I/O-GUARD: Hardware/Software Co-Design for I/O Virtualization with Guaranteed Real-time Performance

Zhe Jiang*[||], Kecheng Yang[†], Yunfeng Ma*, Nathan Fisher[‡], Neil Audsley*, Zheng Dong[‡][§]
*University of York, United Kingdom, [||]ARM Ltd, United Kingdom, [†]Texas State University, USA, [‡]Wayne State University, USA

***Abstract*—For safety-critical computer systems, time-predictability and performance are usually required simultaneously in I/O virtualization. However, both requirements are challenging to achieve due to complex I/O access path and resource management at system level and lack of support from preemptive scheduling at I/O hardware level. In this paper, we propose a new framework, I/O-GUARD, which reconstructs the system architecture of I/O virtualization, bringing a dedicated hardware hypervisor to handle resource management throughout the system. The hypervisor improves system real-time performance by enabling preemptive scheduling in I/O virtualization with both analytical and experimental real-time guarantees. Specifically, I/O-GUARD is a First-of-Its-Kind framework for multi-/many-core I/O virtualization.**

I. Introduction

In safety-critical systems, virtualization has gained momentum in both industry and academia [1], driven by the robust isolation between *Virtual Machines* (*VMs*). This inter-VM isolation prevents fault propagation between different VMs, which satisfies the demands of both safety and security required by safety-critical systems [2].

Input/Output (*I/O*) is a vital part of the safety-critical system, but this has not always been recognized [3]. In safety-critical systems, the I/O often interfaces with physical sensors and actuators that need to either sense a potential hazard in time or make a maneuver to avoid a dangerous scenario [4], [5]. Therefore, it is important to assure that I/O operations behave *correctly*, in a *timely* manner, and most importantly with *secured bandwidths* [5]. For instance, in an autonomous control system, real-time decision making module and driving maneuver control module usually require a series of I/Os to occur timely and accurately during specified periods with guaranteed performance, for the detection of objects [5].

Guaranteeing real-time performance for I/O virtualization is hard due to both system level and I/O hardware level research challenges:
System level research challenges. Conventional I/O virtualization involves *complicated I/O access paths* and *resource management* [1], [5], [6], especially in multi-/many-core architectures. For instance, to access an I/O device in a Network-on-Chip-based many-core virtualized system, I/O operations must pass through the guest Operating System (OS), virtual hardware, *Virtual Machine Monitor* (*VMM*), and arbiters/routers (shown in Figure 1). Such complicated paths introduce significant communication latency and timing variance to I/O operations, compared to a legacy system (which does not support any virtualization features). Moreover, along the access paths, potential resource contentions occur at each system level, which involve additional resource management throughout the entire system. The additional resource management elevates the difficulty of satisfying the timing and performance requirements of I/O virtualization [1].
I/O hardware level research challenges. The implementation of traditional I/O controllers relies on *FIFO queues*, which forbids context switches at the hardware level [3]. Effective scheduling methods, e.g., Preemptive Earliest-Deadline-First (P-EDF) policy, cannot be applied to ensure system predictability [1] by prioritizing I/O tasks according to their importance. This hardware dilemma exacerbates the predictability issues introduced at the system level.
Related work. Existing research efforts aimed at achieving real-time I/O virtualization in multi/many-core systems usually concentrated

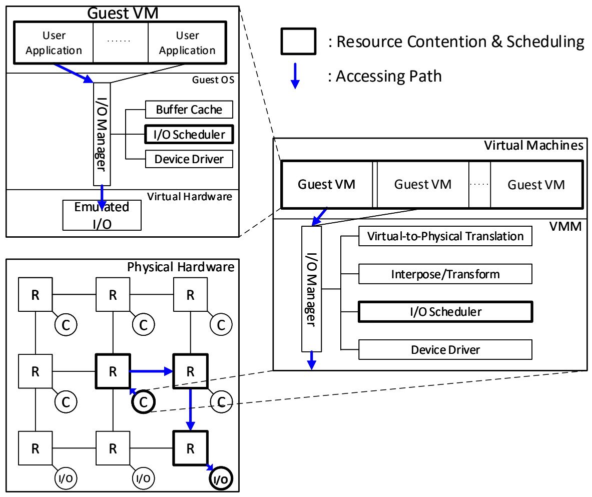


Fig. 1. I/O virtualization involves complex I/O access paths and resource management in all system levels, leading to challenges in guaranteeing its predictability and performance (R: router/arbiter; C: processor core).

on a particular system level. At OS level, Kim *et al.* [7] modified the OS kernel to improve the predictability of I/O scheduling; at VMM level, Gong *et al.* [5] integrated a predictable scheduler into VMM to improve its analyzability. However, it is difficult to ensure the real-time performance of I/O virtualization from a given system level, as the I/O virtualization involves the actual execution of *all* system levels. Moreover, the software methods usually introduce additional computational overhead and complexity, leading to a further reduction of I/O performance [5]. Different from the software methods, Jiang *et al.* [6] proposed *BlueVisor*, a dedicated coprocessor, handling I/O virtualization at hardware level, which improved I/O throughput by introducing paralleling computation for virtualization related functionalities. However, same as the other frameworks, the implementation of the *BlueVisor* remains the FIFO structure at I/O hardware level, which hence cannot guarantee the I/O predictability.
Contribution. In this paper, we propose *I/O-GUARD*, the first system framework to guarantee the real-time performance of multi/many-core I/O virtualization. To achieve this, we present

- A novel *system architecture*, achieving the majority of the virtualization in a *hardware-implemented hypervisor*, allowing the applications in the VMs to access I/Os directly via the hypervisor without the intervention of other system components.
- A new *micro-architecture* for the *I/O-GUARD* hypervisor, enabling *random accesses* of I/O operations and task prioritization.
- A *two-layer scheduler* for the hypervisor, supporting preemptive scheduling methods, with guaranteed real-time performance.
- A *theoretical model* and *analysis* for the proposed framework, demonstrating the improvements to the schedulability brought by *I/O-GUARD* with respect to conventional virtualization.
- Comprehensive *experiments*, including a real-world automotive use case examining overhead, scalability, predictability and performance of *I/O-GUARD* over state-of-the-art I/O virtualization.

The rest of this paper is organized as follows: Sec.II and III detail the system design, followed by the theoretical analysis in Sec.IV.

[§]Corresponding author, dong@wayne.edu.

978-1-6654-3274-0/21/$31.00 © 2021 IEEE

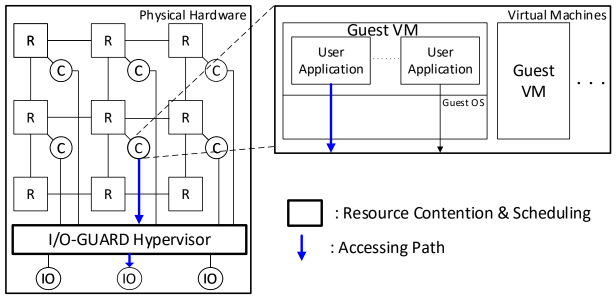
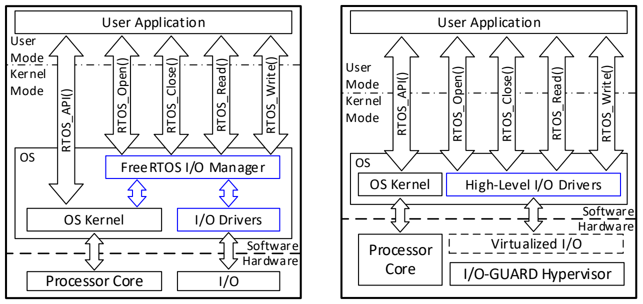


Fig. 2. System architecture of *I/O-GUARD*.



(a) Legacy system (b) *I/O-GUARD*

Fig. 3. RTOSs in legacy system and *I/O-GUARD*.

Sec.V evaluates *I/O-GUARD*, and Sec.VI concludes.

II. *I/O-GUARD*: Architecture

In this work, we make the following assumptions: (i) the hardware platform is a predictability-focused NoC; although *I/O-GUARD* is agnostic to the type of bus, deployment of NoC enhances the predictability of on-chip transactions [6]; (ii) (virtualized) I/O requests and responses transmitted in the hardware are encapsulated as packets using the communication protocol introduced in [8]; (iii) the system elements are synchronized by a single source of timing (global timer).

A. System Architecture

I/O-GUARD changes the system's architecture in both hardware and software levels (Figure 2), compared to a conventional system:
Hardware level. As described in Sec.I, the majority of virtualization in *I/O-GUARD* is achieved by its hypervisor. Hence, in the hardware level, we physically connect the hypervisor to the processors and I/Os. The I/O requests sent from the processors are directly routed to the I/Os via the hypervisor, without involving arbiters/routers. The design details of the hypervisor are presented in Sec.III.
Software level. With the hypervisor, we remove the VMM (which manages I/O virtualization in the conventional architecture) from the software level and directly execute the Real-time Operating Systems (RTOSs) on the processors with full privileges. The RTOSs provide a real-time environment for applications that need timing guarantees. This *bare-metal virtualization* avoids the frequent operating mode switches found in the traditional virtualization (also known as "trap into VMM" [9]), which hence enhances overall system throughput.

In the RTOSs, we replace the I/O manager by new high-level I/O drivers. Figure 3 illustrates the modifications, using FreeRTOS as an example. Different from the legacy system (Figure 3(a)), user applications running in the VM access the virtualized I/Os via the proposed I/O drivers (Figure 3(b)), without the involvement of OS kernel. The implementation of I/O drivers is straightforward, as they only forward the I/O requests to the hypervisor. This *para-virtualization* simplifies the OS kernel by eliminating the (computational and software) overhead caused by the I/O management in the conventional virtualization (evaluated in Sec.V-A).
Compatibility. Although *I/O-GUARD* introduces a new software structure and modifies the OS kernels, the design remains the original OS-application interfaces presented by the legacy systems. Therefore, user applications designed for legacy systems or conventional virtualization can be mapped to the *I/O-GUARD* directly.

B. Working procedure.

Typically, I/O tasks in a system can be either *periodically* or *sporadically*. The periodic I/O tasks are usually determined *before* system execution (named *pre-defined I/O tasks*), e.g., periodic sensor read; and the sporadic I/O tasks are usually generated *during* system execution (named *run-time I/O tasks*), e.g., sporadic body control.

At system initialization, the pre-defined tasks are loaded into the hypervisor with their corresponding start times. During system execution, the hypervisor runs the pre-loaded tasks at the specified times, which guarantees their predictability and performance. At the same time, the hypervisor receives and buffers the run-time I/O tasks requested by the VMs. The hypervisor schedules and executes these run-time tasks when the pre-defined tasks are not occupying the I/O.

In the new system architecture, acquiring system-wide predictability and performance relies on the hypervisor; we therefore present the design details of the *I/O-GUARD* hypervisor in the next section.

III. *I/O-GUARD*: Hypervisor

The design of the *I/O-GUARD* hypervisor keeps modularization in mind, which partitions the hypervisor into two parts:

- **Virtualization manager –** takes charge of the resource management, which decides the execution order of I/O tasks. The design of the virtualization manager is generic to all I/Os.
- **Virtualization driver –** encapsulates the low-level drivers of I/O virtualization, including the instruction/data translation and the I/O control. The design of the I/O driver is specific to the type of connected I/O.

A. Virtualization manager

The design of the virtualization manager (Figure 4) contains two request channels and one response channel. The response channel is pass-through, since the processing speed of the processors is hundreds of times faster than the I/O devices. This means the response channel is not blocked during normal execution. The request channels are respectively designed for pre-defined and run-time I/O tasks, named *Pre-defined I/O task channel* (*P-channel*) and *Run-time I/O task channel* (*R-channel*). We now describe the design of the two channels.
P-channel. The design of the P-channel contains a memory controller, memory banks and an executor. The memory banks store the pre-defined I/O tasks and the corresponding timing information (e.g., the starting time points and the worst-case computation time, etc.), which are loaded during system initialization. We further group this timing information in a look-up table (called *Time Slot Table* σ^*) to record the run-time behaviors of the pre-loaded I/O tasks in each *hyper-period*. During system execution, the executor synchronizes with a global timer and then compares the synchronized results with the time slot table. Once the system executes at a starting time point of a pre-loaded I/O task, the executor loads this task to the connected virtualization driver for execution.
R-channel. The design of the R-channel contains a group of I/O pools, a two-layer scheduler which contains a *local* scheduler (L-Sched) for scheduling real-time tasks in each VM and a *global* scheduler (G-Sched) for allocating free time slots for all VMs, and an executor. The design of the schedulers is agnostic to scheduling methods. Specifically, we use the *preemptive EDF* policy as the scheduling algorithm for both local schedulers and the global scheduler, since it is optimal for uni-processor scheduling. Theoretical results from the two-layer scheduler's real-time performance are discussed in Sec.IV.

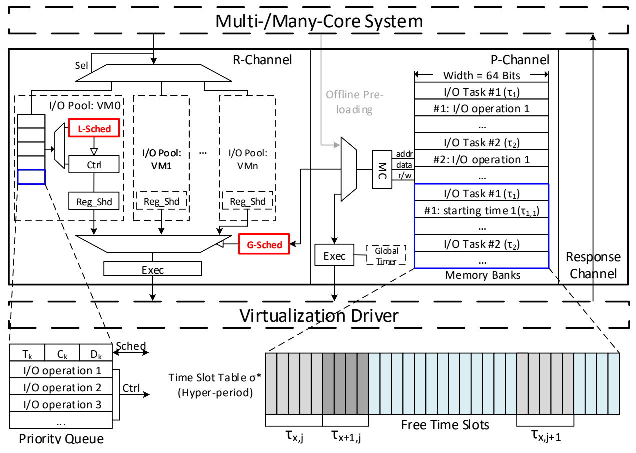


Fig. 4. Micro-architecture of virtualization manager (MC: memory controller).

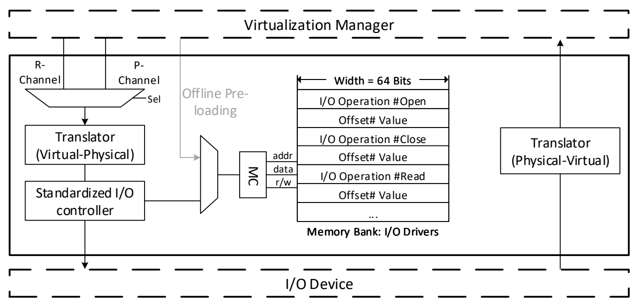


Fig. 5. Micro-architecture of virtualization driver (MC: memory controller).

An I/O pool is associated with a VM, which buffers and schedules the run-time I/O tasks generated by the VM.[1] The design of an I/O pool contains a priority queue, a control logic, a shadow register, and an L-Sched. Different from the conventional FIFO queues, the priority queue has a more complicated structure which introduces an additional slot for each I/O task,[2] storing its *associated parameters* (detailed in Sec.IV). The introduced slot has an accessible interface, allowing the schedulers to read/write these parameters in a timely manner. Moreover, the priority queue supports *random accesses*, which enables the prioritization of the tasks. During execution, the L-Sched keeps checking the status of the tasks, finding the task with the earliest deadline, and requesting the control logic to map the first operation of this I/O task to a shadow register. A G-Sched physically connects to the shadow registers in all I/O pools and the memory banks in the P-channel. It simultaneously compares the deadlines of the I/O operations buffered in the shadow registers and checks free time slots in the time slot table, deciding the next task to be executed and the starting time point. The executor runs the I/O operation selected by the G-Sched and removes it from the priority queue.

B. Virtualization Driver

The design of the virtualization driver contains a pair of open-source real-time translators [6], a standardized I/O controller, and memory banks. The translators are allocated in the request path and the response path, taking charge of the translation of I/O requests and the responding data, respectively. As evidenced in [6], the translator can bound the worst-case time consumption of each translation. During system execution, after receiving an I/O operation from the virtualization manager, the translator (in the request path) firstly translates the I/O operation to bottom-level I/O instructions and then executes them on the I/O controller. Finally, the I/O controller operates the connected I/O device by using the corresponding communication protocol (e.g., SPI, I^{2}C). The drivers of the I/O controller are stored in dedicated memory banks during system initialization.

[1]Partitioning of I/O pools ensures inter-VM isolation at hardware I/O level.
[2]The additionally introduced slots are implemented via registers.

Above we have described the system architecture and design methods of *I/O-GUARD*. In the next section, we detail the schedulability test of the two-layer scheduler to guarantee the system predictability.

IV. Schedulability Test for the Two-layer Scheduler

Our two-layer scheduler is designed to allocate free time slots to the R-Channel I/O operations in a hierarchical manner. In the global layer, available time slots are allocated to n VMs, where each VM i ($1 \le i \le n$) is supported by a periodic server task $\Gamma_i = (\Pi_i, \Theta_i)$ with the interpretation that the server task is invoked every Π_i time slots and receives at least Θ_i time slots between consecutive invocations. The I/O operations from VM i will be executed using the time slots received by VM i. The I/O operations is modeled by a set of sporadic tasks, each of which is denoted $\tau_k = (T_k, C_k, D_k)$. τ_k releases a sequence of I/O operations, or *jobs*, with minimum separation of T_k time slots, where each job completes within C_k time slots of execution and has a deadline at D_k time slots after it is released. We assume *constrained deadlines*, i.e., $\forall k, D_k \le T_k$. Let $\mathcal{T}_i$ denote the task set in VM i, i.e., $\tau_k \in \mathcal{T}_i$ means task k is in VM i. Recall that these jobs are executed *preemptively* at the time-slot level, as described in Sec.III. In the rest of this section, we describe our dual-hierarchy scheduling in company with schedulability analysis.

Supply and demand. We say the *supply* to a set of tasks during a certain time interval as the free time slots available to this set of tasks, and say the *demand* of a set of tasks during a certain time interval as the maximum amount of time slots needed to complete all jobs of these tasks that are released and have a deadline in this time interval. Under preemptive earliest-deadline-first (P-EDF) scheduling, if the demand is at most the supply for *any* time interval, then the deadlines of all tasks in that set are guaranteed to be met [10], [11].

A. Allocating Free Time Slots to VMs (G-Sched)

We let σ^* denote the *Time Slot Table* after P-Channel I/O jobs having been allocated as shown in Figure 4, and let H and F denote the number of total and free time slots in σ^*. Then, this schedule σ^* of length H repeats and results in a (potentially infinitely long) table σ of free time slots to support R-Channel I/O jobs.

Deriving $\mathsf{sbf}(\sigma, t)$. Let the *supply bound function* $\mathsf{sbf}(\sigma, t)$ denote the *minimum* supply to R-Channel I/O jobs in σ during *any* time interval of length t. The value of $\mathsf{sbf}(\sigma, t^*)$ can be obtained for any t^* such that $0 \le t^* \le H-1$ by enumerating a sliding window of length t^* in σ for all cases and there are at most H distinct cases for any given window length t^* since σ repeats σ^* which has a length of H. We store them by a look-up table enum of length H, i.e.,

$$\mathsf{sbf}(\sigma, t) = \mathsf{enum}(t) \text{ for } 0 \le t \le H-1. \tag{1}$$

Also, due to σ strictly repeating σ^*, any time interval of length H in σ must have exact F free time slots no matter where this time interval starts. Therefore,

$$\mathsf{sbf}(\sigma, t) = \mathsf{sbf}(\sigma, t \bmod H) + \left\lfloor \frac{t}{H} \right\rfloor \cdot F \text{ for } t \ge H \tag{2}$$

Thus, $\mathsf{sbf}(\sigma, t)$ for all $t \ge 0$ can be derived by (1) and (2).

On the other hand, we support each VM i by a periodic sever task $\Gamma_i = (\Pi_i, \Theta_i)$ in a manner that all free time slots at which Γ_i is scheduled are devoted to R-Channel I/O jobs from VM i. To ensure that each VM i is guaranteed to receive Θ_i free time slots in every Π_i, the set of tasks $\{\Gamma_i\}$ must be schedulable (i.e., meet all deadlines) on σ. We schedule the task set $\{\Gamma_i\}$ on free time slots in σ by EDF, and the *demand bound function* $\mathsf{dbf}(\Gamma_i, t)$ that denotes the *maximum* demand the periodic implicit-deadline task Γ_i can create in any time interval of length t is calculated by

$$\mathsf{dbf}(\Gamma_i, t) = \left\lfloor \frac{t}{\Pi_i} \right\rfloor \cdot \Theta_i. \tag{3}$$

Thus, the following theorem provides guarantees on the amount of free time slots each VM will receive.

Theorem 1. *Each VM i must be allocated at least Θ_i free time slots in every Π_i time slots, if*

$$\forall t \geq 0, \sum_{i=1}^{n} \mathsf{dbf}(\Gamma_i, t) \leq \mathsf{sbf}(\sigma, t), \tag{4}$$

where $\mathsf{sbf}(\sigma,t)$ and $\mathsf{dbf}(\Gamma_i,t)$ is calculated by (1), (2)*, and* (3).

Note that Theorem 1 does not specify an upper-bound on the $\forall t$. Therefore, we need to check up to the *least common multiple* of all elements in $\{H\} \cup \{\Pi_i\}_{i=1}^{n}$, which can be *exponential* to the input parameters of table σ^* and tasks $\{\Gamma_i\}$. The following theorem provides a *pseudo-polynomial*[3] upper-bound on t for applying Theorem 1.

Theorem 2. *For all systems such that $\frac{F}{H} - \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i} \geq c$ where c is a certain constant such that $c > 0$ (e.g., c = 0.01),* (4) *is true if*

$$\forall t : 0 \leq t < \frac{F \cdot \frac{H-1}{H}}{c}, \sum_{i=1}^{n} \mathsf{dbf}(\Gamma_i, t) \leq \mathsf{sbf}(\sigma, t).$$

Proof. We prove this by showing that

$$\exists t^* \geq 0 \text{ such that } \sum_{i=1}^{n} \mathsf{dbf}(\Gamma_i, t^*) > \mathsf{sbf}(\sigma, t^*) \tag{5}$$

implies $t^* < \frac{F \cdot \frac{H-1}{H}}{c}$. By (2), we have

$$\mathsf{sbf}(\sigma, t^*) \geq \left\lfloor \frac{t^*}{H} \right\rfloor \cdot F \geq \frac{t^* - (H-1)}{H} \cdot F. \tag{6}$$

On the other hand, by (3), we have

$$\sum_{i=1}^{n} \mathsf{dbf}(\Gamma_i, t^*) \leq t^* \cdot \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i}. \tag{7}$$

Thus, by (6) and (7), (5) implies

$$t^* \cdot \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i} > \frac{t^* - (H-1)}{H} \cdot F \Rightarrow t^* < \frac{F \cdot \frac{H-1}{H}}{\frac{F}{H} - \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i}} \Rightarrow t^* < \frac{F \cdot \frac{H-1}{H}}{c}$$

The theorem follows. □

On the limitation of Theorem 2. Please note that, compared to $\frac{F}{H} - \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i} > 0$, the limitation of Theorem 2 only excludes the extremely theoretical case that $\frac{F}{H} - \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i} = \varepsilon$ where $\varepsilon \to 0^+$. On the other hand, $\frac{F}{H} \geq \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i}$ is required anyway, or the system is over-utilized. Therefore, the limitation of Theorem 2 is minimal in practical scenarios (not applicable only when $\frac{F}{H} = \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i}$).

B. Scheduling I/O Jobs within Each VM (L-Sched)

Once the free time slots have been allocated to VMs as described in Sec.IV-A. The R-Channel I/O jobs in each VM can be scheduled and analyzed independently within that VM, where each VM i supported by Γ_i guarantees Θ_i available time slots in every Π_i time slots to the tasks in this VM. This guarantee follows the *periodic resource model* [11]. Therefore, the *supply bound function* $\mathsf{sbf}(\Gamma_i, t)$ denotes the *minimum* supply to R-Channel I/O jobs in VM i in any time interval of length t can be calculated by

$$\mathsf{sbf}(\Gamma_i, t) = \begin{cases} 0 & \text{if } t' < 0 \\ \left\lfloor \frac{t'}{\Pi_i} \right\rfloor \cdot \Theta_i + \theta & \text{if } t' \geq 0 \end{cases} \tag{8}$$

where $t' = t - (\Pi_i - \Theta_i)$ and $\theta = \max\left(t' - \Pi_i \left\lfloor \frac{t'}{\Pi_i} \right\rfloor - (\Pi_i - \Theta_i), 0\right)$.

[3]Informally, that is polynomial to the *values* of the input parameters of table σ^* and tasks $\{\Gamma_i\}$. Please note that, c is a constant and $\frac{H-1}{H} < 1$.

We schedule the task set $\mathcal{T}_i$ on these free time slots available to VM i by EDF, and the *demand bound function* $\mathsf{dbf}(\tau_k, t)$ that denotes the *maximum* demand a task $\tau_k \in \mathcal{T}_i$ can create in any time interval of length t is calculated by

$$\mathsf{dbf}(\tau_k, t) = \left(\left\lfloor \frac{t - D_k}{T_k} \right\rfloor + 1 \right) \cdot C_k. \tag{9}$$

Thus, the following theorem provides a schedulability test for the tasks in each VM i.

Theorem 3. *All I/O jobs from VM i meet their deadlines if*

$$\forall t \geq 0, \sum_{\tau_k \in \mathcal{T}_i} \mathsf{dbf}(\tau_k, t) \leq \mathsf{sbf}(\Gamma_i, t), \tag{10}$$

where $\mathsf{sbf}(\Gamma_i, t)$ and $\mathsf{dbf}(\tau_k, t)$ are calculated by (8) *and* (9).

Again, Theorem 3 does not specify an upper-bound on the $\forall t$, and checking up to the *least common multiple* of all elements in $\{\Pi_i\} \cup \{T_k\}_{\tau_k \in \mathcal{T}_i}$ may results in the schedulability test running in exponential time. The following theorem provides a pseudo-polynomial schedulability test with a minimal limitation similar to that of Theorem 2.

Theorem 4. *For each VM i such that $\frac{\Theta_i}{\Pi_i} - \sum_{\tau_k \in \mathcal{T}_i} \frac{C_k}{T_k} > c'$ where c' is a certain constant such that $c' > 0$ (e.g., c' = 0.01),* (10) *is true if*

$$\forall t : 0 \leq t < \frac{\max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} + 2\Pi_i - \Theta_i - 1}{c'},$$
$$\sum_{\tau_k \in \mathcal{T}_i} \mathsf{dbf}(\tau_k, t) \leq \mathsf{sbf}(\Gamma_i, t).$$

Proof. We prove this by showing that

$$\exists t^* \geq 0 \text{ such that } \sum_{\tau_k \in \mathcal{T}_i} \mathsf{dbf}(\tau_k, t) > \mathsf{sbf}(\Gamma_i, t) \tag{11}$$

implies $t^* < \frac{\max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} + 2\Pi_i - \Theta_i - 1}{c'}$. By (8), we have

$$\mathsf{sbf}(\Gamma_i, t^*) \geq \left\lfloor \frac{t^* - (\Pi_i - \Theta_i)}{\Pi_i} \right\rfloor \cdot \Theta_i \geq \frac{t^* - (\Pi_i - \Theta_i) - (\Pi_i - 1)}{\Pi_i} \cdot \Theta_i$$
$$\geq t^* \cdot \frac{\Theta_i}{\Pi_i} - (2\Pi_i - \Theta_i - 1). \tag{12}$$

The last inequality is because $1 \leq \Theta_i \leq \Pi_i$ implies that $2\Pi_i - \Theta_i - 1 \geq 0$ and $0 < \frac{\Theta_i}{\Pi_i} \leq 1$. On the other hand, by (9), we have

$$\begin{aligned} \sum_{\tau_k \in \mathcal{T}_i} \mathsf{dbf}(\tau_k, t^*) &\leq \sum_{\tau_k \in \mathcal{T}_i} \frac{t^* + (T_k - D_k)}{T_k} \cdot C_k \\ &\leq \sum_{\tau_k \in \mathcal{T}_i} \frac{t^* + \max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\}}{T_k} \cdot C_k \\ &= \sum_{\tau_k \in \mathcal{T}_i} \frac{C_k}{T_k} \cdot (t^* + \max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\}) \\ &\leq \sum_{\tau_k \in \mathcal{T}_i} \frac{C_k}{T_k} \cdot t^* + \max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\}. \end{aligned} \tag{13}$$

The last inequality is because $\sum_{\tau_k \in \mathcal{T}_i} \frac{C_k}{T_k} \leq \frac{\Theta_i}{\Pi_i} \leq 1$ is necessarily required for no over-utilization and $\max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} \geq 0$ holds for constrained-deadline tasks. Thus, by (12) and (13), (11) implies

$$\sum_{\tau_k \in \mathcal{T}_i} \frac{C_k}{T_k} \cdot t^* + \max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} > t^* \cdot \frac{\Theta_i}{\Pi_i} - (2\Pi_i - \Theta_i - 1)$$
$$\Rightarrow t^* < \frac{\max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} + 2\Pi_i - \Theta_i - 1}{\frac{\Theta_i}{\Pi_i} - \sum_{\tau_k \in \mathcal{T}_i} \frac{C_k}{T_k}}$$
$$\Rightarrow t^* < \frac{\max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} + 2\Pi_i - \Theta_i - 1}{c'}$$

The theorem follows. □

V. Evaluation

We now conduct extensive experiments to evaluate *I/O-GUARD*.
Experimental platform. We built *I/O-GUARD* on a Xilinx VC709 evaluation board. The hypervisor was implemented using BlueSpec System Verilog [12] and connected to a 5×5 mesh type open-source NoC [8]. As well as the hypervisor, the NoC also contained 16 MicroBlaze processors [13], memory and I/O peripherals. Each processor supported up to three guest VMs. The software executing on the processors was compiled using a Xilinx MicroBlaze GNU tool-chain [13]. We selected FreeRTOS (v.10.4) as the OS kernel for all VMs, with the modifications introduced in Sec.II-A. Additionally, we introduced three baseline systems (**BS**) running on a similar hardware architecture: ***BS*|*Legacy*** was an NoC system without virtualization support, which left the scheduling related to resource management to the routers, and each processor is deemed as a VM. ***BS*|*RT-XEN*** was a virtualized system established using a Xen hypervisor with real-time patches and I/O enhancement [14]. Both patches and I/O enhancement were implemented in software. ***BS*|*BV*** was a virtualized system built on hardware assistance (*BlueVisor*) introduced in [6], which was reviewed in Sec.I. All architectures ran at 100 MHz.

A. Software Overhead

Experimental setup. The software overhead was evaluated using the run-time memory footprint, with specific consideration of hypervisor, OS kernel and I/O drivers. The legacy OS kernel was fully-featured, but excluded from I/O drivers [15].
Obs 1. Additional software overheads were induced by the conventional I/O virtualization compared to the legacy system. These were considerably improved in *I/O-GUARD*.

This observation is shown in Figure 6. In *BS*|*RT-XEN*, the introduction of a hypervisor and modifications to the OS kernel brought an additional 61 KB (129.8%) memory footprint compared to the legacy system. The hardware-assisted virtualization (*BS*|*BV* and *I/O-GUARD*) effectively reduced this overhead by moving I/O virtualization to the hardware. Compared to *BS*|*BV*, the *I/O-GUARD* entirely eliminated the software overhead of the VMM by directly running the kernels on the processors. For I/O drivers, the complexity of the I/O device determines the its software overhead. For each of the evaluated I/O drivers, *BS*|*RT-XEN* always sustained the most significant software overhead. This overhead was reduced by *I/O-GUARD*, since it integrates the low-level I/O drivers into the hardware.

B. Hardware Overhead

I/O-GUARD requires additional implementation of the hypervisor. Therefore, in this section, we evaluate its hardware overhead.
Experimental setup. We first configured the *I/O-GUARD* to support 16 VMs and 2 I/Os. This means the hypervisor contained 2 groups of virtualization managers and virtualization drivers, where each virtualization manager contained 16 I/O pools (see Sec.III-A). We then compared the hypervisor with two general-purpose processors (MicroBlaze and RISC-V), and two mainstream I/O controllers (SPI, and Ethernet). The MicroBlaze was full-featured, enabling all the performance related functionalities (e.g., pipeline, data cache). The RSIC-V was implemented based on [16], supporting all the functionalities of the MicroBlaze, as well as multi-branch, out-of-order processing and the related functionalities (e.g., branch-prediction). The I/O controllers were chosen from the standard Xilinx IP library.
Obs 2. The design of the hypervisor (of *I/O-GUARD*) was resource-efficient compared to the full-featured processors. Its hardware consumption was slightly higher than commonly used I/O controllers.

As shown in Table I, *I/O-GUARD* required significantly less hardware than full-featured processors: MicroBlaze (56.6% LUTs, 67.8% registers, 77.7% power) and RSIC-V (37.4% LUTs, 18.2% registers, 47.9% power). Due to the hardware-implemented virtualization and drivers, *I/O-GUARD* consumed more hardware than the standard I/O

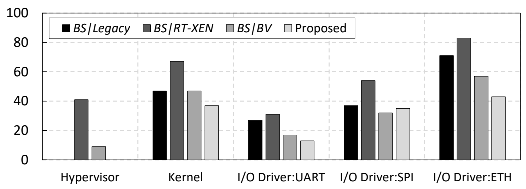


Fig. 6. Run-time software overhead (unit: KB). The software overhead is evaluated via memory footprint, containing segments of BSS, data and text.

TABLE I
Hardware Overhead (Implemented on FPGA)

	LUTs	Registers	DSP	RAM (KB)	Power (mW)
MicroBlaze	4908	4385	6	256	359
RSIC-V	7,432	16,321	21	512	583
SPI	632	427	0	0	4
Ethernet	1321	793	0	0	7
BlueIO	3236	3346	0	256	297
Proposed	2777	2974	0	256	279

controllers. But when compared to *BS*|*BV*, *I/O-GUARD* required the same memory consumption, but less LUTs, registers and DPSs.

C. Case Study

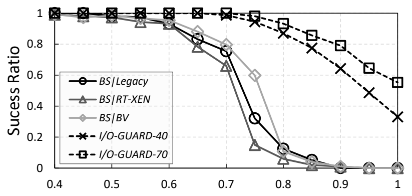
We now use an automotive case study to examine the benefits of the *I/O-GUARD* over a conventional virtualized system framework.
Systems Configurations. To analyze the benefits of *I/O-GUARD*, all introduced systems were examined. We configured *I/O-GUARD* as *I/O-GUARD*-40/70, which pre-loaded $40/70\%$ of I/O tasks into the virtualization manager before run-time. In other words, *I/O-GUARD*-x indicates that $x\%$ of I/O tasks were executed by the P channel and $(1 - x\%)$ of I/O tasks were executed by the R-channel.
Task sets. We introduced three sets of I/O-related tasks:

- 20 automotive safety tasks, selected from the Renesas automotive use case database [17], e.g., CRC, RSA32, etc..
- 20 automotive function tasks, selected from EEMBC benchmark [18], e.g., fast Fourier transform, speed calculation, etc..
- synthetic workloads, selected from EEMBC benchmark, which could be added into system to control overall system utilization.

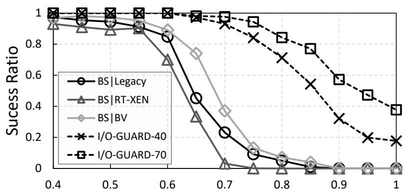
We employed a hybrid-measurement approach to obtain WCETs for all the tasks [19]. The raw data processed by the 40 tasks was randomly generated off-chip and sent to the evaluated systems via an Ethernet controller (1 Gbps) at run-time. The results were sent back via a FlexRay (10 Mbps). Each task had a defined period and implicit deadline, with overall system utilization approximately 40%. In practical systems, the execution time of a task is affected by diverse factors (e.g., cache miss rate); hence, adding synthetic workloads to a system only gives it a *target utilization*.
Experimental Setup We introduced two groups of experimental setups, which activated $4/8$ VMs to execute the experimental task sets and synthetic workloads. In each experimental group, we executed each examined system $1,000$ times under varying target utilization from 40% to 100% (with an interval of 5%). Each execution lasted 100 seconds, which guaranteed that all tasks executed at least 250 times. For fair comparison, we also ensured the data input to the examined systems was identical in each execution.

We evaluated the examined systems using *success ratio* and *I/O throughput*. The *success ratio* recorded the percentage of trials that executed successfully (i.e., without deadline miss of any safety and function task), under a specified target utilization. The *I/O throughput* evaluated the average I/O performance of each examined system.
Obs 3. Introducing *I/O-GUARD* was beneficial.

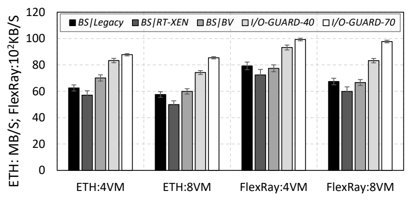
As shown in Figures 7(a), 7(b), and 7(c), with the same configuration, the *I/O-GUARD*s always achieved higher success ratios and I/O throughput compared to the baseline systems. Moreover, we also observed that *I/O-GUARD*-70 consistently outperformed *I/O-GUARD*-40 in both success ratios and I/O throughput, with less experimental variance. This means that pre-loading a higher percentage of I/O tasks into the *I/O-GUARD* before run-time introduces more benefits.



(a) 4-VM system (x-axis: target utilization).

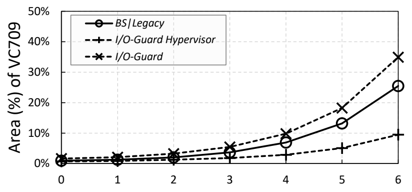


(b) 8-VM system (x-axis: target utilization).

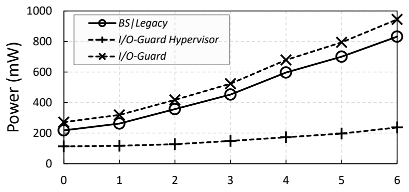


(c) Average I/O throughput.

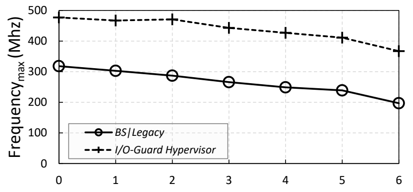
Fig. 7. Automotive case study.



(a) Area consumption.



(b) Power consumption.



(c) Maximum frequency.

Fig. 8. Area, power, and maximum frequency v.s. scaling factor η (The x-axis denotes the scaling factor η).

Obs 4. Increasing the number of VMs significantly reduced the success ratio and I/O throughput of the conventional virtualization. Such issues were effectively eliminated by *I/O-GUARD*.

This observation is shown by the comparison between the results of two experimental groups. In 4-VM *BS*|*RT-XEN* and *BS*|*BV*, significant drops in the success ratios occurred at 70% and 75% of target utilization; whereas these drops moved to 65% target utilization in 8-VM *BS*|*RT-XEN* and *BS*|*BV*. This observation mainly results from the additional on-chip interference and resource contention generated by the introduced VMs and tasks (explained in Sec.I).

In *I/O-GUARD*, the system architecture optimizes the I/O access paths and leaves the resource management to the hypervisor. It hence reduces on-chip interference and manages the I/O resources in a time-predictable manner (achieved via 2-layer scheduler), which improves overall I/O real-time performance. In an 8-VM system, when target utilization approached 100%, *I/O-GUARD*-70 maintained a success ratio which was close to 40% with negligible loss of I/O throughput.

D. Scalability

Since scalability impacts the feasibility of the proposed design, the scalability of *I/O-GUARD*is examined by a varying number of VMs. **Experimental setup.** The same method described in Sec.V-B is adopted to implement the *I/O-GUARD* and *BS*|*Legacy* with a scaling number of basic MicroBlaze processors. Additionally, we introduced a scaling factor: η to control the number of VMs (2^{η}).

First, we compared the scalability of area consumption between the evaluated systems, where the area consumption was normalized by the overall area of the experimental platform. We then examined the scalability of power consumption, calculated as the sum of static and dynamic power. Lastly, we evaluated the maximum frequency of the hypervisor in *I/O-GUARD* and *BS*|*Legacy* using varying η.

Obs 5. The area and power consumption of *I/O-GUARD* were linearly scaled by η. Compared to the legacy system, the area and power consumption of *I/O-GUARD* increased slightly.

As shown in Figure 8(a), when the system scaled with η, the area consumption of both *BS*|*Legacy* and *I/O-GUARD* consistently increased. In all examined cases, although *I/O-GUARD* consumed more area than *BS*|*Legacy*, the additionally introduced area consumption was always bounded within a small margin – less than 20%.

Power consumption is usually determined by four factors: voltage, clock frequency, toggle rate and design area [20]. Because the unified voltage, clock frequency and simulated toggle rate were assigned to the systems being compared, the design area dominated the overall power consumption. As expected, in Figure 8(b), we observed linearly increased power consumption in these systems when η increases.

Obs 6. When the system scaled with η, introducing the hypervisor (in *I/O-GUARD*) did not affect maximum performance.

As shown in Figure 8(c), when the system scaled with η, the maximum frequency of the hypervisor was always greater than the *BS*|*Legacy*. This indicates that the hypervisor did not become a critical path and could not reduce maximum system performance.

VI. Conclusion

This paper proposes a system framework (*I/O-GUARD*) for multi-/many-core I/O virtualization. *I/O-GUARD* introduces a novel system architecture, including both a new hypervisor micro-architecture and a two-layer scheduler, to simultaneously optimize I/O access paths and resource management throughout the system. A theoretical model and schedulability analysis are presented for *I/O-GUARD*, which demonstrate improved schedulability compared to conventional I/O virtualization. As shown in the evaluation, *I/O-GUARD* outperforms state-of-the-art I/O virtualization with varying hardware architectures.

References

[1] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on fpga virtualization," in *International Conference on Field Programmable Logic*, 2018.

[2] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[3] A. Burns and A. J. Wellings, *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*, 2001.

[4] J. Mössinger, "Software in automotive systems," *IEEE software*, 2010.

[5] X. Gong, D. Cao, Y. Li, X. Liu, Y. Li, J. Zhang, and T. Li, "A thread level slo-aware i/o framework for embedded virtualization," *TPDS*, 2020.

[6] Z. Jiang and N. Audsley, "Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems," in *RTAS*, 2018.

[7] N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter, "Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks," in *RTNS*, 2018.

[8] G. Plumbridge, "Blueshell: a platform for rapid prototyping of multiprocessor NoCs and accelerators," *Computer Architecture News*, 2014.

[9] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues."

[10] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *RTSS*, 1990.

[11] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symposium, 2003*, 2003.

[12] "Bluespec System Verilog," https://bluespec.com.

[13] Xilinx, "Microblaze," https://www.xilinx.com/products/microblaze.

[14] S. Xi, M. Xu, C. Lu, L. T. Phan, C. Gill, , and I. Lee, "Real-time multi-core virtual machine scheduling in xen," in *EMSOFT*, 2014.

[15] FreeRTOS, "FreeRTOS official website," http://www.freertos.org/.

[16] S. Mashimo *et al.*, "An open source fpga-optimized out-of-order RISC-V soft processor," in *ICFPT*, 2019.

[17] R. Electronics, "Renesas: Automotive Use Cases," https://www.renesas.com/solutions/automotive.html.

[18] EEMBC, "EEMBC benchmark," https://www.eembc.org/autobench/.

[19] S. Law, M. Bennett, and Hutchesson, "Effective worst-case execution time analysis of DO178C level a software." *Ada User Journal*, 2015.

[20] A. Bellaouar and M. Elmasry, *Low-power digital VLSI design: circuits and systems*. Springer Science & Business Media, 2012.