IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 4 November 2024, accepted 3 December 2024, date of publication 9 December 2024, date of current version 17 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3512777

RESEARCH ARTICLE

BiRtIO: VirtIO for Real-Time Network Interface Sharing on the Bao Hypervisor

JOÃO PEIXOTO, JOSÉ MARTINS, DAVID CERDEIRA, AND SANDRO PINTO
Centro ALGORITMI, LASI, Universidade do Minho, 4800-058 Guimarães, Portugal

Corresponding author: João Peixoto (joaopeixotooficial@gmail.com)

This work was supported in part by national funds through ALGORITMI Research Centre/LASI Universidade do Minho, in part by Fundação para a Ciência e a Tecnologia (FCT) under Project UIDB/00319/2020, and in part by European Union's Horizon Europe Research and Innovation Program under Grant 101070537, project Cross-platform Open Security Stack for Connected Devices (CROSSCON). The work of José Martins and David Cerdeira was supported by FCT under Grant SFRH/BD/146231/2019 and grant SFRH/BD/138660/2018.

ABSTRACT The push to reduce size, weight, power, and cost (SWaP-C) has led to the integration of multiple subsystems with different criticality levels onto a single hardware platform. Virtualization has played a crucial role in merging real-time capabilities with complex environments, facilitating the creation of Mixed-Criticality Systems (MCS). However, consolidating various workloads on a single platform often requires shared peripheral access, such as Network Interface Controllers (NICs), among guest Virtual Machines (VMs). VirtIO provides a standardized interface for device para-virtualization, enabling us to deploy device drivers and sharing logic within dedicated VMs. Despite its advantages, VirtIO can introduce latency and reduced throughput due to its reliance on the hypervisor or other VMs for network access, which may exacerbate resource contention and affect time isolation between VMs. Time-Sensitive Networking (TSN) techniques can mitigate these issues by ensuring reliable, low-latency, and deterministic network communication, although its implementation often depends on proprietary Linux-based solutions. In this paper, we present BiRtIO, an infrastructure designed to support a virtual real-time network for VMs on the Bao hypervisor without compromising safety and security. It leverages the VirtIO standard and incorporates TSN mechanisms available in mainline Linux, such as bandwidth limitation, to enhance performance and predictability in a saturated virtualized network. Experiments conducted on the Xilinx ZCU102 high-performance multicore platform demonstrate that our solution achieves performance and predictability levels suitable for real-time workloads on MCS with limited Ethernet ports.

INDEX TERMS Mixed-criticality systems, real-time, security, time-sensitive networks, VirtIO, virtualization.

I. INTRODUCTION

In recent years, the adoption of embedded and cyber-physical systems has increased exponentially [1], [2], becoming ubiquitous across a wide range of applications, i.e., from single-purpose devices, such as industrial control systems and medical devices, to more complex and multi-functional systems, such as driver assistance and in-vehicle infotainment systems. These systems often have strict timing requirements, particularly for safety-critical tasks [3], [4]. As they become increasingly interconnected, their communication infrastructure must also adhere to these stringent time constraints, requiring low latency and predictable network communication. Real-time networking typically relies on proprietary, Linux-based solutions that necessitate kernel modifications or even bypass the kernel entirely [5], [6]. To standardize real-time communication, the industry has developed TSNs, which currently feature initial support in mainline Linux for TSN standards [5].

Concurrently, there is a growing trend to consolidate multiple separate physical subsystems, each with its own processing unit, into a single, powerful system [7], [8], [9]. This unified system is then partitioned into multiple virtual subsystems through virtualization [10], [11], [12],

The associate editor coordinating the review of this manuscript and approving it for publication was Chuan Heng Foh.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

thereby facilitating the integration of real-time functionalities with more complex environments, creating a MCS that meets established standards like ISO26262 [13]. In this configuration, a hypervisor divides the system into distinct subsystems, each hosted within a dedicated VM, while tightly controlling the resources (e.g., memory, devices, interrupts) accessible to each VM. Existing open-source hypervisors like KVM [14] and Xen [15] have been adapted for embedded systems [16], [17], however, they often fall short of the strict real-time requirements [7]. These hypervisors frequently depend on external libraries and privileged Operating Systems (OSes) to manage their guests, thereby increasing the Trusted Computing Base (TCB) and the potential attack surface [18]. Notwithstanding, the size and monolithic architecture of these OSes contribute to heightened costs and efforts associated with safety certifications [7].

Over the years, Static Partitioning Hypervisor (SPH) architectures have been implemented in MCS to ensure robust isolation and meet real-time requirements [7]. This architecture leverages hardware-assisted virtualization technology to implement a minimal software layer that statically partitions all platform resources among the guests. By exclusively allocating all hardware resources, including Central Process Units (CPUs) and peripheral devices, to each guest, these hypervisors guarantee a high level of determinism and microarchitectural isolation [19]. SPHs, such as Bao [7], [20] and Jailhouse [21], are designed from the ground up to meet these requirements, isolating applications of different criticalities both temporally and spatially [7], [22]. However, when hardware resources are limited, multiple VMs may need access to the same peripheral, such as the NIC. For example, in edge computing environments where computational resources are often constrained, it is common to deploy multiple VMs on a single physical platform, all sharing the same NIC [23]. In virtualization-based systems, one approach to manage this is to provide an interface for VMs to request device operations from a domain that exclusively controls the device, such as the hypervisor or a dedicated VM.

VirtIO [24] is a well-established technology that facilitates device sharing in virtualized environments by providing standardized interfaces for device para-virtualization, splitting a typical device driver into two components: a frontend driver within the guest Operating System (OS) and a backend device that manages actual device access within the hypervisor or dedicated VM. Many automotive-grade hypervisors have integrated VirtIO-based devices.

However, VirtIO presents certain limitations, especially for SPHs. Firstly, VirtIO backends are often tightly integrated with the Virtual Machine Monitor (VMM), as seen in implementations like QEMU [25], lkvm [14], and ACRN [26]. Secondly, the typical design of VirtIO backends necessitates unrestricted access to guest VMs' private memory, which conflicts with SPH principles of memory isolation. Finally, mission-critical TSN applications require deterministic performance. Adding extra logic to mediate network access can result in increased latency and reduced throughput, while sharing physical network resources may cause resource contention and adversely affect time isolation between VMs.

This paper presents BiRtIO, an infrastructure designed to support a virtual real-time network for VMs hosted by the Bao hypervisor. It leverages the VirtIO standard and incorporates TSN techniques, such as bandwidth limitation, available in the mainline Linux OS on the backend VMs. To enhance safety and security, we designed the system to support multiple unprivileged backend VMs, distributing VirtIO backend services and sharing logic across several entities. This approach minimizes potential attack surfaces and reduces the likelihood of system-wide failures by isolating critical functions and limiting the impact of any single VM compromise. We support a range of VirtIO backends, including hypervisor-agnostic solutions like Rust-VMM [27] and well-established options like QEMU backends. To evaluate our virtual network, we conducted tests using both QEMU and Rust-VMM VirtIO-Net devices for network sharing across guests.

In summary, we make the following contributions:

1) We analyze the limitations of the VirtIO interface and the challenges of sharing the same NIC among guests on type-I hypervisors;
2) We present and implement the infrastructure that enables network device sharing on Bao using VirtIO without compromising safety and security;
3) We explore the application of TSN techniques, such as bandwidth limitation, available in the mainline Linux to enhance performance and predictability in VirtIO-based network sharing;
4) We evaluate our solution against the passthrough scenario and KVM hypervisor and analyze its behavior in a saturated virtualized network environment.

II. BACKGROUND

A. BAO HYPERVISOR

Bao [7], [20] is a multi-core embedded hypervisor designed for mixed- criticality systems, where safety-critical workloads run alongside feature-rich OSs. Its main goals are to consolidate workloads while ensuring strong isolation and real-time guarantees, minimizing attack surfaces, and supporting certification. Bao has a small TCB of about 8.4 KSLoC for Armv8, and is a standalone implementation with no external dependencies. Bao also supports cache partitioning via page coloring, improving predictability, reducing cache contention, and mitigating cache-based timing side-channel attacks. Bao follows principles of minimalism and least privilege, enhancing security and making safety and security certifications easier. By assigning each hardware resource, such as CPUs, exclusively to a single VM, Bao ensures high determinism and microarchitectural isolation. Peripherals are also assigned to individual VMs by allowing direct access to Memory-Mapped I/O (MMIO) registers and interrupts.

B. VirtIO IN A NUTSHELL

VirtIO [24] is an open standard that defines a protocol for communication using a para-virtualization approach. It divides a typical device driver into two components: a frontend driver, implemented in the guest OS, and a backend device, typically implemented in the hypervisor or another VM. The frontend driver issues Input/Output (I/O) requests, while the backend device receives them and performs the corresponding I/O operations. The communication channel to transmit I/O information between the frontend and backend entities is implemented through one or more shared queues called *virtqueues*. Due to the benefits of standard device abstractions, functionality, and performance, VirtIO was implemented as the transport abstraction layer across various communication mechanisms [11], [28], [29].

1) VirtIO DEVICE

A VirtIO device consists of several components: (i) device status field, (ii) feature bits, (iii) device configuration space, (iv) *virtqueues*, and (v) notifications. Especially important for our work are *virtqueues* and notifications. *Virtqueues* are shared buffers in guest physical memory that the backend accesses to handle I/O requests and results. Notifications, similar to interrupts, signal events to the driver or device. There are three types: (i) available buffer notifications, (ii) used buffer notifications, and (iii) configuration change notifications. Used buffer and configuration change notifications, sent from device to guest, indicate a newly used buffer or a configuration change. Available buffer notifications, sent from guest to device, are layered on the transport mechanism.

2) VirtIO CONTROL AND DATA PLANES

VirtIO is conceptually divided into two planes: the control plane and the data plane. The control plane handles the driver-driven device initialization and transport register operations. The data plane manages the actual data transfer, using *virtqueues* and notifications. To improve efficiency, protocols like vhost [30] and vhost-user [31] have been developed, offloading data handling from the VMM / hypervisor to the kernel or a standalone user-space application, respectively.

3) VirtIO OVER MMIO

VirtIO supports various transport and device discovery modes, including Peripheral Component Interconnect (PCI) and MMIO. In this work, we focus on the MMIO transport protocol due to its simplicity and ease of integration. The VirtIO specification outlines that all interactions between the frontend driver and the backend device over the MMIO transport are conducted through MMIO read/write operations. Conversely, all interactions between the backend device and the frontend driver occur through interrupts.

C. OVERVIEW OF IEEE TSN STANDARDS

TSN is a set of standards developed by the IEEE Time-Sensitive Networking Task Group [32] that enables networked applications to interact deterministically using standard Ethernet. Applications encompass integrated networks supporting real-time audio and video streaming alongside real-time control streams, commonly used in automotive systems and industrial control environments. Some of these standards have already been thoroughly tested and implemented, while others are still in the early stages of development. Notable standards include the IEEE 802.1Qav [33]. This standard outlines methods to improve Quality of Service (QoS) in Ethernet networks, including forwarding and queuing enhancements for time-sensitive streams. It also defines the Credit-Based Shaping (CBS) algorithm, which regulates Ethernet frame transmission to smooth traffic, reduce bursts, and address starvation issues, improving fairness between flows.

Depending on the application, these standards can be used separately or in various combinations [34]. Still today, TSN networks are often developed with proprietary or special solution-based Linux approaches [35]. However, basic functionalities like time synchronization and traffic scheduling algorithms can already be implemented using mainline Linux [5].

TABLE 1. Gap analysis of VirtIO implementations for KVM, Xen, and ACRN hypervisors. The Bao column represents the current work. ✓: hypervisor implements the feature, –: hypervisor does not implement, N/A: not applicable.

Metric	KVM	Xen	ACRN	Bao
Hypervisor-Agnostic VirtIO Backends	✓	✓	–	✓
Multiple VirtIO Backend VMs	N/A	–	–	✓
Private Guest Memory	–	–	–	✓

III. RELATED WORK

A. VirtIO ON type-I HYPERVISORS

This subsection provides an overview and comparison of VirtIO implementations of KVM, Xen, and ACRN with our solution. KVM is a widely used type-I hypervisor that has significantly contributed to the adoption of the VirtIO standard [24]. Xen and ACRN are two of the most well-known open-source type-I hypervisors supporting VirtIO, both often deployed in mixed-criticality systems. Notably absent from our comparison is Jailhouse, as support for VirtIO over Inter-VM Shared Memory (IVSHMEM) is still a work in progress.

Despite similarities, each hypervisor's distinct requirements and target applications lead to unique VirtIO implementations, with specific characteristics and trade-offs. Table 1 summarizes our analysis.

1) HYPERVISOR-AGNOSTIC VirtIO BACKENDS

There is increasing demand for flexible and standardized hypervisor-agnostic I/O virtualization solutions using the

VirtIO interface, especially in the automotive and embedded sectors. This promotes reusing VirtIO backends and focusing on the development on new VirtIO devices. In this vein, the Rust-VMM and ORKO [36] projects were established to provide foundational components, including VirtIO backends, for different hypervisors. While KVM and Xen support VirtIO backends provided by Rust-VMM, ACRN still relies on proprietary backends.

2) MULTIPLE VirtIO BACKEND VMs

Separating backends into multiple VMs has the advantage of distributing the total risk exposure across multiple entities, thus enhancing the overall system's security. However, VirtIO backends are often integrated into the VMM rather than being deployed in separated VMs. KVM offloads device emulation, including VirtIO backends, to the VMM (e.g., QEMU and Firecracker [37]). Xen requires all VirtIO backends to be placed within a single privileged domain (Dom0). Although Xen can be configured without the privileged domain (Dom0-less), support for VirtIO devices with this configuration is still under development. Similarly, ACRN centralizes all peripheral sharing services within a single and privileged VM (*Service VM*).

3) PRIVATE GUEST MEMORY

In most cases, a VMM has unrestricted access to a guest VM memory. KVM VMMs, such as Qemu and Firecracker, as well as ACRN's *Service VM* feature total access to guest VMs's memory. For Xen, limiting backend VM access to a guest VM has been achieved through grant tables [38]. The use of grant tables enhances isolation by leveraging Xen's existing grant table mechanism, reducing (but not eliminating) backend privileges over the guest. Because grant tables operate with page granularity, when used for VirtIO I/O buffers, they may provide access to security-critical data in cases where I/O buffers are not page-aligned. This method also suffers from reduced performance due to the frequent mapping and unmapping of granted frames and is specific to the Xen environment, potentially limiting its portability. There is an alternative, not yet fully mature, called Argo [39], where the hypervisor manages VirtIO data transfers between the guest and backend VMs. While it maintains the privacy of each guest VM's memory, this solution contradicts the SPH principles of minimality and simplicity and is susceptible to the same memory copy issues as bounce buffers.

B. TSN IN A VIRTUALIZED ENVIRONMENT

The feasibility of TSN virtualization was examined by Leonardi et al. [40], which investigated three approaches for enabling TSN communication on heterogeneous platforms managed by a hypervisor. However, these approaches relied on proprietary implementations and were evaluated based solely on simulated architectural principles. Ferraro et al. [41] proposed a TSN architecture using the Jailhouse hypervisor, focusing on strong isolation and virtual TSN access. However, they did not use the VirtIO interface. Instead, they relied on custom-implemented inter-VM network drivers built on top of IVSHMEM.

Garbugli et al. [42] introduced a novel TSN protocol for virtualized environments, employing precise clock synchronization and Time-Aware Shaping (TAS) [43] to achieve deterministic and ultra-low latency communication between VMs managed by KVM. The study utilized VirtIO and Open vSwitch (OVS) [44] with both kernel-based and kernel-bypassing DPDK datapaths. Their evaluation focuses only on end-to-end latency and jitter measurements between two KVM VMs using the clock synchronization technique and the TAS, not addressing the impact of traffic bandwidth limitation in a saturated network scenario.

IV. BiRtIO OVERVIEW

A. MOTIVATION

Device sharing for type-I hypervisors, particularly through the VirtIO standard, has been a significant focus in academia, aiming to enable device-sharing capabilities among guest VMs [15], [24], [26]. Despite VirtIO providing a standard interface compatible with various hypervisor solutions, its inherent limitations have hampered its widespread adoption. Below, we outline the key reasons that we believe have contributed to the limited adoption of VirtIO for device-sharing in type-I hypervisors, particularly those focused on safety, security, and real-time guarantees.

1) UNRESTRICTED ACCESS CHALLENGES IN VirtIO BACKENDS

The VirtIO operates under the assumption that backends have unrestricted access to the memory allocated for frontends, relying on the premise that the memory pages of both entities are physically located within the same memory region. This requirement stems from the allocation of data-containing buffers by guests, which can be distributed across various segments within the guests' private memory space. This poses a challenge when the backend and frontend are running on separate VMs, as many type-I hypervisors, like Bao, enforce strict memory isolation and do not permit one VM to access the memory of another.

2) CENTRALIZED IMPLEMENTATION OF VirtIO BACKENDS

Typically, the VirtIO backend device is implemented in the VMM. However, providing these services within the VMM violates two principles associated with type-I hypervisors like Bao: (i) the principle of minimality and simplicity, and (ii) the Principle of Least Privilege (PoLP). The principle of minimality and simplicity dictates that only strictly necessary services should be implemented in the hypervisor, thereby reducing the TCB and minimizing the size of the code, which in turn reduces the attack surface. The PoLP states that any software capable of being developed and implemented at an unprivileged level should be prioritized. Flaws or vulnerabilities in services implemented at a higher privilege

level can compromise the system to a greater extent [45]. Moreover, consolidating multiple device drivers within one domain increases the system's susceptibility to potential failures or security breaches, presenting a heightened risk compared to distributing these components across multiple entities.

3) NO READY-TO-USE VirtIO BACKENDS

Various implementations of VirtIO backend devices exist, incorporated into distinct hypervisor projects like QEMU, lkvm, and ACRN. Each of these implementations offers varying levels of maturity, performance, and security, tailored to the specific needs and design philosophies of their respective hypervisors. Notwithstanding, integrating these backends into another hypervisor ecosystem demands additional effort.

4) DETERMINISTIC BEHAVIOR CHALLENGES WHILE SHARING THE SAME NIC

Although VirtIO facilitates device sharing among multiple VMs, mission-critical applications, particularly those dependent on TSN, demand precise and deterministic performance. Introducing an intermediary VM for network access introduces additional processing overhead, which increases latency and decreases throughput. Furthermore, the concurrent use of a single physical network resource by multiple VMs can lead to resource contention, resulting in non-deterministic behavior and weakened temporal isolation between VMs.

B. GOALS

The main goals of BiRtIO are:

- **Safety and Security**. Ensure that VirtIO backends operate within unprivileged VMs;
- **Real-Time and Determinism**. Achieve real-time guarantees in a saturated virtualized network;
- **Flexibility and Agnosticism**. Supporting a diverse array of VirtIO backends, from well-established and rigorously tested options like QEMU, to hypervisor-agnostic applications such as the Rust-VMM project.

C. ARCHITECTURE

The BiRtIO architecture, illustrated in Figure 1, establishes unprivileged VMs to host the VirtIO backends. We modify Bao hypervisor to add a mechanism capable of forwarding VirtIO requests and notifications between the frontend VM, where the frontend driver operates, and the associated backend VM, where the backend device executes the I/O operations. We name our I/O request forwarding system *Remote I/O*. This system is explained in detail in Subsection V-A1.

For security reasons, to prevent the backend VMs from accessing the private memory of the guest VMs, it is crucial to ensure that the I/O buffers allocated by the VirtIO driver (i.e., the frontend driver) are confined to a specific memory region that is accessible to the VirtIO backend (i.e., the backend device). To achieve this, we first establish a shared

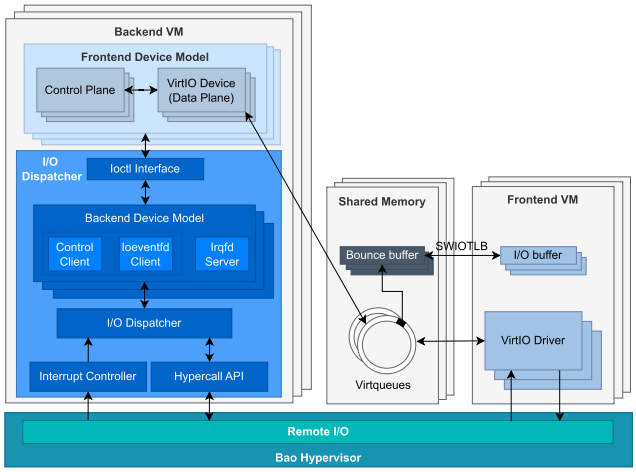


FIGURE 1. BiRtIO architecture. In this setup, the frontend VM issues VirtIO requests, while the backend VM handles interactions with the actual peripheral device (e.g., NIC). The shared memory region is used to pass the VirtIO information between the frontend and backend VMs.

memory region between the frontend and backend VMs to place the I/O buffers. Then, to restrict VirtIO driver I/O buffer allocation, we utilize a Linux mechanism called the restricted Direct Memory Access (DMA) pool. This configuration enables bounce buffering [46], where Linux uses the swiotlb, a memory buffer allocator in the kernel's DMA layer, to bounce the original buffer to a newly allocated buffer (known as a bounce buffer) within our shared memory region, specified by the *restricted-dma-pool* compatible property in the Linux device tree.

On the backend VM, we developed a custom kernel module named *I/O Dispatcher* to support VirtIO backend devices atop Bao. This module mediates the communication between the VirtIO devices and the Remote I/O system. Further details are provided in section V-A2. Details on control plane operations and data plane communication channels are provided in section V-A3.

V. BiRtIO IMPLEMENTATION

A. VirtIO INFRASTRUCTURE

1) REMOTE I/O

Remote I/O is our implementation of VirtIO request/response forwarding between the frontend VM to the associated backend VM in Bao. This feature will be adopted in Bao and included in the next release. Upon a VirtIO request from a frontend driver, the Remote I/O temporarily stores I/O request information, making it accessible to the backend VM via a custom hypercall Application Programming Interface (API) for processing. The data characterizing a VirtIO request includes, among other elements, (i) the address of the accessed MMIO register, (ii) the type of operation (read or write); (iii) the value to be written or read, and (iv) the I/O request state. The Remote I/O system is governed by a finite state machine, with each Virtual CPU (vCPU) featuring an I/O state that defines the state of the VirtIO request. There

are four states: *Free*, *Pending*, *Processing*, and *Complete*. The *Free* state indicates that the I/O request is either not in use, or the Remote I/O system has completed it by updating the accessed system register for read operations and resuming the guest vCPU. The *Pending* state means the I/O request is active and awaiting processing by the backend VM. The *Processing* state signals that the I/O request has been dispatched to the backend VM, but it has not yet finished processing. Lastly, the *Complete* state shows that the backend VM has completed the I/O request, but the Remote I/O system has not yet updated the accessed system register for the read operations and resumed the guest vCPU.

The Frontend-to-Backend and Backend-to-Frontend paths are depicted in Figure 2, in steps 1 to 13 and 14 to 16 (as detailed below).

a: FRONTEND-TO-BACKEND PATH

When the frontend driver accesses a VirtIO MMIO register (1), a trap to Bao occurs (2). Bao decodes the access, processes the access information and updates internal variables (3) (e.g., setting the I/O state to *Pending*) before notifying the backend vCPU of the availability of a new request (4). The vCPU responsible for executing the frontend is halted and transitions to an idle state while awaiting the outcome of the I/O request (5). Bao notifies the backend VM by injecting an interrupt into one of its vCPUs (6). The backend VM requests further information, including access address, operation type, and access width, through the hypercall API (7). Bao then updates the state of the frontend VM to *Processing* and returns the requested information (8). The backend VM processes the request possibly by executing device write or read operations (9) and uses the hypercall API to transmit the MMIO access information back to the hypervisor (e.g., in the case of an MMIO read operation, this information includes the requested value field with the MMIO device value) and updates the I/O request state to *Complete* (10). The hypervisor dispatches a message to the corresponding frontend VM vCPU (11) and updates the register value depending on the operation (12). Finally, the hypervisor activates the frontend vCPU and updates the frontend VM state to *Free* (13).

b: BACKEND-TO-FRONTEND PATH

This path is solely driven by interrupts. First, the backend VM sends a hypercall to the Bao hypervisor with the operation type of *Notify* (14). Then, the hypervisor notifies the frontend guest by injecting the configured VirtIO driver interrupt into the respective VM (15) (16).

2) I/O DISPATCHER

The I/O Dispatcher system is a kernel module that can be loaded into the Linux kernel of the backend VMs. It is responsible for establishing the connection between the Remote I/O system and the Frontend Device Model, offering a unified API to support various VirtIO backends. It comprises five main components: The Ioctl Interface,

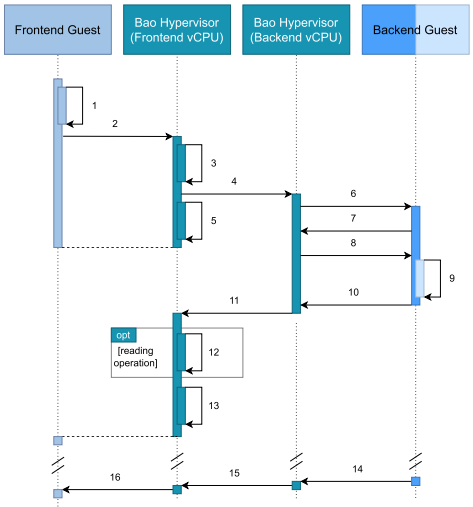


FIGURE 2. Remote I/O frontend-to-backend (1-13) and backend-to-frontend (14-16) paths.

Backend Device Model, Interrupt Controller, Hypercall API, and the I/O Dispatcher. The Ioctl Interface is an interface in the form of an ioctl API [47], responsible for establishing communication between the Frontend Device Model and the Backend Device Model abstraction. The Backend Device Model is a kernel-level abstraction designed to provide all the notification methods and I/O request information needed by the VirtIO device. This subsystem includes: i) *Control Client*: Responsible for interaction with the VirtIO control plane (implemented inside the VMM/Frontend Device Model), facilitating the exchange of all configurations and device features specified by the VirtIO specification; ii) *Ioeventfd Client*: Implementing the publisher side of the frontend-to-backend notification path, it will be responsible for notifying the VirtIO backend through the ioeventfd file descriptor [48] when an available buffer notification is received via an MMIO queue notify register write operation; iii) *Irqfd Server*: The subscriber side of the backend-to-frontend notification path is implemented via the irqfd file descriptor [49]. This implementation is responsible for forwarding notifications between the VirtIO backend and the Bao hypervisor. The Interrupt Controller is responsible for managing and handling interrupts related to I/O operations coming from the Remote I/O system. The Hypercall API facilitates the communication between the backend VM and the Bao hypervisor. Its functionalities include retrieving information about an I/O request (*Ask hypercall*), notifying the guest about a host notification (*Notify* hypercall), and completing write or read I/O operations (*Write* and *Read* hypercalls). Lastly, the I/O Dispatcher functions akin to a scheduler and is responsible to dispatch all I/O operations to/from their respective clients/server.

The workflow sequence can be subdivided into three principal parts, depicted in Figure 3: the initialization (I1-I4), control plane flow (C1-C7) and data plane flow (D1-D5 and H1-H2).

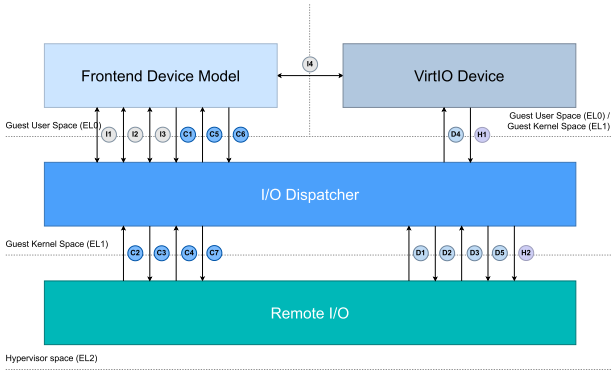


FIGURE 3. I/O Dispatcher initialization (I1-I4), control plane flow (C1-C7) and data plane flow (D1-D5 and H1-H2).

a: INITIALIZATION FLOW

The Frontend Device Model begins by creating the Backend Device Model abstraction on the I/O Dispatcher system by issuing an ioctl (I1). This operation involves several subprocesses, including the creation of the Control and Ioeventfd clients, and the establishment of the Irqfd Server. Next, the Frontend Device Model configures the frontend-to-backend (I2) and backend-to-frontend (I3) communication paths by passing the file descriptors and additional flags that the Ioeventfd Client and Irqfd Server will publish and subscribe to, respectively. More details about the Frontend Device Model, including the I4 initialization step, will be discussed in subsection V-A3.

b: CONTROL PLANE FLOW

The sequence begins with the Frontend Device Model issuing an ioctl to attach its control plane thread to the Control Client, which then waits in an idle state until a new I/O request arrives (C1). When the Remote I/O system injects an interrupt into the backend VM, the interrupt is handled by the Interrupt Controller (C2), and then the I/O Dispatcher system requests the VirtIO access information through the hypercall API (C3, C4). The I/O Dispatcher identifies which client the request belongs to by checking the MMIO address offset field and dispatches the request to the respective client (C5), waking it up. After the Frontend Device Model performs the MMIO write/read operation, it signals the completion of the request by issuing an ioctl and providing the request information (C6). Finally, the I/O Dispatcher system completes the request by sending a hypercall to the Remote I/O system with the request information (C7).

c: DATA PLANE FLOW

Following the VirtIO configuration and feature negotiation phase, the backend device becomes ready to receive notifications via an event file descriptor known as ioeventfd. Steps D1 to D3 mirror the control plane flow as the I/O Dispatcher system acquires the VirtIO request information. Once the VirtIO request information is obtained and it is determined that the MMIO register address corresponds to a queue notify operation, the I/O Dispatcher dispatches the I/O request to the Ioeventfd Client. The Ioeventfd Client then signals the event via the configured ioeventfd (D4). Finally, as the queue notify operation is write-only by the frontend driver, the I/O Dispatcher automatically completes the request via the hypercall API (D5). When the backend device needs to notify the frontend driver (steps H1 and H2 in Figure 3), it raises an irqfd that is captured by the Irqfd Server within the I/O Dispatcher system (H1). The Remote I/O system is then notified about the event via the *Notify* hypercall (H2).

3) FRONTEND DEVICE MODEL

The Frontend Device Model, or simply device model, encompasses all functionalities necessary for managing and configuring a VirtIO device. This includes both the control plane and the data plane, for typical VirtIO devices. The interface between the Frontend Device Model and the I/O Dispatcher system is established via ioctls, as depicted in Figure 3. This facilitates the creation of the Backend Device Model abstraction (I1) and the establishment of communication channels from frontend to backend (I2) and backend to frontend (I3).

To manage the control plane, the Frontend Device Model employs a dedicated thread and uses a specific ioctl to attach it to the Control Client on the I/O Dispatcher system (C1), where it awaits new I/O requests. For the data plane, the Frontend Device Model is responsible for its configuration (I4), executing several actions. These include (i) setting the memory map regions to allow the backend handler to map them, (ii) determining the number of *virtqueues* and their addresses within the memory regions, and (iii) providing the associated file descriptors (ioeventfds and irqfds) to establish the notification path between the backend handler and the I/O Dispatcher system. Depending on the type of backend handler, there are three possible scenarios: (i) for VirtIO devices implemented within the Frontend Device Model, the previous configuration steps are streamlined to simple function calls, (ii) for vhost devices, the configurations are built on top of the vhost protocol and executed through ioctl commands, (iii) for vhost-user devices, the protocol specifies the use of UNIX domain sockets for exchanging the information necessary to configure the backend.

The Frontend Device Model was integrated into both QEMU and Rust-VMM's abstractions. In both implementations, the Frontend Device Model can manage and configure both vhost and vhost-user devices. Specific details of this integration are not discussed in this paper due to space constraints.

B. REAL-TIME VIRTUALIZED NETWORK SETUP

While it is possible to bypass the Linux network stack using custom drivers or user-space network libraries, such as DPDK [50], XDP [51], or other vendor-specific proprietary implementations, we focus on leveraging the standard

mainline Linux capabilities available in the backend VM. This approach emphasizes the reusability of standard Linux interfaces and tools and aims to minimize maintenance efforts associated with third-party components. This section outlines (i) the implementation of a virtual network interface that enables network device sharing by two distinct guest VMs and (ii) the implementation of bandwidth limitation within the backend VM.

1) VIRTUAL NETWORK SETUP

For each frontend VM, a network TAP device — a virtual point-to-point network interface in Linux that operates at the data link layer — must be created within the backend VM. To create the TAP device, we used the *ip tuntap* command. This setup enables user-space applications (such as the VirtIO Net backend) to interact with the TAP device to transmit and receive the Ethernet frames. This interaction is carried out by first opening the */dev/net/tun* device file and issuing the corresponding ioctl command to register the network device with the kernel. Subsequently, write and read operations are performed on the corresponding TAP file descriptor device to transmit and receive Ethernet frames.

To facilitate communication between the TAP devices and the physical network interface (e.g., eth0) in the backend VM, a Linux bridge must be established. A Linux bridge is a virtual network device that connects multiple network interfaces at the data link layer and forwards packets between them. In this setup (Figure 4), when the VirtIO Net driver inside each frontend VM wishes to transmit a packet, the process on the backend VM is as follows: the VirtIO Net device forwards the data to the TAP device; the TAP device then sends the data to the bridge; and finally, the bridge routes the data to the physical NIC driver. Conversely, data returning from the NIC is routed back through the bridge to the TAP device and then to the VirtIO Net device. To create the Linux bridge, we used the *brctl* command.

2) REAL-TIME NETWORK CONFIGURATION

To regulate the rate of outgoing traffic, we first need to divide it into two distinct flows, each assigned a specific priority. Traffic prioritization involves a two-fold process. First, we need to identify and define which packets are high-priority and which are not by assigning a value to the socket buffer priority field (skb->priority). Setting the priority maps the traffic from a socket (or an application) to a socket buffer priority. This is an internal priority for the kernel networking layer, utilized by many other network kernel objects, which will be discussed later. There are several ways to set the priority of a specific traffic flow, such as using various user-space utilities available in the mainline Linux backend VM, e.g., *iptables* and *tc*. In this case, we use the *tc* command, a powerful Linux utility for managing network traffic by configuring QoS parameters, specifically the socket buffer priority action to modify the packet's header. Alongside the socket buffer priority action, we used the *tc filter* utility to identify and filter packets

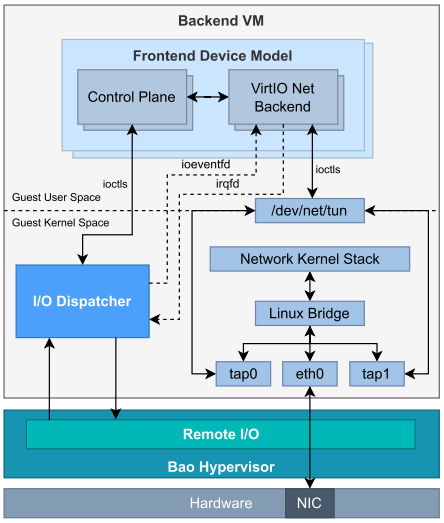


FIGURE 4. Virtual network configuration within the backend VM to enable network sharing for two guest VMs.

based on the source VM's Internet Protocol (IP) address, allowing us to set the priority according to their IP addresses. Next, the second step involves assigning each packet of each prioritized flow to a different transmit queue. This mapping between priorities and queues is achieved through an intermediate mapping object called traffic class and is mainly divided into two steps: mapping a packet from a traffic flow, based on its priority, to a traffic class, and then mapping each traffic class to a different transmit queue. Both steps are managed through the Multi-Queue Priority (MQPRIO) Queuing Discipline (Qdisc) available with the *tc* command. Internally, the MQPRIO assigns one Qdisc — a scheduler that determines how network packets are queued and transmitted - to each queue and maps the traffic classes to these QDiscs.

To regulate the rate of outgoing traffic and ensure that a VM consumes only a predefined portion of the total available network bandwidth, we replace the current queuing discipline with the CBS Qdisc. All these configurations were applied to the physical network interface of the Linux backend VM.

Figure 5 illustrates the flow of network traffic in the backend VM Linux kernel network stack, highlighting how different types of traffic — such as real-time and multimedia — are managed and prioritized from user space down to the NIC. The real-time and multimedia services correspond to the two VirtIO Net backend instances shown in Figure 4. These instances receive packets from the two frontend VMs (not displayed in the figure) and interact with the underlying network stack to transmit Ethernet frames via the corresponding TAP file descriptor, identified as the socket block. Once the TAP device forwards the data to the bridge, and the bridge routes it to the physical NIC driver, each packet is assigned a different priority based on the source VM's IP address, classifying the traffic into two distinct queues. The CBS algorithm then shapes the transmit rate for each traffic queue accordingly. Each traffic class is mapped to a

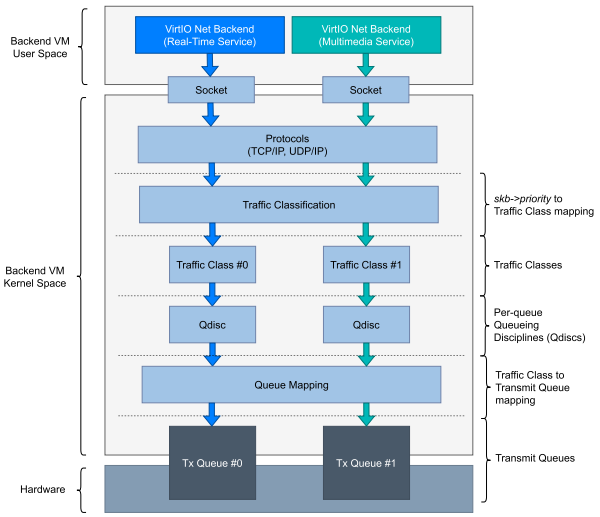


FIGURE 5. Flow of two distinct network traffic streams - real-time and multimedia — through the backend VM Linux kernel network stack.

specific transmit queue, allowing packets to be enqueued for transmission by the hardware layer.

In Subsection VI-B, we provide a comprehensive example of this implementation and evaluate the results both with and without the application of bandwidth limitation in a saturated virtualized network, composed of two guest VMs accessing the same NIC.

VI. EXPERIMENTAL EVALUATION

All experiments were conducted on a Zynq UltraScale+ MPSoC ZCU102 FPGA platform, featuring four ARM Cortex-A53 cores sharing a last-level L2 cache of 1 MiB. The software stack utilized comprised the Zynq UltraScale+ MPSoC Firmware (version 2023.01), Bao (version 1.0), and Linux (version 6.5) for both the backend and frontend VMs. Bao and the Linux VMs were compiled with the GNU AArch64 Toolchain (version 11.2.1 2022.02). For all conducted tests (Table 2), each VM was allocated only one vCPU, and only one backend VM was used.

A. PERFORMANCE ANALYSIS

1) NETWORK LATENCY AND JITTER

To measure networking latency, we used the *ping* tool to initiate and measure round-trip times of Internet Control Message Protocol (ICMP) packets between the ZCU102 and an external laptop. We used both QEMU (version 8.0.3) and Rust-VMM VirtIO-Net devices. To compare our implementation with state-of-the-art solutions, we (i) measured the passthrough scenario where a Linux VM had direct access to the NIC and (ii) ran the same set of tests using KVM in conjunction with the QEMU VMM. In the *passthrough* and *one-guest* configurations, the Linux VM and the guest VM execute ping individually. In the *two-guests* configurations, both Linux guest VMs execute ping simultaneously. Experiments were conducted using various packet sizes, with each experiment transmitting 4,000 packets.

As shown in Figure 6, introducing the VirtIO interface and using the QEMU device model (*bao+qemu (one-guest)*) leads to an average latency increase of 0.30 ms compared to the *passthrough* scenario. However, compared to KVM (*kvm+qemu (one-guest)*), the difference is only 48 μs, resulting in an 18.35% overhead. Bao's higher latency is mainly due to (i) the overhead from bouncing I/O buffers to a shared memory region and (ii) the additional latency from interactions between the frontend and backend VMs. In the KVM setup, the guest VM runs within QEMU, enjoying full access to guest memory and eliminating the need for bounce buffers, which Bao requires for security. Additionally, in KVM, guest-host interactions are more direct since the guest OS runs within the same Linux host. In contrast, Bao's stringent spatial and temporal isolation results in increased execution times for these interactions. Notwithstanding, using the Rust-VMM VirtIO-Net device (*bao+rustvmm (one-guest)*) shows better performance than QEMU (*bao+qemu (one-guest)*), improving by 6.20%. When compared to KVM (*kvm+qemu (one-guest)*), it incurs only an 11.00% overhead. These results can be justified by the complexity of QEMU's software stack, which is higher than the Rust-VMM abstractions.

Despite the increase in latency, Bao's configurations show a reduction in latency variance, or jitter, when compared to both passthrough and KVM scenarios. The *passthrough* setup averages a jitter of 128 μs, while the *bao+qemu (one-guest)* and *bao+rustvmm (one-guest)* setups achieve 122.6 μs and 88.1 μs, respectively. This represents an improvement of 4.22% and 31.2% over the *passthrough* configuration, respectively. This may be explained by the consistent increase in network latency introduced by the VirtIO infrastructure, which results in a slower but more stable connection. Compared to KVM, Bao achieves lower jitter values with both QEMU and Rust-VMM VirtIO-Net backends. Bao's QEMU configuration (*bao+qemu (one-guest)*) improves jitter by 10.05% over KVM (*kvm+qemu (one-guest)*), while the *bao+rustvmm (one-guest)* configuration achieves 35.36% better results. The difference arises because KVM runs guests as standard Linux processes, which subjects them to potential delays from the Linux scheduler. Bao's SPH architecture and its principles of spatial and temporal isolation for VMs mitigate this issue.

In the *two-guests* configurations, there is an average latency increase of 70 μs compared to the *one-guest* configurations. This represents a 14.58% relative increase in latency, a side-effect of the sharing logic. Additionally, it is noted that there is a very small deviation between the two VMs because both are running Linux, which does not guarantee deterministic execution.

In summary, the results show that the introduction of BiRtIO results in an unavoidable latency overhead of approximately 18.35% compared to KVM. Utilizing a simpler device model, such as Rust-VMM, enhances network latency by

TABLE 2. Overview of all test configurations. ✓: the test includes the feature, –: the test does not include the feature.

Configuration	Description	Hypervisor	VirtIO device	Bounce buffers enabled	QoS enabled
passthrough	A Linux VM with direct access to the NIC	–	–	–	–
kvm+qemu (one-guest)	One guest VM accessing the NIC via VirtIO	KVM	QEMU	–	–
bao+qemu (one-guest)		Bao	QEMU	✓	–
bao+rustvmm (one-guest)		Bao	Rust-VMM	✓	–
bao+qemu (one-guest w/o bb)		Bao	QEMU	–	–
bao+rustvmm (one-guest w/o bb)		Bao	Rust-VMM	–	–
kvm+qemu (two-guests)	Two guest VMs sharing the same NIC	KVM	QEMU	–	–
bao+qemu (two-guests)		Bao	QEMU	✓	–
bao+rustvmm (two-guests)		Bao	Rust-VMM	✓	–
bao+qemu (two-guests w/o bb)		Bao	QEMU	–	–
bao+rustvmm (two-guests w/o bb)		Bao	Rust-VMM	–	–
congestion	Real-Time VM + Multimedia VM sharing the same NIC	Bao	Rust-VMM	✓	–
qos-control		Bao	Rust-VMM	✓	✓

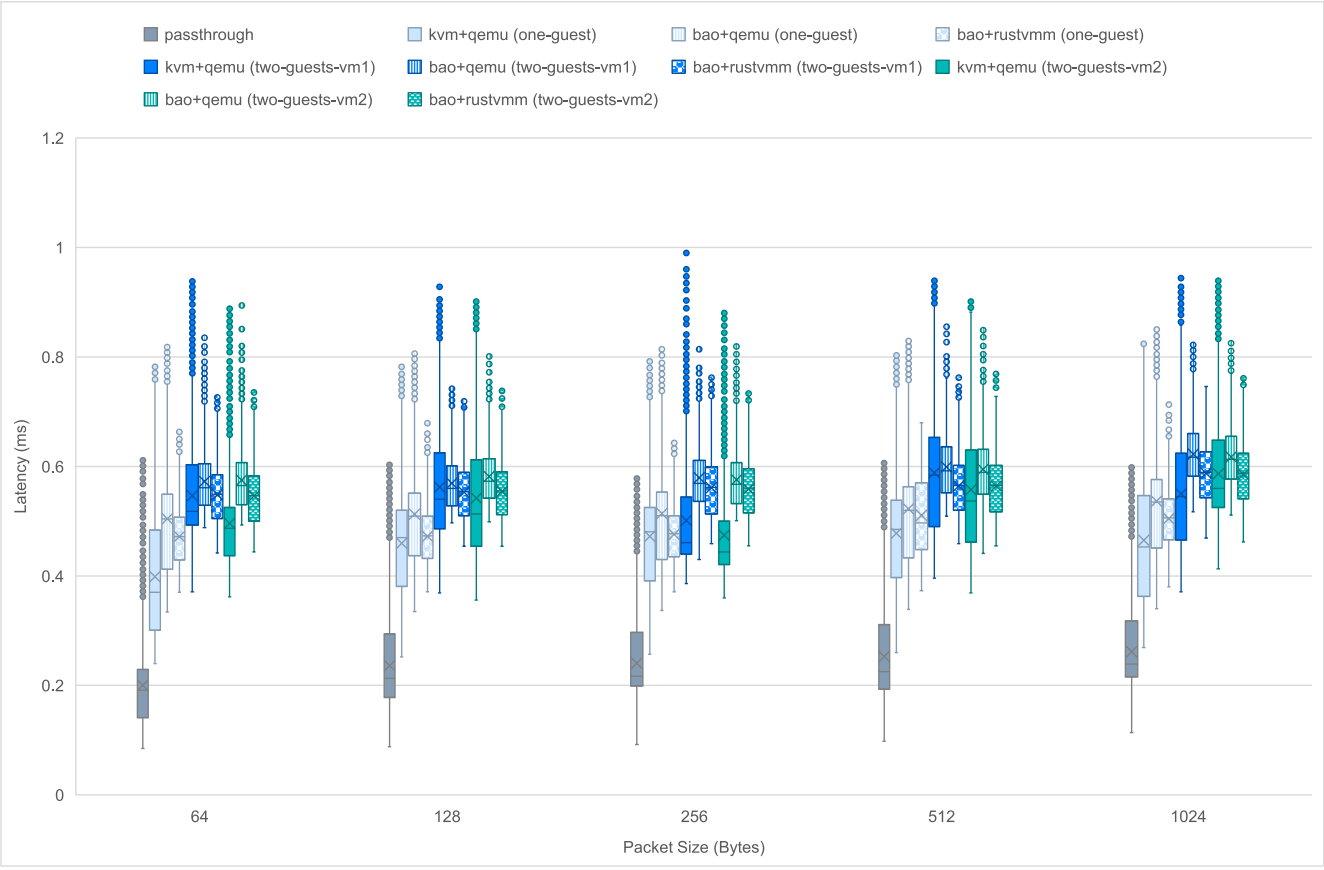


FIGURE 6. Impact of VirtIO infrastructure on latency and jitter. We evaluated two different device models with two distinct VirtIO-Net device implementations — QEMU and Rust-VMM. The results were compared with both the passthrough scenario and the KVM hypervisor utilizing QEMU VMM.

up to 6.20%. However, we are willing to trade performance for improved security guarantees. Despite the increase in latency, BiRtIO enhances network predictability, achieving lower jitter values compared to both the *passthrough* and KVM setups. In the best case (using the Rust-VMM VirtIO-Net device), the jitter is reduced by 35.36% compared to KVM.

2) NETWORK BANDWIDTH

We also measured the maximum achievable bandwidth using *iperf3* across the same configurations mentioned earlier. Additionally, to study the impact of introducing bounce buffers to restrict backend VM access, we evaluated the effect of providing the backend VM with access to the entire address space of the frontend VM(s). Although Bao's core principles emphasize memory isolation between VMs, the frontend VMs entire address spaces were configured as shared memory regions accessible to the backend VM. The server was installed on the external laptop, and the client(s) were installed on the ZCU102. The bandwidth experiments were conducted for 30 minutes.

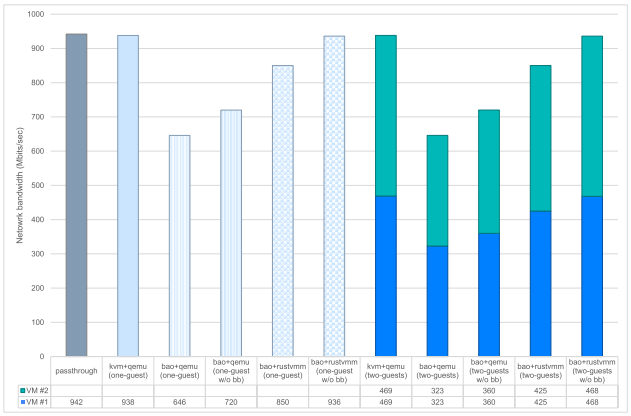


FIGURE 7. Impact of VirtIO infrastructure on network bandwidth. We evaluated two different device models with two distinct VirtIO-Net device implementations – QEMU and Rust-VMM. We also measured the impact of bounce buffers on network bandwidth for both device models. The results were compared with both the passthrough scenario and the KVM hypervisor utilizing the QEMU VMM.

Figure 7 shows that introducing the VirtIO layer (*bao+qemu (one-guest)*) results in a 31.13% overhead compared to the KVM setup (*kvm+qemu (one-guest)*), and 31.42% when compared with the *passthrough* scenario. However, with the Rust-VMM VirtIO-Net device, this overhead is reduced to 9.38% relative to KVM, or 9.77% when compared to the *passthrough* configuration. The disparity is largely due to QEMU's more extensive software stack, which becomes even more pronounced under high-intensity workloads like *iperf3*.

When bounce buffers are eliminated - *bao+qemu (one-guest w/o bb)* and *bao+rustvmm (one-guest w/o bb)* - the performance improves from 646 Mbits/s to 720 Mbits/s and from 850 Mbits/s to 936 Mbits/s, representing improvements of 11.46% and 10.12%, respectively. These findings suggest that the overhead introduced by the VirtIO layer alone is, at its best (using the Rust-VMM VirtIO-Net device), reduced to 0.64% compared to the *passthrough* scenario, or 0.21% when compared to KVM. The performance penalty arises from the CPU-intensive data copying associated with bounce buffers during each VirtIO device DMA transaction, which is particularly detrimental to high-throughput tasks like *iperf3*. In both KVM and Bao *dual-guests* and *dual-guests (w/o bb)* configurations, the bandwidth is evenly shared between the two VMs due to the backend VM round-robin scheduler.

In summary, the results show that the introduction of BiRtIO incurs a 9.77% overhead on network bandwidth at its best (using the Rust-VMM VirtIO-Net device). However, 93.45% of this overhead is attributed to the use of bounce buffers, indicating that the VirtIO infrastructure itself only imposes a performance loss of 0.64% compared to the *passthrough* scenario, and 0.21% when compared to KVM. Similar to network latency and jitter, the choice of the device model significantly impacts the overall available network bandwidth.

B. TSN ANALYSIS IN A VIRTUALIZED ENVIRONMENT

In this subsection, we evaluate the effectiveness and feasibility of using open-source TSN mechanisms available in mainline Linux to enhance performance and predictability in a saturated virtualized network. The setup, depicted in Figure 8, comprises two Linux frontend VMs, each utilizing the Linux VirtIO-Net driver as the network interface, and one Linux backend VM that runs the Rust-VMM VirtIO-Net backends with direct access to the physical NIC. The choice of Rust-VMM over QEMU was driven by its superior performance in terms of latency, jitter, and bandwidth. We configured the first frontend VM to function as a real-time workload (e.g., control system) and the second frontend VM to function as a general-purpose workload (e.g., multimedia system). The established requirements for both real-time and multimedia traffic are described next.

a: REAL-TIME TRAFFIC

The real-time service must send 20 Transmission Control Protocol (TCP) messages of 64 bytes per second with a round-trip latency that should always be less than 2 milliseconds. The average jitter must be lower than 250 microseconds. These values were based on Blanco et al.'s work [52], specifically for low-complexity and static driving real-time services such as sensors and actuators, which, according to the authors, feature hard deadlines, require reactivity within a few milliseconds, involve low data complexity and size, and follow a periodic pattern;

b: MULTIMEDIA TRAFFIC

The multimedia network bandwidth should operate at a minimum of 100 Mbit/s. The value of 100 Mbit/s was drawn from the work of Al Mtawa et al. [53], where it is noted that Ultra-High Definition (4K) streaming requires at least 25 Mbit/s. By reserving 100 Mbit/s for the multimedia traffic, we ensure that the ''video stream'' will occur without interruptions.

The real-time traffic was simulated using the *ping* command, and the multimedia traffic was generated using the *iperf3* utility. In Figures 9 and 10, the *congestion* case illustrates the network bandwidth achieved by the multimedia VM and the real-time traffic behavior, respectively. As shown in Figure 9, the multimedia requirements were met. The multimedia network bandwidth saturates the entire network, practically reaching the maximum available bandwidth of 850 Mbit/s. The 2 Mbit/s difference, not visible in the *congestion* chart, is utilized by the real-time VM to transmit the 50 ms periodic messages. However, the *congestion* configuration in Figure 10 reveals that the real-time requirements were not satisfied. The real-time traffic failed to meet both round-trip latency and jitter criteria. The round-trip time ranged from 0.650 ms to 8.115 ms, with an average of 3.188 ms, while the average jitter was 2.986 ms. This means that, when the multimedia system saturates the

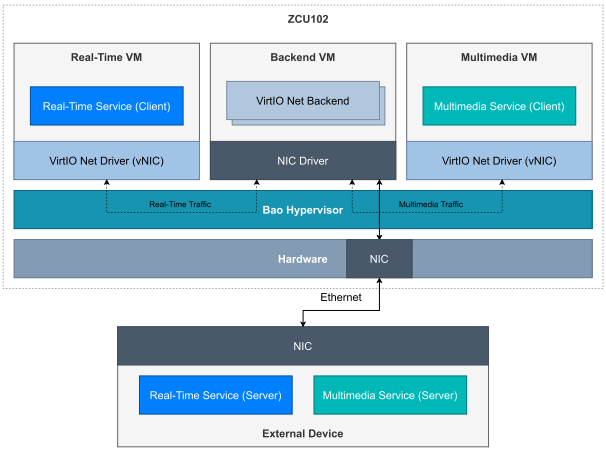


FIGURE 8. Triple-VM architecture for TSN-based networks: (i) a VM running a real-time service, (ii) a VM running a multimedia service, and (iii) a backend VM mediating communication between the guest VMs and an external laptop via Ethernet.

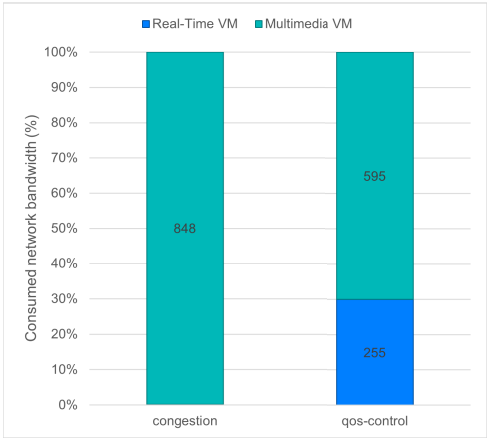


FIGURE 9. Network bandwidth in multimedia and real-time VMs: Congestion (without bandwidth limitation) vs. QoS Control (with bandwidth limitation).

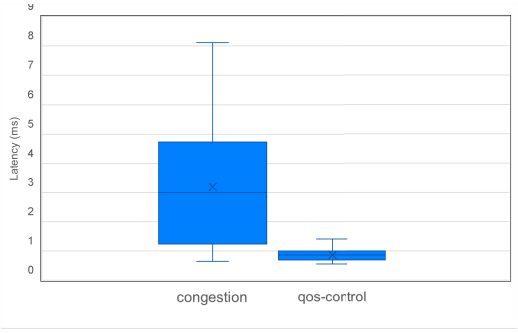


FIGURE 10. Real-time traffic behavior: Congestion (without bandwidth limitation) vs. QoS Control (with bandwidth limitation).

network, the average real-time round-trip time exceeds the acceptable limit by 59.4% and the jitter exceeds by 1094%.

Following the assessment of the *congestion* scenario, a QoS implementation based on the bandwidth limitation was performed, as detailed in section V-B. Initially, two traffic queues with different priorities were created to separate the traffic between the two workloads. Real-time traffic was assigned the highest priority, while multimedia traffic was given the lowest priority. Subsequently, the multimedia traffic was shaped to utilize no more than 70% of the total available bandwidth to prevent network saturation and ensure sufficient bandwidth allocation for real-time traffic. The *qos-control* case in Figure 9 shows the measured network bandwidth for both multimedia and real-time VMs. The multimedia VM achieves 595 Mbit/s, while the real-time VM attains 255 Mbit/s, resulting in a precise 7:3 ratio.

The *qos-control* configuration in Figure 10 illustrates the impact of applying bandwidth limitation on enhancing the predictability and reducing latency of real-time traffic. The results show significant improvement in the values distribution, with round-trip times ranging from 0.555 ms to 1.413 ms and averaging 0.863 ms. The average jitter was measured at 243.96 μs. Note that the network bandwidth limitation could be split more fairly, considering that the multimedia VM requires only 100 Mbit/s, approximately 12% of the total available bandwidth in a dual-VM configuration. However, since the real-time traffic involves transferring low-complexity and low-size data periodically, reserving more bandwidth for the real-time VM would be inefficient.

These findings demonstrate that it is possible to leverage the mainline Linux ecosystem inside the backend VM to implement bandwidth limitation, enabling both guest VMs to consistently meet predefined performance requirements in a saturated virtualized network environment.

C. IMPACT OF REMOTE I/O ON BAO's TCB

We measure the size of Bao's TCB before and after implementing the modifications required for the Remote I/O system. To measure the lines of code we use Tokei [54]. To measure the size we use the size tool provided by the toolchain. The results are presented in Table 3. The Remote I/O system performs all its functions with an addition of only 540 lines of code in the hypervisor, which represents a 4.05% increase. Regarding memory usage, we analyze the size of the program segments in the binary. An increase in size is detectable, as expected. This increase corresponds to 3.92 KiB or 6.36%.

However, although Bao's TCB increase was relatively small, introducing at least one backend VM with a full OS such as Linux could impose several security concerns. The dependency of one VM to provide services, such as networking, to another VM expands the frontend VM's TCB, thereby increasing its attack surface. This situation could lead to a scenario where, even if the frontend VM is not compromised, its dependence on another VM — which hypothetically could be compromised — could adversely affect it indirectly. Another example is when one VM doesn't want to share its data (e.g., sensor information) with another VM, despite needing to share the same device. In this specific

TABLE 3. Remote I/O impact on system TCB.

Lines of Code		Size (bytes)	
Bao	Bao w/ Remote I/O	Bao	Bao w/ Remote I/O
13238	13778	59165	63181

scenario, the data could be end-to-end encrypted to prevent sharing sensitive information with the respective backend VM, with decryption occurring only after the data reaches the receiver.

VII. CONCLUSION AND FUTURE WORK

In this paper we presented the deployment of real-time networks on a virtualized environment supported by Bao. We developed a device-sharing solution using the VirtIO interface, which will be adopted in Bao and included in the next release, all while adhering to Bao's safety and security principles. We leveraged Linux to host the VirtIO backends to benefit from both readily available VirtIO support and existing TSN mechanisms implementation. The analysis of the impact of our solution demonstrates performance and predictability levels suitable for MCS with limited Ethernet ports.

In future work we will explore replacing bounce buffering with alternative mechanisms, as copying data between buffers is responsible for the majority of the performance degradation. Moreover, we intend to explore (i) additional TSN protocols to enhance real-time communication capabilities within the VirtIO environment and (ii) the performance and predictability implications of bypassing the Linux kernel through custom drivers such as DPDK or XDP. Lastly, we will explore replacing mainline Linux as the OS for backend VMs with a Linux-based unikernel to improve memory footprint, and potentially enhance security and performance by minimizing the attack surface and overhead associated with full-fledged Linux VMs.

REFERENCES

[1] N. Jazdi, "Cyber physical systems in the context of industry 4.0," in *Proc. IEEE Int. Conf. Autom., Quality Test., Robot.*, May 2014, pp. 1–4.
[2] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on TrustZone-enabled microcontrollers? Voilá!" in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2019, pp. 293–304.
[3] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, "μRTZVisor: A secure and safe real-time hypervisor," *Electronics*, vol. 6, no. 4, p. 93, Oct. 2017.
[4] S. Pinto, J. Martins, J. Lopes, M. Abreu, and A. Tavares, "SecSSy hypervisor: Security-safety synergy for aerospace," in *Proc. Data Syst. Aerosp. (DASIA)*, 2017, pp. 1–8.
[5] Linutronix. (2022). *TSN and Linux*. [Online]. Available: https://www.linutronix.de/PDF/DS_2022_TSN_Linux_V1.0_E_A4.pdf?m=1670755560&m=1666685856&amp%3B=
[6] T. Zhang, G. Wang, C. Xue, J. Wang, M. Nixon, and S. Han, "Time-sensitive networking (TSN) for industrial automation: Current advances and future directions," *ACM Comput. Surv.*, vol. 57, no. 2, pp. 1–38, Feb. 2025.
[7] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *Proc. Workshop Next Gener. Real-Time Embedded Syst.*, Jan. 2020, p. 14.
[8] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the key," in *Proc. Euromicro Conf. Real-Time Syst.*, vol. 76, Jan. 2017, p. 22.
[9] D. Costa, L. Cuomo, D. Oliveira, I. M. Savino, B. Morelli, J. Martins, A. Biasci, and S. Pinto, "IRQ coloring and the subtle art of mitigating interrupt-generated interference," in *Proc. IEEE 29th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2023, pp. 47–56.
[10] S. Pereira, J. Sousa, S. Pinto, J. Martins, and D. Cerdeira, "Bao-Enclave: Virtualization-based enclaves for arm," in *Proc. IEEE World Forum Internet Things (WF-IoT)*, Oct. 2022, pp. 1–6.
[11] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto, "TZ- VirtIO: Enabling standardized inter-partition communication in a trustzone-assisted hypervisor," in *Proc. IEEE 27th Int. Symp. Ind. Electron. (ISIE)*, Jun. 2018, pp. 708–713.
[12] B. Sá, J. Martins, and S. Pinto, "A first look at RISC-V virtualization from an embedded systems perspective," *IEEE Trans. Comput.*, vol. 71, no. 9, pp. 2177–2190, Sep. 2022.
[13] M. Cinque, L. De Simone, and A. Marchetta, "Certify the uncertified: Towards assessment of virtualization for mixed-criticality in the automotive domain," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun. 2022, pp. 8–11.
[14] A. Qumranet, Y. Qumranet, D. Qumranet, U. Qumranet, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 225–314.
[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.
[16] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 333–348. [Online]. Available: https://dl.acm.org/doi/10.1145/2644865.2541946
[17] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones," in *Proc. 5th IEEE Consum. Commun. Netw. Conf.*, Jan. 2008, pp. 257–261.
[18] D. G. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proc. 4th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, Mar. 2008, pp. 151–160.
[19] D. Ottaviano, F. Ciraolo, R. Mancuso, and M. Cinque, "The Omnivisor: A real-time static partitioning hypervisor extension for heterogeneous core virtualization over MPSoCs," in *Proc. Euromicro Conf. Real-Time Syst.*, 2024, pp. 7:1–7:27.
[20] J. Martins and S. Pinto, "Shedding light on static partitioning hypervisors for arm-based mixed-criticality systems," in *Proc. IEEE 29th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, May 2023, pp. 40–53.
[21] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no VM exits! (Almost)," in *Proc. Workshop Operating Syst. Platforms Embedded Real-Time Appl.*, Jan. 2017. [Online]. Available: https://arxiv.org/abs/1705.06932
[22] B. Cilku and P. Puschner, "Towards temporal and spatial isolation in memory hierarchies for mixed-criticality systems with hypervisors," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Sep. 2013, pp. 25–28.
[23] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The Internet of Things, fog and cloud continuum: Integration and challenges," *Internet Things*, vols. 3–4, pp. 134–155, Oct. 2018.
[24] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008.
[25] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, Apr. 2005, p. 41.
[26] H. Li, X. Xu, J. Ren, and Y. Dong, "ACRN: A big little hypervisor for IoT development," in *Proc. 15th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Apr. 2019, pp. 31–44.
[27] A. Florescu. (2019). *Building the Virtualization Stack of the Future With Rust-VMM*. [Online]. Available: https://opensource.com/article/19/3/rust-virtual-machine
[28] S. Patni, J. George, P. Lahoti, and J. Abraham, "A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial," in *Proc. 1st Int. Conf. Next Gener. Comput. Technol. (NGCT)*, Sep. 2015, pp. 6–9.

[29] F. Diakhaté, M. Pérache, R. Namyst, and H. Jourdren, "Efficient shared memory message passing for inter-VM communications," in *Proc. Euro-Par Workshops-Parallel Process.*, 2008, pp. 53–62.
[30] E. P. Martín. (2019). *Deep Dive Into Virtio-Networking and Vhost-Net*. [Online]. Available: https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net
[31] E. P. Martín and A. M. Zapata. (2019). *A Journey to the Vhost-Users Realm*. [Online]. Available: https://www.redhat.com/en/blog/journey-vhost-users-realm
[32] T. T. Group. (2024). *Time-Sensitive Networking Task Group*. [Online]. Available: https://1.ieee802.org/tsn/
[33] IEEE (2007). *802.1Qav—Forwarding and Queuing Enhancements for Time-Sensitive Streams*. [Online]. Available: https//www.ieee802.org/1/pages/802.1av.html
[34] L. Zhao, P. Pop, and S. Steinhorst, "Quantitative performance comparison of various traffic shapers in time-sensitive networking," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 3, pp. 2899–2928, Sep. 2022.
[35] A. Alliance. (2019). *TSN Documentation Project for Linux*. [Online]. Available: https://tsn.readthedocs.io/
[36] Linaro. (2023). *Project Orko: Empowering SOAFEE With Virtio Magic*. [Online]. Available: https://linaro.atlassian.net/wiki/spaces/ORKO/overview
[37] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proc. USENIX Symp. Networked Syst. Design Implement.*, Jan. 2020, pp. 419–434.
[38] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Upper Saddle River, NJ, USA: Prentice-Hall, 2007.
[39] C. Clark. *Virtio & Argo: A Linux VirtioTransport Driver on Xen*. Accessed: Jun. 5, 2024. [Online]. Available: https://www.mail-archive.com/xen-devel@lists.xenproject.org/msg82602.html
[40] L. Leonardi, L. L. Bello, and G. Patti, "Towards time-sensitive networking in heterogeneous platforms with virtualization," in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, vol. 1, Sep. 2020, pp. 1155–1158.
[41] D. Ferraro, L. Palazzi, F. Gavioli, M. Guzzinati, A. Bernardi, B. Rouxel, P. Burgio, and M. Solieri, "Time-sensitive autonomous architectures," *Real-Time Syst.*, vol. 59, no. 4, pp. 568–608, Dec. 2023.
[42] A. Garbugli, L. Rosa, L. Foschini, A. Corradi, and P. Bellavista, "A framework for TSN-enabled virtual environments for ultra-low latency 5G scenarios," in *Proc. ICC - IEEE Int. Conf. Commun.*, May 2022, pp. 5023–5028.
[43] I Framework. (2024). *Time-Aware Shaping*. [Online]. Available: https://inet.omnetpp.org/docs/showcases/tsn/trafficshaping/timeawareshaper/doc/index.html#:~:text=Time%2Daware%20shaping%20works%20by,windows%20based%20on%20their%20priority
[44] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vSwitch," in *Proc. USENIX Conf. Networked Syst. Design Implement.*, May 2015, pp. 117–130.
[45] J. H. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, Jan. 1975.
[46] J. Corbet. (2023). *A More Dynamic Software I/O TLB*. [Online]. Available: https://lwn.net/Articles/940973/
[47] T-Kernel Develop. Community. (1979). *Ioctl Based Interfaces*. [Online]. Available: https://docs.kernel.org/driver-api/ioctl.html
[48] G. Haskins. (2009). *KVM: Add Ioeventfd Support*. [Online]. Available: https://patchwork.kernel.org/project/kvm/patch/1251028605-31977-23-git-send-email-avi@redhat.com/
[49] (2017). *KVM Irqfd Introduction*. [Online]. Available: https://xzpeter.org/htmls/2017_12_07_kvm_irqfd/kvm_irqfd_implementation.html
[50] L. Foundation. (2024). *DPDK: Powering the Future of Network Software & Applications*. [Online]. Available: https://www.dpdk.org/
[51] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," in *Proc. 14th Int. Conf. Emerg. Netw. EXperiments Technol.*, Dec. 2018, pp. 54–66.
[52] D. F. Blanco, F. Le Mouël, T. Lin, and M.-P. Escudié, "A comprehensive survey on software as a service (SaaS) transformation for the automotive systems," *IEEE Access*, vol. 11, pp. 73688–73753, 2023.
[53] Y. A. Mtawa, A. Haque, and B. Bitar, "The mammoth Internet: Are we ready?" *IEEE Access*, vol. 7, pp. 132894–132908, 2019.
[54] Tokei. (2015). *Tokei: Count Your Code Quickly*. [Online]. Available: https://docs.rs/tokei/latest/tokei/



JOÃO PEIXOTO received the bachelor's degree in electronics and computer engineering. He is currently pursuing the master's degree in embedded systems and computers and control, automation, and robotics with the Embedded Systems Research Group, Universidade do Minho, Portugal. His research interests include virtualization, operating systems, security for embedded systems, and compilers.



JOSÉ MARTINS received the master's degree in electronics and computer engineering. He is currently pursuing the Ph.D. degree with the Embedded Systems Research Group, Universidade do Minho, Portugal. He is also a Teaching Assistant with the Embedded Systems Research Group, Universidade do Minho. During the master's degree, he was a Visiting Student with the University of Würzburg, Germany. He has a significant background in operating systems and virtualization for embedded systems. He is the main author of the Bao hypervisor.



DAVID CERDEIRA received the Ph.D. degree in electronics and computer engineering. He is currently a Researcher with the Embedded Systems Research Group, Universidade do Minho, Portugal. His work focuses on trusted execution environments, embedded systems, embedded security, and virtualization. He has contributed to these fields through several peer-reviewed publications.



SANDRO PINTO received the Ph.D. degree in electronics and computer engineering. He is currently an Associate Research Professor with the Centro ALGORITMI, Universidade do Minho, Portugal. He has a deep academic background and several years of industry collaboration focusing on virtualization, security for embedded systems, and IoT systems. He has published more than 100 peer-reviewed articles and skilled presenter with speaking experience in top-tier academic and industrial conferences.