B_a^* = \max\{B_a^{\text{HC}}, B_a\}$, where B_a is defined in Lemma 2 and B_a^{HC} in Eq. (13).

I/O Para-Virtualization causes a different pipeline of events with respect to the case of pass-through I/O, which is discussed in the following.

Input Data Delivery Latency. Bounding the IDDL for the case of I/O Para-Virtualization requires considering additional delays introduced by the I/O VM. By rule V1, when new data is received by d_f, it instructs the DMA to copy the data into the I/O buffers. Upon completion, a H-ISR is triggered, which in turn triggers a VM-ISR in the I/O VM. The latter inserts an input request in the queue $Q_{i,f}^{\text{IN}}$ of the I/O VM. The I/O VM is then responsible for performing the request. After the operation is dispatched by the I/O scheduler, the corresponding data become available for other VMs in the buffers in the hypervisor memory. In summary, the IDDL of a request $r_{j,y}^f \in Q_{i,f}^T$, with $\mathcal{M}(\tau_j) = v_i$, is bounded similarly as Eq. (3) by summing the delays in the stages of the pipeline: $L^{IDD}(e_{j,y}^{\text{f}}) = sz(r_{j,y}^f) \cdot \lambda_{\text{DMA}}^{\text{IN}} + R(\ell_x) + D_{i,f}^{T*}$, where $D_{i,f}^{T*}$ is a bound on the delay caused by the I/O VM (Theorem 1), $\ell_x = (\sigma_h, \sigma_v)$, with $\sigma_v \in S_k^{\text{VM}}$ and $\sigma_h \in S_k^{\text{H}} \wedge \mathcal{F}(\sigma_v) = \sigma_h$. As in Section V, $R(\ell_x)$ is bounded by Lemma 4 (after updating the blocking bound with B_a^*) , since Lemmas 3 and 4 do not depend on the I/O mechanism.

Output Data Delivery Latency. By rule C2, when a task $\tau_j \in \Gamma_i^{\text{VM}}$ performs an output operation $r_{j,y}^f$, it issues a hypercall to copy data from the memory space of the task to the hypervisor, with a cost $\lambda^{\text{CP}} \cdot sz(r_{j,y}^f)$. Still due to rule C2, an output request $r_{j,y}^f$ is inserted in a queue $Q_{i,f}^{\text{OUT}}$. Consequently, $r_{j,y}^f$ may be subject to a delay due to the I/O manager, which is bounded by $D_{i,f}^{T*}$ (Theorem 1). The ODDL pipeline terminates with the execution of the hypervisor-level ISR that is triggered by rule V3, and the corresponding VM-level ISR σ_v. Overall, the ODDL is bounded by the following composition of delays:

$L^{ODD}(r_{j,y}^f) = sz(r_{j,y}^f) \cdot \lambda^{\text{CP}} + D_{i,f}^{T*} + R(\ell_x)$, which extends Eq. (5) to account for I/O Para-Virtualization. The response time $R(\ell_x)$ for the ISR chain $\ell_x = (\sigma_h, \sigma_v)$ is bounded by Lemma 4 (again, after updating the blocking bound as discussed above). The H-ISR σ_h that notifies the completion of the I/O operation is triggered as a consequence of both the execution of τ_j, the time spent in the I/O manager, and the completion of the DMA transfer. Consequently, the

arrival curve of σ_h can be obtained from the one of τ_j as $\eta_h(\delta) = \eta_j(\delta + R_j + D^{T*}_{i,f})$.

Input Processing Latency. As for pass-through I/O, we distinguish two cases for a task $\tau_j \in \Gamma^{\text{VM}}_i$ performing I/O operations: **(i)** τ_j is triggered by an I/O event or, **(ii)** otherwise. In both cases, the IPL shares the same pipeline of events of the IDDL but with one additional stage, where the data is consumed by τ_j. In case (i), τ_j is triggered just after the I/O scheduler completes processing a specific input request. Its arrival curve $\eta_j(\delta)$ requires then to account for a release jitter due to the pipeline of events composed of: **(a)** the notification of $e^{\text{f}}_{j,y}$ by the VM-level ISR $\mathcal{I}(e^{\text{f}}_{j,y}) = \sigma_v$, and **(b)** the processing delay $D^{T*}_{i,f}$ due to the I/O manager. Hence, the arrival curve of task τ_j can be derived from the one of the ISR $\mathcal{I}(e^{\text{f}}_{j,y}) = \sigma_v$ associated with $e^{\text{f}}_{j,y}$ as $\eta_j(\delta) = \eta_v(\delta + R_v + D^{T*}_{i,f})$. The same pipeline of events is considered to bound the IPL, by composing the delays as $L^{IP}_S(e^{\text{f}}_{j,y}) = sz(e^{\text{f}}_{j,y}) \cdot \lambda^{\text{IN}}_{\text{DMA}} + R(\ell_x) + D^{T*}_{i,f} + R_j$, where R_j bounds the response time of τ_j, and $\ell_x = (\sigma_h, \sigma_v)$. When the release of τ_j is asynchronous with respect to the I/O event, as in Section V, we consider periodic tasks. Similarly to the case of pass-through I/O (Eq. (9)), the IPL can be bounded as $L^{IP}_A(e^{\text{f}}_{j,y}) = sz(e^{\text{f}}_{j,y}) \cdot \lambda^{\text{IN}}_{\text{DMA}} + R(\ell_x) + D^{T*}_{i,f} + T_j + R_j$, by observing that a task activated asynchronously is subject to the same pipeline of events of case (i) but it can also experience a worst-case sampling delay of up to T_j time units.

C. I/O Para-Virtualization with Shared Buffers

The timing parameters under I/O Para-Virtualization with shared buffers can be bounded in similar way as above. The key difference with the previous case is that, due to rule CS1 and CS2, VMs communicate with the I/O VM by means of shared buffers and not by means of the hypervisor memory. This choice provides two advantages. The first one is a reduced priority-inversion blocking with respect to the case without shared buffers, since there is no need for VMs $v_i \in \mathcal{V} \setminus \{v_{io}\}$ to write and read the data buffers using hypercalls. The second advantage is a reduced number of data copies. Indeed, since data is stored in shared buffers, it can be directly produced/consumed in/from such buffers.

Consequently, schedulability can be checked by computing the response times as discussed in Section V for pass-through I/O, with the advantage of using C_j in place of $\overline{C}_j$. IDDL, ODDL, and IPL can be bounded as discussed in Section VI-B for the scenario without shared buffers, as the pipeline of events that occur in the two cases remains the same, but saving $sz(r^f_{j,y}) \cdot \lambda^{\text{CP}}$ units of time in the ODDL, as data is produced directly in the shared memory buffer (i.e., $L^{ODD}(r^f_{j,y}) = D^{T*}_{i,f} + R(\ell_x)$).

VII. Experimental Results

This section presents the results of two experimental studies carried out to evaluate the performance of the three I/O virtualization schemes discussed in the paper. The analysis has been implemented in the *pyCPA* tool [78]. Before starting, we introduce a baseline setup common to both the studies.

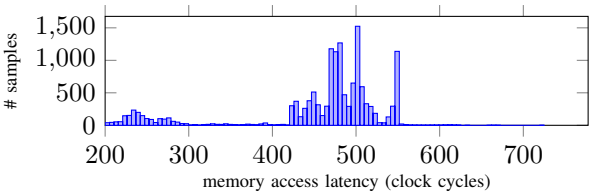

Fig. 5: Read delays ($\geq$ 200) on a Cortex A-53 of the Zynq Ultrascale+.

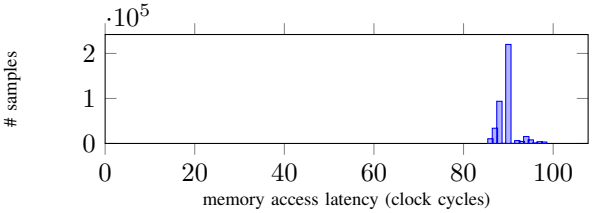

Fig. 6: Write delays on a Cortex A-53 of the Zynq Ultrascale+.

Baseline setup. We consider the metrics presented in Sec. IV-D for two tasks in a case-study, an autonomous driving application taken from the 2019 WATERS Industrial Challenge [10] by Bosch: a *Lidar task* performing an input request to a lidar (via Ethernet), and a *CAN task* writing data to a CAN device. We assume the lidar task to run in a feature-rich VM (e.g., running Linux) named F-VM and the CAN task to run in a safety-critical VM (running a real-time OS) named S-VM. For each task, we monitored the performance metrics of two I/O transactions with a size equal to the maximum size allowed by the Ethernet and CAN protocols, respectively.

For each core p_k, we considered two ISRs for the timer, one for the hypervisor and one for the VM, periodically released every 1 ms and with a WCET randomly generated with uniform distribution in the interval $[1, 10]$ μs for the hypervisor ISRs, $[1, 20]$ μs for the timer ISR of S-VM, and $[10, 30]$ μs for the timer ISR of F-VM. For each core, n interfering tasks have been synthetically generated. Tasks have been assigned to a periodic arrival curve [71], where the period T_j has been randomly generated in the interval $[1, 100]$ ms. The *randfixedsum* [79] algorithm has been used to derive the individual utilizations $U_j = C_j/T_j$ given a total utilization U^I reserved for interfering tasks on each core. WCETs have been derived from the utilizations and the periods, i.e., as $C_j = U_j \cdot T_j$. Priorities have been assigned according to deadline monotonic. The length of the longest NIR of each task and ISR has been generated in two intervals: `low`, i.e., $\omega_j \in [0, 0.1 \cdot C_j]$ and `medium`, i.e., $\omega_j \in [0, 0.2 \cdot C_j]$. The amount of data $sz(r^f_{j,y})$ transfered by each I/O request $r^f_{j,y}$ is generated in $[1, sz_{\text{MAX}}(r^f_{j,y})] = [1, 1500]$ bytes. Each interfering task τ_j performs a request to an exclusively assigned device. For each request, a pair of hypervisor-level and VM-level ISRs are generated. WCETs are randomly chosen in the intervals: $[1, 10]$ μs for hypervisor-level ISRs, $[10, 40]$ μs for VM-level ISRs of F-VM, and $[1, 30]$ μs for VM-level ISRs of S-VM. For each graph, the results are averaged over 100 iterations.

Measurements from a real platform. The parameter λ^{CP} has

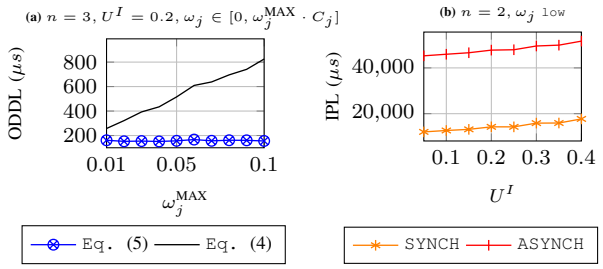

Fig. 7: Analytical latencies for pass-through I/O.

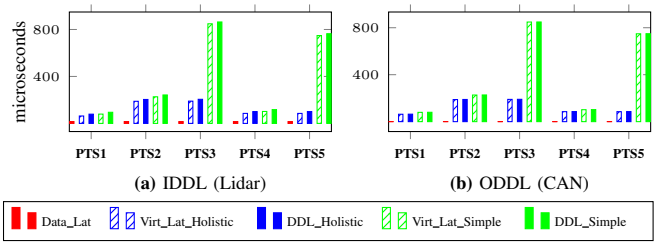

Fig. 8: Latencies in different scenarios for pass-through I/O.

been experimentally measured on a real platform, the Zynq UltraScale+ MPSoC by Xilinx (ZCU102 board) equipped with a MTA8ATF512 16-bank DDR memory, on a Cortex A-53 core running at 1.2 GHz. To this end, we measured the time required to read and write a 64-bit word from/to a DRAM over 400k iterations. The measurements have been performed by placing memory barriers before and after a read (or write) instruction, and measuring the elapsed cycles with the Performance Monitor Unit of the core. During the experiment, caches have been disabled, and the other cores generated intensive memory traffic to stimulate the worst-case scenarios that maximize the delay at the memory controller (according to our best knowledge and reverse engineering, given that its internal details are not available to us), also considering domino effects caused by transactions re-ordering (e.g., those directed to "open-rows" are privileged) and write batching. Figures 5 and 6 show the distribution of memory access delays for reads and writes (in clock cycles), respectively, measured in the experiments. The maximum measured delays are $\lambda_{64}^{\text{RD}} = 604.16\ ns$ and $\lambda_{64}^{\text{WR}} = 81.67\ ns$, for read and writes, respectively. Given the two individual delays λ_{64}^{RD} and λ_{64}^{WR}, we set time λ^{CP} required to copy a byte to $\lambda^{\text{CP}} = (\lambda_{64}^{\text{RD}} + \lambda_{64}^{\text{WR}})/8 = 85.74\ ns$ for the next experiments. For the sake of simplicity, we modeled the delay of memory transactions performed by DMA in the same way. Hence, we set $\lambda_{\text{DMA}}^{\text{OUT}} = \lambda_{64}^{\text{RD}}/8 = 75.52\ ns$ and $\lambda_{\text{DMA}}^{\text{IN}} = \lambda_{64}^{\text{WR}}/8 = 10.21\ ns$, approximating the time required by the DMA to copy from the device memory to the I/O buffers with the time required to perform a write in the DDR, and the time for copying from the I/O buffers to the device as a read.

A. Pass-through I/O

Comparing the bounds. The first experimental study targets pass-through I/O and considers the baseline system with two VMs. Figure 7 addresses two representative configurations.

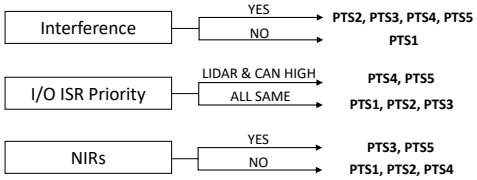

Fig. 9: Scenarios used in the experiment of Figure 8.

In Figure 7(a), for each task and ISR, the parameter ω_j has been generated in the interval $[0, \omega_j^{\text{MAX}} \cdot C_j]$, and ω_j^{MAX} has been varied in $[0, 0.1]$ with steps of 0.01, comparing the ODDL bounds of Eqs. (4) and (5) for the I/O operation of the CAN task. As ω_j^{MAX} increases, the length of non-interruptible regions increases and hence priority-inversion blocking. The simple CPA-based bound of Eq. (4) suffers from a higher blocking because it accounts for the individual blocking of each ISR in the path. Since VM-level ISRs can also be blocked by tasks, which have normally a larger WCETs and ω_j, the latency bound is consistently higher than Eq. (5). Figure 7(b) compares the IPL for the Lidar task, in both cases in which it is triggered synchronously (Eq. (7)) or asynchronously (Eq. (9)). As expected, both latencies increase as U^I increase, and the asynchronous case provides a consistently higher IPL.

Exploring the sources of the delay. Figure 8 reports on a breakdown of the IDDL (inset (a)) and ODDL (inset (b)) in two different source of delays: data latency (e.g., $sz(r_{j,y}^f) \cdot \lambda^{\text{CP}}$), and virtualization latency (i.e., due to the response time of the ISRs). Also the sum of the two is reported, denoted as DDL (either IDDL or ODDL). Worst-case latencies obtained by summing the response times of individual ISRs are denoted as "simple" (e.g., Eq. (4)), while those obtained with a holistic analysis of the ISR chain is denoted as "holistic" (e.g., Eq. (5)). Five scenarios have been tested (PTS1,..., PTS5): they are summarized in Figure 9.

In PTS1, there is no interfering task on each VM. Clearly, it determines the lowest bound, and the virtualization latency is limited to 60 μs. In the other cases PTS2, ..., PTS5, for each core, an interfering load due to $n = 4$ tasks with $U^I = 0.2$ is considered. In PTS2 and PTS3, all the ISRs handling I/O are assigned to the same priority (one priority for VM and hypervisor level ISRs, respectively), whereas in PTS4 and PTS5 the ISRs handling Lidar and CAN data are assigned to a higher priority. PTS2 and PTS4 consider interfering tasks not causing NIRs, whereas PTS3 and PTS5 consider this case with $\omega_j \in [0, 0.1 \cdot C_j]$. Figure 8 highlights that higher priorities for Lidar and CAN ISRs lead to lower DDLs, which derive from a smaller virtualization latency. Moreover, comparing the cases with and without NIRs (PTS3 and PTS5 vs. PTS2 and PTS4) shows that the "simple" bounds are very sensitive to NIRs, producing very high DDL. This case study highlights the potential of the proposed analysis: the possibility of decomposing the delays in different components, shining a light on the sources of a high latency. This may also drive the application designer in performing a design-space exploration of the parameters (e.g., ISRs priorities) to guarantee given

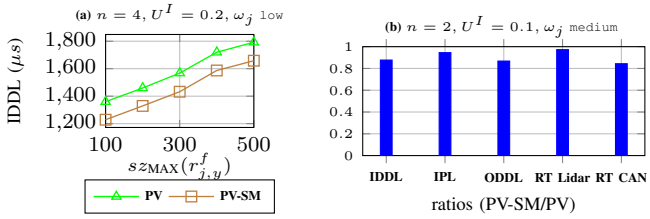

Fig. 10: Analytical latencies for I/O Para-Virtualization with and without shared buffers (PV and PV-SM, respectively).

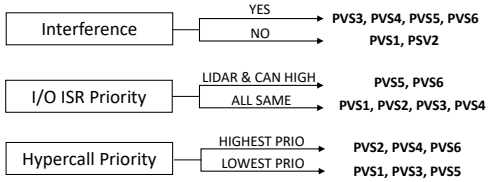

Fig. 11: Scenarios used in the experiment of Figures 12 and 13.

worst-case latency bounds.

B. I/O Para-Virtualization

Comparing the bounds. In this second study, we enriched the baseline setup to include an additional VM, called *R-VM*, which acts as a replica for *F-VM*, thus needing to share the same I/O devices. Figure 10(a) shows the IDDL of the Lidar task when $sz_{\text{MAX}}(r^f_{j,y})$ is varied, using I/O Para-Virtualization with and without buffers in shared memory, denoted as PV-SM and PV, respectively. In both cases, with a higher $sz_{\text{MAX}}(r^f_{j,y})$, the IDDL increases, as the I/O manager incurs in higher delays to serve interfering requests. Furthermore, PV-SM incurs in lower delays, as it is affected by a lower blocking and number of copies in memory. The same trend emerges from Figure 10(b), which shows all the metrics presented in Section IV-D for a representative configuration.

Exploring the sources of delay. Figure 12 and 13 considers a breakdown of the DDL into data latency, virtualization latency (due to the ISRs), and latency due to the I/O manager, for I/O ParaVirtualization with and without shared buffers. Six cases have been considered (summarized in Figure 11): no interference (PVS1, PVS2) vs. interference (PVS3, ..., PVS6), with $n = 4$ tasks with $U^I = 0.2$ for each core (p_{io} excluded), hypercalls at lowest H-ISR priority (PVS1, PVS3, PVS5) vs. hypercalls at highest hypervisor-ISR priority (PVS2, PVS4, PVS6), all the ISRs due to I/O at the same priority (PVS1, ..., PVS4) vs. ISRs due to Lidar and CAN I/O at a higher priority (PVS5, PVS6). Figure 12 and 13 shows that the I/O manager delay plays a key role in determining the worst-case latency, especially in the scenarios with interference (PVS3, ..., PVS6). Furthermore, Figure 12 shows that, also in this case, increasing the priority of I/O ISRs of Lidar and CAN reduces the virtualization latency. Figure 13 shows a reduction of the virtualization latency as an effect of the hypercall priority. This effect is not shown in Figure 12 because, as discussed in Section VI-C, the I/O para-virtualization with shared buffers does not require hypercalls to perform copies, providing an advantage with respect to the case in which memory buffers are allocated in the hypervisor memory.

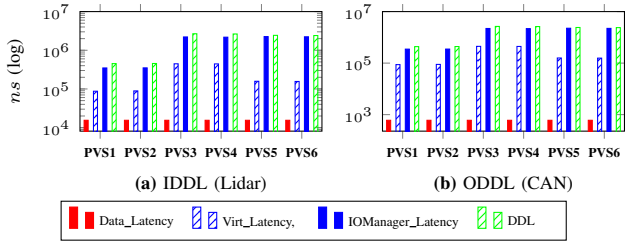

Fig. 12: Delays in different scenarios for I/O Para-Virtualization (with shared buffers). Log-scale on the y axis.

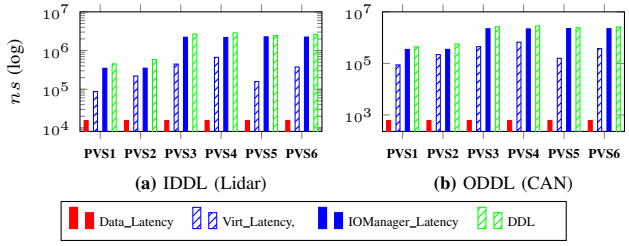

Fig. 13: Delays in different scenarios for I/O Para-Virtualization (no shared buffers). Log-scale on the y-axis.

VIII. Limitations, Extensions, and Conclusions

This paper provided a general and flexible model, which allowed to study the timing effects of different I/O virtualization techniques and compare their worst-case performance. Most importantly, this work enables a fine-grained understanding of the sources of worst-case latency, allowing a design-space exploration of the system parameters, and guiding the application designer in configuring a vast amount of parameters (e.g., ISRs and tasks priorities) to meet timing constraints.

Future work aims at extending this paper to overcome some of its limitations and assumptions. For instance, interesting research can be carried out to consider other I/O scheduling policies, e.g., fixed-priority, which may be suitable to prioritize the traffic of highly-critical and latency-sensitive VMs. Different I/O management schemes may be also considered, e.g., the case of a master VM handling I/O traffic while also serving other loads. Techniques to bound the buffer sizes are also worth to investigate. Overhead-aware analyses [80] may be integrated with this work in the future. For the sake of conciseness in the notation, we assumed only one task for each VM to access each I/O device. This assumption can already be relaxed in our analysis framework by providing one buffer for each pair task/device (instead of VM/device), at the expense of additional space in main memory. More complex techniques can be used to improve the precision in analyzing the I/O manager by exploiting the parallelism between CPU and DMA copies, e.g., similarly to [81, 82]. Finally, future research should also focus on implementations of the considered techniques and hardware-assisted virtualization.

ACKNOWLEDGMENT

This work has been partially supported by Huawei and the Italian Ministry of University and Research (MIUR), under the SPHERE project funded within the PRIN-2017 framework (grant no. 93008800505).

REFERENCES

[1] G. Heiser, "Virtualizing embedded systems - why bother?" in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 901–905.

[2] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the Key," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, vol. 76, 2017, pp. 4:1–4:22.

[3] G. Cicero, A. Biondi, G. Buttazzo, and A. Patel, "Reconciling security with virtualization: A dual-hypervisor design for arm trustzone," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, 2018, pp. 1628–1633.

[4] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, 2018, pp. 1651–1657.

[5] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 1–14.

[6] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 155–166.

[7] 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 *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[8] H. Pérez, J. J. Gutiérrez, S. Peiro, and A. Crespo, "Distributed architecture for developing mixed-criticality systems in multi-core platforms," *Journal of Systems and Software*, vol. 123, pp. 145 – 159, 2017.

[9] A. Biondi, F. Nesti, G. Cicero, D. Casini, and G. Buttazzo, "A safe, secure, and predictable software architecture for deep learning in safety-critical systems," *IEEE Embedded Systems Letters*, 2019.

[10] A. Hamann, D. Dasari, F. Wurst, I. Sañudo, N. Capodieci, P. Burgio, and M. Bertogna, "WATERS Industrial Challenge 2019," https://www.ecrts.org/forum.

[11] M. Danish, Y. Li, and R. West, "Virtual-cpu scheduling in the quest operating system," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 169–179.

[12] E. Missimer, K. Missimer, and R. West, "Mixed-criticality scheduling with i/o," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016.

[13] Y. Li, R. West, Z. Cheng, and E. Missimer, "Predictable communication and migration in the quest-v separation kernel," in *2014 IEEE Real-Time Systems Symposium*, 2014, pp. 272–283.

[14] 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, Jun. 2016.

[15] A. Golchin, S. Sinha, and R. West, "Boomerang: Real-time i/o meets legacy systems," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2020.

[16] A. Masrur, S. Drossler, T. Pfeuffer, and S. Chakraborty, "Vm-based real-time services for automotive control applications," in *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2010.

[17] Z. Jiang and N. Audsley, "VCDC: The Virtualized Complicated Device Controller," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[18] 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 *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 326–338.

[19] D. Münch, O. Isfort, K. Mueller, M. Paulitsch, and A. Herkersdorf, "Hardware-based I/O virtualization for mixed criticality real-time systems using PCIe SR-IOV," in *2013 IEEE 16th International Conference on Computational Science and Engineering*, Dec 2013.

[20] D. Münch, M. Paulitsch, and A. Herkersdorf, "IOMPU: Spatial separation for hardware-based I/O virtualization for mixed-criticality embedded real-time systems using non-transparent bridges," in *2015 IEEE 17th International Conference on High Performance Computing and Communications*, Aug 2015.

[21] R. Pellizzoni and M. Caccamo, "Toward the predictable integration of real-time cots based systems," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007, pp. 73–82.

[22] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha, "Coscheduling of cpu and i/o transactions in cots-based embedded systems," in *2008 Real-Time Systems Symposium*, 2008, pp. 221–231.

[23] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on wcet analysis of real-time embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 400–415, 2010.

[24] S. Bak, E. Betti, R. Pellizzoni, M. Caccamo, and L. Sha, "Real-time control of i/o cots peripherals for embedded systems," in *2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 193–203.

[25] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha, "Real-time i/o management system with cots peripherals," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 45–58, 2013.

[26] 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," ser. RTNS '18, 2018.

[27] S. Abedi, N. Gandhi, H. M. Demoulin, Y. Li, Y. Wu, and L. T. X. Phan, "Rtnf: Predictable latency for network function virtualization," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[28] I. Sañudo, R. Cavicchioli, N. Capodieci, P. Valente, and M. Bertogna, "A survey on shared disk i/o management in virtualized environments under real time constraints," *SIGBED Rev.*, vol. 15, no. 1, p. 57–63, 2018.

[29] L. Abdallah, M. Jan, J. Ermont, and C. Fraboul, "Reducing the contention experienced by real-time core-to-i/o flows over a tilera-like network on chip," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[30] K. Singer, K. Agrawal, and I.-T. Lee, *Scheduling I/O Latency-Hiding Futures in Task-Parallel Platforms*, 01 2020, pp. 147–161.

[31] G. Ara, L. Abeni, T. Cucinotta, and C. Vitucci, "On the use of kernel bypass mechanisms for high-performance inter-container communications," in *High Performance Computing*, M. Weiland, G. Juckeland, S. Alam, and H. Jagode, Eds., 2019.

[32] G. Ara, T. Cucinotta, L. Abeni, and C. Vitucci, "Comparative evaluation of kernel bypass mechanisms for high-performance inter-container communications," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, 2020.

[33] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed i/o." *Intel technology journal*, vol. 10, no. 3, 2006.

[34] "PCI-SIG. SR-IOV website. http://pcisig.com/."

[35] Z. Jiang, "Real-time i/o system for many-core embedded systems," Ph.D. dissertation, University of York, August 2018. [Online]. Available: http://etheses.whiterose.ac.uk/22608/

[36] Z. Jiang and N. C. Audsley, "GPIOCP: timing-accurate general purpose I/O controller for many-core real-time systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 806–811.

[37] 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, 2019.

[38] Z. Jiang, N. C. Audsley, and P. Dong, "Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018.

[39] D. Münch, M. Paulitsch, O. Hanka, and A. Herkersdorf, "MPIOV: Scaling hardware-based I/O virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (multi-core) multi-processor systems," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015.

[40] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[41] A. Golchin, Z. Cheng, and R. West, "Tuned pipes: End-to-end throughput and delay guarantees for usb devices," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 196–207.

[42] "Project ACRN, 2019, https://projectacrn.org/."

[43] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The xtratum approach," in *2010 European Dependable Computing Conference*, April 2010.

[44] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no vm exits!(almost)," *OSPERT 2017, the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017.

[45] M. Beckert, M. Neukirchner, R. Ernst, and S. M. Petters, "Sufficient temporal independence and improved interrupt latencies in a real-time hypervisor," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[46] R. Pellizzoni, M.-Y. Nam, R. M. Bradford, and L. Sha, "A network calculus based analysis for the pci bus," Tech. rep., University of Illinois at Urbana-Champaign, http://ece . . . , Tech. Rep., 2008.

[47] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric os for multi-core embedded systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

[48] R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems," *Real-Time Systems*, 2019.

[49] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011.

[50] G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Embedded Real Time Software and Systems ERTSS'14*.

[51] C. Maia, G. Nelissen, L. M. Nogueira, L. M. Pinho, and D. G. Pérez, "Schedulability analysis for global fixed-priority scheduling of the 3-phase task model," in *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2017.

[52] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Memory feasibility analysis of parallel tasks running on scratchpad-based architectures," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018.

[53] "ARM Generic Interrupt Controller, Architecture version 2.0. https://developer.arm.com/docs/ihi0048/b/arm-generic-interrupt-controller-architecture-version-20-architecture-specification."

[54] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, Nov 2018.

[55] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*, 2020.

[56] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019.

[57] S. A. Rashid, G. Nelissen, S. Altmeyer, R. I. Davis, and E. Tovar, "Integrated analysis of cache related preemption delays and cache persistence reload overheads," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 188–198.

[58] J. Xiao, S. Altmeyer, and A. Pimentel, "Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2017.

[59] J. Xiao and A. D. Pimentel, "Citta: Cache interference-aware task partitioning for real-time multi-core systems," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20, 2020.

[60] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," vol. 48, no. 2, Nov. 2015.

[61] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," vol. 52, no. 3, Jun. 2019.

[62] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real-Time Systems*, vol. 54, no. 3, pp. 607–661, 2018.

[63] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[64] A. Biondi, G. C. Buttazzo, and M. Bertogna, "Schedulability analysis of hierarchical real-time systems under shared resources," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1593–1605, May 2016.

[65] B. B. Brandenburg, "Multiprocessor real-time locking protocols: A systematic review," *arXiv preprint arXiv:1909.09600*, 2019.

[66] A. Wieder and B. B. Brandenburg, "On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks," in *2013 IEEE 34th Real-Time Systems Symposium*, 2013, pp. 45–56.

[67] F. Nemati and T. Nolte, "Resource sharing among real-time components under multiprocessor clustered scheduling," *Real-time systems*, vol. 49, no. 5, pp. 580–613, 2013.

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

[69] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, no. 2-3, p. 117–134, 2012.

[70] J. C. Palencia and M. G. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *Proceedings 19th IEEE Real-Time Systems Symposium*, 1998.

[71] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst, "Model composition for scheduling analysis in platform design," in *Proceedings 2002 Design Automation Conference*, June 2002.

[72] M. Jersak, *Compositional Performance Analysis for Complex Embedded Applications*.

[73] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/S approach," *IEEE Proceedings - Computers and Digital Techniques*, March 2005.

[74] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*.

[75] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Berlin, Heidelberg: Springer-Verlag, 2001.

[76] S. Schliecker and R. Ernst, "A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems," in *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '09, 2009.

[77] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007.

[78] J. Diemer, P. Axer, and R. Ernst., "Compositional performance analysis in python with pyCPA," in *In Proceedings of WATERS'12*, 2012.

[79] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pp. 6–11.

[80] L. T. X. Phan, M. Xu, J. Lee, I. Lee, and O. Sokolsky, "Overhead-aware compositional analysis of real-time systems," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 237–246.

[81] S. Wasly and R. Pellizzoni, "Hiding memory latency using fixed priority scheduling," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 75–86.

[82] D. Casini, P. Pazzaglia, A. Biondi, M. Di Natale, and G. Buttazzo, "Predictable memory-cpu co-scheduling with support for latency-sensitive tasks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.