Towards Hard Real-Time and Energy-Efficient Virtualization for Many-core Embedded Systems

Zhe Jiang, Kecheng Yang, Yunfeng Ma, Nathan Fisher, Neil Audsley, Zheng Dong§

Abstract—In safety-critical computing systems, the I/O virtualization must simultaneously satisfy different requirements, including time-predictability, performance, and energy-efficiency. However, these requirements are challenging to achieve due to complex I/O access path and resource management at the system level, lack of support from preemptive scheduling at I/O hardware level, and missing an effective energy management method. 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. Furthermore, we also present a dedicated energy management unit to adjust *I/O-GUARD*'s dynamic energy using frequency scaling. Associated with that, a frequency identification algorithm is proposed to find the appropriate executing frequency at run-time. As shown in experiments, *I/O-GUARD* simultaneously improves the predictability, performance and energy-efficiency compared to the state-of-the-art I/O virtualization.

Index Terms—Real-time Systems, I/O Virtualization, Energy-Efficiency, Schedulability, Scalability, Hardware/Software Co-design.

1 Introduction

Safety-critical systems have stringent assurance and verification requirements that are absolutely essential to life-critical applications, including medical, automotive, aerospace and industrial automation [1]–[3]. In safety-critical systems, virtualization has gained increasing momentum, driven by the robust isolation between different Virtual Machines (VMs) [4]. Such inter-VM isolation prevents fault propagation between different VMs, which satisfies the demands of both safety and security required by safety-critical systems [1].

As a part of safety-critical systems, *Input/Output* (*I/O*) is vital but has not been widely recognized [5], [6]. Specifically, 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 [7], [8]. Therefore, it is important to assure that I/O operations behave *correctly*, in a *timely* manner, and most importantly with *secured bandwidths* [8]. 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 [8].

It is very difficult to guarantee predictability and performance of I/O virtualization, especially for multi-/many-core architectures, such as *Network-on-Chip* (*NoC*). This is because of the research challenges (C.x) listed below:

C.1: System level. Conventional I/O virtualization involves *complicated I/O access paths* and *resource management* [4], [8], [9], 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 Fig. 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 extra resource management elevates the difficulty of satisfying the real-time requirements of I/O virtualization [4].

C.2: I/O hardware level. The implementation of traditional I/O controllers relies on *FIFO queues*, which forbids context switches at the hardware level [5]. Effective scheduling methods, e.g., Preemptive Earliest-Deadline-First (P-EDF) policy, cannot be applied to ensure system predictability [4] by prioritizing I/O tasks according to their importance.

C.3: Energy-efficiency. Safety-critical systems are usually implemented on embedded computing platforms, in which the energy is usually constrained. A methodology presented to solve the above challenges must be energy-efficient. Different from the processor and memory virtualization, an effective energy management method of I/O virtualization is still missing.

Contributions. We propose *I/O-GUARD*, a scalable and energy-efficient system framework, guaranteeing the real-time performance of multi/many-core NoC-based I/O virtualization. To this end, we introduce a novel system architecture, including both a new hypervisor micro-architecture and a two-layer scheduler, which simultaneously optimize I/O access paths and resource management throughout the system. Moreover, we present a dedicated energy management unit to support run-time frequency scaling of the I/O

- *Zhe Jiang is with Computer Science Department, University of Cambridge, United Kingdom, CB3 0FD.*
- *Kecheng Yang is with the Department of Computer Science, Texas State University, San Marcos, TX 78666, United States.*
- *Yunfeng Ma is with Computer Science Department, University of York, United Kingdom, YO10 5GH.*
- *Neil Audsley is with the Department of Computer Science, City, University of London, United Kingdom, EC1V 0HB.*
- *Zheng Dong and Nathan Fisher are with the Department of Computer Science, Wayne State University, Detroit, MI, 48202, United States.*

This work was supported in part by the U.S. National Science Foundation under Grants CNS-2103604, CNS-2140346, CNS-2038609, IIS-1724227, CCF-2118202 and CNS-2104181, in part by a start-up Grant from Wayne State University, in part by start-up and REP grants from Texas State University.

§. Corresponding author, dong@wayne.edu.

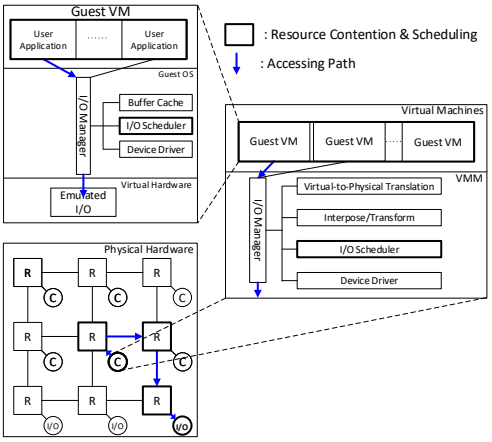


Fig. 1. System architecture of conventional I/O virtualization (R: router/arbiter; C: processor core).

virtualization, and a frequency identification algorithm to find the appropriate run-time frequency. Corresponding to the new system, we present a theoretical model and schedulability analysis to demonstrate the improved schedulability compared to conventional I/O virtualization. At last, we examined *I/O-GUARD* using different metrics.

Paper organization. The rest of the paper is organized as follows: Sec. 2 gives the overview of *I/O-GUARD*, followed by the design details in Sec. 3 and Sec. 4. Sec. 5 presents the theoretical model and optimization to select *I/O-GUARD*'s configurations. Sec. 6 evaluates *I/O-GUARD*-MCS. Sec. 7 reviews the related work and Sec. 8 concludes.

2 *I/O-GUARD*: Overview

To ensure the real-time performance and energy-efficiency of I/O virtualization, we present a new system framework, *I/O-GUARD*, employing a *hardware-implemented* hypervisor with a dedicated energy management unit. The hypervisor realizes the majority of I/O virtualization and manages resource contentions throughout the system. The energy management unit adjusts *I/O-GUARD*'s dynamic energy using run-time frequency scaling. Associated with the new system framework, we present a frequency identification algorithm and the analysis framework to find the optimized configurations of the *I/O-GUARD*, further optimizing its real-time performance and energy-efficiency.

2.1 Context

In this paper, we have the following assumptions: (i) the proposed design and theoretical analysis only focus on I/O virtualization; (ii) hardware platform is a predictability-focused NoC. Although *I/O-GUARD* has been integrated and evaluated with a NoC-based many-core system [10], integrating *I/O-GUARD* with other complex interconnects still needs further investigation. (iii) I/O requests and responses transmitted in the hardware are encapsulated as packets using the communication protocol introduced in [10], where the I/O requests and responses are transmitted using full-duplex channels. Each transaction contains header and payload packets. The header packet is used for handshake, and the payload packet is used for data exchange.

2.2 Design Concepts.

The *I/O-GUARD* design has four Design Concepts (DC.x):

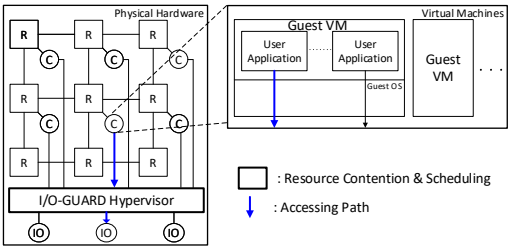


Fig. 2. System architecture of *I/O-GUARD*.

DC.1: New system architecture. *I/O-GUARD* presents a novel system architecture, archiving the majority of I/O virtualization via a *hardware-implemented* hypervisor and allowing the applications (in VMs) to access I/Os directly via the hypervisor without intervening other system components. This new system architecture simplifies I/O access paths and minimizes resource contentions/management compared to conventional virtualization.

DC.2: New hypervisor micro-architecture. *I/O-GUARD* hypervisor presents a new micro-architecture, enabling random accesses of I/O operations and task prioritization at hardware level. Based on the new micro-architecture, preemptive scheduling methods are applied, guaranteeing real-time performance of I/O virtualization.

DC.3: Run-time energy management. *I/O-GUARD* contains a dedicated energy management unit, supporting online dynamic frequency scaling to the hypervisor and the controlled I/O devices.

DC.4: Energy-aware real-time scheduling. In light of the new design, an energy-aware system model and the associated schedulability analysis are presented to ensure *I/O-GUARD*'s predictability, performance, and energy-efficiency, simultaneously.

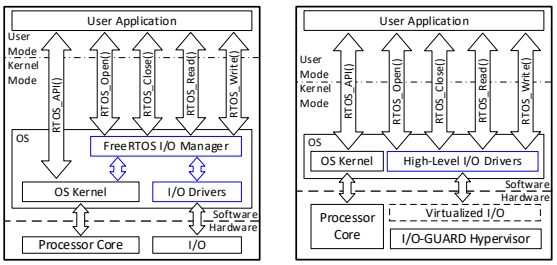
2.3 System Architecture

The *I/O-GUARD* has architecture changes (Fig. 2) in both hardware and software layers compared to the conventional I/O virtualization (Fig. 1).

Hardware level. As described in DC.1, 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 detailed in Sec.3.

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" [11]), which hence enhances overall system throughput.

In the RTOSs, we replace I/O manager by new high-level I/O drivers. Fig. 3 illustrates the modifications, using FreeRTOS as an example. Different from the legacy system (Fig. 3(a)), user applications running in the VM access the virtualized I/Os via the proposed I/O drivers (Fig. 3(b)), reducing 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*



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

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

simplifies the OS kernel by eliminating the (computational and software) overhead caused by the I/O management in the conventional virtualization (evaluated in Sec.6.1).

2.4 Compatibility

Software applications. 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, allowing the software applications to use the same system calls to access the I/Os. Therefore, user applications designed for legacy systems or conventional virtualization can be mapped to the *I/O-GUARD* directly.

Other virtualization types. As stated in Sec. 2.1, *I/O-GUARD* only supports I/O virtualization. To enable other types of virtualization, *e.g.*, memory virtualization, the corresponding (software or hardware) hypervisor is required. The straightforward solution of integrating *I/O-GUARD* in a mature hypervisor is replacing the hypervisors' I/O management modules using the high-level drivers presented by *I/O-GUARD* (same as the example given by Fig. 3).

2.5 Programming Model and Working procedure

In real-time systems, I/O tasks are released by the software tasks, and an I/O task usually contains multiple I/O operations. Typically, I/O tasks are classified into periodic and sporadic tasks. The periodic I/O tasks are released with fixed time intervals by the software tasks, usually determined before system execution and hence also termed pre-defined I/O tasks. Differently, the sporadic I/O tasks are triggered by run-time events, often generated during system execution and hence also termed run-time I/O tasks.

To ensure the real-time predictability for both categories of I/O tasks, *I/O-GUARD* loads the pre-defined tasks into the hypervisor with their corresponding start times at initialization,*e.g.*, during firmware boot, using the configuration interfaces. During system execution, the software tasks are not required to release the pre-defined tasks and the hypervisor can directly run these tasks at the specified times (synchronized with the software tasks), guaranteeing the pre-defined tasks' predictability and performance. At the same time, the run-time tasks can be sent to the hypervisor using the execution interfaces (*i.e.*, the high-level I/O drivers illustrated in Fig. 3(b), and the hypervisor then schedules and executes the sporadic tasks when the periodic tasks are not occupying the I/O.

As introduced, the design of *I/O-GUARD* relies on its hypervisor. The hypervisor is designed from two dimensions to ensure the system's real-time performance and energy-efficiency, respectively. In the following sections, we describe the real-time features (in Sec. 3) and energy-efficient features (in Sec. 4) of *I/O-GUARD* hypervisor, respectively.

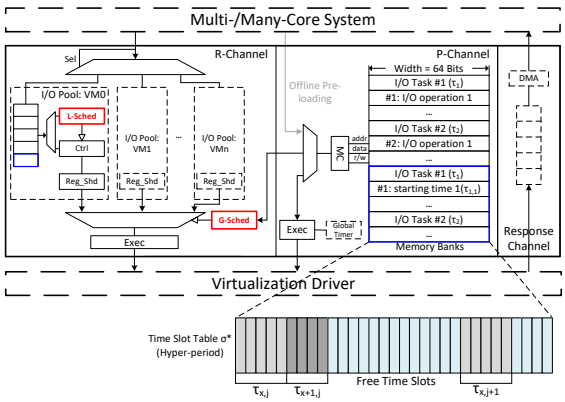


Fig. 4. Micro-architecture of virtualization manager (MC: memory controller; dash lines: optional modules).

3 *I/O-GUARD*: Real-time Features

The design of the *I/O-GUARD* hypervisor mainly contains the virtualization manager and virtualization driver, managing the resource contentions and realizing the hardware-level I/O virtualization:

- **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 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.

The virtualization manager and driver are associated with each I/O device. Hence, to support a new I/O device in *I/O-GUARD*, the corresponding virtualization manager and driver are required to be added. We now detail the design of these two modules below.

3.1 Virtualization manager

The design of the virtualization manager (Fig. 4) contains two *request* channels and one *response* channel. 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*); and the response channel was designed for I/O responses. We now describe the design of the two channels.

3.1.1 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*. Note that, in the time slot table, I/O tasks' WCET is accounted for in terms of time slots. 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.

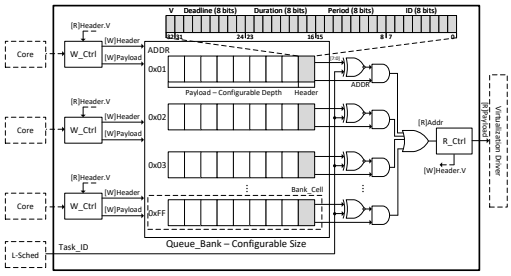


Fig. 5. Micro-architecture of a priority queue (W: write; R: read; W_Ctrl: write controller; R_Ctrl: read controller; V: Validness).

3.1.2 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 run-time tasks in each VM and a *global* scheduler (G-Sched) for allocating free time slots for all VMs, and an executor for tasks' execution. The design of the schedulers is agnostic to scheduling methods. Specifically, we use the *preemptive EDF* policy as the scheduling algorithm for both local and the global schedulers, since it is optimal for uni-processor scheduling. Theoretical results from the two-layer scheduler's real-time performance are discussed in Sec.5.

An I/O pool is associated with a VM, which buffers and prioritizes the run-time I/O tasks generated by the VM, selecting the I/O tasks with the highest priority.[1] The design of an I/O pool contains a priority queue, a control logic, a shadow register, and an L-Sched. The priority queue buffers the I/O requests sent from the associated VM. During execution, the L-Sched keeps checking the status of the tasks, finding the task with the earliest deadline, and mapping 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, using the virtualization driver, and removes it from the priority queue.

Priority queue. Unlike the conventional FIFO queues, the priority queue has a more complicated structure, enabling random access. The design of the priority queue comprises a queue bank and multiple queue controllers (see Fig. 5).

A *bank cell* is the essential element of a queue bank, with a unique address starting from 0x01.[2] The bank cell design has two parts: a *payload FIFO* and a *cell header*. The payload FIFO stores the transferred content (*e.g.*, I/O operations), and the cell header stores the parameters associated to the I/O tasks, including task ID (bits 0 - 7), period (bits 8 - 15), duration (bits 16 - 23) and deadline (bits 24 - 31). In addition, we use the cell header's highest bit (bit 32) to indicate the validity of this bank cell. The depth of a payload FIFO is configurable, providing flexibility for customization.

We also use write/read controllers to store/fetch the content in the queue bank. A write/read controller contains two interfaces connected to the cell headers and payload FIFOs, respectively. In the *storing procedure*, the write controller first reads the cells' headers to check the cell validity (*i.e.*, *header.bit[32]*). If the controller finds an unused bank cell (*i.e.*, *header.bit[32]* == 0), it sets the cell *header.bit[32]* to 1 and then starts to *push* the transfer content to the payload FIFO. In the *fetching procedure*, the read controller pulls the content from a payload FIFO using its address, and then writes the cell's *header.bit[32]* to 0. As introduced above, the the I/O pool relies on the L-Sched and G-Sched to prioritize the tasks, scheduling the tasks using their IDs. Hence, we present a combinational logic circuit connecting the read controller and the queue bank to convert the IDs into the address of a specific bank cell in a fixed single clock cycle.

1. Partitioning of I/O pools is required, as it ensures inter-VM isolation at hardware I/O level.
2. We use the address 0x00, which indicates an invalid address.

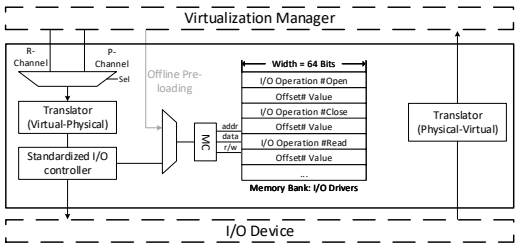


Fig. 6. Micro-architecture of virtualization driver.

3.1.3 Response channel

In coping with the NoC interconnect [10] used in *I/O-GUARD* (Sec. 2.1), the I/O responses can be directly sent back to the processors' communication buffers or the shared memory. We proposed three design options for the response channel: (i) pass-through design, where the I/O responses are sent to the processors directly without buffering; (ii) priority queue based design, where the I/O responses are buffered in a priority queue and sent to the processor's communication buffers based on their priorities; (iii) DMA based design, where the I/O responses are sent to the shared memory using a DMA.

Option (i) is designed for low-speed I/Os, *e.g.*, UART and I^2C transmitters. The low-speed I/Os' are usually hundreds of times slower than the processors; hence the response channel is unlikely blocked during normal execution. When resource contentions occur in the pass-through design, contention management entirely relies on the routers of the NoC. Option (ii) is designed for high-speed I/Os, *e.g.*, FlexRay transmitters, where the priority queues can buffer and prioritize the blocked I/O responses. Option (iii) is designed for the I/Os required to transmit a large amount of data, *e.g.*, Ethernet transmitters, allowing the I/Os to send the data directly to the shared memory.

3.2 Virtualization Driver

The design of the virtualization driver contains a pair of open-source real-time translators [9], a standardized I/O controller, and memory banks (see Fig. 6). 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 [9], 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^2C). The drivers of the I/O controller are stored in dedicated memory banks during system initialization.

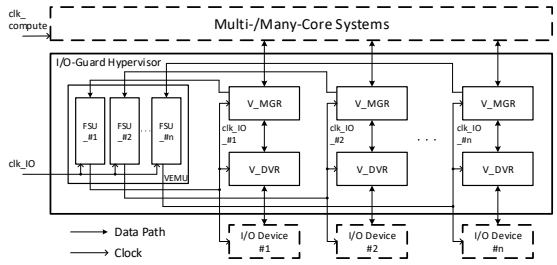


Fig. 7. Top-level energy management framework (V_MRG: virtualization manager; V_DVR: virtualization driver).

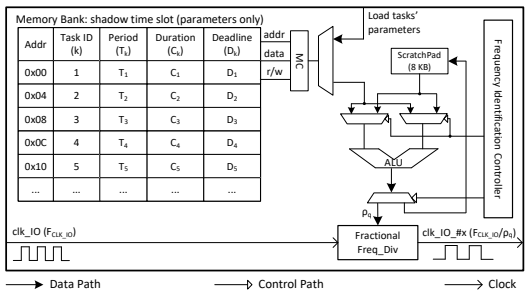


Fig. 8. Top-level micro-architecture of FSU.

I/O-GUARD hypervisor involves additional hardware implementation compared to a conventional virtualization, potentially increasing overall energy consumption. To mitigate this issue, we introduce the energy-efficient features of the *I/O-GUARD* hypervisor in the next sections.

4 *I/O-GUARD*: Energy-Efficient Features

To ensure the *I/O-GUARD*'s energy-efficiency, we first present the dynamic energy model and then introduce the energy management framework used by *I/O-GUARD*.

4.1 Dynamic Energy Model

As identified by the best-known power model, a hardware element simultaneously consumes both *static* and *dynamic* power [12]–[14]. The static power is consumed by leakage current and the dynamic power is generated by run-time activities. The dynamic power ($P(q)$) of an element at time slot q is calculated using Eq. (1):

$$P(q) = \alpha_q \cdot \beta \cdot V_q^2 \cdot f_q \tag{1}$$

In the equation, f_q and V_q are the element's executing frequency and supply voltage. β indicates the element's load capacitance, which is dependent on its physical implementation [13], [15]. For the same element, the β is usually constant at all time slots. α_q stands for the element's switching activities, determined by the element's run-time status. From the system-level perspective, an element's run-time statuses can be simplified into two categories: executing status (EX, for short) and non-executing status (N-EX, for short). We denote the element's run-time status at q as S_q, where $S_q \in \{\text{EX}, \text{N-EX}\}$. While executing at EX, the element usually involves more switch activities compared to executing at N-EX, *i.e.*, $\alpha^{\text{EX}} > \alpha^{\text{N-EX}}$.[3] For an I/O system, $\alpha^{\text{N-EX}}$ is linear increasing with α^{EX} [16], [17]; hence, $\alpha^{\text{N-EX}}$ is presented as $\gamma \cdot \alpha^{\text{EX}}$ ($\gamma < 1$), where γ denotes the percentage of switching activities when the I/O device stays at N-EX compared to at EX. The value of γ is varied with the types of I/O devices and specific use cases [18].

Therefore, the total energy consumption ($E(t)$) of the element during the time period t is presented as:

$$E(t) = E^{\text{EX}}(t) + E^{\text{N-EX}}(t) \tag{2}$$

$E^{\text{EX}}(t)$ and $E^{\text{N-EX}}(t)$ stand for the element's energy consumption at EX and N-EX, calculated by following equations.

$$E^{\text{EX}}(t) = \sum_{\forall q \in t \wedge S_t = \text{EX}} P(q), \tag{3}$$

3. At N-EX, many internal modules of *I/O-GUARD* still work, generating switching activities, *e.g.*, checking transactions from processors.

$$E^{\text{N-EX}}(t) = \sum_{\forall q \in t \wedge S_t = \text{N-EX}} P(q) \cdot \gamma \tag{4}$$

4.2 Overview of Energy Management

In *I/O-GUARD*, we partition the hardware elements into two clock domains, *i.e.*, *compute domain* and *I/O domain*, see Fig. 7.

The *compute domain* has processors, memory, and a NoC, driven by a *high-frequency* clock source (*i.e.*, $clk\_compute$). Existing technologies, *e.g.*, [12]–[15], can be adopted to manage the energy consumption of the elements.

The *I/O domain* contains *I/O-GUARD* hypervisor and I/O devices, driven by a clock source with relatively *lower* frequency (*i.e.*, clk_IO). The frequency of clk_IO is denoted as F_{clk_IO}. Existing energy management IPs are not dedicately designed for the I/O system/virtualization [16], which can neither manage the energy effectively nor ensure the *I/O-GUARD*'s predictability. Hence, we present a *Virtualization Energy Management Unit* (*VEMU*), guaranteeing the energy-efficiency and predictability of the I/O domain simultaneously. The VEMU contains multiple Frequency Scaling Units (FSUs), shown in Fig. 7. Each FSU drives the clock ($clk\_IO\_\#x$) for an I/O device ($\#x$) and its associated virtualization manager and virtualization driver.

In this paper, we set the clock frequency to the high-frequency (*i.e.*, clk_compte) and the low-frequency (*i.e.*, clk_IO) as 100MHZ and 75MHZ, respectively. In practice, the high-frequency could be higher (usually more than 500 MHz for modern micro-controllers). We set the high-frequency as 100 Mhz is because FPGA designs tend towards lower clock frequencies for prototyping.

Frequency scaling. An FSU manages the energy consumption using *frequency scaling*. Specifically, an FSU receives the clk_IO and divides it with a specific ratio. We denote the frequency dividing ratio as ρ_q, where $\rho_q \in [1, \frac{3}{2}, 2, \frac{5}{2}, ..., 10]$. The FSU then exports the divided clock as $clk\_IO\_\#x$ with frequency $\frac{F_{clk\_IO}}{\rho_q}$. As the power consumption at any time slot is linear increase with the executing frequency (see Eq. (1)), an FSU dynamically adjusts the frequency of $clk\_IO\_\#x$ using three phases:

- **Phase** 1 **-** FSU communicates with virtualization manager to obtain the information of buffered I/O tasks.
- **Phase** 2 **-** FSU determines the *maximum* ρ_q that ensures all buffered tasks can be completed before their deadlines. If none of the tasks is required to be executed, FSU sets ρ_q as 10 to enable the lowest running frequency.
- **Phase** 3 **-** FSU increases/decreases the frequency of $clk\_IO\_\#x$ to $\frac{F_{clk\_IO}}{\rho_q}$.

To support these three executing phases, we introduce the design of FSU in Sec. 4.3 and present a *frequency identification algorithm* to find ρ_q in Sec. 5.3.

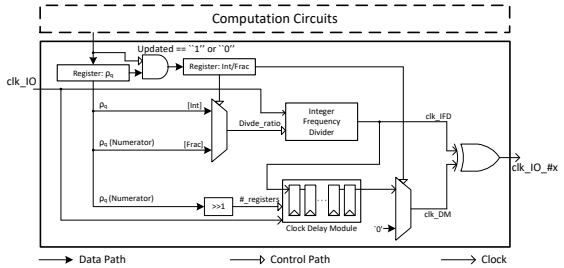


Fig. 9. Micro-architecture of fractional frequency divider.

4.3 Frequency Scaling Unit (FSU) Design

The top-level micro-architecture of FSU is illustrated in Fig. 8, which contains three main elements: a memory unit, computation circuits, and a *Fractional* Frequency Divider (FFD). The memory unit behaves like a *shadow* time slot table (σ), but only storing the parameters of the I/O tasks. At run-time, the computation circuits obtain the parameters of I/O tasks from the memory unit (Phases 1). With the required parameters, the computation circuits calculate ρ_q for the controlled elements using *frequency identification algorithm* (Phase 2); and transmits ρ_q to FFD. The FFD then adjusts clock frequency of the controlled elements, correspondingly (Phase 3).

Below, we present the design details of the computation circuits and the FFD, respectively.

Computation Circuits. The computation circuits contain both data and control paths. The data path has an ALU, a memory controller, and a scratchpad (8 KB). The ALU is used for calculation; the memory controller collects task parameters, and the scratchpad buffers temporary data during calculation. The control path is implemented using a Finite State Machine (FSM), managing the data flow in data path to realize the frequency identification algorithm when the shadow time slot table is updated.

Once the computation is done, the ALU outputs the calculated ρ_q with a notification to the FFD.

Fractional frequency divider (FFD). The FFD is designed mainly based on an integer frequency divider (IFD) and a delay module (see Fig. 9). The IFD is a ready-built IP, which can divide the frequency of an input clock by an integer (*i.e.*, $1, 2, 3, ...$) with fixed 50% duty cycle. The delay module contains a chain of registers, and the registers are triggered at both rising and falling clock edges. The delay module has two responsibilities: (i) cooperating with the IFD to adjust the frequency of $clk\_IO\_\#x$ and (ii) tuning the duty cycle of $clk\_IO\_\#x$ to make it close to 50%.

A frequency update of $clk\_IO\_\#x$ is triggered by the updated ρ_q sent by computation circuits. Once the FFD receives the update, it first checks the type of received ρ_q (*i.e.*, integer or fraction). If the ρ_q is an integer, the delay module are disabled, and the ρ_q is passed to the IFD, allowing the IFD to fully control the clock frequency. If the ρ_q is a fraction, both delay module and IFD is active, and the *numerator* of ρ_q is passed to both IFD and delay module (right-shifting one bit). The numerator determines the divide ratio of the IFD and the delay stages of the delay module. At last, an XOR gate combines the outputs from both IFD and delay module, generating $clk\_IO\_\#x$. Fig. 10 illustrates the generation of $clk\_IO\_\#x$ while $\rho_q = \frac{3}{2}$.

4.4 Discussion: Clock Gating and Voltage Scaling

As well as frequency scaling, clock gating and voltage scaling are both effective mechanisms to improve the hardware's energy-efficiency [18]. Specifically, clock gating entirely blocks the switch activities of the sequential logic, effectively decreasing the α_q in Eq. (1). Voltage decreases the supply voltage of the hardware, *i.e.*, V_q in Eq. (1). As evidenced in [12]–[16], voltage scaling could be more energy-efficient compared to frequency scaling, since the power consumption is a convex increasing function of the element's supply voltage (see Eq. (1)). In future work, we plan to integrate the clock gating and voltage scaling in *I/O-GUARD*, and further investigate the effectiveness of these different mechanisms.

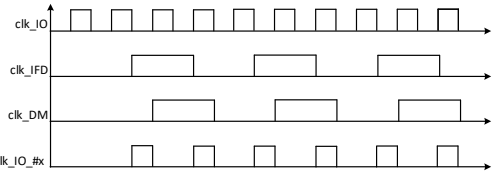


Fig. 10. Example: $\rho_q = \frac{3}{2}$, with duty cycle 33.33%.
(clk_IFD: clock from IFD; clk_DM: clock from delay module.

5 Energy-aware 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 ($1 \leq i \leq n$) 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 exact 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 with unit I/O frequency, *i.e.*, $\rho_q = 1$, and has a deadline at D_k time slots after it is released. That is, when $\rho_q > 1$, each job of τ_k takes up to $\lceil \rho_q C_k \rceil$ time slots to complete (as the frequency and therefore the processing speed is scaled down to $1/\rho_q$ of the unit one). We assume *constrained deadlines*, i.e., $\forall k, D_k \leq T_k$. 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.3. Also, this analysis focuses on each individual I/O device and therefore is similar to a uniprocessor scheduling problem. 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 [19], [20].

5.1 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. Please note that t stands for number of time slots and therefore must be an integer. The value of $\mathsf{sbf}(\sigma, t^*)$ can be obtained for any t^* such that $0 \leq t^* \leq H-1$ 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.,

$$\mathsf{sbf}(\sigma, t) = \mathsf{enum}(t) \text{ for } 0 \leq t \leq H-1. \tag{5}$$

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,

$$\mathsf{sbf}(\sigma, t) = \mathsf{sbf}(\sigma, t \bmod H) + \left\lfloor \frac{t}{H} \right\rfloor \cdot F \text{ for } t \geq H \tag{6}$$

Thus, $\mathsf{sbf}(\sigma, t)$ for all $t \geq 0$ can be derived by (5) and (6).

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

$$\mathsf{dbf}(\Gamma_i, t) = \left\lfloor \frac{t}{\Pi_i} \right\rfloor \cdot \Theta_i. \tag{7}$$

Theorem 1. *Each VM i is guaranteed to receive Θ_i time slots every Π_i time slots, if*

$$\forall t \geq 0, \sum_{i=1}^{n} \mathsf{dbf}(\Gamma_i, t) \leq \mathsf{sbf}(\sigma, t), \tag{8}$$

where $\mathsf{sbf}(\sigma, t)$ *and* $\mathsf{dbf}(\Gamma_i, t)$ *is calculated by* (5), (6), *and* (7).

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*[4] 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 constant such that $c > 0$ (e.g., $c = 0.01$),* (8) *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

$$\exists t^* \geq 0 \text{ such that } \sum_{i=1}^{n} \mathsf{dbf}(\Gamma_i, t^*) > \mathsf{sbf}(\sigma, t^*) \tag{9}$$

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

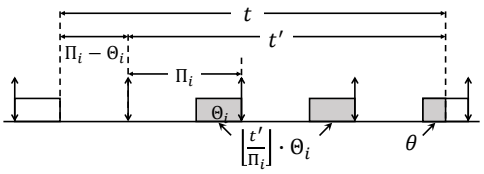


Fig. 11. An illustration of the scenario where (15) is derived.

implies $t^* < \frac{F \cdot \frac{H-1}{H}}{c}$. By (6), we have

$$\mathsf{sbf}(\sigma, t^*) \geq \left\lfloor \frac{t^*}{H} \right\rfloor \cdot F \geq \frac{t^* - (H-1)}{H} \cdot F. \tag{10}$$

On the other hand, by (7), we have

$$\sum_{i=1}^{n} \mathsf{dbf}(\Gamma_i, t^*) \leq t^* \cdot \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i}. \tag{11}$$

Thus, by (10) and (11), (9) implies

$$t^* \cdot \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i} > \frac{t^* - (H-1)}{H} \cdot F \tag{12}$$

$$\Rightarrow t^* < \frac{F \cdot \frac{H-1}{H}}{\frac{F}{H} - \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i}} \tag{13}$$

$$\Rightarrow t^* < \frac{F \cdot \frac{H-1}{H}}{c} \tag{14}$$

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 $\frac{F}{H} = \sum_{i=1}^{n} \frac{\Theta_i}{\Pi_i}$).

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

Once the free time slots have been allocated to VMs as described in Sec.5.1, 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* [20]. 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

$$\mathsf{sbf}(\Gamma_i, t) = \begin{cases} 0 & \text{if } t' < 0 \\ \left\lfloor \frac{t'}{\Pi_i} \right\rfloor \cdot \Theta_i + \theta & \text{if } t' \geq 0 \end{cases} \tag{15}$$

where $t' = t - (\Pi_i - \Theta_i)$ and $\theta = \max\left(t' - \Pi_i \left\lfloor \frac{t'}{\Pi_i} \right\rfloor - (\Pi_i - \Theta_i), 0\right)$. This supply bound function presented in (15) was derived and reasoned in [20]. The minimum-supply scenario that result in (15) is illustrated in Fig. 11

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

$$\mathsf{dbf}(\tau_k, t) = \left(\left\lfloor \frac{t - D_k}{T_k} \right\rfloor + 1 \right) \cdot \lceil \rho_q C_k \rceil. \tag{16}$$

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*

$$\forall t \geq 0, \sum_{\tau_k \in \mathcal{T}_i} \mathsf{dbf}(\tau_k, t) \leq \mathsf{sbf}(\Gamma_i, t), \tag{17}$$

where $\mathsf{sbf}(\Gamma_i, t)$ *and* $\mathsf{dbf}(\tau_k, t)$ *are calculated by* (15) *and* (16)*.*

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 $\frac{\Theta_i}{\Pi_i} - \sum_{\tau_k \in \mathcal{T}_i} \frac{\lceil \rho_q C_k \rceil}{T_k} > c'$ where c' is a certain constant such that $c' > 0$ (e.g., $c' = 0.01$), (17) 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

$$\exists t^* \geq 0 \text{ such that } \sum_{\tau_k \in \mathcal{T}_i} \mathsf{dbf}(\tau_k, t) > \mathsf{sbf}(\Gamma_i, t) \tag{18}$$

implies $t^* < \frac{\max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} + 2\Pi_i - \Theta_i - 1}{c'}$. By (15), we have

$$\begin{aligned} \mathsf{sbf}(\Gamma_i, t^*) &\geq \left\lfloor \frac{t^* - (\Pi_i - \Theta_i)}{\Pi_i} \right\rfloor \cdot \Theta_i && (19) \\ &\geq \frac{t^* - (\Pi_i - \Theta_i) - (\Pi_i - 1)}{\Pi_i} \cdot \Theta_i \\ &\geq t^* \cdot \frac{\Theta_i}{\Pi_i} - (2\Pi_i - \Theta_i - 1). && (20) \end{aligned}$$

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 (16), we have

$$\begin{aligned} \sum_{\tau_k \in \mathcal{T}_i} \mathsf{dbf}(\tau_k, t^*) &\leq \sum_{\tau_k \in \mathcal{T}_i} \frac{t^* + (T_k - D_k)}{T_k} \cdot \lceil \rho_q C_k \rceil \\ &\leq \sum_{\tau_k \in \mathcal{T}_i} \frac{t^* + \max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\}}{T_k} \cdot \lceil \rho_q C_k \rceil \\ &= \sum_{\tau_k \in \mathcal{T}_i} \frac{\lceil \rho_q C_k \rceil}{T_k} \cdot (t^* + \max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\}) \\ &\leq \sum_{\tau_k \in \mathcal{T}_i} \frac{\lceil \rho_q C_k \rceil}{T_k} \cdot t^* + \max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\}. \end{aligned} \tag{21}$$

The last inequality is because $\sum_{\tau_k \in \mathcal{T}_i} \frac{\lceil \rho_q C_k \rceil}{T_k} \leq \frac{\Theta_i}{\Pi_i} \leq 1$ 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 (20) and (21), (18) implies

$$\sum_{\tau_k \in \mathcal{T}_i} \frac{\lceil \rho_q C_k \rceil}{T_k} \cdot t^* + \max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} > t^* \cdot \frac{\Theta_i}{\Pi_i} - (2\Pi_i - \Theta_i - 1)$$

$$\Rightarrow t^* < \frac{\max_{\tau_k \in \mathcal{T}_i}\{T_k - D_k\} + 2\Pi_i - \Theta_i - 1}{\frac{\Theta_i}{\Pi_i} - \sum_{\tau_k \in \mathcal{T}_i} \frac{\lceil \rho_q C_k \rceil}{T_k}}$$

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

The theorem follows. □

5.3 Frequency Identification Algorithm

We are now ready to present our *frequency identification algorithm* that select the largest ρ_q (so that the minimum frequency) such that the system is still schedulable.

In this frequency identification algorithm, we assume that the period of each VM (*i.e.*, Π_i) is pre-selected and given while the budget (*i.e.*, Θ_i) can be adjusted to maximize the schedulability. That is, system designers need select such periods according to the application and system requirements and limitations first, *e.g.*, considering context-switch overheads. Then. for each selected combination of the periods of VMs, our *frequency identification algorithm* is able to identify the minimum needed operation frequency as well as the budgets for each VM. If multiple candidates of the period selection may be considered, the *frequency identification algorithm* can be applied multiple times to each candidate and therefore identify the best periodic selection from the candidates.

For clarity and conciseness, we describe the frequency identification algorithm as follows, so that we do not need to present fairly standard code/pseudo-code structure (*e.g.*, binary search) nor re-present mathematical formulas we just derived in the above subsections. Please note that, the following 1) to 4) are not four subsequent steps; instead, they are four layers.

1) We can determine the largest schedulable ρ_q by a binary search on all its possible candidates (*i.e.*, $\{1, \frac{3}{2}, 2, \frac{5}{2}, ..., 10\}$).
2) For each given ρ_q, we determine the YES/NO for schedulability by Theorem 1, where the time slot table σ is given and VM parameters $\{(\Pi_i, \Theta_i^*)\}$ are determined below.
3) Under the given ρ_q, for each VM i with given Π_i and its associated task set $\mathcal{T}_i$, we can determine the minimum required budget Θ_i^* by a binary search for Θ_i in the range $[0, \Pi_i]$.
4) Under the given ρ_q, i, Π_i, and $\mathcal{T}_i$, for each Θ_i, we can determine the YES/NO for schedulability by Theorem 3.

In terms of the time complexity of our frequency identification algorithm, it can be analyzed as follows.

- The largest schedulable ρ_q is obtained when 1) has been completed. Due to the binary search, 1) consists of at most $\mathcal{O}(\log N_\rho)$ iterations of 2), where N_ρ denote the number of candidates to be selected as ρ_q. — There are polynomial number of iterations of 2).
- Each iteration of 2) completes by applying Theorem 1. According to Theorem 2, it takes pseudo-polynomial time.
- Moreover, for each iteration of 2), we need do one iteration of 3) to determine the $\{\Theta_i^*\}$. 3) consists $\mathcal{O}(n)$ binary

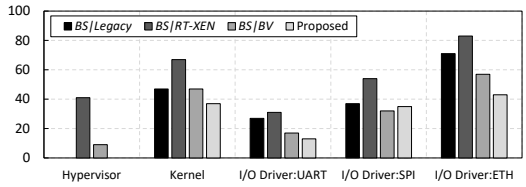


Fig. 12. Run-time software overhead (unit: KB). The software overhead is evaluated via memory footprint, containing BSS, data and text.

searches, where n is the number of VMs, and each binary search consists of at most $\mathcal{O}(\log \Pi_i)$ iterations of 4). In total, each iteration of 3) consists of at most $\mathcal{O}(n \log \Pi_{\max})$ iterations of 4), where $\Pi_{\max} = \max_i \Pi_i$. — 3) consists of a polynomial number of iterations of 4).

- Each iteration of 4) completes by applying Theorem 3. According to Theorem 4, it takes pseudo-polynomial time.

Thus, to sum up, our frequency identification algorithm runs in pseudo-polynomial time.

6 EVALUATION

We now conduct experiments to evaluate *I/O-GUARD*.

Experimental platform. We built two variants of *I/O-GUARD* on a Xilinx VC709 evaluation board , configured the size of the priority queues to 16, and adopted the pass-through design for the response channel. The only difference between the variants is the deployment (with or without) of VEMU (detailed in Sec. 4.2). We use *I/O-GUARD*|*E* to indicate the system with VEMU. In each *I/O-GUARD* variant, the hypervisor was implemented using BlueSpec System Verilog [21] and connected to a 5×5 mesh type open-source NoC [10]. As well to the hypervisor, the NoC also contained 16 MicroBlaze processors [22], 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 [22]. We selected FreeRTOS (v.10.4) as the OS kernel for all VMs, with the modifications introduced in Sec.2.3. Additionally, we introduced three baseline systems (**BS**s) 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 a server-based scheduling scheme [23]. Following the observations given by [24], we configured the hypervisor with a global EDF scheduling policy. We also removed the redundant checking modules in the I/O drivers to enhance the I/O performance. Both patches and I/O enhancement were implemented in the software. ***BS*|*BV*** was a virtualized system built on hardware assistance (*BlueVisor*) introduced in [9], which was reviewed in Sec. 7. For all architectures, we set the clock frequency to the compute domain (*i.e.*, clk_compte) and I/O domain (*i.e.*, clk_IO) as 100MHZ and 75MHZ, respectively.

6.1 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 [25]. Note, *I/O-GUARD*|*E* was not examined, as the VEMU does not involve any software implementation.

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 12. 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 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.

6.2 Hardware Overhead

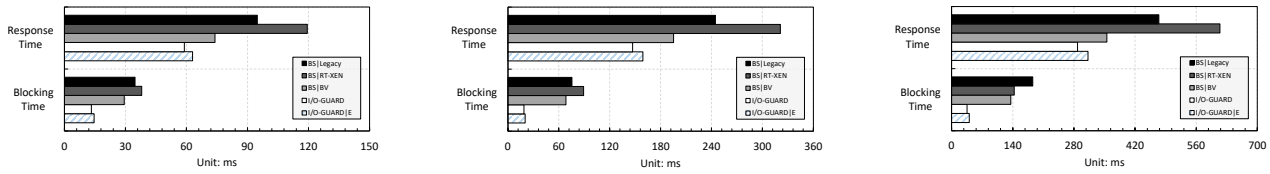
I/O-GUARD requires additional implementation of the hypervisor. Therefore, we evaluate its hardware overhead.

Experimental setup. We first configured the *I/O-GUARD* to support 16 VMs and x I/Os ($x \in [2, 4, 8]$). This means the hypervisor contained x groups of virtualization managers and virtualization drivers, where each virtualization manager contained 16 I/O pools (see Sec. 3.1). We used *I/O-GUARD*(|*E*)-x to denote the *I/O-GUARD*(|*E*) variants, where x indicates the number of I/Os.

We compared the hypervisors with two general-purpose processors (MicroBlaze and RISC-V), two mainstream I/O controllers (SPI, and Ethernet), and another hardware-implemented hypervisor (*i.e.*, BlueVisor used in *BS*|*BV*). The MicroBlaze was full-featured, enabling all the performance related functionalities (e.g., pipeline, data cache). The RSIC-V was implemented based on [26], 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. For fair comparison, we also configured BlueVisor to support 2 I/Os. All the elements were synthesized using Vivado (v2020.2).

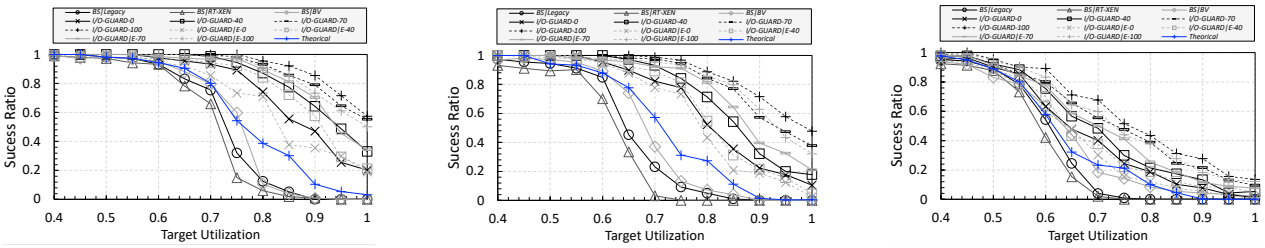
Obs 2. The design of the hypervisor (of *I/O-GUARD*) was resource-efficient, compared to the full-featured processors. Its hardware consumption was higher than commonly used I/O controllers.

As shown in Table 1, *I/O-GUARD*-2 required significantly less hardware than full-featured processors: MicroBlaze (56.6% LUTs, 67.8% registers, 100.0% RAM) and RSIC-V (37.4% LUTs, 18.2% registers, 50.0% RAM). Due to the hardware-implemented virtualization and drivers, *I/O-GUARD*-2 consumed more hardware than the standard I/O controllers. But when compared to *BS*|*BV*, *I/O-GUARD* required the same memory consumption, but less LUTs, registers and DPSs. We also reported that the hardware consumption of the *I/O-GUARD* and *I/O-GUARD*|E is linearly increased when the number of I/O devices increases. This is because the virtualization manager and driver in *I/O-GUARD* hypervisor are independently associated with each I/O device, *i.e.*, when adding a new I/O device, a group of a virtualization manager and a virtualization driver are required to be instantiated (described in Sec. 4).



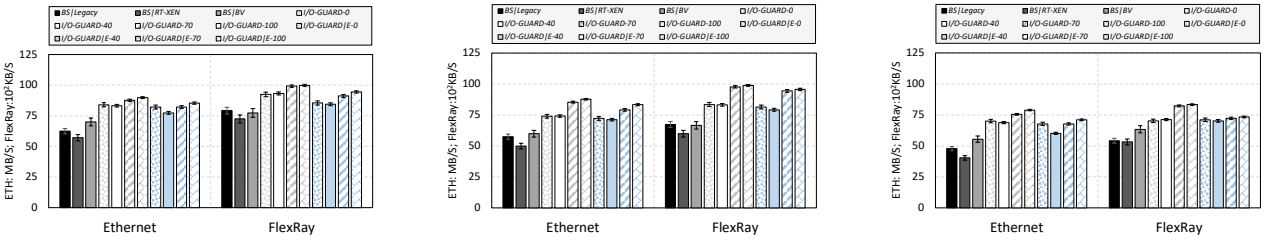
(a) 4 requesters and 2 operators. (b) 8 requesters and 2 operators. (c) 16 requesters and 2 operators.

Fig. 13. Synthetic workloads: average I/O response and blocking time (x-axis: ms).



(a) 4-VM system. (b) 8-VM system. (c) 16-VM system.

Fig. 14. Automotive case study: success ratios of different systems (x-axis: target utilization).



(a) 4-VM system. (b) 8-VM system. (c) 16-VM system.

Fig. 15. Automotive case study: Average I/O throughput (The error bars indicate the experimental variances).

TABLE 1
Hardware Overhead (Implemented on FPGA)

	LUTs	Registers	DSP	RAM (KB)
MicroBlaze	4,908	4,385	6	256
RSIC-V	7,432	16,321	21	512
SPI	632	427	0	0
Ethernet	1,321	793	0	0
BlueVisor	3,236	3,346	0	256
I/O-GUARD-2	2,777	2,974	0	256
I/O-GUARD|*E*-2	3,164	3,273	0	288
I/O-GUARD-4	4,915	5,055	0	512
I/O-GUARD|*E*-4	5,137	5,344	0	576
I/O-GUARD-8	8,422	9,457	0	1024
I/O-GUARD|*E*-8	9,457	10,145	0	1,152

Obs 3. Although bringing a VEMU slightly increased overall hardware overhead, *I/O-GUARD*|*E* still required less LUTs and registers than the other hardware hypervisor.

This observation is given by the comparison between the two *I/O-GUARD* variants and BlueVisor. Introducing VEMU in *I/O-GUARD* hypervisor brought additional 387 (13.9%) LUTs, 299 (10.1%) registers, and 32 KB (11.1%) RAM. Even with the VEMU, *I/O-GUARD* hypervisor still required less LUTs (2.2%) and registers (2.2%) compared to BlueVisor.

6.3 Synthetic Workloads: I/O Response Time and I/O Blocking Time

We examined the *I/O response time* and *blocking time* of the *I/O-GUARD* using *synthetic* workloads.

Experimental setup. We deployed 4/8/16 VMs as I/O requesters and configured 2 Ethernet controllers (1 Gbps, with loop-back mode) as I/O operators. During the experiments, a requester randomly generated 2-4 I/O requests (with 0.5 - 1.5 KB data) to each operator and assigned a unique priority to each I/O request. The operators acknowledged requesters for requests by looping them back. The operator paused until it had no outstanding request and then started to reissue new requests. Synthetic transaction workloads such as this provide traffic patterns close to practical applications and facilitate *behavior* observation. We examined the systems using average *response* and *blocking* time of I/O requests. The response time of an I/O request records the time duration from issue to completion. The blocking time of an I/O request indicates the duration of time it is blocked by requests with lower priority. Experiments were executed 1,000 times.

Obs 4. *I/O-GUARD* had the best real-time performance. This benefit was consistently magnified when the system scaled with deploying more elements in the system.

This observation by given in Fig. 13, which is summarized from two perspectives. First, I/O requests in *I/O-GUARD* always had the shortest response time, indicating that *I/O-GUARD* has the highest I/O throughput. This benefit is given by the optimization of system architecture in *I/O-GUARD* (see Sec. 2.3). Second, the I/O requests in *I/O-GUARD* always suffered the least blocking time. This is because *I/O-GUARD* achieves requests' prioritization and real-time scheduling at the hardware level (see Sec. 3.1), ensuring I/O requests are operated based on their importance and the system executes predictably.

Obs 5. Deploying the energy management module slightly decreased the real-time performance of *I/O-GUARD*|*E*. Such reduction is trivial.

This observation is shown in the comparison between the *I/O-GUARD* and *I/O-GUARD*|*E* of three experimental groups in Fig. 13. When the systems were configured with the same settings, the *I/O-GUARD*|*E* was outperformed by the *I/O-GUARD* in most cases, indicating that introducing VEMU slightly affects *I/O-GUARD*'s real-time performance. Such performance reduction was bounded in a small margin: about $3\% - 5\%$.

6.4 Case Study: Overall Real-time Performance

We use an automotive case study to examine the benefits of the *I/O-GUARD* over conventional virtualized systems.

Systems Configurations. We configured *I/O-GUARD*(|*E*) as *I/O-GUARD*(|*E*)-x (x ∈ [0, 40, 70, 100]), which pre-loaded $x\%$ of I/O tasks into the virtualization manager before run-time. In other words, *I/O-GUARD*(|*E*)-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 [27], e.g., RSA32, etc..
- 20 automotive function tasks, selected from EEMBC benchmark [28], e.g., FFT, speed calculation, etc..
- synthetic workloads, selected from EEMBC benchmark, can be optionally added to control overall utilization.

We employed a hybrid-measurement approach to obtain WCETs for all the task. 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 fetching of the raw data was packed as I/O tasks in *I/O-GUARD* variants. The results were sent back via a FlexRay (10 Mbps), which were formed as sporadic I/O tasks. Each task had a randomly defined period and implicit deadline, with overall system utilization approximately 40%. Since 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*. Note that the random generation of the task parameters may result in the taskset being physically unschedulable.

Experimental Setup We introduced three groups of experimental setups, which activated $4/8/16$ 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 during I/O processing. We also performed the schedulability tests (Sec. 5) on the given tasksets to check the consistency between theoretical analysis and practical evaluation. Based on the experimental results in Sec. 6.3, we configured the *I/O-GUARD* without VEMU to ensure the best real-time performance.

Obs 6. Introducing *I/O-GUARD* improved system-level real-time performance. Such benefit was slightly decreased by the energy management module in *I/O-GUARD*|*E*.

As shown in Fig. 14 and 15, with the same configurations, the *I/O-GUARD* variants always achieved higher success ratios and I/O throughput compared to the baseline systems. However, such improvements were slightly decreased in *I/O-GUARD*|*E*, due to the involvement of VMEU. This observation is aligned to the experiments using synthetic workloads, *i.e.*, Obs. 5. The results also shows that *I/O-GUARD*(|*E*)-100 consistently outperformed other *I/O-GUARD*(|*E*) variants in both success ratios and I/O throughput, with less experimental variance, meaning that pre-loading a higher percentage of I/O tasks into the *I/O-GUARD* before run-time introduces more benefits. While comparing the theoretical results with the experimental results in Fig. 6.4, we reported that the *I/O-GUARD* variants consistently outperformed the theoretical results, which demonstrates the consistency between theoretical analysis and the experimental results.

Obs 7. *I/O-GUARD*(|*E*)-0 had the worst real-time performance in *I/O-GUARD*(|*E*) variants, but still outperformed the baseline systems.

This observation is given by Fig. 14. With the same configurations, *I/O-GUARD*(|*E*)-0 achieved the lowest success ratio in the *I/O-GUARD*(|*E*) variants, but still higher than the baseline systems. In *I/O-GUARD*(|*E*)-0, none I/O task was pre-loaded in *I/O-GUARD* hypervisor, and such improvement is acquired by the novel architecture presented by *I/O-GUARD* (see Sec. 2), simplifying the I/O access paths.

Obs 8. Increasing the number of VMs reduced the success ratio and I/O throughput of the conventional virtualization. *I/O-GUARD* effectively reduced such issues.

This observation is shown by the comparison between the results of three experimental groups in Fig. 14 and 15. 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 60% target utilization in 16-VM *BS*|*RT-XEN* and *BS*|*BV*. Moreover, *BS*|*RT-XEN* and *BS*|*BV* also suffered from an approximate 20% reduction of I/O throughput. This observation mainly results from the additional on-chip interference and resource contention generated by the introduced VMs and tasks (see Sec.1).

In *I/O-GUARD* and *I/O-GUARD*|*E*, 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*(|*E*)-100 maintained a success ratio which was close to 45% (35%) with negligible loss of I/O throughput. For the experiments with 16 VMs, *I/O-GUARD*s still kept outperforming the baseline systems.

6.5 Power Distribution and Energy Efficiency

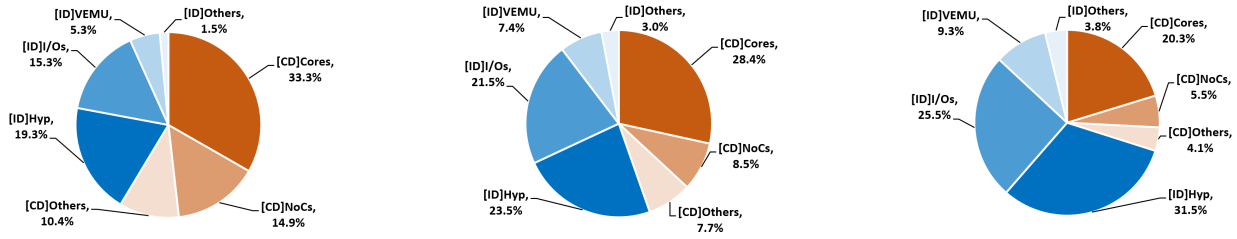
We now evaluate power distribution and energy efficiency of *I/O-GUARD*|*E*.

Experimental Setup. We configured *I/O-GUARD*|*E* to support 4 processors and 2/4/8 I/Os (Ethernet). We first adopted the method described in Sec. 6.2 to synthesize the systems and reported their power distributions. We then executed the case study described in Sec. 6.4 and recorded the clock frequency in the I/O domain. With that, we calculated the *dynamic* energy consumption using the method described in Sec. 4. We executed the experiments 100 times. Note that, we set γ as 0.5 (median number) and normalized the experimental results by *I/O-GUARD*.

Obs 9. With the increasing number of I/Os, the I/O virtualization dominated the entire system's energy consumption.

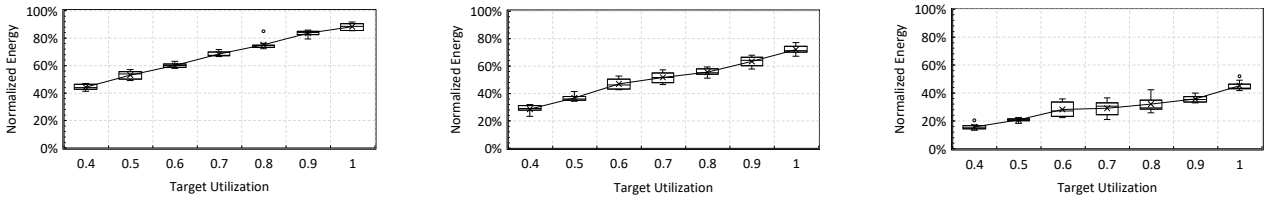
This observation is shown by comparing the three experimental groups in Fig. 16. When the number of I/Os scaled from 2 (Fig. 16(a)) to 8 (Fig. 17(c)), the power distribution of the I/O domain was increased from 39.9% to 70.1%, becoming to dominate the entire system's energy consumption.

Obs 10. Implementing VEMU in *I/O-GUARD* effectively reduced the overall *dynamic* energy consumption.



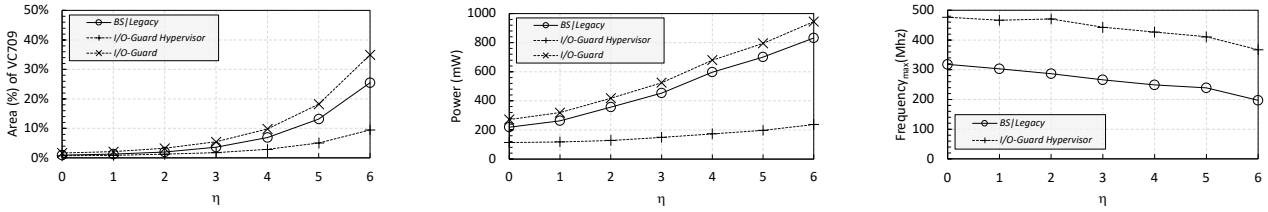
(a) 2-I/O *I/O-GUARD|E*. (b) 4-I/O *I/O-GUARD|E*. (c) 8-I/O *I/O-GUARD|E*.

Fig. 16. Power distribution of *I/O-GUARD|E* with different numbers of I/Os. CD: Compute Domain; ID: I/O Domain.



(a) 2-I/O *I/O-GUARD|E*. (b) 4-I/O *I/O-GUARD|E*. (c) 8-I/O *I/O-GUARD|E*.

Fig. 17. Dynamic energy consumption of *I/O-GUARD|E* with different numbers of I/Os.



(a) Area consumption. (b) Power consumption. (c) Maximum frequency.

Fig. 18. Area, power, and maximum frequency v.s. η (*I/O-GUARD*: entire *I/O-GUARD* system; *I/O-GUARD* hypervisor: hypervisor in *I/O-GUARD*).

This observation is given by Fig. 17. As shown in Fig. 17(a), 17(b), and 17(c), *I/O-GUARD|E* saved about 56%, 73% and 70% dynamic energy consumption in the best case. However, such benefits were slightly reduced with the increase of I/O utilization, since the VEMU must provide a relatively high frequency when the *I/O-GUARD* hypervisor and I/O devices were busy.

6.6 Scalability

We acknowledge that the scalability impacts the feasibility of the proposed design, the scalability of *I/O-GUARD* is lastly examined by a varying number of VMs.

Experimental setup. The same method described in Sec. 6.2 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 11. The *I/O-GUARD*'s area consumption was linearly scaled by $log_2\eta$ (the number of VMs), and the power consumption was linearly scaled by η. Compared to the legacy system, *I/O-GUARD* increased such consumption, slightly.

As shown in Figure 18(a), when the system scaled with $log_2\eta$, the area consumption of both *BS|Legacy* and *I/O-GUARD* linearly 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%. As described in Sec. 4.1, power consumption is usually determined by four factors: voltage, clock frequency, toggle rate and design area [29]. 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. In Fig. 18(b), we observed linearly increased power consumption in these systems when η increases.

Obs 12. When the system scaled with η, brining the hypervisor (in *I/O-GUARD*) did not affect maximum performance.

As shown in Figure 18(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.

Limitations of scalability. Although the evaluation demonstrates the hardware scalability of *I/O-GUARD*, deploying a hardware hypervisor in *I/O-GUARD* still increased the hardware overhead compared to the software-based solutions, *i.e.*, *BS|Legacy* and *BS|RT-XEN*. As shown in Fig. 18(a) and 18(b), with the same configurations, *I/O-GUARD* always consumed more area and power compared to *BS|Legacy*. The software-based solutions usually suffer fewer limitations in area, power and frequency, because the system only needs to add more processors when the number of VMs increases.

7 Related Work

A range of research efforts and industrial products aiming to achieve real-time I/O virtualization in multi-core and many-core systems have been proposed, with a different focus and design philosophies.

Software-level Optimization. Research frameworks in the software level mostly focus on VMM/hypervisor. For instance, RT-XEN [23] and Quest-V [30] have been developed which improve I/O predictability by integrating a predictable scheduler into the hypervisor. Each of the projects presents results to show they improve predictability in the VMM layer, and such innovations are key to the overall goal

of predictable I/O virtualization. However, without consideration of other system layers (research challenge $C.1$), real-time virtualization can not be guaranteed from the system-level perspective. Different from the previously reviewed work, Kim *et al.* [31] proposed a methodology to enhance I/O predictability from the OS level. This work proposed optimized memory management (*i.e.*, isolating dynamic-memory allocations) and improved IPC-related mechanisms (*i.e.*, selective LLC bypass, and concurrency elimination) to improve the system performance and predictability. This work makes numerous improvements but has the same drawbacks when the system as a whole is considered.

Hardware-level Optimization. Work focusing on the hardware level has mostly contributed to the predictability of on-chip communication. For instance, Burns *et al.* [32] and Plumbridge *et al.* [10] adopted different scheduling algorithms to optimize predictable communication flow in many-core systems, such as via an on-chip network. This work, and others like it, assist the system designer to develop predictable traffic flows, although they focus entirely on the communications network making it challenging to cast virtualization-related overheads into the context of real-time virtualization. With consideration of both virtualization and hardware implementation, Single Root I/O virtualization [33] proposed a set of hardware enhancements for PCIe devices. Rather than relying on the VMM to intervene on I/O instructions, it moved the intervention for performant data movement to the I/O device itself, for tasks such as packet classification and address translation. However, these research efforts did not aim to provide system-level predictability, although it is a contributing factor.

System Structure Optimization. Considering the entire system, Intel's VT-D and AMD's IOMMU optimized the access paths in the I/O virtualization, providing direct communication channels between the VMs and the underlying hardware. However, these technologies have not been developed for real-time application scenarios. Based on the concept of "VMM-bypass virtualization", Jiang *et al.* [9] proposed *BlueVisor*, a dedicated coprocessor, handling I/O virtualization at the 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 , leading the timing-bound of the I/O behaviors to become very pessimistic. (*i.e.*, research challenge $C.2$). Distinct from VMM-bypass virtualization, Siemens has presented a static partitioning virtualization architecture, named Jailhouse [34]. It statically allocates all system resources, including I/Os, at initialization time by exclusively assigning each to a single partition. Jailhouse replaces run-time memory allocation and physical-to-virtual CPU assignment with a 1:1 mapping, which effectively reduces system overhead and ensures that system performance is close to the native system. Such physical separation provides strong isolation between different VMs. However, this separation is contrary to a fundamental concept of virtualization. Since the hardware can only be accessed by a specific partition, it is not really *shared* between VMs.

Timing Analysis for I/O Virtualization. In real-time systems, there has been existing work modeling and providing the timing guarantee for I/O virtualization. In the context of Quest-V, Danish et al. [35] proposed a Sporadic Servers (SS) and Priority Inheritance Bandwidth Preserving Server (PIBS) to handle I/O operations. Following this work, Missimer et al. [30] further presented a theoretical model and schedulability analysis for the SSs and PIBSs, ensuring the I/Os' predictability in the context of Quest-V. However, the hardware-assisted I/O virtualization was not considered. Schwaricke *et al.* [36] presented a "broker-based" real-time communication framework for the VMs, and provided guidelines to system designers on the dimensioning of the system regulation to achieve maximum bandwidth while preserving the I/O flow schedulability. Same as [35] and [30], this work only focused on the software-based I/O virtualization. As a summation of real-time I/O virtualization, Casini *et al.* [37] grouped I/O virtualization into three categories: pass-through virtualization, para-virtualization with I/O VMs, and para-virtualization I/O VMs and shared buffers. With that, this work presented the theoretical model and schedulability analysis for each category, providing the timing guarantees for I/O virtualization. However, same with the other work, hardware-assisted was not discussed.

8 Conclusion

This paper proposes a system framework for multi-/many-core NoC-based 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. *I/O-GUARD* contains a dedicated energy management unit to adjust the energy consumption of the I/O virtualization using frequency scaling. Associated with that, a frequency identification algorithm is proposed to find the appropriate clock frequency at runtime. A theoretical model and schedulability analysis are presented for *I/O-GUARD*, which demonstrates 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. Also, the *I/O-GUARD* design is energy efficient.

References

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

[2] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems," in *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 2011.

[3] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange, "Run-time control to increase task parallelism in mixed-critical systems," in *Proc. ECRTS*. IEEE, 2014.

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

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

[6] Z. Jiang *et al.*, "Mcs-iov: Real-time i/o virtualization for mixed-criticality systems," in *Proc. RTSS*.

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

[8] 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.

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

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

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

[12] R. Nathuji, K. Schwan, A. Somani, and Y. Joshi, "Vpm tokens: virtual machine-aware power budgeting in datacenters," *Cluster computing*, vol. 12, no. 2, pp. 189–203, 2009.

[13] J. Doweck *et al.*, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, 2017.
[14] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," *ACM SIGPLAN Notices*, 2016.
[15] P. S. Bhojwani, J. D. Lee, and R. N. Mahapatra, "Sapp: Scalable and adaptable peak power management in nocs," in *Proceedings of ISLPED*, 2007.
[16] J. Haj-Yahya, M. Alser, J. Kim, A. G. Yağlıkçı, N. Vijaykumar, E. Rotem, and O. Mutlu, "Sysscale: Exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 227–240.
[17] S. Safari, S. Hessabi, and G. Ershadi, "Less-mics: A low energy standby-sparing scheme for mixed-criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4601–4610, 2020.
[18] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.
[19] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *RTSS*, 1990.
[20] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symposium, 2003*, 2003.
[21] "Bluespec System Verilog," https://bluespec.com.
[22] Xilinx, "Microblaze," https://www.xilinx.com/products/microblaze.
[23] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *2011 the Ninth ACM International Conference on Embedded Software*. IEEE, pp. 39–48.
[24] S. Xi, M. Xu, C. Lu, L. T. Phan, C. Gill, O. Sokolsky, and I. Lee, "Real-time multi-core virtual machine scheduling in xen," in *2014 International Conference on Embedded Software*. IEEE, 2014, pp. 1–10.
[25] FreeRTOS, "FreeRTOS website," http://www.freertos.org/.
[26] S. Mashimo *et al.*, "An open source fpga-optimized out-of-order RISC-V soft processor," in *ICFPT*, 2019.
[27] R. Electronics, "Renesas: Automotive Use Cases," https://www.renesas.com/solutions/automotive.html.
[28] EEMBC, "EEMBC benchmark," https://www.eembc.org/autobench/.
[29] A. Bellaouar and M. Elmasry, *Low-power digital VLSI design: circuits and systems*. Springer Science & Business Media, 2012.
[30] E. Missimer, K. Missimer, and R. West, "Mixed-criticality scheduling with i/o," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016, pp. 120–130.
[31] 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.
[32] A. Burns, L. Indrusiak, N. Smirnov, and J. Harrison, "A novel flow control mechanism to avoid multi-point progressive blocking in hard real-time priority-preemptive nocs," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 137–147.
[33] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, 2012.
[34] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no vm exits!(almost)," *arXiv preprint arXiv:1705.06932*, 2017.
[35] 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*. IEEE, 2011, pp. 169–179.
[36] G. Schwäricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A real-time virtio-based framework for predictable inter-vm communication," in *Proc. RTSS*, 2021.
[37] D. Casini, A. Biondi, G. Cicero, and G. Buttazzo, "Latency analysis of i/o virtualization techniques in hypervisor-based real-time systems," in *Proc. RTAS*. IEEE, 2021.

Zhe Jiang received his Ph.D. from University of York (2019). He is currently working as the system design engineer of Central Engineering Department in ARM Ltd and visit research associate in University of York. He is research interests include safety-critical system, system architecture, and system micro-architecture. He can be reached at: zhejiang.uk@gmail.com

Kecheng Yang received the BE degree in computer science and technology from Hunan University in 2013, and the MS and PhD degrees from the University of North Carolina at Chapel Hill in 2015 and 2018, respectively. He is an assistant professor in the Department of Computer Science at Texas State University. His research interests include real-time systems and scheduling algorithms. He received an Outstanding Paper Award and the Best Student Paper Award at the 40th IEEE RTSS, and an Oustanding Paper Award at the 26th RTNS.

Yunfeng Ma, engineering professor, has his BSc. degree in Electronic and Information Engineering awarded by Huaqiao University and a MSc. degree in Digital Systems Engineering and Ph.D in Computer Science awarded by the University of York. His research interests include cyber-physical critical systems and unmanned system swarm.

Nathan Fisher received the Ph.D. from the University of North Carolina at Chapel Hill in 2007, and M.S. degree from Columbia University in 2002, and the B.S. degree from the University of Minnesota in 1999, all in computer science. He is an Associate Professor with the Department of Computer Science, Wayne State University. His research interests include real-time and embedded computer systems and approximation algorithms. He was the recipient of the NSF CAREER Award in 2010.

Neil C. Audsley is currently Deputy Dean of the School of Mathematics, Computer Science and Engineering at City, University of London. His research interests include high performance real-time systems; real-time computer and memory architectures; real-time operating systems and their acceleration on FPGAs; timing analysis.

Zheng Dong received the BS degree from Wuhan University, China, in 2007, the MS degree from University of Science and Technology of China, in 2011, and the PhD degree from the University of Texas at Dallas, USA, in 2019. He is an assistant professor with the Department of Computer Science, Wayne State University, Detroit, Michigan. His research interests are in real-time embedded computer systems and connected autonomous driving systems. His current research focus is on multiprocessor scheduling theory and hardware-software co-design for real-time applications. He received the Outstanding Paper Award at the 38th IEEE RTSS. He is a member of the IEEE Computer Society.