Courier: Host-Transparent Direct Network I/O for Secure Containers in Cloud VMs

Hang Huang
Huazhong University of Science and Technology
Wuhan, Hubei, China
hanghuang@hust.edu.cn

Zhucan Ye
Huazhong University of Science and Technology
Wuhan, Hubei, China
yezhucan@hust.edu.cn

Zhiheng Tao
Ant Group
Hangzhou, Zhejiang, China
junchuan.tzh@antgroup.com

Hui Lu
The University of Texas at Arlington
Arlington, Texas, USA
hui.lu@uta.edu

Wenlong Hou
Ant Group
Hangzhou, Zhejiang, China
houwenlong.hwl@antgroup.com

Wei Guo
Alibaba Group
Hangzhou, Zhejiang, China
xiaofeng.gw@alibaba-inc.com

Quan Xu
Alibaba Group
Hangzhou, Zhejiang, China
wutu.xq@alibaba-inc.com

Jiang Liu
Alibaba Group
Hangzhou, Zhejiang, China
liu.jiang@alibaba-inc.com

Jiangshan Lai
Ant Group
Hangzhou, Zhejiang, China
jiangshan.ljs@antgroup.com

Jia Rao
The University of Texas at Arlington
Arlington, Texas, USA
jia.rao@uta.edu

Zhuo Huang
Huazhong University of Science and Technology
Wuhan, Hubei, China
huangzhuo@hust.edu.cn

Song Wu
Huazhong University of Science and Technology
Wuhan, Hubei, China
wusong@hust.edu.cn

Chen Yu
Huazhong University of Science and Technology
Wuhan, Hubei, China
yuchen@hust.edu.cn

Hai Jin
Huazhong University of Science and Technology
Wuhan, Hubei, China
hjin@hust.edu.cn

Abstract

Secure containers strengthen isolation by encapsulating conventional OS-level containers within lightweight virtual machines (microVMs), and have been widely adopted to execute untrusted, multi-tenant workloads in modern cloud platforms. In practice, these platforms are increasingly deployed atop cloud virtual machines to exploit cloud elasticity and cost efficiency, which places secure containers within a nested virtualization stack. Although prior work has significantly advanced nested CPU and memory virtualization, network I/O remains a critical bottleneck. Existing para-virtualized networking solutions incur substantial overhead due to repeated cross-layer mediation, while hardware-assisted device passthrough depends on IOMMU virtualization support that is often unavailable in public clouds.

This paper presents Courier, a direct network I/O architecture that enables passthrough-like, high-performance networking for secure containers in cloud environments. Courier rethinks nested network virtualization by introducing an in-guest shadow IOMMU to enable secure device passthrough with a shorter network stack and zero-copy data transfer, without requiring host involvement. In addition, Courier incorporates several data-plane techniques, including direct-kick notifications, used-ring passthrough, and efficient interrupt handling, to further improve packet processing efficiency. We implement Courier in a production system and evaluated it using both microbenchmarks and real-world applications. The results show that Courier significantly improves both throughput and tail latency, consistently outperforming state-of-the-art virtio-net and vhost-net by up to 9.36x and 96.7%, respectively, while achieving near-native performance under realistic workloads.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857111

CCS Concepts: • **Software and its engineering → Operating systems**; *Input / output*.

Keywords: Secure containers, Network, I/O virtualization

ACM Reference Format:
Hang Huang, Zhucan Ye, Zhiheng Tao, Hui Lu, Wenlong Hou, Wei Guo, Quan Xu, Jiang Liu, Jiangshan Lai, Jia Rao, Zhuo Huang, Song Wu, Chen Yu, and Hai Jin. 2026. Courier: Host-Transparent Direct Network I/O for Secure Containers in Cloud VMs. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857111

1 Introduction

While conventional Linux containers offer lightweight packaging, fast startup, and close-to-native performance, their dependence on a shared host kernel fundamentally limits isolation [13], making them ill-suited for hostile *multi-tenant* settings [30, 32, 49]. Consequently, cloud platforms, e.g., Alibaba MaxCompute [7], have adopted *secure containers*, such as RunD [36], which encapsulate each container inside a lightweight virtual machine (microVM), combining container agility with hardware-enforced strong isolation.

For high-performance secure container execution, bare-metal servers are attractive because they allow the secure-container runtime to operate directly on physical servers, efficiently multiplexing microVM-isolated secure containers from different tenants through hardware-assisted virtualization techniques. However, relying exclusively on bare-metal servers is impractical in public clouds because their provisioning is slow and their elasticity is limited under bursty demand [40]. For example, in Alibaba Cloud, starting a bare-metal server (e.g., ecs.ebmg9i.48xlarge with 192 vCPUs and 768 GiB memory) from launch to readiness takes 232.4 seconds, whereas a VM instance with the same resources (e.g., ecs.g9i.48xlarge) takes 22.4 seconds; a smaller VM instance with one-quarter of the resources (e.g., ecs.g9i.12xlarge) takes only 19 seconds. Moreover, bare-metal server allocation may fail in some regions due to provisioning shortages.

To accommodate demand surges without continuously over-provisioning bare-metal servers, cloud platforms adopt heterogeneous resource pools [19] that combine statically provisioned bare-metal capacity with on-demand VMs [3]. Since VMs remain the primary tenancy unit in public clouds, offering fast startup and elastic provisioning, secure containers are increasingly deployed *inside* these cloud VMs [17].

This results in a three-layer execution stack that comprises the host (L_0), the guest VM (L_1), and the nested microVM (L_2). While recent work has significantly advanced nested CPU and memory virtualization [16, 29], nested network I/O remains a critical bottleneck. Specifically, nested secure containers primarily rely on *para-virtualized* (PV) networking mechanisms [31, 48] (Figure 1), such as virtio-net [2] and vhost-net [27]. In such nested setups, each packet must traverse a split front-end/back-end I/O stack across L_2 and L_1, incurring repeated privilege transitions and redundant data movement across virtualization boundaries, substantially inflating tail latency under high packet rates [49, 52]. This limitation is increasingly problematic for deploying latency-sensitive workloads [33], such as microservices [65], RPC-driven services [60], and AI inference [25, 69], which account for a non-trivial fraction of production datacenter workloads (e.g., 8.4% in the Google cluster trace, counting latency sensitivity classes 2 and 3 [35], and a dominant portion of the Alibaba GPU cluster trace [63]).

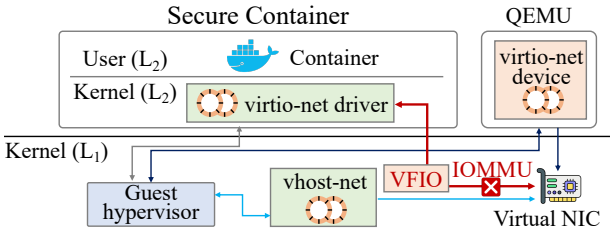


Figure 1. Network virtualization within cloud VMs

A natural alternative is direct network *device passthrough*, which has been extensively studied in conventional, single-level virtualization as an effective means of achieving higher network I/O performance than PV solutions [26, 41]. Passthrough-based designs can deliver near-native throughput by granting VMs direct access to the *network interface controller* (NIC) device. These approaches commonly rely on the *Virtual Function I/O* (VFIO) framework [1] to grant guests exclusive device control, while leveraging the *Input–Output Memory Management Unit* (IOMMU) to safely confine *Direct Memory Access* (DMA) to guest-assigned memory regions. Similar to the *Memory Management Unit* (MMU) in virtual memory translation, the IOMMU translates device-issued *I/O virtual addresses* into *host physical addresses* during DMA, enabling efficient and secure device assignment [45].

However, enabling device passthrough in nested virtualization poses fundamental challenges for IOMMU-based address translation. In nested deployments, memory virtualization spans multiple address spaces, from *host physical addresses* (HPA), to the cloud VM's *guest physical addresses* (GPA_{L_1}), and further to the nested microVM's *physical addresses* (GPA_{L_2}). Device drivers in the nested guest register IOVAs that correspond to GPA_{L_2}. In contrast, the host hypervisor (i.e., the only entity with sufficient privilege to configure the IOMMU) has visibility limited to the translation between GPA_{L_1} and HPA. This semantic gap between the address space expected by the device and the translations available on the host prevents the hypervisor from installing correct IOMMU mappings for nested device passthrough.

Jiangshan Lai, Hui Lu and Song Wu are corresponding authors. Hang Huang, Zhucan Ye, Zhuo Huang, Song Wu, Chen Yu and Hai Jin are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

To bridge this gap, IOMMU virtualization has been proposed to enable correct DMA address translation in nested virtualization. *Hardware-assisted* solutions [42], such as ARM SMMU [47], support two-stage DMA translation, in which an IOVA (or GPA_{L_2}) is first translated to GPA_{L_1}, and then from GPA_{L_1} to a HPA. However, contemporary Intel and AMD processors do not support such multi-stage address translation in their IOMMU implementations, severely limiting the practicality of this approach in mainstream cloud infrastructures. *Software-based* alternatives, including vIOMMU [11] and coIOMMU [53], emulate nested IOMMU functionality by constructing low-level IOMMU mappings that translate IOVA (or GPA_{L_2}) directly to HPA, in a manner similar to shadow paging in memory virtualization. Systems such as DVH [38] build upon host-side IOMMU emulation to enable direct connectivity between NICs exposed to L_1 and L_2, thereby shortening the L_2 network datapath. HyperTurtle [71] further extends this design by enabling L_1 to partially reclaim datapath control via eBPF filters on L_0. Xen's PV-IOMMU also enables Dom0 to manage DMA mappings through hypercalls, with planned extension to DomU [14, 15].

These approaches require privileged management of IOMMU-enforced translation state, coupling DMA translation to host-managed memory through explicit L_1–L_0 coordination. Such support—particularly vIOMMU—is an architectural and operational commitment, not a feature toggle. L_0 must emulate and secure a guest-facing IOMMU, validate untrusted mapping requests, compose nested translations, install hardware mappings, and coordinate IOTLB invalidations. These mappings must remain coherent with page pinning, reclamation, memory overcommitment, migration, and device reset; stale mappings threaten DMA isolation, while adversarial churn expands the L_0 attack surface and denial-of-service exposure [59]. This complexity has limited their deployment in public clouds.

In this paper, we present Courier, a direct network I/O architecture that enables secure containers to achieve passthrough-like access to virtio-based network devices inside cloud VMs, *without* requiring any modification to the host hypervisor (L_0). Courier 's key departure from prior IOMMU-virtualization approaches is to move DMA enforcement from L_0's IOMMU/page-table layer to L_1's virtio submission boundary. This design is enabled by a key observation: Although nested secure containers cannot directly rely on hardware IOMMU virtualization for device passthrough, the virtio interface exposes packet I/O through *software-visible* virtqueues, where each I/O request is represented as a descriptor chain that explicitly enumerates the guest physical addresses of the corresponding DMA buffers. Such virtio descriptor chains make DMA regions visible and remappable at a software-controlled point (e.g., I/O submission).

In nested deployments, these descriptors refer to GPA_{L_2} addresses, whereas the hardware IOMMU can only translate the enclosing VM's physical space (GPA_{L_1}). Courier resolves this semantic gap by interposing within the cloud VM (L_1) and providing in-guest DMA remapping semantics through a *VFIO-assisted shadow IOMMU*. Specifically, in Courier, the nested guest (L_2) continues to allocate and operate on its original virtqueue using unmodified virtio semantics. The virtio device, however, never directly consumes this L_2 virtqueue. Instead, Courier constructs a *shadow virtqueue* in L_1 memory that serves as the sole DMA-visible interface to the virtio device. Courier keeps the shadow virtqueue synchronized with the L_2 virtqueue, rewriting descriptor addresses from GPA_{L_2} into device-valid GPA_{L_1} before exposing them to the device. As a result, the device performs DMA exclusively against L_1-managed memory, enabling *zero-copy* packet delivery (to and from the L_2 nested guest) without host involvement. Beyond DMA correctness, Courier further removes virtualization overheads on the critical virtqueue interaction path. By enabling direct-kick notification, used-ring passthrough, and efficient interrupt handling, Courier eliminates cross-layer mediation and allows nested secure containers to interact with the device through a substantially shortened fast path.

We have implemented Courier in a production environment with ~1.7K lines of in-guest code changes and deployed it inside public-cloud VMs on unmodified provider hosts. Courier requires only standard virtio PCI devices exposed to L_1 and manageable through VFIO, with BAR access, interrupt delivery, and ordinary DMA to GPA_{L_1} addresses. All Courier-specific VFIO and shadow-IOMMU logic resides in L_1. By enabling passthrough-like network I/O for secure containers inside cloud VMs, Courier bridges the gap between bare-metal performance and VM elasticity. It enables cloud platforms to absorb demand surges with on-demand VMs, avoid resource waste from bare-metal over-provisioning, and sustain high performance for network-intensive, latency-sensitive workloads across heterogeneous resource pools. Extensive micro- and macrobenchmarks show that Courier outperforms virtio-net and vhost-net by up to 9.36× and 96.7%, respectively, while maintaining robust performance across a diverse set of real-world applications, demonstrating its practicality in existing cloud environments.

2 Background

2.1 Secure Containers

Cloud platforms execute massive, untrusted tenant-supplied workloads, such as *user-defined functions* (UDFs) and serverless applications [18, 62], through container-based abstractions. Conventional OS-level containers, such as RunC [56], provide fast startup and strong portability, but they rely on a shared host kernel. Although kernel mechanisms such as namespaces, cgroups, and Linux Security Modules provide logical isolation [56], the shared-kernel model remains vulnerable to privilege-escalation [54] and container-escape attacks [4–6]. A single compromised container can potentially

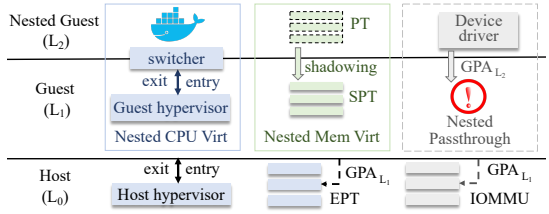


Figure 2. Host-transparent nested virtualization support

affect co-located tenants or even the host system, making OS-level containers insufficient for multi-tenant settings.

To strengthen isolation, cloud platforms have adopted secure containers, such as Kata containers and RunD [36], which encapsulate each container inside a lightweight virtual machine (microVM). Like conventional VMs, secure containers require hardware virtualization support (e.g., Intel VT-x or AMD-V), providing privilege separation between the host and the VM through root and non-root execution modes, and enable memory virtualization via second-level address translation mechanisms, such as EPT or NPT, assisting the CR3 register in translating *guest virtual addresses* (GVA) to *host physical addresses* (HPA). For I/Os, device passthrough techniques can deliver near-native performance, meanwhile the IOMMU enforces DMA isolation for microVMs. Together, these mechanisms allow secure-container runtimes to run efficiently on bare-metal servers [21], i.e., physical machines where the runtime can access hardware resources directly and use hardware virtualization to multiplex microVM-isolated secure containers from different tenants. This enables secure containers to serve as a strongly isolated execution substrate for multi-tenant cloud workloads.

2.2 Deployment within Cloud VMs

High bare-metal performance alone does not make bare-metal deployment a practical default for secure containers in public clouds. Our empirical measurements on Alibaba Cloud show that provisioning a large bare-metal server can take 232.4 seconds, over 10.4x longer than a resource-equivalent VM instance at 22.4 seconds and 12.2x longer than a smaller VM instance; bare-metal requests may also fail when regional capacity is insufficient [40]. These provisioning delays and capacity constraints motivate *deploying secure containers inside cloud VMs*, which offer faster elasticity, better cost efficiency, and flexible tenant-managed orchestration, such as Kubernetes-on-VM [10].

This results in a three-layer nested virtualization stack: the host (L_0), the cloud VM (L_1), and the nested microVM (L_2). In conventional nested virtualization, the L_0 hypervisor virtualizes hardware virtualization features for the L_1 guest hypervisor, allowing L_1 to create and manage L_2 microVMs/secure containers. Many L_1 virtualization operations, such as VMX control operations and nested address-translation updates, must be trapped, validated, translated, or emulated by L_0 before their effects can be enforced on the underlying hardware. As the L_0 host must remain on critical execution paths, including VM entry and exit, the L_0-mediated nested virtualization incurs substantial performance overhead, Moreover, enabling nested virtualization in production environments requires extensive modifications – e.g., over 10k lines of code – to the L_0 hypervisor, increasing cloud infrastructure complexity. As a result, although major cloud providers widely adopt KVM as the L_0 hypervisor, its nested virtualization support is generally disabled for cloud VMs.

To overcome this limitation, state-of-the-art guest hypervisors, such as PVM [29], introduce an efficient software-based virtualization layer within cloud VMs (L_1), enabling immediate secure container deployment on production cloud platforms (including Alibaba Cloud) – without relying on nested virtualization support from the L_0 host hypervisor. In Figure 2, the L_1 guest hypervisor emulates VMX functionality for L_2 guests through a software component, known as *the switcher*, which resides in both the L_1 and L_2 address spaces and coordinates VM entry and exit transitions. Since modern hardware exposes only a single level of root and non-root privilege separation, nested guests/secure containers run in non-root mode (ring 3) to enforce strict privilege isolation. To support nested memory virtualization, the L_1 guest hypervisor further employs a shadow paging mechanism. Rather than loading the L_2 page table directly into the hardware CR3 register to translate GVA_{L_2} into GPA_{L_2}, the CR3 register instead points to a *shadow page table* (SPT) that mirrors the L_2 page table while directly translating GVA_{L_2} to GPA_{L_1}. This design enables correct physical memory access for nested secure containers without invoking the L_0 host.

2.3 Para-virtualized Networking

Despite advances in nested CPU and memory virtualization through virtualized VMX support and nested address translation, network I/O remains a major unresolved bottleneck in nested secure container deployments.

In today's cloud environments, nested secure containers primarily rely on *para-virtualized* (PV) networking, most notably virtio-net [2] and vhost-net [27]. As shown in Figure 1, they adopt a split front-end/back-end architecture, where the front-end network driver executes inside the guest, while the back-end network device resides in the hypervisor or host kernel and interacts with the guest through shared memory regions defined by the virtio specification. This software-centric design does not rely on specialized hardware virtualization features, such as IOMMU-assisted DMA remapping, thereby providing strong deployment flexibility.

The core data-transfer mechanism of virtio devices is *virtqueue* – i.e., a shared-memory ring buffer that enables asynchronous communication between the guest driver and the host device. We focus our discussion on a *split virtqueue* design, as it is the dominant deployment model in major cloud environments that conforms to the virtio 1.0 specification. A split virtqueue, organized in the form of a vring, consists of three key data structures in guest memory: (1) a

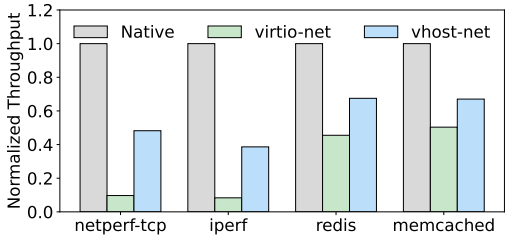


Figure 3. Para-virtualized networking overhead

descriptor table, which is an array describing buffer addresses and associated metadata; (2) an available ring, maintained by the device driver to indicate descriptors that are ready for device consumption; and (3) a used ring, maintained by the device to return descriptors that have been processed.

This vring-based interface underpins the two widely used virtio devices – virtio-net [2] and vhost-net [27]. Both adopt similar control-plane mechanisms for feature negotiation, queue initialization, and interrupt delivery, while diverging primarily in their data-plane processing paths. For virtio-net, during packet transmission (TX), the device driver enqueues buffer descriptors into the available ring and invokes `virtqueue_kick` to notify the backend device through a virtual I/O write. This notification triggers a VM exit and transfers control to QEMU in userspace. QEMU subsequently retrieves packet data from guest memory, performs data copy, and forwards the packets to the host network stack. This data path incurs substantial overhead, as each packet typically induces a VM exit, a context switch into the QEMU process, and multiple memory copies across guest memory, QEMU userspace, and the host kernel space, ultimately resulting in limited throughput and increased latency. In contrast, vhost-net mitigates this overhead by relocating the backend into the host kernel. After initialization, a dedicated vhost-net kernel thread directly polls the virtqueues. On the TX path, a guest notification still triggers a VM exit; yet, packet processing is performed entirely in kernel space, allowing packet data to be copied directly from guest memory to the host backend, eliminating QEMU involvement.

3 Motivation

While PV networking enables network connectivity for secure containers running inside cloud VMs, its performance remains far from optimal, limiting practical adoption. To quantify the associated performance degradation, we measure the network throughput of Kata Containers (running on the PVM hypervisor) using either virtio-net or vhost-net within a cloud VM, and compare the results with conventional RunC containers that directly utilize the host network stack (denoted as Native). Our evaluation (Figure 3) reveals that both virtio-net and vhost-net incur significant throughput losses relative to native containers, with virtio-net experiencing up to 91.7% degradation and vhost-net up to 61.4%.

3.1 Limitation: Missing Nested Passthrough

A natural alternative to PV networking is to use direct device passthrough, which can deliver near-native performance in single-level virtualization – by allowing a guest to directly access the NIC through VFIO [1], while relying on the IOMMU to safely confine the DMA. Unfortunately, passthrough becomes fundamentally problematic in nested virtualization: In a nested stack, memory spans multiple address spaces: *host physical addresses* (HPA), the cloud VM's *guest physical addresses* (GPA_{L_1}), and the nested microVM's *physical addresses* (GPA_{L_2}). Device drivers inside L_2 register DMA buffers using GPA_{L_2} addresses, yet the hardware IOMMU – managed exclusively by L_0 – only provides translation for GPA_{L_1}. This semantic mismatch prevents the host from installing correct DMA mappings for nested guests, rendering conventional VFIO passthrough impractical.

Hardware-assisted IOMMU virtualization [42] could, in principle, support multi-stage DMA translation, but such capabilities are largely unavailable on mainstream cloud servers. Software-based approaches such as vIOMMU [11] emulate nested DMA translation, but require tight coordination with the L_0 hypervisor and conflict with key cloud features such as memory overcommitment. Consequently, modern cloud providers do not expose IOMMU capabilities to tenants, making direct passthrough still infeasible for nested secure containers.

3.2 Opportunity: In-Guest DMA Remapping

After revisiting the shadow paging design of nested memory virtualization, which enables the correct generation of GPA_{L_1} (from GVA_{L_2}) for low-level EPT translation and thereby eliminates L_0 involvement, we identify a key opportunity. In a similar vein, *DMA remapping* can be achieved by first translating IOVA (GPA_{L_2}) into device-acceptable GPA_{L_1} before delivering it to the device, since L_2 memory is physically part of L_1 memory and can be accessed via the corresponding GPA_{L_1}. As a result, the low-level backend in L_0 only needs to manage DMA mappings for L_1 without concerning itself with the internal IOVA from nested guests. Because DMA regions are typically defined within virtqueues (according to the virtio specification), capturing and translating them for DMA remapping becomes feasible. Therefore, by providing in-guest DMA remapping semantics analogous to IOMMU, this approach can enable direct nested network passthrough.

Realizing this opportunity, however, introduces several non-trivial challenges in DMA management. First, it is a must to ensure that DMA requests (delivered to the device) align exactly with L_2's expectations. Second, the interactions between nested guest device drivers and the device, such as kicks and completion notifications, must be correctly maintained. Third, the approach must be compatible with the existing memory management mechanisms of secure containers. To this end, we present Courier, our solution for

addressing these challenges and enabling high-performance, direct network I/O for nested secure containers.

3.3 Threat Model

Courier operates entirely within cloud VMs (L_1) without requiring any awareness or modification of the host (L_0), enabling seamless deployment in public clouds while preserving the integrity and security properties of the underlying infrastructure. As Courier targets in-guest secure container deployments for cloud platforms, we assume that the guest VM (L_1) is trusted and managed by the platform and orchestration frameworks (e.g., Kubernetes), whereas untrusted user workloads execute inside secure containers (L_2). Accordingly, the entire L_2 environment, including both user space and kernel space, is considered potentially malicious and may attempt to compromise the shared L_1 guest VM.

Our focus is on software-level vulnerabilities in the L_1 software stack that may be exposed to malicious L_2 through nested virtualization interfaces. Since Courier does not modify existing nested CPU or memory virtualization mechanisms in modern guest hypervisors, it naturally inherits their CPU and memory protection guarantees. Accordingly, we consider a threat model in which a potentially malicious L_2 attempts to compromise the shared L_1 through the direct network I/O path introduced by Courier. This threat model fundamentally motivates the careful design of both Courier's control plane and data plane. Passthrough devices are managed through an in-guest shadow IOMMU, which preserves isolation from L_1 while enabling safe DMA access to L_2 memory using translated L_1 physical addresses (§4.2). To defend against malicious DMA requests from L_2, such as accesses to invalid or unauthorized addresses, Courier strictly validates each DMA mapping before it becomes visible to either the L_1 shadow virtqueue or the device. Subsequent L_2 updates to its descriptor cannot interfere with in-flight DMA operations. To minimize risks arising from memory sharing for direct I/O support, Courier maps only the minimal required L_1 memory into L_2 with *read-only* permissions, ensuring that no sensitive L_1 state is exposed (§4.3). In addition, Courier integrates software IOTLB invalidation to ensure that memory reclamation in L_1 cannot interfere with in-flight DMA operations (§4.4).

4 Design and Implementation

4.1 Overview: Courier Architecture

Figure 4 shows the architecture of Courier. Courier enables a network-intensive L_2 secure container to achieve passthrough-like direct access to its virtio-net device, without requiring any IOMMU virtualization support from the L_0 host. The key idea is to interpose at L_1 and provide in-guest DMA remapping by maintaining a *shadow virtqueue* that mirrors the L_2 virtqueue while rewriting descriptor addresses from GPA_{L_2} to device-valid GPA_{L_1}. As a result, the virtio

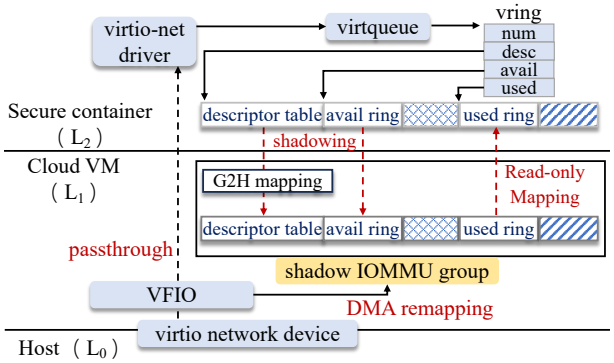


Figure 4. Courier architecture

device performs DMA solely against L_1-managed memory, while the L_2 driver continues to operate on an unmodified virtqueue interface. This design eliminates the multi-layer mediation/synchronization overhead of virtio-net/vhost-net and enables near-native network performance for secure containers in production cloud VMs.

Courier, functioning within the cloud VM (L_1), comprises (i) an L_1 shadow IOMMU layer integrated with VFIO to maintain guest-to-host DMA mappings, and (ii) lightweight hooks in the L_2 virtio-net frontend to propagate virtqueue updates. The device never directly consumes the L_2 vring; instead, Courier presents a shadow vring in L_1 memory as the DMA contract with the virtio device. Building on this mechanism, Courier introduces three optimizations to approach true direct I/O performance: direct-kick notification to bypass user-level mediation, used-ring passthrough to eliminate completion copying, and a software IOTLB flush mechanism to safely support dynamic memory reclamation.

4.2 Shadow IOMMU

Courier enables a nested secure container (L_2) to interact with its virtio-net device through a passthrough-like path – by introducing a shadow IOMMU mechanism inside the L_1 guest that provides DMA remapping semantics without requiring any support from the L_0 host.

As discussed earlier, the L_2 driver registers DMA buffers using GPA_{L_2} addresses, while the device can only issue DMA requests against addresses valid in the L_1 physical space (GPA_{L_1}). Courier bridges this mismatch by maintaining a shadow virtqueue and continuously translating L_2 DMA-visible addresses into device-consumable L_1 addresses. Supporting this model requires coordinated changes in both the control plane (i.e., queue setup and DMA mapping) and the data plane (i.e., descriptor submission and completion).

Control Plane. The shadow IOMMU at L_1 must remain synchronized with all DMA-related operations initiated by the L_2 virtio-net frontend.

Figure 5 illustrates the Courier's control-plane workflow. When an L_2 virtio-net driver probes a device, it issues standard virtio control operations, including virtqueue creation and configuration, followed by I/O writes to the PCI BAR

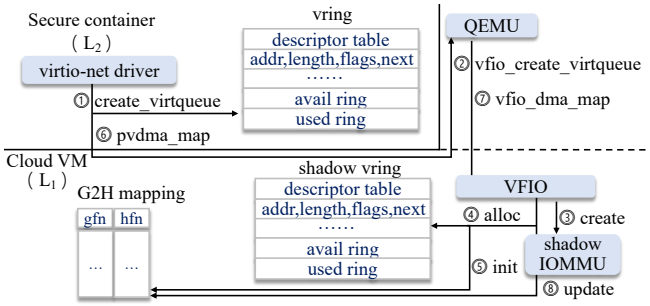


Figure 5. Courier control plane

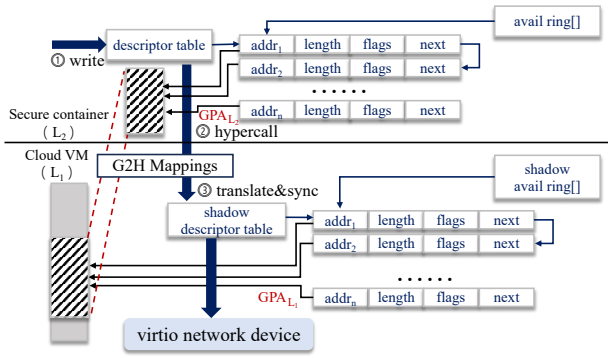


Figure 6. Courier data plane with shadow vring

to apply the changes (①). These operations are *intercepted and redirected* to the control plane at L_1. In particular, upon receiving a `create_virtqueue` request from L_2, Courier allocates a corresponding shadow virtqueue in L_1-managed memory. This allocation is performed through QEMU, which forwards the request to VFIO via ioctl interfaces (②). VFIO is then responsible for maintaining the shadow IOMMU (③), which includes creating the corresponding shadow vring (④) that mirrors the L_2 vring – including the descriptor table, available ring, and used ring – and establishing the *guest-to-host* (G2H) mapping (⑤) to track DMA regions from GPA_{L_2} to GPA_{L_1} for subsequent DMA translation.

To avoid eagerly pinning all guest pages, Courier follows the on-demand DMA mapping model used in production systems. When L_2 invokes `pvdma_map`, the operation exits to the VMM to dynamically pin the required pages and call `vfio_dma_map` to update IOMMU mappings (⑥). Courier intercepts these mapping requests and forwards them to the shadow IOMMU maintained by VFIO, where G2H mappings are updated based on GPA_{L_2} and the corresponding GPA_{L_1} (⑦-⑨). As a result, the shadow IOMMU consistently maintains the necessary G2H mappings, ensuring that subsequent data-plane DMA operations can be safely and efficiently translated to device-visible L_1 physical addresses.

Data Plane. While the control plane establishes virtqueues and maintains correct DMA mappings, Courier must also correctly multiplex DMA requests issued by the virtio-net driver during data-plane operations. To achieve this, Courier leverages the shadow IOMMU to synchronize virtqueue state across layers and ensure DMA correctness.

Courier allows the L_2 virtio-net driver to continue operating on its native vring structures, where each descriptor contains GPA_{L_2} addresses, thus remaining compatible with the virtio programming model. However, the device never directly consumes the L_2 vring. Instead, Courier presents the device with the shadow vring in L_1 memory as the authoritative DMA contract. As illustrated in Figure 6, updates to the descriptor table and available ring – both written by the L_2 driver – are intercepted during virtqueue kicks and propagated into the shadow vring at L_1 (② and ③).

To prevent TOCTOU attacks [28], Courier makes an L_1-private snapshot the sole input to descriptor validation, address translation, and shadow-vring construction; it never validates descriptors in mutable L_2 memory and later rereads them. Each captured chain is checked for bounds, alignment, and structural consistency. For every descriptor, Courier verifies that the complete buffer range lies within pinned G2H mappings that permit the requested DMA access. A chain captured during an L_2 race remains untrusted and is accepted only if the captured state passes the same checks.

After validation, Courier rewrites buffer addresses from GPA_{L_2} to device-valid GPA_{L_1} addresses. If an L_2 buffer spans non-contiguous L_1 pages, Courier splits it into multiple physically contiguous shadow descriptors. Only descriptors derived from the validated snapshot are published to the device-visible shadow vring. Consequently, later modifications to the original L_2 descriptors cannot affect in-flight DMA, matching hardware descriptor-fetch semantics.

The device performs DMA directly against the underlying L_2 memory through validated GPA_{L_1} addresses, without copying packet data. By combining snapshot-based validation, address rewriting, and shadow-vring synchronization, Courier preserves unmodified virtio semantics while decoupling the data plane from nested-virtualization constraints.

4.3 Direct Network I/O Enhancement

While the shadow IOMMU enables correct passthrough-style DMA for nested secure containers, it does not by itself eliminate all virtualization overhead on the critical I/O path.

In particular, virtio devices still rely on frequent queue notifications, completion propagation, and interrupt coordination, which can introduce excessive world switches between L_2, the L_1 hypervisor, and the user-level VMM. To approach true direct network I/O performance, Courier introduces three complementary techniques: direct-kick notification, used-ring passthrough, and shadow-flag passthrough. These mechanisms together shorten the virtqueue fast path, eliminate redundant exits, and reduce completion overhead.

Direct kick. In a conventional virtualized setup, the L_2 guest driver is not trusted to directly access device registers. Hence, writes to the PCI BAR are mediated through the hypervisor and the VMM, where such accesses are emulated. Although most PCI BAR `iowrites` occur during device initialization and are therefore infrequent, the *kick* operation lies on the

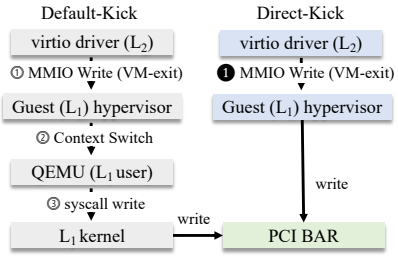


Figure 7. Default-kick vs direct-kick

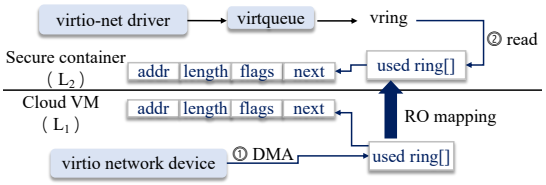


Figure 8. Used-ring passthrough

critical data path. A kick simply writes the updated queue index to a device register to notify the device of newly available descriptors; it is issued at a high frequency.

Under the default virtio execution model, each kick incurs multiple world switches (Figure 7, left). The request propagates from the L_2 guest to the L_1 kernel, then to the L_1 user-space VMM, and finally back to the L_1 kernel – incurring *three* world switches – before the PCI BAR iowrite is performed. Despite the simplicity of the operation, this process introduces substantial overhead.

Courier shortens this path (Figure 7, right) by exploiting the trust boundary provided by the guest hypervisor [1]. It constructs a redundant mapping for the PCI BAR within the guest hypervisor address space. This mapping enables the hypervisor to directly translate a guest virtqueue kick into the corresponding shadow virtqueue kick. As a result, the kick operation can be completed by writing to the PCI BAR in the hypervisor execution context, without trapping into the user-space VMM or triggering additional world switches. This approach substantially reduces the latency and overhead of virtqueue notifications, bringing the kick path closer to that of native device passthrough, with only a single exit from the nested guest to the guest hypervisor.

Used-ring passthrough. Courier enables the device to DMA packet buffers directly into L_2 memory; a remaining challenge is how the L_2 driver observes request completions.

In virtio, completion metadata is returned through the *used ring*. In the shadow vring design, the device updates the shadow used ring maintained at L_1, while the used ring in L_2 memory is never directly consumed by the device. As a result, the L_2 driver cannot directly detect completed requests.

A naive approach would be to reuse the guest used ring as the shadow used ring. However, this approach violates a fundamental virtio requirement that the descriptor table, available ring, and used ring reside in physically contiguous memory. Sharing only the used ring would therefore break the virtio memory layout invariants. An alternative is to copy entries from the shadow used ring back to the guest used ring at well-defined synchronization points, such as upon interrupt delivery. However, this would reintroduce additional exits, memory copies, and synchronization complexity – precisely the overheads Courier aims to avoid.

[1]We adapt the production-grade PVM [29] guest hypervisor for nested CPU and memory virtualization.

Courier instead observes that the used ring contains only completion indices and does not expose sensitive buffer contents or DMA addresses. Leveraging this insight, Courier exposes the shadow used ring directly to the L_2 guest through a *read-only* mapping in the guest MMU page tables. As shown in Figure 8, the L_2 vring is modified to repoint its used-ring reference to the passthrough shadow used ring, rather than its native used ring. This design allows the L_2 driver to directly observe device completions using its standard virtqueue and vring semantics, without copying or trapping, thereby substantially reducing completion-path overhead.

Shadow flags passthrough. Virtio drivers commonly use the *flags field* in the available ring to control interrupt behavior. By setting or clearing these flags, the driver can instruct the device to suppress or resume interrupts around batched request processing. However, when the queue is empty, toggling these flags has no observable effect on device behavior.

With shadow vring, writes to the available-ring flags would normally trap into L_1 so that the corresponding shadow-ring flags in shadow vring can be updated and propagated to the device. To avoid unnecessary VM exits in common cases, Courier exposes the shadow flags directly to the L_2 driver. The driver applies a simple technique: when setting the flags, it first checks whether the queue is empty; when clearing the flags, it checks whether the shadow flags are already cleared. If no effective state change is required, the operation is skipped. By allowing the guest to access shadow flags directly, Courier eliminates redundant traps on the interrupt control path while preserving correct synchronization between the L_2 driver and the device.

Preserving isolation. Crucially, these fast-path optimizations do not weaken Courier 's isolation boundary. Courier maps only the shadow used ring and the flags field of the shadow available ring into L_2, both read-only; all remaining shadow-queue state and G2H mappings stay private to L_1. The former exposes only completion metadata, while the latter reveals only the current interrupt-suppression state, allowing L_2 to avoid redundant traps without modifying the flags directly. L_1's memory-virtualization layer enforces these permissions independently of L_2's page tables. Courier thus exposes the minimum state needed to accelerate the fast path while preventing a malicious L_2 from forging completions or corrupting device-visible queue state.

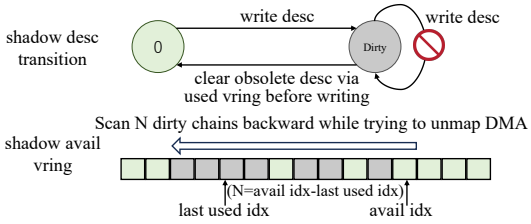


Figure 9. Software IOTLB flush

4.4 Memory Elasticity Management

To improve resource utilization, cloud platforms often reclaim or dynamically resize guest memory to support memory overcommitment. When reclaimed DMA regions have previously been exposed to a passthrough device for DMA access via `pvdma_unmap`, the system must ensure that the device can no longer access such memory, as failing to do so could lead to data corruption or compromise isolation. Since hardware-level IOTLB support is typically unavailable to enforce unmapping correctness, Courier introduces a software-based IOTLB flush mechanism (Figure 9) to safely handle `pvdma_unmap` operations and maintain DMA safety.

In Courier, potential DMA targets are recorded in the shadow descriptor table. Before reclaiming a memory region, Courier first removes the corresponding guest-to-host (G2H) mappings from the shadow IOMMU, preventing any future DMA requests from reaching the reclaimed pages. However, this step alone is insufficient, as the device may still hold outstanding requests referencing descriptors that have already been posted in the shadow descriptor table.

Courier enforces that the final unmapping action can only be performed after all in-flight descriptors have been scanned and confirmed not to reference memory intended for unmapping. Scanning the entire shadow descriptor table to identify in-flight requests is impractical, because virtio descriptor tables are circular and continuously reused. To address this issue, Courier introduces a descriptor sanitization mechanism that incrementally tracks descriptor liveness. Before inserting a new descriptor into the shadow descriptor table, Courier clears descriptors that have already been consumed by the device, as indicated by the shadow used ring. New descriptors are written only to descriptor slots that have been explicitly zeroed. As a result, any remaining non-zero descriptors correspond precisely to requests that have not yet been consumed by the device.

In addition to limiting the scan to dirty descriptors, Courier further narrows the search using the used and available indices. The used index delimits descriptors whose completions have been reported, whereas the available index delimits those submitted to the device. Courier therefore scans only the outstanding interval between these indices.

This interval remains conservative under both device-side prefetching and concurrent completion. A descriptor fetched by the device but not yet reported in the used ring remains within the interval because the used index has not advanced past it. If a completion races with the scan, the captured used index may cause a newly completed descriptor to be scanned unnecessarily, but it cannot exclude any descriptor still in flight. Thus, the scan may conservatively retain a page but cannot release one still accessible to the device.

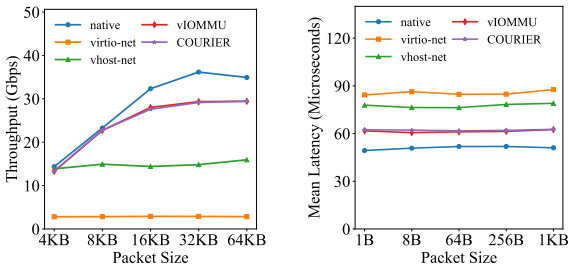
If any outstanding descriptor references the target region, Courier rejects the `pvdma_unmap` request and terminates the L_2 guest. Because termination alone does not cancel previously issued DMA, Courier also blocks subsequent kicks and keeps the affected pages pinned until the device is reset or confirmed quiescent. Only then can the pages be safely released. By combining proactive G2H invalidation, conservative descriptor-liveness tracking, and reset-aware page release, Courier supports dynamic memory reclamation without weakening DMA isolation.

5 Evaluation

We implemented Courier with little in-guest changes. The L_2 updates involve lightweight kernel patches (8 files changed, 230 insertions) to interact with the shadow IOMMU, as well as adjustments to the virtio datapath (4 files changed, 67 insertions). The L_1 implementation of Courier is integrated into the guest hypervisor (PVM) as a self-contained module (8 files changed, 1389 insertions, 3 deletions), providing shadow queue management, address translation, and DMA safety enforcement. Note that no modifications are required at L_0. Although our prototype is built on the Alibaba PVM guest hypervisor, the design can be ported to KVM guest hypervisor when deployed in VMs supporting nested virtualization (e.g., Google Cloud).

Testbed and configurations. All experiments were conducted on a dual-socket Intel(R) Xeon(R) Platinum 8369B server with 128 physical cores (2.90 GHz) and 256 GB of RAM across two NUMA nodes, running Linux kernel 5.10 (i.e., the primary kernel version in our production environment) across L_0, L_1, and L_2. Notably, we also implemented Courier to support Linux kernel 6.6 for broader production deployment. Each L_1 VM was instantiated using QEMU with 32 vCPUs and 128 GB of memory, leveraging KVM hardware-assisted virtualization. The L_1 VMs were equipped with *four* virtio-net PCI devices implemented via host VFIO-based passthrough with pre-established IOMMU mappings. To enable a direct comparison with existing IOMMU virtualization approaches, we attached *four* additional virtio-IOMMU PCI devices when evaluating vIOMMU-based device passthrough. Secure containers were provisioned using Kata Containers 3.0, typically configured with 8 vCPUs and 16 GB of memory unless otherwise specified. Client requests were issued from the L_0 host and processed by servers running in L_1 or L_2. *Transparent Huge Pages* (THP) were enabled in L_1 to reduce address translation overhead.

Baselines for Comparison. We evaluated five configurations to characterize the impact of nested network virtualization and quantified the benefits of Courier: (1) **native** runs



(a) TCP throughput (b) UDP latency

Figure 10. Network performance measured by Netperf

RunC containers directly on the L_1 kernel network stack without nested virtualization; (2) **virtio-net** deploys secure containers (L_2) using standard virtio-net paravirtualized networking; (3) **vhost-net** enables vhost-net offloading for secure containers; (4) **vIOMMU** uses vIOMMU-based network device passthrough [2]; (5) **Courier** provides direct network passthrough for secure containers. These baselines isolate nested-virtualization overheads and compare paravirtualized with direct network I/O. We treat vIOMMU as an L_0-assisted performance oracle rather than a tenant-deployable baseline: once L_0 installs GPA_{L_2}-to-HPA DMA mappings, L_2 submits GPA_{L_2} addresses directly, yielding a near-optimal datapath. All results are averaged over 10 runs.

5.1 End-to-End Network Performance

We first evaluated Courier's end-to-end network performance using Netperf and iperf3, measuring throughput and latency across diverse packet sizes and workloads.

Varying packet size. We varied TCP packet sizes from 4 KB to 64 KB and UDP payloads from 1 B to 1 KB. As shown in Figure 10, virtio-net and vhost-net fail to scale with packet size due to persistent per-packet overheads from cross-world queue notifications and indirect memory accesses. These overheads dominate in high-throughput regimes and ultimately limit effective bandwidth utilization. Vhost-net approaches native throughput at packet size of 4 KB due to virtqueue batching, but this benefit diminishes at larger packet sizes as other para-virtualization overheads become the dominant bottleneck. In contrast, native, vIOMMU, and Courier scale with packet size until saturating physical bandwidth, benefiting from direct network I/O.

Courier improves the TCP throughput by up to 9.36x over virtio-net and 96.7% over vhost-net, while remaining within 3% of vIOMMU – *without requiring any L_0 support.* Compared to native, Courier incurs at most a 19.4% throughput reduction due to the shadow virtqueue update associated with kick operations. This degradation reflects the inherent trade-off between performance and isolation in secure containers – enforcing DMA isolation requires DMA remapping, and cross-world interception and validation introduce additional overhead. Nevertheless, Courier achieves substantially higher performance than PV networking approaches. Meanwhile, the UDP latency remains largely insensitive to packet size, with Courier reducing latency by up to 28.5% and 20.7% over virtio-net and vhost-net and achieving performance comparable to vIOMMU.

[2]Note that vIOMMU depends on support from the L_0 hypervisor and can interfere with cloud features such as memory overcommitment. Therefore, cloud providers typically disable vIOMMU support for cloud VMs.

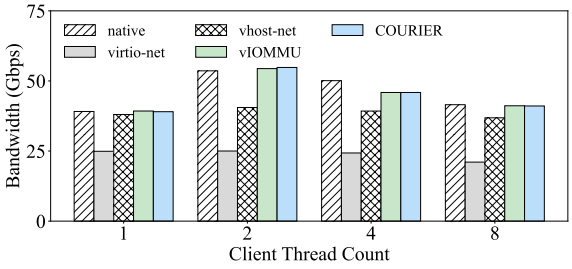


Figure 11. Network bandwidth measured by iperf3

Overall, these results demonstrate that Courier provides a practical alternative to vIOMMU with comparable performance while avoiding additional complexity in L_0.

Varying network workloads. We further evaluated the network bandwidth under varying network loads using `iperf3`. Each experiment was run for 10 seconds, and the number of concurrent client threads was varied to simulate varying levels of network loads. As shown in Figure 11, the bandwidth does not increase monotonically with the workload in all configurations. Instead, peak bandwidth is consistently achieved with two client threads, after which additional threads lead to performance degradation. This behavior stems primarily from the single-threaded design of the iperf3 server, which limits its ability to efficiently handle higher concurrency. At the optimal point of two client threads, Courier outperforms virtio-net and vhost-net by 119.2% and 35.3%, while achieving performance comparable to native and vIOMMU. Beyond this point, Courier continues to maintain competitive throughput. The modest performance degradation under higher concurrency indicates that Courier introduces minimal additional contention in the critical data path.

In summary, these results demonstrate that Courier effectively supports high network loads and preserves bandwidth scalability comparable to native and vIOMMU, without introducing additional scalability bottlenecks.

5.2 Real-world Applications

We then evaluated Courier using Redis and Memcached, two widely deployed in-memory key–value stores. Redis was evaluated using `redis-benchmark`, which issues one million read requests, each accessing a 256 byte value. Memcached was evaluated using `memaslap`, which generates requests with a get/set ratio of 0.9, 64-byte keys, and 1 KB values for 10 seconds using eight workers. Both workloads were configured to be read-intensive in order to minimize interference from memory virtualization effects. We varied the number of concurrent clients to assess throughput scalability.

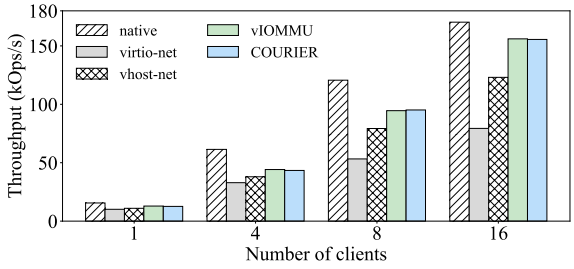


Figure 12. The performance of Redis

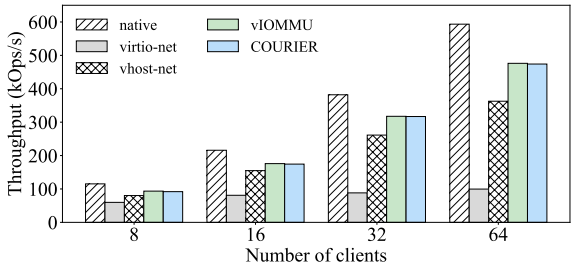


Figure 13. The performance of Memcached

As shown in Figures 12 and 13, the throughput increases steadily with the number of concurrent clients across all evaluated configurations. Nevertheless, virtio-net exhibits markedly poor scalability. Its throughput increases only marginally as the client count grows, which indicates the existence of substantial bottlenecks in the PV network stack. These limitations primarily stem from the data-plane processing overhead introduced by the user-level VMM, which restricts efficient packet handling under high concurrency. In contrast, Courier delivers significant performance improvements even at low concurrency levels. Courier improves the throughput by 24.5% for Redis with 1 client and by 53.6% for Memcached with 8 clients. As concurrency increases, these advantages become more pronounced. Throughput improves by 95.93% for Redis with 16 clients and by 3.76x for Memcached with 64 clients. This trend indicates that Courier effectively mitigates contention in the VMM network data path and sustains scalable performance as request parallelism increases. Vhost-net achieves higher throughput than virtio-net due to kernel-level data-plane offloading, but it still suffers from a longer network stack and multiple cross-world data copies, which introduce increasing overhead as request rates grow. Courier delivers up to 26.4% higher throughput for Redis with 16 clients and 30.7% higher throughput for Memcached with 64 clients.

CPU utilization of L_1 guest confirms that Courier 's throughput gains do not incur higher CPU costs. With one Redis client, Courier uses 183.44% CPU, comparable to virtio-net (181.79%) and vIOMMU (183.52%), but 14.8% lower than vhost-net (215.26%). With eight Memcached clients, Courier uses 257.87% CPU, comparable to virtio-net (257.58%) and vIOMMU (250.32%), but 23.9% lower than vhost-net (338.79%). Although kernel-level data-plane offloading improves vhost-net throughput over virtio-net, it incurs substantially higher CPU utilization. In contrast, Courier outperforms both paravirtualized baselines while retaining virtio-net-level CPU efficiency.

Overall, Courier consistently delivers higher throughput across a wide range of client counts. These results demonstrate that Courier effectively supports scalable, network-intensive applications while maintaining performance competitive with L_0-assisted IOMMU virtualization mechanisms.

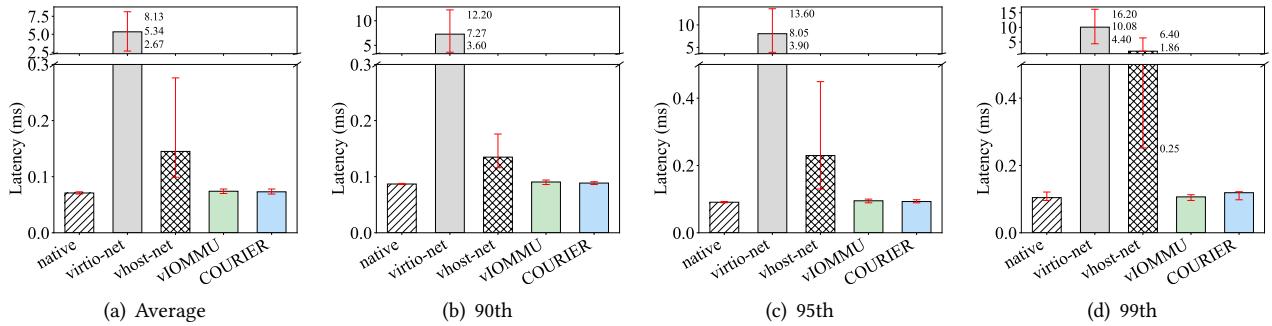
5.3 Cloud Workloads

We also evaluated Courier under realistic cloud workloads. We used the Data Caching benchmark from CloudSuite [24], which models a production-grade social-network caching service. The benchmark is built on Memcached and replays request traces derived from a Twitter dataset. It follows a two-tier architecture consisting of one or more Memcached servers and a set of clients issuing cache requests. The cache is first populated during a warm-up phase to establish a steady-state working set, after which clients execute a read-intensive phase (a get/set ratio of 0.8) for 20 seconds under a fixed *service-level objective* (SLO) to stress networking performance. Experiments were conducted using both single-server and four-server deployments to evaluate scalability and interference effects. Notably, in the four-server deployment, each server was hosted in a separate secure container with a distinct network device. During execution, we recorded average latency as well as tail latencies.

As shown in Figures 14 and 15, in single-server and four-server deployments, virtio-net and vhost-net exhibit substantial latency jitter. First, virtio-net fails to satisfy the 1 ms QoS requirement across all latency metrics, while the 99th-percentile latency of vhost-net still significantly exceeds the QoS threshold. These effects arise from the deeper network stack and additional data copies inherent in PV networking, which amplify the tail latency under load. In contrast, native, vIOMMU, and Courier consistently meet the 1 ms latency target in the single-server configuration.

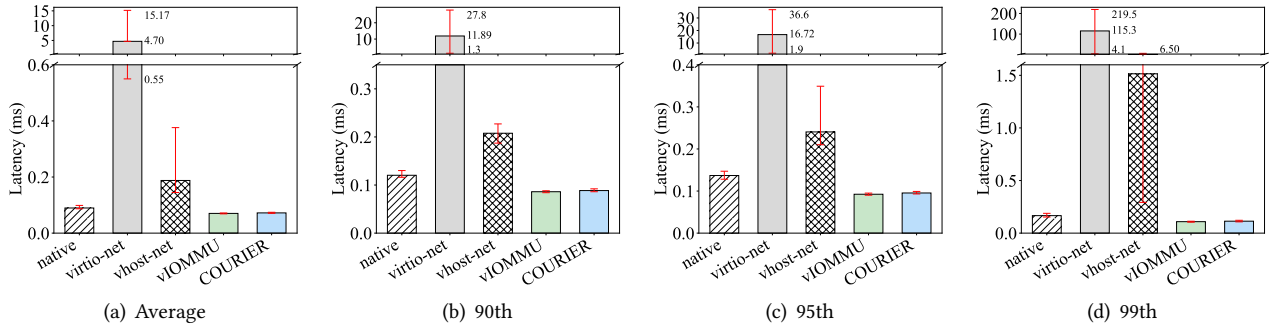
Scaling to four servers further exposes the limitations of PV networking. The 99th-percentile latency of virtio-net increases from 10.08 ms to 115.3 ms with substantially greater variability. Vhost-net likewise exhibits higher average, 90th-, and 95th-percentile latencies, while its 99th-percentile latency remains above the SLO. These results indicate that contention and cross-layer processing overhead grow as multiple virtualized network stacks execute concurrently.

Native RunC also degrades in the four-server deployment: its 99th-percentile latency increases by 60%, as insufficient isolation permits interference among co-located servers [43]. In contrast, Courier maintains stable average and tail latencies as the deployment scales, outperforming native RunC by up to 31.7% while remaining comparable to vIOMMU. Overall, Courier avoids both PV-networking scalability bottlenecks and cross-container interference, providing strong isolation with consistently low tail latency.



(a) Average (b) 90th (c) 95th (d) 99th

Figure 14. The performance of data caching with one server



(a) Average (b) 90th (c) 95th (d) 99th

Figure 15. The performance of data caching with four servers

5.4 Ablation Study

Finally, we examined packet-processing overhead to understand how Courier's design reduces per-packet cost by first quantifying the impact of Courier's mediation-removal mechanisms and then decomposing the transmit path to attribute overhead to individual processing components.

Mediation removal. We evaluated the impact of Courier's mechanisms for eliminating mediation on the virtqueue interaction path. As described in §4, Courier's shadow IOMMU and shadow virtqueue with used-ring passthrough construct the correctness foundation for passthrough-style DMA and constitute the `base` configuration. Accordingly, we focused on the remaining two mechanisms – direct kick (`kick`) and efficient interrupt management (`intr`) – and evaluated their impact by enabling each mechanism individually relative to the base. We used Netperf to measure TCP throughput and UDP latency, with packet sizes of 16 KB and 1 B.

As shown in Table 1, direct kick significantly improves performance by bypassing user-level VMM and VFIO redirections, increasing TCP throughput by 143.4% and reducing mean UDP latency by 40.2%. Efficient interrupt management further eliminates unnecessary exits from L_2 to L_1, yielding an additional 20.9% throughput improvement and a 13.9% latency reduction. When combined, the two mechanisms improve the TCP throughput by 166.4% and reduce the mean UDP latency by 44%. Overall, these results demonstrate that direct kick and interrupt management play complementary and important roles in sharpening the critical direct I/O path.

Table 1. Effectiveness of Courier Optimizations

Configuration	Throughput (Mbps)	Mean Latency (μs)
base	9962.9	112.2
base + kick	24248.8	67.0
base + intr	12048.5	96.66
base + kick + intr	26543.3	62.8

Table 2. Breakdown of per-packet overheads (in ns)

	`Guest syscall`	`sys_sendto (start_xmit)`	`Host syscall`	`handle_tx`	Total
native	0	1713 (707)	103	0	1816
virtio-net	268	3486 (2137)	103	3131	6988
vhost-net	268	3214 (2110)	0	3131	6613
vIOMMU	268	1980 (822)	0	0	2248
Courier	268	2464 (1233)	0	0	2732

To validate the correctness of these optimizations, we ran Netperf continuously for 24 hours while periodically cycling the memory size between 4 GB and 16 GB to emulate elastic memory management. Netperf ran uninterrupted, with no failures observed in either L_1 or L_2. Analysis of L_1 runtime behavior confirms that software IOTLB flush correctly handles DMA mappings. We therefore enabled all optimizations for Courier in all subsequent experiments.

Cost breakdown. To understand how much cost each key component contributes along the critical path, we quantified per-packet processing overhead using a UDP workload with

a 1-byte payload. Table 2 shows a breakdown of the transmit-side cost. We mainly measured (1) the system call overhead associated with user–kernel context switches when packet requests were issued from user-space applications, (2) the in-kernel processing cost incurred in the `sys_sendto` path, and (3) the cost of the `start_xmit` function implemented by the device driver, responsible for transmitting `sk_buffs` to the network device. For PV networking, we also accounted for the overhead of `handle_tx`, which lies on the critical path and processes guest packets by traversing the host network stack before device delivery. System call overhead was measured separately using a `no-op` system call, while all other components were obtained via kernel instrumentation.

Compared to the native and vIOMMU, Courier introduces less than 1 μs of additional overhead on the transmit path, primarily due to the necessary virtqueue synchronization incurred during kick operations in the shadow IOMMU implementation. Although Courier incurs slightly higher overhead than vIOMMU, it achieves this without requiring any L_0-provided nested DMA remapping support, making it practical for deployment in public clouds. Despite this, Courier reduces transmit-side cost by 61% and 58.7% compared to virtio-net and vhost-net, respectively. The higher overhead of the PV approaches stems from their longer transmit paths, involving additional `handle_tx` processing in L_1. Moreover, their receive paths incur extra overhead due to packet processing through the host network stack and interrupt injection into the guest, leading to higher receive-side latency. In contrast, Courier simplifies the receive path via used-ring passthrough, allowing DMA results to be directly consumed by the nested guest with minimal overhead.

6 Discussion and Future Work

Hybrid NIC-aware placement. Courier is designed to fit existing cloud networking abstractions rather than replace them. In public clouds, the practical scale of direct NIC access is mainly bounded by the number of NICs that can be exposed to a guest, such as ENIs. A common VM, for example ecs.g9i.12xlarge (32 vCPUs with 8 ENIs), can already host multiple secure containers with dedicated NIC access. This makes Courier particularly suitable for hybrid placement strategies [46]: network-intensive secure containers can use Courier, while others can continue to use PV networking. Such placement improves deployment density without assigning scarce NIC resources to every workload.

L_1-enforced isolation. Courier also preserves a clear isolation boundary under realistic cloud assumptions. It treats L_2 as untrusted and keeps DMA control within the trusted L_1. Specifically, Courier validates DMA mappings, restricts device access to verified memory regions, and mitigates TOCTOU attacks [28] through controlled shadow-state management and explicit mapping invalidation. Thus, Courier focuses on providing strong memory and device isolation for in-guest direct I/O. QoS isolation is orthogonal to this design and can be provided by existing provisioning mechanisms, such as assigning dedicated ENIs to individual containers.

Beyond network I/O. Courier opens a broader direction for in-guest direct I/O. Unlike prior systems, Courier requires no host modifications, which makes it deployable in today's public clouds. Although this paper focuses on networking, the design is not limited to NICs. We have extended Courier to support additional virtio-based devices, including virtio-blk and virtio-nvme. A natural next step is to generalize the design to a wider class of I/O devices, enabling comprehensive in-guest direct I/O support while preserving the same deployment model and isolation guarantees.

7 Related Work

Cloud network virtualization. Virtualized networking forms a critical backbone of multi-tenant cloud infrastructures [9, 12, 23, 61]. Early software-defined solutions established the widely adopted virtio-net standard, while kernel-bypass frameworks such as vhost-user with DPDK [52] further enhanced I/O performance. To reduce latency and CPU overhead, hardware-assisted techniques [34, 39, 70] offload the data plane directly onto SmartNICs. SR-IOV further allows a single physical NIC to be split into multiple virtual functions (VFs) each functioning as an independent network device [22, 58]. Beyond single-node optimizations, recent work has also explored cloud-scale orchestration [31, 50, 57, 67]. In contrast, Courier targets direct network I/O for nested secure containers inside cloud VMs.

Containers networking. Containers have become a cornerstone of modern cloud infrastructures, motivating extensive studies of their network performance [8, 20, 51]. Traditional container networking relies on centralized bridge interfaces, creating throughput and latency bottlenecks that motivate eBPF-based fast datapaths [44, 64]. As secure containers emerge as a hardened alternative, network virtualization remains critical: they typically use para-virtualized networking, while selectively enabling device passthrough for network-intensive workloads. SKernel [20] decouples the secure container guest kernel into SKernel-R for host-assisted management and SKernel-D for passthrough I/O. Systems such as FastIOV [41] demonstrate the benefits of SR-IOV-enabled networking for secure containers and reduce the startup overhead associated with device passthrough. In contrast, Courier redefines virtio interaction semantics for secure containers inside the cloud VM.

Nested network virtualization. Prior work has explored software- and hardware-assisted techniques for nested CPU and memory virtualization [16, 37, 55, 66, 68]. Xen's PV-MMU direct-paging design offers an alternative to shadow paging by allowing guests to construct page tables that Xen validates before hardware use [15]. Recent Xen PV-IOMMU work extends this model to DMA, allowing Dom0 to manage IOMMU contexts and mappings through hypercalls with hardware-IOMMU enforcement; nested DomU semantics

remain under development [14]. Despite differing in how they construct translation state, Xen PV-MMU/PV-IOMMU and conventional PT/IOPT shadowing all rely on privileged software to validate or attach hardware-enforced state.

This dependence is particularly consequential for network I/O. Although paravirtualized device stacks remain usable in nested guests, device passthrough requires safe remapping of L_2 DMA addresses. vIOMMU [11] emulates guest IOMMU operations in L_0 and underpins systems such as DVH [38] for nested device passthrough. HyperTurtle [71] further reduces emulation overhead using eBPF filters in L_0. These systems nevertheless introduce new L_0 mechanisms and couple DMA translation with host-managed memory state, limiting cloud deployability. Courier instead relocates enforcement to the virtio protocol boundary: L_1 validates and translates L_2 descriptors into an L_1-owned shadow virtqueue, providing nested DMA-remapping semantics without L_0 support.

8 Conclusion

Secure containers running within cloud VMs face a critical network I/O bottleneck due to nested virtualization, where para-virtualized solutions incur high cross-layer overhead and hardware-assisted passthrough is often unavailable. Courier overcomes this limitation by virtualizing IOMMU semantics within cloud VMs and providing direct, passthrough-like access to virtio network devices without modifying the host hypervisor. By combining a shadow IOMMU with data-plane optimizations – including direct-kick notifications, used-ring passthrough, and efficient interrupt handling – Courier eliminates redundant mediation, enables zero-copy data paths, and preserves strong DMA isolation. The evaluation demonstrates that Courier consistently outperforms state-of-the-art solutions across microbenchmarks and real-world workloads, confirming its practicality for high-performance networking in nested secure container deployments on modern cloud platforms.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their constructive feedback and suggestions. This work is supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under grant JYB2025XDXM103, Hubei Provincial Natural Science Foundation of China (No. 2026AFA002), Science and Technology Plan Project of Hubei Province under grant 2025CSA056, and the Hubei Provincial Natural Science Foundation of China under grant 2026AFB102. The corresponding author is Jiangshan Lai, Hui Lu and Song Wu.

References

[1] 2020. *VFIO*. https://www.kernel.org/doc/html/v5.10/driver-api/vfio.html.

[2] 2020. *VIRTIO-Net*. https://github.com/torvalds/linux/blob/v5.10/drivers/net/virtio_net.c.

[3] 2024. *Bare metal and virtual servers*. https://www.ibm.com/think/insights/bare-metal-and-virtual-servers-combining-benefits-for-competitive-advantage

[4] 2025. *CVE-2025-31133*. https://access.redhat.com/security/cve/cve-2025-31133.

[5] 2025. *CVE-2025-52565*. https://access.redhat.com/security/cve/cve-2025-52565.

[6] 2025. *CVE-2025-62725*. https://access.redhat.com/security/cve/cve-2025-62725.

[7] 2026. *Alibaba Maxcompute*. https://www.alibabacloud.com/help/en/maxcompute/security-and-compliance/maxcompute-security-white-paper/.

[8] Ubaid Abbasi, El Houssine. Bourhim, Mouhamad Dieye, and Halima Elbiaze. 2019. A performance comparison of container networking alternatives. *IEEE Network* 33, 4 (2019), 178–185.

[9] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. In *Proceedings of the Symposium on Cloud Computing (SoCC)*. 121–127.

[10] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2018. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing* 11, 2 (2018), 430–447.

[11] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU emulation. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[12] Saif Ansary, Antonio Barbalace, Ho-Ren Chuang, Thomas Lazor, and Binoy Ravindran. 2017. A distributed operating system network stack and device driver for multicores. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2646–2649.

[13] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure linux containers with intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 689–703.

[14] Teddy Astie. 2024. IOMMU Subsystem Redesign and PV-IOMMU Interface. (2024). https://lists.xenproject.org/archives/html/xen-devel/2024-06/msg01571.html

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 164–177.

[16] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[17] Anna Berenberg and Brad Calder. 2022. Deployment archetypes for cloud applications. *The ACM Computing Surveys* 55, 3 (2022), 1–48.

[18] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand container loading in AWS lambda. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 315–328.

[19] Miguel Caballer, Marica Antonacci, Zdeněk Šustr, Michele Perniola, and Germán Moltó. 2021. Deployment of elastic virtual hybrid clusters across cloud sites. *Journal of Grid Computing* 19, 1 (2021), 4.

[20] Xiaohu Chai, Keyang Hu, Jianfeng Tan, Tiwei Bie, Guotao Tan, Tianyu Zhou, Anqi Shen, Dawei Shen, Xinyao Yang, Xin Chen, Xu Wang, Feng Yu, Zhengyu He, Dong Du, Yubin Xia, Kang Chen, and Yu Chen. 2026. SKernel: An elastic and efficient secure container system at scale with a split-kernel architecture. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 605–623.

[21] Kevin Cheng, Spoorti Doddamani, Tzi-Cker Chiueh, Yongheng Li, and Kartik Gopalan. 2020. Directvisor: Virtualization for bare-metal cloud. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 45–58.

[22] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *The Journal of Parallel and Distributed Computing* 72, 11 (2012), 1471–1480.

[23] Nolan P. Donoghue, Bridger Hahn, Helen Xu, Thomas M. Kroeger, David Zage, and Rob Johnson. 2015. Tracking network events with write optimized data structures. In *Proceedings of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. 1–7.

[24] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *The ACM Sigplan Notices* 47, 4 (2012), 37–48.

[25] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency serverless inference for large language models. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 135–153.

[26] Stefano Garzarella, Giuseppe Lettieri, and Luigi Rizzo. 2015. Virtual device passthrough for high speed VM networking. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 99–110.

[27] Abel Gordon, Nadav Har'El, Alex Landau, Muli Ben-Yehuda, and Avishay Traeger. 2012. Towards exitless and efficient paravirtual I/O. In *Proceedings of the Annual International Systems and Storage Conference (SYSTOR)*. 1–6.

[28] Gui-Dong Han, Jia-Ju Bai, Qiu-Ji Chen, and Jiqiang Lu. 2026. Static Detection of TOCTOU Bugs Caused by Kernel Races. In *Proceedings of the USENIX Security Symposium (Security)*. 831–847.

[29] Hang Huang, Jiangshan Lai, Jia Rao, Hui Lu, Wenlong Hou, Hang Su, Quan Xu, Jiang Zhong, Jiahao Zeng, Xu Wang, Zhengyu He, Weidong Han, Jiang Liu, Tao Ma, and Song Wu. 2023. PVM: Efficient shadow paging for deploying secure containers in cloud-native environment. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 515–530.

[30] Zhuo Huang, Hao Fan, Junhui Peng, Qi Wu, Song Wu, Chen Yu, Hai Jin, Qiming Liu, Wei Yang, and Shuo Yu. 2025. WAF: An efficient webAssembly-based execution environment for user-defined functions. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1966–1980.

[31] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2015. NetVM: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management* 12, 1 (2015), 34–47.

[32] Omar Jarkas, Ryan Ko, Naipeng Dong, and Redowan Mahmud. 2025. A container security survey: Exploits, attacks, and defenses. *The ACM Computing Surveys* 57, 7 (2025), 1–36.

[33] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 152–166.

[34] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and TV Lakshman. 2017. UNO: Uniflying host and smart NIC offload for flexible packet processing. In *Proceedings of the Symposium on Cloud Computing (SoCC)*. 506–519.

[35] Jack Li, Calton Pu, Yuan Chen, Vanish Talwar, and Dejan Milojicic. 2015. Improving preemptive scheduling with application-transparent checkpointing in shared clusters. In *Proceedings of the Annual Middleware Conference (Middleware)*. 222–234.

[36] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 53–68.

[37] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. 2017. NEVE: Nested virtualization extensions for ARM. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 201–217.

[38] Jin Tack Lim and Jason Nieh. 2020. Optimizing nested virtualization performance using direct virtual hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 557–574.

[39] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 243–259.

[40] Likai Liu, Fu Xiao, Lei Han, Weibei Fan, Xin He, and Zheng Wu. 2026. NBBM: An efficient SmartNIC-Based architecture for bare-metal management in cloud platforms. *IEEE Trans. Comput.* (2026).

[41] Yunzhuo Liu, Junchen Guo, Bo Jiang, Yang Song, Pengyu Zhang, Rong Wen, Biao Lyu, Shunmin Zhu, and Xinbing Wang. 2025. FastIOV: Fast startup of passthrough network I/O virtualization for secure containers. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 720–735.

[42] Chen Lv, Fuxin Zhang, Xiang Gao, and Chen Zhu. 2022. LA-vIOMMU: An efficient hardware-software co-design of IOMMU virtualization. In *Proceedings of the IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 246–253.

[43] Kaesi Manakkal, Nathan Daughety, Yu Sun, Marcus Pendleton, and Hui Lu. 2026. vBOIDs: Taming Chaos via Coarse-grained Scheduling Abstraction for Containers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[44] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez. Bernal, Matteo Bertrone, and Yunsong Lu. 2021. A framework for eBPF-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 133–151.

[45] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. 2015. Utilizing the IOMMU scalably. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 549–562.

[46] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. 2006. Network-aware operator placement for stream-processing systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 49–49.

[47] Antonis Psistakis. 2025. IOMMU support for virtual-address remote DMA in an ARMv8 environment. *arXiv preprint arXiv:2511.19258* (2025).

[48] Kaushik Kumar Ram, Alan L Cox, Mehul Chadha, and Scott Rixner. 2013. Hyper-Switch: A scalable software virtual switching architecture. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 13–24.

[49] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 419–433.

[50] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and efficient NIC packet scheduling. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 33–46.

[51] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. An analysis and empirical study of container networks. In *Proceedings of the IEEE*

International Conference on Computer Communications (INFOCOM). 189–197.

[52] Jianfeng Tan, Cunming Liang, Huawei Xie, Qian Xu, Jiayu Hu, Heqing Zhu, and Yuanhan Liu. 2017. VIRTIO-USER: A new versatile channel for kernel-bypass networks. In *Proceedings of the Workshop on Kernel-Bypass Networks*. 13–18.

[53] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. 2020. coIOMMU: A virtual IOMMU with cooperative DMA buffer tracking for efficient memory management in direct I/O. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 479–492.

[54] Alexander Van't Hof and Jason Nieh. 2022. BlackBox: a container security monitor for protecting containers on untrusted operating systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 683–700.

[55] Lluís Vilanova, Nadav Amit, and Yoav Etsion. 2019. Using smt to accelerate nested virtualization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 750–761.

[56] Xingyu Wang, Junzhao Du, and Hui Liu. 2022. Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes. *Cluster Computing* 25, 2 (2022), 1497–1513.

[57] Xin Wang, Deguo Li, Zhihong Wang, Lidong Jiang, Shubo Wen, Daxiang Kang, Engin Arslan, Peng He, Xinyu Qian, Bin Niu, Jianwen Pi, Xiaoning Ding, Ke Lin, and Hao Luo. 2025. Byte vSwitch: A high-performance virtual switch for cloud networking. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 1417–1432.

[58] Yun Wang, Liang Chen, Jie Ji, Xianting Tian, Ben Luo, Zhixiang Wei, Zhibai Huang, Kailiang Xu, Kaihuan Peng, Kaijie Guo, Ning Luo, Guangjian Wang, Shengdong Dai, Yibin Shen, Jiesheng Wu, and Zhengwei Qi. 2025. To PRI or not to PRI, that's the question. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 75–89.

[59] Yaohui Wang, Ben Luo, and Yibin Shen. 2023. Efficient memory overcommitment for I/O passthrough enabled VMs via fine-grained page meta-data management. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 769–783.

[60] Yongwei Wu, Teng Ma, Maomeng Su, Mingxing Zhang, Kang Chen, and Zhenyu Guo. 2019. RF-RPC: Remote fetching RPC paradigm for RDMA-enabled network. *IEEE Transactions on Parallel and Distributed Systems* 30, 7 (2019), 1657–1671.

[61] Tong Xing, Hesam Tajbakhsh, Israat Haque, Michio Honda, and Antonio Barbalace. 2022. Towards portable end-to-end network performance characterization of smartnics. In *Proceedings of the ACM SIGOPS Asia-Pacific Workshop on Systems*. 46–52.

[62] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, Íñigo Goiri, Eli Cortez, Terry Yang, Victor Rühle, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2023. Snape: Reliable and low-cost computing with mixture of spot and on-demand vms. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 631–643.

[63] Lingyun Yang, Yongchen Wang, Yinghao Yu, Qizhen Weng, Jianbo Dong, Kan Liu, Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, , Menglei Zheng, Lanlan Xi, Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, and Wei Wang. 2025. GPU-Disaggregated serving for deep learning recommendation models at scale. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 847–863.

[64] Wanqi Yang and Pengfei Chen. 2025. eProbe: eBPF-enhanced Accurate Container Status Probing in Cloud-native Systems. *IEEE Transactions on Services Computing* (2025).

[65] Jianing You, Kang Chen, Laiping Zhao, Yiming Li, Yichi Chen, Yuxuan Du, Yanjie Wang, Luhang Wen, Keyang Hu, and Keqiu Li. 2025. AlloyStack: A library operating system for serverless workflow applications. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 921–937.

[66] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 203–216.

[67] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A platform for high performance network service chains. In *Proceedings of the Workshop on Hot topics in Middleboxes and Network Function Virtualization*. 26–31.

[68] Xiantao Zhang and Eddie Dong. 2012. Nested virtualization update from Intel. *Xen Summit* (2012), 11.

[69] Yinchao Zhang, Su Yao, Yong Feng, Kang Chen, Tong Li, Zhuotao Liu, Yi Zhao, Lexuan Zhang, Xiangyu Gao, Feng Xiong, Qi Li, and Ke Xu. 2025. Pegasus: A universal framework for scalable deep learning inference on the dataplane. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. 692–706.

[70] Zongpu Zhang, Jiangtao Chen, Banghao Ying, Yahui Cao, Lingyu Liu, Jian Li, Xin Zeng, Junyuan Wang, Weigang Li, and Haibing Guan. 2024. Hd-iov: Sw-hw co-designed i/o virtualization with scalability and flexibility for hyper-density cloud. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 834–850.

[71] Ori Ben Zur, Jakob Krebs, Shai Aviram Bergman, and Mark Silberstein. 2025. Accelerating nested virtualization with HyperTurtle. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 987–1002.