SG-IOV: Socket-Granular I/O Virtualization for SmartNIC-Based Container Networks

Chenxingyu Zhao
University of Washington
Seattle, Washington, USA
cxyzhao@cs.washington.edu

Hongtao Zhang
University of Washington
Seattle, Washington, USA
hongtaoz@cs.washington.edu

Jaehong Min
University of Washington
Seattle, Washington, USA
jaehongm@cs.washington.edu

Shengkai Lin*
Shanghai Jiao Tong University
Shanghai, China
jefflin@sjtu.edu.cn

Wei Zhang
University of Connecticut
Storrs, Connecticut, USA
wei.13.zhang@uconn.edu

Kaiyuan Zhang
University of Washington
Seattle, Washington, USA
kaiyuanz@cs.washington.edu

Ming Liu
University of Wisconsin-Madison
Madison, Wisconsin, USA
mgliu@cs.wisc.edu

Arvind Krishnamurthy
University of Washington
Seattle, Washington, USA
arvind@cs.washington.edu

Abstract

I/O Virtualization (IOV) is a cornerstone of cloud computing, with container networking as a critical form of IOV in modern cloud paradigms. While container networks serve as feature-rich infrastructure, they incur a high CPU tax yet leave room for efficiency improvement. A natural idea is to offload container networks onto hardware such as SmartNICs via IOV interfaces. However, existing IOV mechanisms, such as SR-IOV, are misaligned with container requirements: limited device scalability versus high container density, packet-layer abstraction versus application-layer processing demands, and coarse-grained virtualization versus fine-grained container workloads.

In this work, we propose Socket-Granular I/O Virtualization (SG-IOV), a new IOV mechanism to offload container networks efficiently. SG-IOV provisions socket-level devices, supports size-varying transformations over a message-stream abstraction, and enables granular virtualization. Realizing this vision is non-trivial: Finer granularity stresses device scalability, while size-varying message processing complicates IO buffer management. Additionally, many fine-grained devices strain hardware virtualization enforcement. To overcome these challenges, the core principle of SG-IOV is that software mediates signals (*e.g.,* descriptors) while accelerators touch the payload. This separation enables several innovations: resource multiplexing to support scalability, adaptive handling of size-varying tasks, and intermediate queuing to enforce virtualization. We prototype SG-IOV on the latest NVIDIA BlueField-3 and deliver an end-to-end container network solution. Our evaluations show that SG-IOV scales beyond 4K devices (10x more than VFs). Compared to Cilium, SG-IOV saves up to 1.9 cores per 10Gbps of traffic. Beyond core savings, SG-IOV achieves 53% higher bandwidth and reduces latency by up to 48%.

CCS Concepts: • **Hardware → Networking hardware**; • **Networks → Programmable networks**; *Cloud computing*; • **Software and its engineering →** *Operating systems.*

Keywords: SmartNIC; I/O Virtualization; Container Network; Cloud Computing

ACM Reference Format:
Chenxingyu Zhao, Hongtao Zhang, Jaehong Min, Shengkai Lin, Wei Zhang, Kaiyuan Zhang, Ming Liu, and Arvind Krishnamurthy. 2026. SG-IOV: Socket-Granular I/O Virtualization for SmartNIC-Based Container Networks. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3779212.3790218

1 Introduction

Container networking, as a critical form of network I/O Virtualization (IOV), underpins modern cloud paradigms, such as serverless computing and microservices. Driven by the diverse demands of containerized clouds and the advent of network technologies, container networks [17, 18, 25, 33, 35] are rapidly evolving. These solutions not only provide essential network connectivity among housed containers but also continuously embed features such as overlay virtualization, security enhancements, application-level proxies, and additional functionalities spanning the entire OSI-7 layers.

*Work done while at the University of Washington.

ASPLOS '26, March 22–26, 2026, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790218

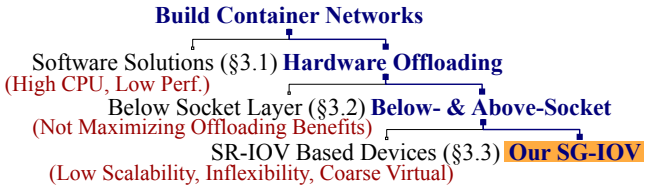


Figure 1. Design Space for Building Container Networks.

Enriching features of container networks adds value but also imposes efficiency burdens in terms of low performance and CPU consumption, contending with tenant workloads. Thus, we investigate how to build container networks that are both feature-rich and efficient by navigating the design space shown in Figure 1. On the first branching point, we first consider CPU-based software solutions. According to Red Hat and NVIDIA's cost model [11], container infrastructures would use eight CPU cores per server in a mid-scale data center with 51K servers, yielding $68.5M capital expenses. This aligns with our empirical evaluation (§3.1) that sustaining 10K requests per second consumes around six host cores when processing L7 HTTP with Cilium [18] and the Envoy [24] proxy. Despite consuming many cores, software solutions still exhibit low performance when using general-purpose CPUs for several domain-specific tasks, such as securing transfers (experiments in §3.1). For core savings and acceleration, the idea of offloading container networks onto hardware devices (*e.g.,* SmartNICs [13, 14, 29, 57, 67, 89, 105, 117, 121]) naturally comes to mind.

While hardware offloading paves the way for efficiency, a key design choice is determining at which layer to divide the labor of the container networking stack between the host and hardware, accordingly designing IOV devices that streamline network functionality execution with maximal hardware acceleration. One insight is that intercepting closer to the application layer is more beneficial. The key reason is that higher-layer interception maximizes offloading opportunities; for example, intercepting at the L5 socket enables offloading of L2 Ethernet tunneling and L3/L4 transport. Moreover, because the L5 socket is message-aware rather than packet-oriented, it integrates seamlessly with application-layer processing (*e.g.,* HTTP) and flexibly supports L5+ offloading, further amplifying the offloading benefits.

Although interfacing the offloading device at higher layers is beneficial, current IOV mechanisms fall short in provisioning virtual devices at the upper layers (*e.g.,* L5 socket). Specifically, we examine the *de facto* IOV mechanism, SR-IOV [1], and identify the following limitations:

Device Granularity Gap: SR-IOV (Single Root I/O Virtualization), originating in the cloud VM era, typically supports only a limited number of devices (*i.e.,* PF/VF) due to hardware resource constraints, with even the latest SmartNICs operating at the scale of 100s of VFs [29, 40]. Meanwhile, containers, driven by their lightweight nature, have much higher density requirements compared to cloud VMs, with several industry practices pushing to deploy thousands of containers per node [44, 86, 110]. Clearly, a gap exists between the number of VFs and the container density, which is likely to persist or even increase in the future. Several techniques (*e.g.,* Sub-Functions/SFs [96]) aim to increase device count, yet hardware constraints remain (§3.3) and the following issues are unaddressed for both SR-IOV and SFs.

Interface Abstraction Mismatch: NICs provide the host with SR-IOV VFs or SFs, which are typically L2 Ethernet devices with packetized interfaces. However, as mentioned before, we aim at the upper-layer interface of the L5 socket with message abstraction. This mismatch between L2 packets and L5+ messages results in the non-offloadability of overlay transport (L3/L4) and upper-layer tasks (L7). Alternatively, it requires converting L5 messages into L2 packets on the host and then reversing the process on the SmartNIC to rebuild the messages from L2 to L5, creating cyclic redundancy between the intra-node host and SmartNIC domains.

Insufficient Composition and Virtualization of Accelerators: SR-IOV primarily provides coarse-grained mechanisms (*i.e.,* per-device) to partition hardware resources such as doorbells, interrupts, and queues. However, it lacks support for fine-grained virtualization of more complex accelerators, such as isolating and scheduling traffic over cryptographic or DMA engines. In containerized networks, multi-tenancy and multi-flow scenarios are common, with traffic patterns that shift rapidly over short time scales. As a result, efficient virtualization mechanisms for accelerators are both more necessary and more challenging, issues that SR-IOV or SF does not provide adequate mechanisms to address.

In this work, we propose **Socket-Granular I/O Virtualization (*SG-IOV*)**, a new IOV mechanism to offload container networks efficiently. Compared to SR-IOV, we envision SG-IOV as scalable, message-aware, and capable of fine-grained virtualization. To realize this vision, SG-IOV adopts a core principle in which *SmartNIC-embedded cores mediate only signal operations without touching data, while SmartNIC-side accelerators handle data processing*. The software-mediated signal plane manages stateful metadata (*e.g.,* transfer descriptors via TX/RX queues) without touching payloads, while hardware accelerators within the data plane handle payload operations statelessly. This separation enables the following innovations to overcome several challenges:

Scalable Device Interface (§4 & §5.1): SG-IOV provisions socket-granular IOV devices for containerized applications. The key challenge is how to support many more devices than SR-IOV VFs, preserving performance while using the same or even fewer hardware resources (*e.g.,* TX/RX queues). We address this by multiplexing TX/RX queues across multiple devices. Meanwhile, to preserve performance, software mediates message-level signaling (coalescing multiple packets), and we design efficient signaling mechanisms accordingly.

Flexible Message-Aware Abstraction (§5.2): SG-IOV devices support a streaming abstraction with ring buffers to handle input and processing of unbounded message streams. The

key challenge is that while sockets allow *unbounded* streams, ring buffers for payload are *bounded* and wrap around, and hardware accelerators often require contiguous buffer regions. We address this using *a Recursive Strategy* enabled by the programmability of the signal plane.

Granular Accelerator Virtualization (§6.1): While composing accelerators, a key problem is where and how to enforce fine-grained accelerator virtualization. We address this by introducing intermediate queuing between the separated signal and data planes. SG-IOV tailors the *Dominant Resource Fairness (DRF)* [70] policy and derives an *Equal Bandwidth Demand* property for scheduling heterogeneous hardware.

We prototype SG-IOV on NVIDIA BlueField-3 SmartNICs. We characterize several hardware primitives and implement IOV devices as emulated PCIe devices (§7). Based on our IOV devices, we develop an end-to-end CNI (Container Network Interface [69]) solution, *SGIOV-CNI*, which can support the advanced secure container runtime. SG-IOV is compatible with POSIX sockets, requires no modifications to container images, and complies with orchestration tools such as Docker and Kubernetes.

We have the following results from evaluation (§8):

1) On scalability, SG-IOV scales over 4K devices (an order of magnitude more than SR-IOV VFs), and when scaling the number of sockets to 4K, latency increases by only 2.8x, demonstrating strong scalability, compared to SR-IOV scaling to 128 VFs with a 3.75x increase in RDMA latency.

2) On flexibility, we demonstrate zero-copy and full-copy data transfers as size-preserving jobs, and in-motion AES en-/decryption as size-varying jobs. For end-to-end container networks, we compare SG-IOV with the versatile and high-performance Cilium [18] and SRIOV-CNI [46] over SFs.

3) Regarding core savings, SG-IOV offloading compute-intensive tasks such as transport over tunnels, L3/L5 security tasks, and L7 HTTP processing results in substantial benefits, with savings of ~1.9 host cores per 10Gbps of traffic.

4) More than core savings, SG-IOV achieves up to 53% higher bandwidth for plaintext transfers and 12.4x higher bandwidth for encrypted transfers compared to Cilium. Regarding latency, SG-IOV achieves up to 48% lower latency using real-world applications.

We make the following contributions: 1) We characterize container networks, latest SmartNICs, and SR-IOV-based offloading mechanisms to uncover new problems and opportunities (§2-§3); 2) We design and implement SG-IOV, a new IOV mechanism for SmartNICs (§4–§7), which is scalable, flexible, and supports fine-grained virtualization; 3) We develop SGIOV-CNI as a container network solution and demonstrate its performance gains through end-to-end tests and microbenchmarks with secure containers (§8).

2 Background

In this section, we present background on container networking, I/O Virtualization, and SmartNICs.

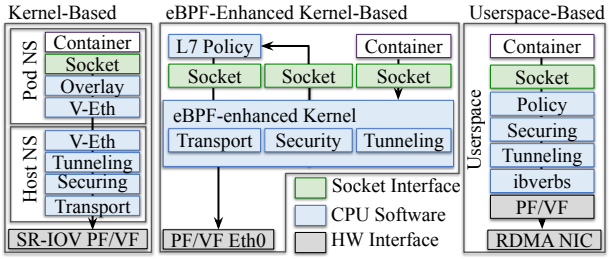
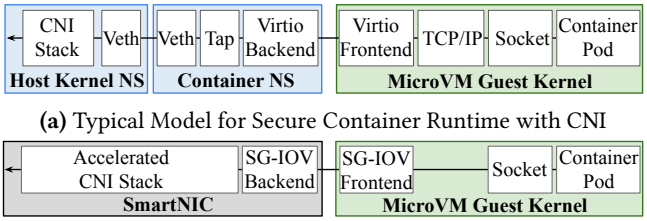


Figure 2. Container Network Interface (CNI) Solutions.



(a) Typical Model for Secure Container Runtime with CNI

(b) Our SG-IOV Deployment Model

Figure 3. Secure Container Runtime Complicating CNI.

2.1 Container Networking

Container Network Interface (CNI): Most container networks build overlay interfaces to provide socket APIs, virtual IPs, and namespaces (NS) independent of the host, enhancing portability and isolation. As container-based paradigms (*e.g.,* service meshes) evolve rapidly [12, 15, 26, 56, 88], CNIs increasingly embed features across OSI layers to deliver infrastructure services and ease operations. As Figure 2 shows, we classify them into the following types: 1) Kernel-based solutions, such as Flannel [25], Weave [53] and more [17, 35] rely on the in-kernel network stack; 2) eBPF-enhanced kernel-based solutions like Cilium [18] harness eBPF [23] to implement some functionalities hooked to the in-kernel data path. Notably, despite being kernel-based, Cilium still utilizes some userspace proxies like Envoy to handle L7 processing; 3) User-space solutions, such as FreeFlow [76] and more [85, 100, 115], which use the kernel-bypass network stack (like RDMA) to provide the container's network solutions. For these types of CNIs, we highlight three computation-intensive tasks as follows:

- Transport Over Tunnels, which provides the essential intra-/inter-server connectivity while handling the isolation and multi-tenancy via tunneling (*e.g.,* VxLAN);
- Security, which guarantees confidentiality, integrity, and authenticity of the transmitted data via crypto protocols. Given the importance of security, container networks continually integrate cryptographic mechanisms throughout the entire stack (*e.g.,* L3 IPsec, L5 TLS, and service-level admission), all of which are computationally complex;
- Application-Level Processing, which enforces network policies at the application level via HTTP load-balancing, service-level metering/rate-limiting, access control, and more. Especially in service mesh scenarios, it is common to use a *sidecar* to handle these L7 tasks [104, 110].

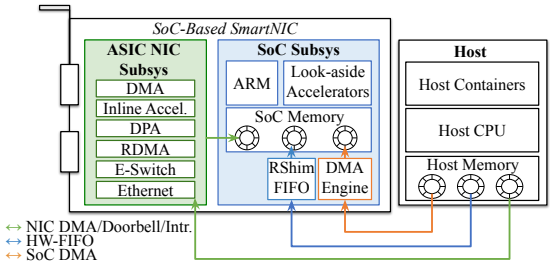


Figure 4. Block Diagram of NVIDIA BlueField-3 SmartNIC.

Size-Varying/Preserving Jobs: We classify these functionalities by whether they change data size: size-preserving jobs (*e.g.,* transport) keep sizes unchanged, while size-varying jobs (*e.g.,* encryption) add metadata or alter size.

Secure Containers Further Complicate CNI: Beyond traditional containers, hyperscalers widely deploy secure container runtimes (*i.e.,* sandboxing runtimes) for enhanced security isolation. Examples include AWS Firecracker [55], Azure using Kata [42], and more [27, 34, 43, 86, 90, 122]. Simply put, a secure container runtime (shown in Figure 3a) utilizes MicroVMs within the user space and deploys container pods over these MicroVMs instead of directly over the host kernel. Secure container runtimes further complicate the CNI because socket syscalls are no longer handled first by the host kernel. Instead, syscalls are initially processed within the guest OS, traversing the transport stack and the Virtio [52] device driver. On the host side, the Virtio backend is typically connected to a veth device. From there, the packet follows the standard CNI data path, as in traditional container runtimes. Given the enhanced security and growing adoption of secure containers, we primarily target SG-IOV in this setting, as shown in Figure 3b. SG-IOV exposes pass-through devices to the guest kernel.

2.2 I/O Virtualization

I/O Virtualization (IOV) is the mechanism for provisioning multiple virtual devices to the host with physical NICs. IOV mechanisms follow a *split front-end/back-end device model*, where the host front end exposes a device interface (*e.g.,* SR-IOV PF/VF [1] or Sub-Functions [96]) and the device back end executes datapath logic (*e.g.,* signal handling and DMA). The front end and back end interact through hardware TX/RX queues for descriptor exchange. IOV has extensive literature [1, 45, 50, 75, 76, 80, 90, 120], especially for cloud VMs. Due to space constraints, we defer the IOV primer to Appendix §A.1.

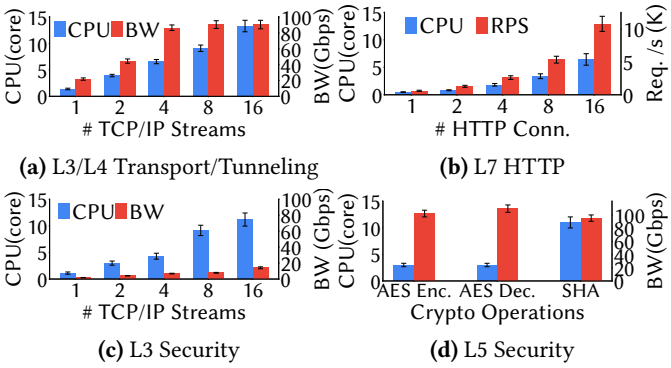
2.3 SoC-based SmartNICs

The programmability of SmartNICs enables new opportunities for building new IOV devices. In this project, we focus on SoC-based SmartNICs (*e.g.,* NVIDIA BlueField-3/BF3). SmartNICs are commonly fabricated as PCIe add-in cards. As Figure 4 shows, we present the BF3 as an example. They enclose an ASIC NIC and a system-on-chip (SoC) that typically co-packages a multi-core processor, a memory subsystem, and a collection of hardware acceleration engines. In §3.4, we introduce some emerging capabilities of BF3 and present a detailed characterization in §A.2.

	SW-CNI[1]	SR-IOV	SF[2]	NSaaS[3]	TOE[4]	SG-IOV
HW Offload	✗	✓	✓	✗	✓	✓
PCIe Device Pass-through	✗	✓	✓	✗	✗	✓
Scalability	✓	✗	✓	✓	✓	✓
Flexibility[5]	✓	✗	✗	✓	✗	✓
Granular HW Virtualization	✗	✗	✗	✗	✗	✓

[1]Software CNI; [2]Sub-Function; [3]Network Stack as a Service; [4]TCP Offload Engine; [5]Flexibility: Support size-varying jobs spanning the Layer-7 stack.

Table 1. Comparison of Approaches for Building CNIs.



(a) L3/L4 Transport/Tunneling **(b)** L7 HTTP

(c) L3 Security **(d)** L5 Security

Figure 5. CPU Cost of Software CNI Solutions.

3 Motivation

This section presents the motivation and challenges of efficient CNI design and summarizes design goals versus prior solutions in Table 1.

3.1 Why Offload Container Networking?

CNI Overhead Characterization: In Figure 5, we benchmark the aforementioned tasks (§2.1) in terms of computation costs. Given the widespread adoption in practice, we use Cilium [18] for our measurement study and use the Kubernetes built-in tool to monitor containers. Our testbed consists of two servers connected with 100GbE networking and equipped with a typical x86 CPU (testbed details in §8.1).

In Figure 5a, we measure the overhead of transport over tunnels. We use containerized *iperf3* [32] to generate multi-stream TCP/IP traffic and enable VxLAN tunneling. We can saturate a 100GbE line rate while consuming 6.6 CPU cores.

In Figure 5b, we measure the overhead of application-level processing. We use *wrk* application [9] to generate HTTP requests to *nginx* [38] backend servers. We measure the CPU cost for the Envoy L7 proxy, a common component of Cilium that executes application-level processing, such as HTTP request load balancing and URL rewriting. The Envoy proxy consumes 6.4 CPU cores for 10K requests per second (RPS).

In Figures 5c and 5d, we measure the overhead of securing transfers. In Figure 5c, we enable IPsec. By default, Cilium uses a software-based IPsec implementation with in-kernel ip-xfrm [31]. As the results indicate, we cannot saturate a

100GbE line rate while consuming 4.3 CPU cores for 6.8Gbps of encrypted traffic. Besides L3 security, container networks actively promote the implementation of upper-layer security mechanisms; for example, Cilium is attempting to integrate mTLS [28]. To our knowledge, the encryption for Cilium mTLS is still under development; it currently supports only mTLS authentication. Thus, we assess the building blocks utilizing OpenSSL [7]. AES-GCM-256 encryption and decryption consume 3 cores each to achieve 100Gbps, while SHA-256 authentication needs 11.0 cores for 100Gbps (1024B messages for authentication).

Takeaway: The functionality set of container networks is extensive and requires significant upper-layer processing. They are computation-intensive and exhibit suboptimal performance (*e.g.,* securing transfers) for software CNI solutions, as summarized under *SW-CNI* in Table 1.

3.2 Yes, Offloading — Which Layer to Intercept and Offload onto Hardware?

As mentioned before, hardware offloading paves the way for efficient CNIs. A key question is: at which layer should the datapath be split between the host CPU and hardware? As measured in §3.1, intercepting closer to the application (*i.e.,* at higher layers) has the following advantages:

• Maximize the benefits of offloading: As shown in Figure 5, CNIs commonly incur costs across L2 tunneling (*e.g.,* VxLAN), L3/L4 transport, and L5/L7 processing. To fully realize offloading gains, both below- and above-L5 processing must be embedded in hardware;

• Message Awareness: The L5 socket interface operates directly on messages, aligning with application protocols like HTTP and avoiding costly packet reassembly;

• Batch Processing: A message chunk typically contains multiple packets, allowing more efficient batched operations than per-packet processing and reducing metadata overhead (*e.g.,* headers and trailers in cryptographic records).

Comparing with Network-Stack–as–a–Service (NSaaS): Intercepting traffic at the socket message layer is similar to prior NSaaS systems (*e.g.,* SNAP [91] and NetKernel [93]), but SG-IOV differs in several key aspects: 1) Deployment Scenario: SG-IOV targets CNIs spanning the full OSI Layer-7 stack, whereas prior NSaaS systems primarily focus on L4/L3 transport; 2) Hardware Acceleration: SG-IOV adopts an accelerator-centric model to offload computation, whereas prior NSaaS systems are CPU-centric; 3) Application-NIC Interactions: SG-IOV utilizes PCIe devices as the SW-HW offloading interface, whereas host-only NSaaS solutions rely on memory sharing between applications and network stacks.

3.3 Yes, Upper Layer — How to Build an L5+ Device?

After deciding to offload both below- and above-L5 functions, the next question is how to build such virtual devices. Next, we examine SR-IOV, the *de facto* IOV method, and NVIDIA Sub-Functions (SF) for building L5+ devices to offload CNIs.

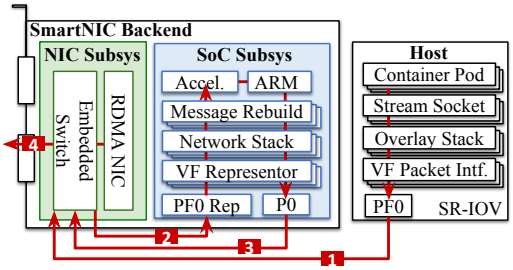


Figure 6. Offloading CNI through SR-IOV SmartNICs.

Issue #1 Limited Scalability at the Front-End: The number of VFs is inherently limited, as each VF requires dedicated hardware resources (*e.g.,* TX/RX queues, doorbells, and interrupts). Meanwhile, the number of container pods per node continues to increase [44, 86, 110], particularly for micro-services and service meshes.

Issue #2 Inflexible Packet-Oriented Abstraction SR-IOV VFs typically provide a packetized interface (*i.e.,* L2) to the host. However, we desire a socket-level stream abstraction to perform message-layer processing (*i.e.,* L5/L7). This inflexibility is evident in both ASIC NICs and SmartNICs:

ASIC NICs, such as NVIDIA ConnectX [2], now support offloading of some upper-layer tasks [99]. However, a packet-oriented abstraction imposes non-trivial constraints. Specifically, *Pismenny et al.* [99] propose novel *autonomous offloading* for L5+ processing (*e.g.,* TLS), which still requires size-preserving constraints and tight coordination between kernels and NICs (*e.g.,* kernel skipping offloaded crypto operations). Such complexity arises primarily because L5 messages like TLS records often span multiple fixed MTU-sized packets, which necessitates state tracking (*e.g.,* packet loss). Size-preserving designs simplify ASIC NICs, but CNI application-level processing often violates this constraint (*e.g.,* security, compression, and HTTP processing).

For SoC SmartNICs (Figure 6), packet-oriented abstraction also incurs some inflexibility. Here, we use BF3 as an example. Other types of SmartNICs exhibit similar limitations (see Appendix §A.6). As shown in Figure 6, the workflow from container to VF and VF to the SmartNIC SoC involves transforming message streams into packets, then reconstructing messages on the SmartNIC to process them. This *cyclic stream-to-packet transformation* is generally inefficient, and is particularly redundant for intra-node communication. Notably, RDMA is not a panacea for this overhead, due to contention between intra- and inter-node traffic (see §8.2).

Issue #3 Inadequate Back-End Accelerator Virtualization: SR-IOV enforces multi-tenant isolation and QoS control at a per-device granularity, such as allocating bandwidth across all VFs. This rigidity limits finer-grained isolation over hardware resources (*e.g.,* per-message). Further, SR-IOV primarily virtualizes PCIe-related resources (*e.g.,* TX/RX QPs) rather than the on-NIC accelerators. Although SmartNICs embed rich accelerators, debate remains over where to enforce such virtualization—runtime, driver, or accelerator [63, 74, 87, 123, 124].

Comparing with NVIDIA Sub-Functions (SF): At the host front end, SFs expose more devices than SR-IOV via finer-grained resource partitioning, but each SF still consumes dedicated queues, making QP count a hard scalability limit. At the back end, SFs inherit Issues #2 and #3 from SR-IOV, as they remain packet-oriented and lack support for fine-grained (*e.g.,* per-message) accelerator virtualization.

Comparing with TCP Offload Engines (TOE): Besides SR-IOV, prior work [62, 77, 108] explores other I/O interfaces, such as socket syscall in TOE. However, TOE falls short for CNIs: 1) TOE focuses on L4 TCP offloading, whereas SG-IOV targets richer below- and above-L5 CNI functionality; 2) TOE exposes offloading via tight integration with kernel TCP syscalls, while SG-IOV uses PCIe devices for passthrough to container runtimes; 3) TOE is fixed to TCP acceleration, whereas SG-IOV integrates diverse accelerators and supports fine-grained virtualization over heterogeneous hardware.

3.4 SmartNIC Opportunities

Besides CNI-driven demands, emerging SmartNIC capabilities also enable new IOV designs. We summarize the key capabilities here and defer detailed characterization to §A.2.

Capability #1 Emulating PCIe Devices: SmartNICs such as BF3 support the emulation of PCIe devices [41]. On the host front-end, these emulated devices are hot-pluggable and compatible with the VFIO framework [51] for accessing PCIe resources, such as doorbells, MSI-X interrupts, and stateful regions. On the back-end, the SoC handles doorbell callbacks and raises interrupts. Notably, BF3 uses the Datapath Accelerator (DPA) to handle callbacks and interrupts.

Capability #2 Efficient Host–SmartNIC Communication Channels: SmartNICs provide efficient intra-node communication mechanisms, including PCIe-based primitives such as doorbells and MSI-X, as well as network mechanisms like the BF3 representor switching, RDMA, DPDK, and more. Also, BF3 supports the *RShim* HW-FIFO for intra-node communications (details in §A.2). The capabilities enable tighter host–SmartNIC cooperation; we highlight the following communication mechanisms from our characterization:

BF3 Finding #1: BF3 has HW-assisted FIFO queues, which enable deterministic and consistently low-latency communication between the host and the SoC ARM cores.

Capability #3 Unified Management for SoC/Host Memory: SmartNIC SDKs (*e.g.,* DOCA [41]) support registering both host-side and SoC-side memory through unified APIs like DOCA mmap and X-GVMI (details in §A.2). This allows the SoC to flexibly use these buffers for DMA transfers, cross-node transports (*e.g.,* RDMA), or as source/destination buffers for hardware accelerators. Notably, we highlight two findings on memory management and data movement:

BF3 Finding #2: BF3 has dual DMA engines, and the SoC DMA incurs little interference with the NIC DMA.

BF3 Finding #3: BF3 supports the X-GVMI technique: SoC cores can initiate Host-to-Host RDMA on behalf of the host to enable efficient delegator-initialized zero-copy.

4 SG-IOV Overview and Interface

4.1 Overview

In this work, we present *Socket-Granular I/O Virtualization* (SG-IOV), a new IOV mechanism for SmartNICs that efficiently offloads container networks. Figure 7 and Table 2 present an overview of the SG-IOV architecture and highlight the following design principles:

$\mathcal{P}1$: Software Mediates Signaling while Hardware Accelerates. As shown in Figure 7a, SG-IOV adopts a hybrid model where software mediates stateful signaling while hardware performs stateless acceleration. The signal plane runs on device-embedded cores like SoC ARM and interacts with host CPUs and other nodes to retrieve states (*e.g.,* socket buffer head/tail). After synchronizing states, the signal plane generates acceleration job descriptors, which the data plane schedules and submits to accelerators. Such separation enables several key techniques, as summarized in Table 2.

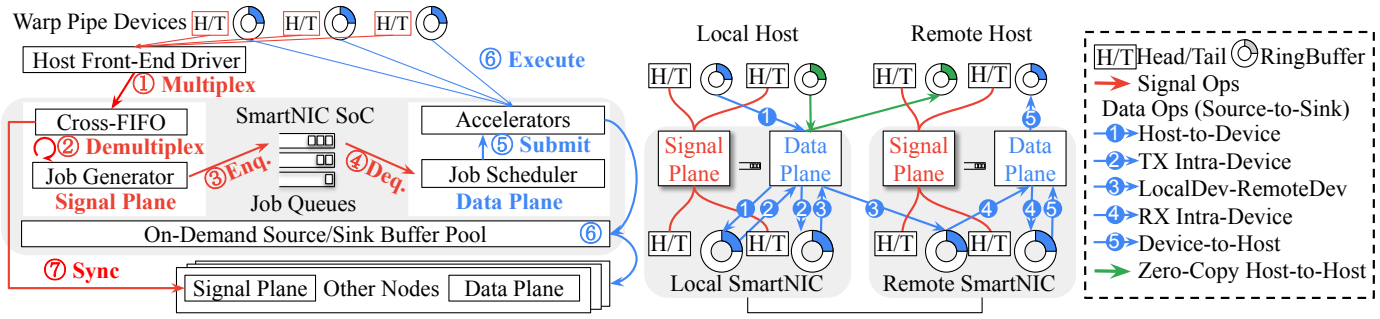
$\mathcal{P}2$: Socket Stream Oriented. SG-IOV provides stream interfaces that seamlessly comply with the socket API and its underlying socket ring buffers. Furthermore, the streaming interface allows SG-IOV to interpose on upper-layer (L5+) processing, which requires awareness of the message rather than just the packets (*e.g.,* message-level TLS and L7 proxy).

4.2 Warp Pipe and Properties

Warp Pipes [6] are compositional units to build SG-IOV devices. Warp pipes implement a simple yet effective stream abstraction with a pair of ring buffers as data structures and a clean set of operations (*e.g.,* read/write). The warp pipe has three key concepts: *source buffer*, *sink buffer*, and *transformations*. Source and sink buffers are typically implemented with ring buffers, which can serve read/write of data payloads without constraining the message size. To boost efficiency and simplify concurrency control, we apply the single-producer and single-consumer (SPSC) model and then utilize lock-free source and sink buffers. Transformations represent operations that convert a data stream from a source buffer into a sink buffer; we will exemplify them later (§6.2).

Flexible Source/Sink Placement: A key property of the warp pipe is that we allow flexible placement of source and sink buffers. Specifically, we allow source and sink buffers of the same warp pipe to be placed on different domains within the same server node or even across nodes (as Figure 7b shows). For example, the source buffer can be on the host CPU side, while the sink buffer is on the SmartNIC SoC. Another example is that we can place the source buffer on the

[6] The name is inspired by Super Mario, where Mario magically transforms to enter the next stop through green warp pipes.



(a) Workflow of Separated Signal and Data Planes **(b)** Composing Warp Pipes with Separated Signal and Data Planes

Figure 7. Architecture Overview of SG-IOV with Warp Pipe Interface (§4.2), Signal Plane (§5) and Data Plane (§6).

Techniques	Prior Limitations and Challenges	Solution Overview	SmartNIC Capabilities
Software-Enhanced Scalable Interface (§5.1)	*Issue #1 (§3.3)*: Limited scalability of SR-IOV-based SW–HW interfaces, where each device dedicates at least one physical QP. *Challenge:* How to scale hardware IOV devices while preserving performance?	Through software mediation, we can define signaling protocols to multiplex a single signaling channel (*i.e.,* one physical QP) across multiple virtual devices. The multiplexing (①,② in Fig. 7a) enables scalable SW–HW interfacing.	*Capabilities #1 and #2 (§3.4)*: Emulating PCIe Device and Efficient Host–SmartNIC Communication Channels.
Flexible Message-Aware Job Generation (§5.2)	*Issue #2 (§3.3)*: Inflexible packet-oriented abstractions for message-aware operations. *Challenge:* How to support size-varying jobs for unbounded message streams over bounded buffers?	Before enqueue (③), the software-based signal plane can adaptively generate jobs based on buffer availability, especially for challenging cases of size-varying jobs.	Lightweight yet general-purpose compute of SmartNIC SoC cores.
Fine-Grained Accelerator Scheduling (§6.1)	*Issue #3 (§3.3)*: Inadequate and coarse-grained accelerator virtualization. *Challenge:* How to enforce effective per-message virtualization policies?	By separating signal and data planes, an intermediate queuing system enables fine-grained (per-message) job scheduling (④) across accelerators.	Lightweight control model of SmartNIC SoC cores.
Zero-Touch Efficiency (§6.2)	*Issue in General*: Inefficiency of software-involved data operations on host CPU or SoC ARM cores. *Challenge:* How to enable software control without touching payloads?	The signal plane and ARM cores generate jobs without touching data payloads; only accelerators operate on payloads (⑤,⑥), enabling accelerator-centric datapaths.	*Capability #3 (§3.4)*: Unified management for SoC/Host memory with zero-copy support.

Table 2. SG-IOV Key Techniques Enabled by the Principles of Signal/Data Plane Separation and Stream Orientation.

local host and the sink buffer on the remote host (green line in Figure 7b). Such flexible placements lay the foundation for SG-IOV to support the offloading of various network functionalities onto rich accelerators.

Chainable Composition: Warp pipes naturally support chainable composition. In Figure 7b (blue line), multiple warp pipes can be composed in series to create function-rich pipelines, where the sink of the current warp pipe serves as the source for the next one (no additional copy and no violation of SPSC). Each warp pipe in the pipeline can execute different transformations. An IOV device is backed by one pipeline. Figure 7b illustrates how cross-domain (host/SmartNIC) and cross-node datapaths are composed via chainable execution.

5 Signal Plane: Scalability and Flexibility

In this section, we introduce the stateful signal plane [7], which is designed to synchronize stream buffer states and generate transformation jobs accordingly.

[7]We use the term "signal plane" rather than "control plane" because the signal plane mainly serves the socket data APIs (*e.g.,* `send`/`recv`). Thus, we intend to avoid the misunderstanding that the signal plane is designed for control APIs (*e.g.,* `connect`/`accept`).

5.1 Scalable Signal Synchronization

Memory View of Ring Buffer Pools: As Figure 7 shows, the host and the offloading device maintain their own ring buffers in memory to support the flexible placement of source and sink buffers on demand. The host and device monitor their own *Memory View*, tracking three key states for each ring buffer: head pointer, tail pointer, and data buffer address, which can help infer data availability or space occupancy. Notably, this data buffer address points to the socket's ring buffer, making the host-side memory footprint of each warp pipe identical to that of the application's socket buffer.

Intra- and Inter-Node Synchronization: To offload host computation, SG-IOV shifts most signal-plane tasks (*e.g.,* generating accelerator jobs) to the SmartNIC SoC. This requires synchronizing the memory view (*i.e.,* buffer header/tail) with the host and other nodes to ensure timely and efficient job generation. We design lightweight protocols and data structures to make this synchronization scalable.

Limitations of Legacy SQ/CQ Synchronizations: We first refer to the legacy Submission Queue (SQ) and Completion Queue (CQ) model, which is widely used to support

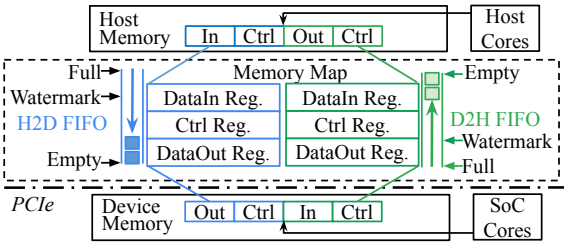


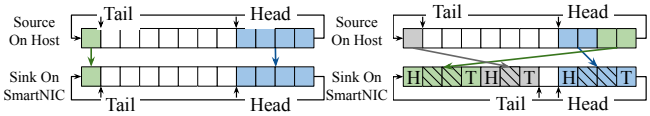
Figure 8. Cross-FIFO Data Structure for Memory View Sync. The processor writes to the Data-In register to enqueue data and reads from the Data-Out register to dequeue. Each FIFO applies the SPSC model. Upon each access, pointers shift to the next position in the FIFO queues. The Control Register indicates queue occupancy and a watermark level, which can be set to interrupt when the queue reaches a prescribed level. Besides interrupts, the processor can poll the register.

front-end and back-end interactions in PCIe functions such as network and NVMe devices. However, SG-IOV introduces the following new requirements, necessitating extensions to the traditional SQ/CQ interaction model: 1) SG-IOV supports not only intra-node warp pipes (*i.e.,* host-to-SmartNIC) but also cross-node warp pipes, such as host-to-host (through the ASIC NIC, bypassing the SoC) or SmartNIC-to-SmartNIC. Therefore, it is necessary to extend the legacy SQ/CQ interactions, originally designed for PCIe, to operate across cross-node domains; 2) With legacy SR-IOV, each device is provisioned with at least one dedicated QP, so the limited hardware QPs cap the device count.

New Extensions of SG-IOV: We address the above limitations through the following extensions.

• *Cross-FIFO Data Structure:* To efficiently transfer the signals for intra-node or inter-node synchronization, we apply one simple yet effective data structure, named *Cross-FIFO*. It is compatible with the legacy SQ/CQ model over PCIe and, thanks to its simple design, can also be implemented using cross-node transport such as RDMA. Cross-FIFO is inspired by the design of UART FIFO [49], a low-cost and widely utilized hardware circuit [4, 5, 10]. In Figure 8, the Cross-FIFO consists of a pair of FIFO queues that enable full-duplex signaling. The Cross-FIFO design is efficient to implement using PCIe transfer (*e.g.,* MMIO and DMA), hardware circuits (*e.g.,* UART), or emulation with RDMA (implementation in §7).

• *Scalable Multiplexing over Cross-FIFO*: Unlike SR-IOV, which dedicates QPs to each VF/PF, SG-IOV allows multiple warp pipes to share a single QP for signal transfer. The rationale is that warp pipes operate at the message level, where message sizes are typically larger than packets. As a result, the signaling and synchronization frequency is naturally lower than in traditional SR-IOV-based PCIe network devices, which operate at a smaller packet level. Thus, we propose multiplexing multiple warp pipes over the same Cross-FIFO, which is key to ensuring the scalability of SG-IOV. In addition to reduced signaling frequency enabled by large message sizes, another key enabler is that both the host front-end and SmartNIC back-end in SG-IOV are software



(a) DMA Example **(b)** Encryption Example

Figure 9. Size-Preserving and Size-Varying Job Examples. For the size-varying encryption job: In the first iteration (blue ones), the sink has four units of the first chunk, with two used for data after accounting for the header and trailer. In the second iteration (green ones), the source wraps around while the sink does not, processing the last two units before the boundary at the source. In the third iteration (grey ones), neither buffer wraps around, making job generation trivial.

components. This allows us to define a custom signal format and embed per-device information (*e.g.,* ring buffer ID) into each signal. As a result, a single Cross-FIFO can carry signals for multiple devices, and the back-end can *demultiplex* them based on the encoded signal fields. Specifically, each signal is represented as a 64-bit word encoding three fields:

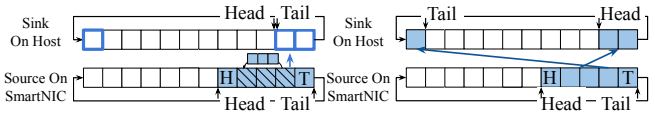
Signal Type (16b)	*RingBuffer ID (16b)*	*Head/Tail Pointer (32b)*

5.2 Flexible Message-Aware Job Generator

After synchronizing the latest memory views from the host or remote devices, the signal plane next generates jobs (*e.g.,* DMA jobs or encryption jobs) to feed into the hardware engines. The job generator's task is to determine which chunk of data in the source buffer and which chunk of space in the sink buffer should be used for each specific transformation job. As Figure 9 shows, SG-IOV flexibly supports two types of transformation jobs: Size-Preserving and Size-Varying jobs. Size-preserving one is a trivial case of size-varying jobs; next, we focus on the more general size-varying cases:

Size-Varying Job: As shown in Figure 9, this type of transformation varies the data length from source to sink buffers, either by increasing it (*e.g.,* appending a header and a trailer during encryption) or decreasing it (*e.g.,* removing a header and a trailer during decryption). This size variability complicates the logic of the job generator in several ways. First, the sink ring buffer advances its head and tail pointers differently from the source buffer due to size changes. Second, in the case of a ring buffer wrap-around, we need to carefully address a common hardware accelerator constraint: For a single job, the input data buffer must be located in a contiguous memory region, and the output space buffer must also be in a contiguous memory region. Given this *continuity constraint*, the job generator must handle eight cases arising from three binary conditions: whether the source wraps, whether the sink wraps, and whether the data size increases or decreases. We unify all cases with a clean strategy:

• *Recursive Strategy for General Wrap-Around Cases:* First, if the data in the source buffer or the space in the sink buffer wraps around, we divide them at the wrap-around boundary. This results in at most two data chunks and two space chunks. The first chunk is logically positioned before the



(a) Step-1: Transform In-Place **(b)** Step-2: DMA Chunks

Figure 10. In-Place Chunking with Decryption as an Example. An encrypted message (3 units) is decrypted into a sink buffer with only 2 contiguous units before wrap-around. Since the crypto engine requires continuity, the message is first decrypted in-place in the source buffer, then split into two DMA jobs. This avoids extra buffering and preserves integrity.

boundary, and the second chunk is positioned after it. Next, we pair the first data chunk with the first space chunk to generate a job. This step is straightforward since it involves contiguous regions without wrap-around complexities. After generating a job, we recursively repeat the chunking process on the remaining data and space. Each iteration pairs the first continuous data chunk with the first continuous space chunk, generating a new job. This recursive process iterates until we reach a case where neither the source buffer nor the sink buffer wraps around, leaving all remaining data and space in continuous regions. Figure 9b shows a case where both source and sink buffers wrap around. The recursive strategy works for all cases except the following one.

• *In-Place Chunking:* Figure 10 shows a special case of size-decreasing jobs where the source does not wrap but the sink does; we address this with in-place chunking.

6 Data Plane: Fine-Grained Virtualization

6.1 Job Queues and Scheduler

Enqueue Operations: As Figure 11 shows, we maintain a queuing system where each warp pipe is assigned a FIFO queue. Once jobs are generated, the signal plane enqueues them into the corresponding queue. Notably, a tenant may use multiple warp pipes, each with its own job queue.

Dequeue Operations and Job Scheduler: Within the data plane, the job scheduler dequeues job descriptors from the queuing system. The job scheduler supports Round-Robin, Priority, and more software-defined policies. Next, we introduce a policy specifically designed for SG-IOV.

Dominant Resource Fairness Policy: Considering that a single tenant may utilize multiple accelerators (Figure 11), we revisit the *Dominant Resource Fairness (DRF)* policy for heterogeneous resources [70, 71]. The key idea of the DRF policy is to ensure that all tenants receive an equal share of their dominant resource, where the dominant resource is the one they use most heavily relative to total capacity. When applying DRF to the SG-IOV context, we observe a natural and useful property, termed *Equal-Bandwidth Demand*: each multi-accelerator tenant consumes the same bandwidth on all accelerators it uses. This holds because, in a pipelined datapath of multiple accelerators, each *stage* processes data at the same bandwidth. Leveraging this property, we derive

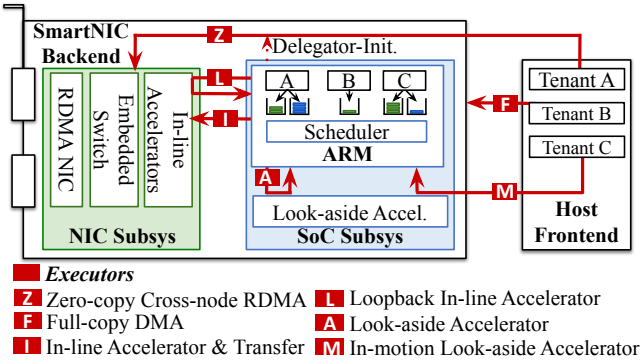


Figure 11. Job Executors and Scheduler of SG-IOV Data Plane. Description for different types of executors in Table 3.

Type	Description
Z	**Zero-Copy Cross-Node RDMA:** We use RDMA for cross-node transfers with zero-copy host-to-host data movement, avoiding staging on the SmartNIC. SG-IOV also supports *Delegator-Initialized* RDMA, where SoC cores submit jobs to QPs on behalf of the host.
F	**Full-Copy DMA:** We use DMA to implement a typical transformation to move data between the host and the device.
I	**In-line Acceleration and Transfer:** We use in-line hardware within the NIC subsystem, such as the Flow Engine, for VxLAN tunneling.
L	**Loopback In-line Acceleration:** This is similar to the above one, but it does not send packets to an external domain.
A	**Look-aside Acceleration:** For this type of transformation (*e.g.,* encryption), both input and output buffers are on the SoC subsystem.
M	**In-Motion Look-aside Acceleration:** This operates similarly to the above one but differs in that its input buffer resides on the host, and the output buffer is in the SoC.

Table 3. Description of Data Transformation Types.

a clean, closed-form allocation that conforms to the DRF. We present the formula here and more details in Appendix §A.3.

DRF under Equal-Bandwidth Demand: *Consider K accelerators with processing bandwidths $C_1, \ldots, C_K$. Tenant t uses a subset $S_t \subseteq \{1, \ldots, K\}$ at equal bandwidth on each resource in S_t. For tenant t, let the bottleneck capacity be $Z_{S_t} \triangleq \min_{j \in S_t} C_j$.*

Per-tenant allocation (per used resource):

$$a_t = s^* \cdot Z_{S_t} \quad \text{where} \quad s^* = \min_j \frac{C_j}{\sum_{t: j \in S_t} Z_{S_t}},$$

where the sum $\sum_{t: j \in S_t} Z_{S_t}$ is taken over all tenants that use resource j, adding each tenant's bottleneck capacity Z_{S_t}.

Example: Consider two accelerators with capacities $C_{\text{RDMA}} = 100$ Gbps and $C_{\text{Crypto}} = 60$ Gbps. Tenant T1 uses only {RDMA}, giving $Z_{S_1} = 100$, while Tenant T2 uses {RDMA, Crypto}, giving $Z_{S_2} = 60$. The common dominant share is

$$s^* = \min\left(\frac{100}{100+60}, \frac{60}{60}\right) = 0.625.$$

Thus, T1 receives $a_1 = 62.5$ Gbps on RDMA, and T2 receives 37.5 Gbps on both RDMA and Crypto.

6.2 Job Executors

Stateless Executor Abstraction. Executor provides SG-IOV's unified abstraction for heterogeneous accelerators. We design it to be stateless: after dequeuing a job descriptor, the executor completes the job using only descriptor fields

(*e.g.,* source and sink addresses). All stateful tasks—buffer tracking, dependency handling, and QoS enforcement—are delegated to the signal plane and scheduler.

Accelerators for Transformations. In Figure 11, with SoC-based SmartNICs, SG-IOV supports accelerators of the following types: 1) Look-aside Accelerators; 2) In-line Accelerators; 3) Data Transfer Engines; and 4) General-purpose SoC cores. We describe transformations in Table 3.

SG-IOV Control APIs: While SG-IOV focuses on I/O-intensive data APIs, auxiliary control APIs are in A.4.

7 Implementation

We prototype SG-IOV with NVIDIA BlueField-3. We comprehensively characterize BF3 based on SG-IOV's requirements (details in §A.2). We also explore other hardware, such as Marvell Octeon (details in §A.6).

7.1 SG-IOV Device Implementation

Due to space limits, we present only key implementation aspects here; further details, including optimization, dataplane implementation, and SDK usage, are in Appendix A.5.

PCIe Emulated Device: For intra-node host-to-SmartNIC warp pipes, we leverage the DOCA Device Emulation Library [41] to implement emulated PCIe physical functions. On the host front-end, we use the VFIO framework [51] to query the device's BAR regions and configure MMIO for doorbells and eventfd for MSI-X interrupts. To support signal-word transfers from host to device, doorbells allow inline messages in 32-bit units. In contrast, MSI-X interrupts do not support inline messaging; therefore, we use either a PCIe BAR stateful region or DMA to transfer messages from device to host.

Mediated Pass-through Device: Atop emulated physical functions, SG-IOV further scales by provisioning host-to-SmartNIC warp pipes as mediated pass-through devices. A device provisioning daemon runs on the host front-end to manage these mediated devices. The daemon uses the Cross-FIFO interface of the physical device to relay signals to the SmartNIC. Importantly, the daemon mediates only signal-plane operations; the data-plane ring buffer is directly mapped, avoiding any additional data copy.

SG-IOV Internal Modules. We implement SG-IOV internal modules in about 10.3K lines of C/C++ code, with detailed modularity shown in Table 8 of Appendix A.5.

7.2 SG-IOV Container Network Interface (CNI)

We build *SGIOV-CNI*, a container network solution atop SG-IOV that is POSIX-compliant, requires no container image changes, and integrates with Docker and Kubernetes.

SGIOV-CNI for Secure Containers: We primarily support SGIOV-CNI for secure containers, widely adopted by hyperscalers. As described in §3.1, secure containers run inside a MicroVM with a guest kernel. Warp pipes, implemented as pass-through devices, are exposed to the guest kernel (Figure 3b). We implement a lightweight warp pipe driver within the guest kernel, which integrates with the socket layer to support socket-related syscalls invoked by containerized applications. This driver interacts with the device provisioning daemon, which is integrated as a component of the VMM used by secure container runtimes to manage MicroVMs. To implement QP sharing, the daemon intercepts guest-kernel MMIO operations to PCIe resources (*e.g.,* registers and QPs), remaps them in software, and performs QP multiplexing by encoding MicroVM-issued QP actions (*e.g.,* TX descriptor submission) with a compact signal (§5.1). On the SmartNIC, the backend performs demultiplexing: it fetches MMIO commands from the warp pipes, extracts the signal fields, and generates the corresponding transformation job (*e.g.,* DMA).

SGIOV-CNI for Traditional Containers: Although we focus on secure containers, SGIOV-CNI also integrates with Linux kernel-based runtimes; details are in Appendix A.5.

8 Evaluation

8.1 Setup

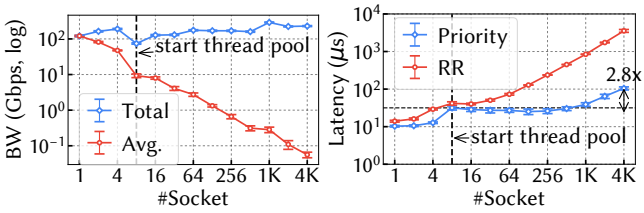
Testbed: Our testbed consists of two servers connected to a 100GbE switch. Each server has an NVIDIA BlueField-3 (B3220) with a 100GbE cable. Each server is equipped with two Intel Xeon Gold 6346 processors, 384GB DDR4, and PCIe Gen4×16 links. The host machines run Ubuntu 22.04 with kernel 5.15, and DOCA v3.0.

Measurements: We report bandwidth, latency, and CPU utilization; each test runs for 5 minutes with a 10-second warm-up, is repeated five times, and reports standard deviations as error bars in the figures.

Baselines: For end-to-end comparisons with SGIOV-CNI, we have two baselines: 1) We use *Cilium* (v1.16) [18], a popular, versatile, and high-performance CNI adopted by hyperscalers [19, 20]; 2) We build a baseline using NVIDIA *Sub-Functions* (*i.e.,* Scalable Functions, SF) [96], configured following the official user guide [95]. Then, we deploy the *SR-IOV CNI Plugin* [47], a standard CNI that provisions SR-IOV PFs/VFs and SFs [65, 94] for containers. Notably, the SR-IOV CNI provides only device passthrough and thus lacks native support for essential CNI functionalities such as virtualization, L7 HTTP proxy, and traffic security. To support these functionalities, we follow the official DOCA VNF (virtual network function) reference application [22] to implement them on the SmartNIC SoC. Overall, the baseline using SFs follows the datapath illustrated in Figure 6.

8.2 Micro-Benchmarking SG-IOV

8.2.1 Evaluate Scalability. In Figure 12, we measure the number of concurrent devices that SG-IOV can support. We use POSIX `send()/recv()` over warp pipes while measuring the host-to-device bandwidth and the round-trip latency. For



(a) Scalability (Bandwidth) **(b)** Scalability (Latency)

Figure 12. Benchmarking SG-IOV Scalability.

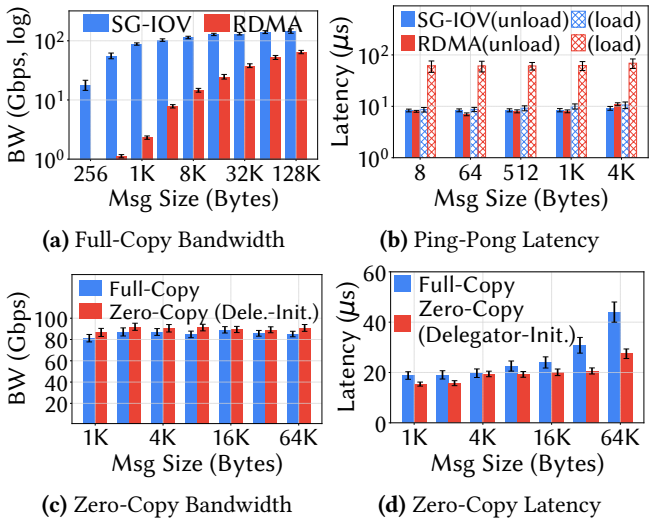
bandwidth tests, we use 128KB messages, and for latency, 4KB messages (varying sizes in later experiments).

In Figure 12a, we scale the number of sockets up to 4K, keeping an aggregate bandwidth of up to 190Gbps (PCIe Gen4×16) across all settings, which is sufficient to match the 100GbE network line rate. Moreover, each socket can fairly share bandwidth without starvation, even at the massive scale of 4K sockets. While there are no fundamental limitations for scaling beyond 4K, we stop since the 4K source/sink buffers already consume a relatively large 8GB of memory. Note that the number of sockets far exceeds the number of CPU cores, so we use a thread pool to multiplex all socket operations with a fixed number of cores (8 cores in our setup). Importantly, these 8 CPU cores are dedicated to the traffic generator and do not contribute to the resource overhead of SG-IOV. The SG-IOV resource setup remains consistent, as described earlier, regardless of the number of sockets.

In Figure 12b, we present latency results while scaling the number of sockets. We try two job scheduler policies: priority scheduling in a work-conserving manner and round-robin (RR). For priority scheduling, we measure the latency of the socket with higher priority while it competes with other lower-priority sockets. While scaling the number of sockets by 500x, from 8 to 4K, the latency increases by only 2.8x, demonstrating that SG-IOV can not only support a large number of concurrent sockets but also execute the workflow with the expected priority. The 2.8x increase in latency is mainly due to the work-conserving manner, which ensures the system remains busy. For comparison, scaling SR-IOV to 128 VFs results in a 3.75x increase in RDMA latency (`ib_write` [6]). For the RR scheduling, the latency increases as the sockets fairly share the processing resources.

8.2.2 Evaluate Flexibility and Efficiency. We evaluate both size-preserving data transfers and size-varying transformations. Firstly, for data transfers (Figure 13), we evaluate the efficiency of full-copy host-to-device data transfers and zero-copy host-to-host data transfers (delegator initiates).

Figure 13a presents the throughput when transferring data from the host to the SmartNIC SoC. A single H2D warp pipe can achieve a maximum throughput of 150Gbps (D2H performance is symmetric). For comparison, we also present results for RDMA-based solutions where RDMA implements both signaling and intra-node data copying. SG-IOV, by default,



(a) Full-Copy Bandwidth **(b)** Ping-Pong Latency

(c) Zero-Copy Bandwidth **(d)** Zero-Copy Latency

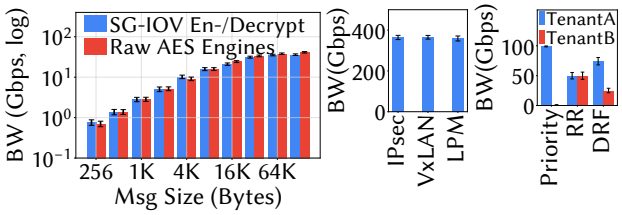
Figure 13. Benchmarking Data Transfer Performance of SG-IOV: Full-Copy Host-to-Device and Zero-Copy Host-to-Host.

utilizes BF3 HW-FIFO for signaling and DMA for intra-node data copying, which outperforms RDMA-based solutions.

Figure 13b presents the latency results of the warp pipe. To measure the latency, we use a ping-pong test where the host produces the data, and the device acts as a bouncer. The round-trip time is around 8μs. The completion time of D2H and H2D DMA jobs (2×4 μs) accounts for the majority. Additionally, we benchmarked the DMA-based solution against RDMA-based ping-pong between the host and the device. In unloaded scenarios without inter-server traffic, "RDMA (unload)" achieves similar latency compared to the DMA-based warp pipe. However, under loaded conditions, such as when generating 100Gbps inter-server traffic with *perftest* [6] `ib_read`, RDMA ping-pong latency significantly escalates due to contention in the NIC. SG-IOV maintains low latency under loaded conditions, thanks to HW-FIFO and DMA being less impacted by cross-server RDMA.

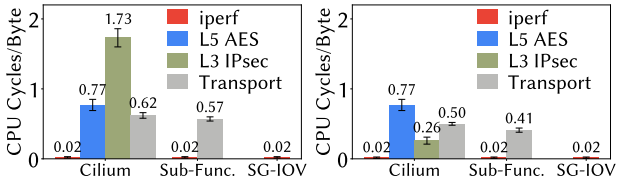
In Figures 13c and 13d, we present results for the zero-copy inter-node transformations. As described in §6.2, we support Full-Copy (FC) and Delegator-Initialized Zero-Copy (ZC) for inter-node transfers. The results indicate that Zero-Copy Mode achieves lower latency and higher throughput by bypassing the SmartNIC SoC subsystem. Especially for large messages, the savings in data movement costs are notable with Zero-Copy Mode. Despite the data movement costs, Full-Copy Mode is also beneficial for some scenarios requiring going through the SoC subsystem to utilize diverse look-aside accelerators and other SoC units.

Next, we evaluated size-varying transformations. In Figure 14a, we present the results for the in-motion look-aside acceleration. We use two warp pipes; one is for H2D with AES-GCM encryption as the transformation, allowing us to write plaintext data on the host and then read the encrypted data from the SmartNIC SoC, while the other is for D2H with AES-GCM decryption as the transformation. This



(a) In-motion Look-aside **(b)** Inline Task **(c)** Scheduler

Figure 14. Benchmarking Advanced Transformations.



(a) Sender **(b)** Receiver

Figure 15. Host CPU Core Savings from Offloading Cases: L3/L5 Securing Tasks and Transport over Tunnels.

pair of warp pipes is used for message-level encryption in container networks. We show throughput results for the loopback test, taking into account both encryption and decryption. We use the raw stress-test throughput of the AES engine as the baseline, where jobs are executed directly without any system integration. The results show that SG-IOV not only integrates crypto engines but also sustains their peak hardware capacity.

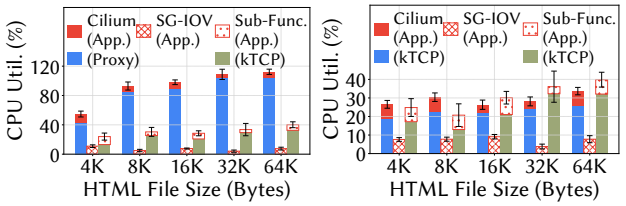
Figure 14b shows loop-back in-line benchmarks for IPsec, VxLAN, and Longest Prefix Matching (LPM). The results indicate that we achieved nearly a 400Gbps processing rate with in-line accelerations for SoC-initialized loop-back tests, near the peak bandwidth of the BF3 on-board interconnect.

8.2.3 Evaluate Scheduling. Figure 14c illustrates two tenants contending for an AES-GCM engine. SG-IOV supports accelerator isolation via Priority, Round Robin, and DRF scheduling. For the DRF configuration, one tenant uses only RDMA, while the other uses RDMA (~100 Gbps for 128 KB) plus look-aside encryption (~40 Gbps for 128 KB), yielding an approximate bandwidth ratio of 7:3 for RDMA under the DRF policy. The results demonstrate that SG-IOV can faithfully enforce resource allocations according to each policy.

8.3 End-to-End Evaluating Container Networks

In this section, we conduct an end-to-end evaluation of the SGIOV-CNI. For the baselines, we use Cilium and the CNI over NVIDIA Sub-Functions (*abbr.* Sub-Func. or SF), with setup in §8.1. We focus on offloading three typical tasks: transport over tunnels, L7 HTTP proxy, and secure transfers.

8.3.1 Saving Host CPU Cycles. Figure 15 shows core savings from offloading transport over tunnels and L3/L5 security tasks. In this test, we use *iperf3* [32] (under default settings, 128KB message) as a containerized application and



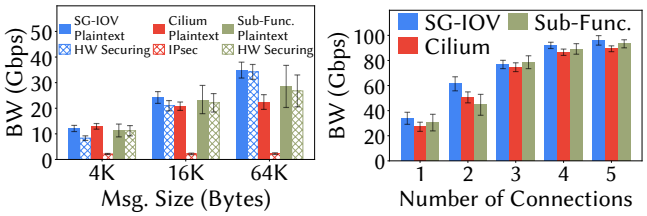
(a) Core Saving (With Envoy) **(b)** Core Saving (Bypass Envoy)

Figure 16. Host CPU Core Savings from Offloading Case: Application-Layer Processing of HTTP Proxy.

linux-perf [8] for reporting CPU cycle counts. We configure Cilium to enable the built-in functionalities: in-kernel TCP/IP, VxLAN, and IPsec. As mentioned in §3.1, Cilium has not enabled end-to-end L5 encryption. Thus, we implement message-level encryption within the containerized applications using the OpenSSL AES-GCM cryptographic suite. For SFs, we offload VxLAN tunneling and traffic security (L5 AES/L3 IPsec) using the DOCA VNF application on the SmartNIC SoC. We present core utilization in terms of CPU Cycles Per Byte, a more consistent metric across different CPU generations than CPU utilization. Compared with Cilium, offloading results in a saving of 4.65 cycles per byte at the sender (Figure 15a) and receiver side (Figure 15b). On our testbed, equipped with a CPU @3.10GHz, these cycle savings translate to a total saving of ~1.9 cores per 10Gbps for both sender and receiver combined. Compared with SFs, SG-IOV saves 0.98 cycles per byte at both the sender and receiver, since SFs expose only L2 packet interfaces and still require in-kernel L4/L3 TCP/IP processing to serve sockets for containers. The overhead is illustrated in Figure 6 and discussed as *Issue #2* in §3.3.

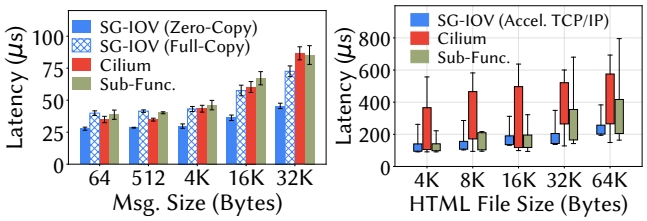
Figure 16a demonstrates offloading application-level processing. Cilium typically uses an Envoy proxy for HTTP processing: requests are generated by *wrk* [9] in a container, pass through Envoy for load balancing and URL rewriting, and are served by an *nginx* [38] backend server. We divide the host CPU usage into two segments: application processing and the proxy. For SG-IOV, the Envoy is shifted onto the SmartNIC, leaving only the containerized application to run on the host. Notably, Envoy is CPU-intensive. For instance, with a single connection and a 64KB response length, Envoy alone utilizes nearly one full CPU core (104.6%). For SFs, we offload the Envoy to the SmartNIC via DOCA VNF to intercept traffic and handle L7 processing, eliminating host CPU overhead for proxying compared with Cilium. However, SFs still incur host-side kernel TCP/IP (kTCP) processing, yielding smaller offloading benefits than SG-IOV.

Figure 16b shows host core savings from offloading the kernel data path without Envoy under the same workloads. In this test, even though HTTP requests are directly sent out through the kernel network stack while bypassing Envoy, the host kernel still consumes CPU cycles for the network stack. To operate the single connection continuously requesting



(a) Single Connection **(b)** Multi Connections

Figure 17. Throughput of the SG-IOV Container Network.



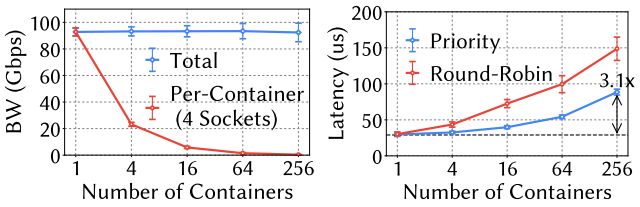
(a) NPtcp Latency **(b)** Nginx HTTP Latency

Figure 18. Latency of the SG-IOV Container Network.

64KB-size HTML files, the kernel network stack consumes about 0.24 host CPU cores. Similarly, for SFs, kTCP also consumes host CPU cycles.

8.3.2 Increasing Throughput. Figure 17 shows throughput results. In Figure 17a, we set up a single connection between two containers using *iperf3*. For the SF CNI, we configure DOCA VNF for plaintext forwarding and use DOCA Flow [21] with E-Switch hardware acceleration for encrypted traffic. For plaintext stream transmission, SG-IOV achieves higher throughput. For example, with a 128KB message size, SG-IOV sustains 38.0Gbps, which is 53% higher than Cilium thanks to SG-IOV's efficient inter-node transfers. SG-IOV achieves up to 22% higher bandwidth than SFs by streamlining the datapath through direct exposure of message-level devices to containers, while the SF CNI still incurs host CPU overhead for in-kernel message-to-packet processing. For encrypted data, Cilium, which relies on host x86 cores for IPsec, is limited to a maximum of 3Gbps for a single connection with a 1500B MTU. Even with an increased MTU of 9000B, throughput remains around 8.8Gbps. In contrast, SG-IOV with hardware encryption maintains high throughput (up to 37.2Gbps, 12.4x higher than Cilium). Moreover, SFs deliver comparable encrypted-traffic bandwidth to SG-IOV, as both use hardware-accelerated encryption, while SG-IOV further streamlines the datapath by avoiding host CPU involvement in message-to-packet processing.

Figure 17b displays the throughput performance as the number of connections increases. SG-IOV maintains throughput scalability similar to Cilium. In this test, we utilize plaintext transfer with a message size of 128KB. On our 100GbE testbed, both SG-IOV and Cilium reach over 90Gbps. SG-IOV surpasses both Cilium and SFs in terms of higher throughput.



(a) Throughput **(b)** Latency

Figure 19. SG-IOV Perf. with Increasing Container Count.

Memory per Socket	RingBuffer (Configurable)	1 MB × 2 (RD/WR)
	Metadata	256 B
	Job Queue	64B × 128 Entries

Table 4. Resource Accountability for SG-IOV Devices.

8.3.3 Reducing Latency. In Figure 18, we measure network latencies with real-world containerized applications. In Figure 18a, we measure latencies using *NPtcp* [39] in containers. SG-IOV achieves lower latency than Cilium, especially when using delegator-initialized zero-copy (§6.2) for inter-node transfers. For example, this results in a 48% reduction in latency for a 32KB message. Moreover, even in full-copy mode, SG-IOV achieves performance on par with Cilium, thanks to efficient signaling and intra-/inter-node transfers. SG-IOV outperforms SFs because CNI over SFs incurs an inefficient *cyclic stream-to-packet transformation* (shown in Figure 6). Notably, we even carefully optimized the SF CNI by offloading only lightweight functions to the SmartNIC SoC and enabling DPDK and DOCA Flow acceleration. SG-IOV, especially in zero-copy mode, employs a more streamlined datapath in which only control operations are handled by the SmartNIC SoC, enabling low-latency data transfer.

In Figure 18b, we measure latencies using containerized *wrk* to generate HTTP requests for backend *Nginx* servers. We show HTTP request latencies (min, P25, P75, P99) in a box plot. For Cilium, HTTP requests pass through the in-kernel network stack. Since legacy *Nginx* lacks RDMA support, SG-IOV instead uses an accelerated user-space TCP/IP stack over ARM cores (XLIO library [54] from NVIDIA). The results show that SG-IOV achieves lower latency than Cilium, particularly in terms of tail latency (*e.g.*, a 46% reduction for a 4KB HTML response). SG-IOV achieves comparable average latency to SFs but better tail latency, as SFs rely on host CPU involvement for L5-to-L2 CNI processing, introducing additional variability.

8.3.4 Scalability with Increasing Container Counts. In Figure 19a, we measure throughput using *iperf3* with a setup similar to the experiments in Figure 17. For each container, we configure four concurrent sockets, which can achieve reasonably high performance, as shown in Figure 17b. In Figure 19a, we report both aggregate and per-container bandwidth, showing that SG-IOV CNI scales well with increasing container counts while maintaining high throughput and fair per-container bandwidth with low variance.

In Figure 19b, we measure latency using containerized *NPtcp* with a setup similar to Figure 18a (64B messages). As demonstrated in the microbenchmark (Figure 12b), SG-IOV supports multiple scheduling policies, including priority and round-robin. Under priority scheduling, the latency of a high-priority container increases by only 3.1× as container count scales to 256, consistent with the microbenchmark (Figure 12b). Under round-robin scheduling, latency increases as expected due to bandwidth sharing among containers.

Table 4 summarizes the memory footprint of SG-IOV devices to analyze scalability. The main cost is read/write ring buffers. We use a 1MB default buffer size, similar to Linux, and SG-IOV can scale to many devices by reducing buffer sizes. Other overheads are modest: 256B per device for metadata (*e.g.,* head/tail pointers) and 8KB per device for job queues, which can scale given BF3's 32GB memory.

8.3.5 Cost–Benefit Analysis of SmartNIC Offloading.

Besides performance gains, we compare the host CPU and SmartNIC from the following aspects:

• CapEx: We compare capital expenditure (CapEx) using price quotes for our testbed, sourced from a single OEM to avoid cross-vendor variability. We normalize costs to the CPU price. One BF3 SmartNIC costs 39.1% of the CPU price, while a dual-port 100GbE NIC (Intel E810) costs 10.8%, resulting in a 28.3% SmartNIC premium over a conventional NIC. As shown in §8.3.1, processing 100 GbE CNI traffic consumes 29.7% of host CPU cores (1.9 cores per 10Gbps for 64 cores). Thus, SmartNICs are CapEx-competitive with host CPUs, consistent with prior observations [11, 59];

• Operation Profits: SG-IOV frees host CPU cores for revenue-generating workloads, a benefit sustained throughout the server life cycle and aligned with major cloud providers' motivation to reduce the I/O Virtualization tax [14, 68, 119];

• OpEx (Energy): We compare operating expenditure (OpEx) via energy consumption. Using *IPMI* [48], we measure runtime power under the same setup as Figure 17 (five connections at line-rate). The Cilium baseline consumes about 397 W, while SG-IOV consumes 290W, yielding a 107W reduction. Normalized to the 410W CPU TDP (Thermal Design Power), this corresponds to a 26.1% power saving;

• Interference: SmartNIC offloading reduces interference between CNI functionalities and co-located applications. With the same setup as Figure 16a, co-locating *iperf3* with *Envoy* on one core reduces throughput by up to 49.8%, which SmartNIC offloading helps avoid. Furthermore, from a security perspective, the SmartNIC is isolated from the host CPU, providing a strong trust model [16].

9 Related Work

I/O Virtualization: OCP S-IOV (contributed by Intel) [45] and HD-IOV [120] enhance SR-IOV scalability for ASIC NICs. We target SmartNICs with a software-mediated backend; unlike S-IOV and HD-IOV that allocate one QP per device, SG-IOV can multiplex within a QP and further adds support for size-varying transformations and fine-grained virtualization. FastIOV [90] speeds up IOV device setup for secure containers, an orthogonal optimization that could also benefit SG-IOV pass-through devices. We also refer to NVMe IOV studies [41, 73, 80] for background on PCIe device emulation.

Host-NIC Interaction: As noted earlier, *Pismenny et al.* [99] propose novel *autonomous offloading* for ASIC NICs to support L5+ processing, but it relies on size-preserving transformations. NICA [64] demonstrates in-line acceleration on FPGA-based SmartNICs, while SG-IOV primarily leverages SoC-based SmartNICs and integrates additional accelerators such as look-aside engines. FlexDriver [63] attaches drivers to accelerators, while SG-IOV integrates accelerators directly into the datapath. ASNI [113], Enso [103], and more [66, 98] explore metadata and payload buffer layouts, as well as host-NIC interactions. While prior works emphasize size-preserving NIC transport, SG-IOV uses software at the front- and back-end to flexibly design metadata/payload handling and integrate diverse accelerators.

Container Networks: Cloud providers evolve K8S Service CNIs (*e.g.,* Azure with enhanced security [3]), yet CPU overhead and performance issues persist, which SGIOV-CNI addresses. CNI has many solutions [17, 25, 53, 76, 125]; SG-IOV uses efficient offloading to deliver equivalent rich features. For microservice, CNI is an essential component [60, 72, 78, 101, 106], which can benefit from SGIOV-CNI improvements. Beyond containers, SG-IOV PCIe device potentially supports other virtualized OS such as Virtual Machines and Unikraft [79, 83, 118].

SmartNIC Offloading: In SR-IOV offloading, AccelNet [67] uses VFs for host tenants and AWS EFA (*i.e.,* Nitro) exemplifies emulated PCIe devices. Unlike these VM-oriented paradigms, SG-IOV targets containerized clouds. SmartNIC offloading covers many use cases [58, 74, 77, 82, 84, 89, 92, 97, 102, 105, 109, 111, 116, 117]. Solar [92] and BurstCBS [109] apply signal/data plane separation to storage; in a similar spirit, SG-IOV develops different techniques for CNIs.

10 Conclusion

This paper presents SG-IOV, a new IOV mechanism to offload container networks onto SmartNICs. SG-IOV is scalable, flexible, and provides granular virtualization. SGIOV-CNI can help containerized cloud reap substantial core savings and boost performance. We envision SG-IOV as an abstraction for more I/O scenarios and offloading devices.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was supported in part by NSF grants CNS-2212193, CNS-2213387, CNS-2504469, CNS-2106199, CNS-2212192, CAREER-2339755, CNS-2530909, and NEUTC.

References

[1] 2011. Intel PCI-SIG SR-IOV Primer. https://www.intel.com/content/www/us/en/content-details/321211/pci-sig-sr-iov-primer-an-introduction-to-sr-iov-technology.html.

[2] 2022. ConnectX SmartNICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[3] 2024. Advanced Container Networking Services: Enhancing security and observability in AKS. https://azure.microsoft.com/en-us/blog/advanced-container-networking-services-enhancing-security-and-observability-in-aks/.

[4] 2024. ARM TX FIFO. https://developer.arm.com/documentation/ddi0194/h/functional-overview/primecell-ssp-functional-description/transmit-fifo.

[5] 2024. Intel TX FIFO. https://www.intel.com/content/www/us/en/docs/programmable/683617/21-1/tx-fifo-and-rx-fifo.html.

[6] 2024. Linux-rdma perftest . https://github.com/linux-rdma/perftest.

[7] 2024. OpenSSL. https://www.openssl.org/.

[8] 2024. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[9] 2024. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk.

[10] 2024. xilinx TX FIFO. https://docs.xilinx.com/r/en-US/am017-versal-gtm-transceivers/Using-the-TX-FIFO.

[11] 2025. Accelerating Cloud-Ready Infrastructure and Kubernetes with Red Hat OpenShift and the NVIDIA BlueField DPU. https://developer.nvidia.com/blog/accelerating-cloud-ready-infrastructure-and-kubernetes-with-red-hat-openshift-and-bluefield-dpu/.

[12] 2025. Alibaba Cloud Container Service for Kubernetes. https://www.alibabacloud.com/product/kubernetes.

[13] 2025. AMD Pensando. https://www.amd.com/en/products/data-processing-units/pensando.html.

[14] 2025. AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[15] 2025. Azure Container Instance. https://azure.microsoft.com/en-us/products/container-instances/.

[16] 2025. BlueField Modes of Operation. https://docs.nvidia.com/doca/sdk/bluefield-modes-of-operation/index.html.

[17] 2025. Calico. https://www.tigera.io/project-calico/.

[18] 2025. Cilium. https://cilium.io/.

[19] 2025. Cilium cluster-wide network policy for Google Kubernetes Engine (GKE). https://docs.cloud.google.com/kubernetes-engine/docs/how-to/configure-cilium-network-policy.

[20] 2025. Configure Azure CNI Powered by Cilium in Azure Kubernetes Service (AKS). https://learn.microsoft.com/en-us/azure/aks/azure-cni-powered-by-cilium.

[21] 2025. DOCA Flow. https://docs.nvidia.com/doca/sdk/doca+flow/index.html.

[22] 2025. DOCA Simple Forward VNF Application Guide. https://docs.nvidia.com/doca/sdk/doca-simple-forward-vnf-application-guide/index.html.

[23] 2025. eBPF. https://ebpf.io/.

[24] 2025. Envoy. https://www.envoyproxy.io/.

[25] 2025. Flannel. https://github.com/coreos/flannel/.

[26] 2025. Google Kubernetes Engine . https://cloud.google.com/kubernetes-engine.

[27] 2025. gVisor. https://gvisor.dev/.

[28] 2025. Improving the security of Cilium Mutual Authentication. https://cilium.io/blog/2024/03/20/improving-mutual-auth-security/.

[29] 2025. Intel Infrastructure Processing Unit. https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.

[30] 2025. Introduction to vDPA kernel framework. https://www.redhat.com/en/blog/introduction-vdpa-kernel-framework.

[31] 2025. ip-xfrm — Linux manual page. https://man7.org/linux/man-pages/man8/ip-xfrm.8.html.

[32] 2025. iperf3. https://github.com/esnet/iperf.

[33] 2025. Istio service mesh. https://istio.io/.

[34] 2025. Kata Containers. https://katacontainers.io/.

[35] 2025. Kube-OVN. https://www.kube-ovn.io/.

[36] 2025. Marvell Octeon. https://www.marvell.com/products/data-processing-units.html.

[37] 2025. Netronome Agilio. https://netronome.com/agilio-smartnics/.

[38] 2025. Nginx. https://www.nginx.com/.

[39] 2025. NPtcp. https://manpages.ubuntu.com/manpages/xenial/man1/NPtcp.1.html.

[40] 2025. NVIDIA BlueField. https://www.nvidia.com/en-in/networking/products/data-processing-unit/.

[41] 2025. NVIDIA DOCA Software Framework. https://developer.nvidia.com/networking/doca.

[42] 2025. Pod Sandboxing with Azure Kubernetes Service (AKS) . https://learn.microsoft.com/en-us/azure/aks/use-pod-sandboxing.

[43] 2025. Quark Container Runtime. https://github.com/QuarkContainer/Quark.

[44] 2025. Red Hat Blog: Running 2500 pods per node. https://www.redhat.com/en/blog/running-2500-pods-per-node-on-ocp-4.13.

[45] 2025. Scalable I/O Virtualization Revision. https://www.opencompute.org/documents/ocp-scalable-io-virtualization-technical-specification-revision-1-0-version-1-3-pdf.

[46] 2025. SR-IOV CNI plugin. https://github.com/k8snetworkplumbingwg/sriov-cni.

[47] 2025. SR-IOV Network Device Plugin for Kubernetes. https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin.

[48] 2025. Supermicro IPMI Utilities. https://www.supermicro.com/en/solutions/management-software/ipmi-utilities.

[49] 2025. UART FIFO. https://www.realdigital.org/doc/ef3498f424aaa1b20b5199907ba12b4c.

[50] 2025. vDPA: virtio Data Path Acceleration. https://vdpa-dev.gitlab.io/.

[51] 2025. Virtual Function I/O. https://docs.kernel.org/driver-api/vfio.html.

[52] 2025. Virtual I/O Device (VIRTIO) Version 1.3. https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.html.

[53] 2025. Weave Net. https://github.com/weaveworks/weave.

[54] 2025. XLIO-Accelerated IO SW library,. https://github.com/Mellanox/libxlio.

[55] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[56] Amazon. 2025. AWS Fargate. https://aws.amazon.com/fargate/.

[57] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, Tommaso Pimpo, Ariff Premji, Pranjal Shrivastava, and Rishabh Tewari. 2022. Bluebird: High-performance SDN for Bare-metal Cloud Services. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 355–370. https://www.usenix.org/conference/nsdi22/presentation/arumugam

[58] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. 2023. Disaggregating Stateful Network Functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1469–1487. https://www.usenix.org/conference/nsdi23/presentation/bansal

[59] Idan Burstein. 2021. NVIDIA Data Center Processing Unit (DPU) Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 1–20.

[60] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, et al. 2024. Yuanrong: A production general-purpose serverless system for distributed applications in the cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 843–859.

[61] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. 2024. Demystifying datapath accelerator enhanced off-path smartnic. *arXiv preprint arXiv:2402.03041* (2024).

[62] Andy Currid. 2004. TCP offload to the rescue: Getting a toehold on TCP offload engines—and why we need them. *Queue* 2, 3 (2004), 58–65.

[63] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. 2022. Flexdriver: A network driver for your accelerator. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 1115–1129.

[64] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 345–362.

[65] F5. 2025. Configure SR-IOV Network Device Plugin. https://clouddocs.f5.com/bigip-next-for-kubernetes/latest/sriov-plugin.html.

[66] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: toward per-Core 100-Gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3445814.3446724

[67] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.

[68] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.

[69] Cloud Native Computing Foundation. 2025. CNI - the Container Network Interface. https://github.com/containernetworking/cni.

[70] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Helsinki, Finland) *(SIGCOMM '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2342356.2342358

[71] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 323–336.

[72] Anyesha Ghosh, Neeraja J. Yadwadkar, and Mattan Erez. 2024. Fast and Efficient Scaling for Microservices with SurgeGuard. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41406.2024.00103

[73] Peter-Jan Gootzen, Jonas Pfefferle, Radu Stoica, and Animesh Trivedi. 2023. DPFS: DPU-powered file system virtualization. In *Proceedings of the 16th ACM International Conference on Systems and Storage*. 1–7.

[74] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. 2020. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of the ACM SIGCOMM*. 681–693.

[75] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. 2020. MASQ: RDMA for virtual private cloud. In *Proceedings of the ACM SIGCOMM*. 1–14.

[76] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds.. In *NSDI*. 113–126.

[77] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. 2023. Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023*. USENIX.

[78] Samuel Kounev, Cristina Abad, Ian Foster, Nikolas Herbst, Alexandru Iosup, Samer Al-Kiswany, Ahmed Ali-Eldin Hassan, Bartosz Balis, Andre Bauer, Andre Bondi, et al. 2021. Toward a definition for serverless computing. (2021).

[79] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. doi:10.1145/3447786.3456248

[80] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. 2020. FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 955–971.

[81] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. 2024. Sabre: Hardware-Accelerated snapshot compression for serverless MicroVMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 1–18.

[82] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 36–51. doi:10.1145/3445814.3446696

[83] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 467–482. doi:10.1145/3503222.3507759

[84] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. 2020. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 591–605.

[85] Xiaoyu Li, Ran Shu, Yongqiang Xiong, and Fengyuan Ren. 2024. Software-based Live Migration for Containerized RDMA. In *Proceedings of the 8th Asia-Pacific Workshop on Networking* (Sydney, Australia) *(APNet '24)*. Association for Computing Machinery, New York, NY, USA, 52–58. doi:10.1145/3663408.3663416

[86] 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 *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 53–68.

[87] 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 14th USENIX Conference on Operating Systems Design and Implementation*. 243–259.

[88] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1505–1519. https://www.usenix.org/conference/nsdi23/presentation/liu-david

[89] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM SIGCOMM*. 318–333.

[90] 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 Twentieth European Conference on Computer Systems*. 720–735.

[91] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. 2019. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 399–413.

[92] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. 2022. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference (Sigcomm 22)*.

[93] Zhixiong Niu, Qiang Su, Peng Cheng, Yongqiang Xiong, Dongsu Han, Keith Winstein, Chun Jason Xue, and Hong Xu. 2021. NetKernel: Making network stack part of the virtualized infrastructure. *IEEE/ACM Transactions on Networking* (2021).

[94] NVIDIA. 2025. DOCA Documentation: Kubernetes Using SR-IOV. https://docs.nvidia.com/doca/sdk/kubernetes-using-sr-iov/index.html.

[95] NVIDIA. 2025. DOCA Documentation: Scalable Functions. https://docs.nvidia.com/doca/sdk/scalable-functions/index.html.

[96] NVIDIA. 2025. Scalable functions (aka sub functions). https://github.com/Mellanox/scalablefunctions.

[97] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. 2022. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 761–777.

[98] Solal Pirelli and George Candea. 2020. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 225–241.

[99] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. 2021. Autonomous NIC offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. 18–35.

[100] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. 2021. MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications.. In *USENIX Annual Technical Conference*. 47–63.

[101] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. 2022. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 780–794.

[102] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 772–787.

[103] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S Berger, James C Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. 2023. Ensō: A Streaming Interface for NIC-Application Communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 1005–1025.

[104] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 969–985. https://www.usenix.org/conference/osdi23/presentation/saokar

[105] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 740–755.

[106] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[107] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.

[108] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 87–102.

[109] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2024. Burstable cloud block storage with data processing units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 783–799.

[110] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, et al. 2024. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 860–875.

[111] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. sRDMA: efficient NIC-based authentication and encryption for remote direct memory access. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 691–704.

[112] The Unified Communication X Library 2025. The Unified Communication X Library. http://www.openucx.org.

[113] Nikita Tyunyayev, Clément Delzotti, Haggai Eran, and Tom Barbette. 2025. ASNI: Redefining the Interface Between SmartNICs and Applications. *Proceedings of the ACM on Networking* 3, CoNEXT2 (2025), 1–22.

[114] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 987–1004.

[115] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. 2022. KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 121–136.

[116] Yunming Xiao, Diman Zad Tootaghaj, Aditya Dhakal, Lianjie Cao, Puneet Sharma, and Aleksandar Kuzmanovic. 2024. Conspirator: SmartNIC-Aided Control Plane for Distributed ML Workloads. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 767–784. https://www.usenix.org/conference/atc24/presentation/xiao

[117] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, TS Eugene Ng, et al. 2023. Unleashing SmartNIC packet processing performance in P4. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1028–1042.

[118] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. doi:10.1145/3477132.3483569

[119] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-density multi-tenant bare-metal cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 483–495.

[120] 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 Nineteenth European Conference on Computer Systems*. 834–850.

[121] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2025. White-Boxing RDMA with Packet-Granular Software Control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 427–449. https://www.usenix.org/conference/nsdi25/presentation/zhao-chenxingyu

[122] Chenxingyu Zhao, Yulin Sun, Ying Xiong, and Arvind Krishnamurthy. 2023. Quark: A High-Performance Secure Container Runtime for Serverless Computing. *arXiv preprint arXiv:2309.12624* (2023).

[123] Jiechen Zhao, Ran Shu, Katie Lim, Zewen Fan, Thomas Anderson, Mingyu Gao, and Natalie Enright Jerger. 2024. Accelerator-as-a-Service in Public Clouds: An Intra-Host Traffic Management View for Performance Isolation in the Wild. arXiv:2407.10098 [cs.OS] https://arxiv.org/abs/2407.10098

[124] Jiechen Zhao, Ran Shu, Katie Lim, Zewen Fan, Thomas Anderson, Mingyu Gao, and Natalie Enright Jerger. 2024. Arcus: SLO Management for Accelerators in the Cloud with Traffic Shaping. *arXiv preprint arXiv:2410.17577* (2024).

[125] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 331–344.

A Appendix

A.1 I/O Virtualization (IOV) Primer

Para Virtualization (PV) enables a split front-end/back-end architecture for virtual devices, where the front-end resides in the guest OS kernel and the back-end lives in the Virtual Machine Monitor (VMM) or host (*e.g.,* vhost). Virtio is a representative PV standard supporting devices like virtio-net. Originally designed for interaction between guest and VMM, Virtio's virtqueues now can map onto hardware-accelerated queues over PCIe via Virtio Data Path Acceleration (vDPA), as supported in emerging SmartNICs [29, 57].

Device Pass-through assigns PCIe devices to VMs or containers, allowing direct access to doorbells and interrupts, while enabling devices to perform DMA to guest memory without host kernel involvement. To address the scalability limitations of assigning all hardware resources (*e.g.,* TX/RX queues) to a single physical function, SR-IOV [1] introduces Physical Functions (PFs) and their associated Virtual Functions (VFs). Each PF/VF is allocated a subset of the device's hardware resources, making SR-IOV a hardware-defined virtualization approach.

Mediated Pass-through enables flexible composition of hardware resources through a software mediation layer, such as Intel's Virtual Device Composition Module (VDCM) [45]. VDCM software can remap hardware resources, such as TX/RX queues, to multiple mediated devices (MDEVs), enabling finer-grained resource partitioning than SR-IOV. Moreover, MDEVs can expose the full feature set of the underlying device, similar to SR-IOV PFs/VFs, and can achieve near-native performance, as only slow-path operations (*e.g.,* control path) are mediated.

SmartNIC-based IOV: Emulating virtual devices with SmartNICs brings key benefits for cloud infrastructure by offloading I/O stacks to specialized accelerators, while preserving the host's standard PCIe functions. Originating in the cloud VM era, hyperscalers have widely adopted such devices; for instance, AWS EFA over Nitro [14], Microsoft AccelNet [67], and Alibaba eRDMA over CIPU [119].

A.2 BlueField-3 Characterization

In this work, we focus primarily on the NVIDIA BlueField-3 (BF3), as shown in Figure 4. In §3.4, we discuss the capabilities we leverage from BF3, and next we benchmark the primitives used in SG-IOV. Besides the BF3, we also consider other hardware devices (Appendix A.6).

SG-IOV Requirements for BF3: Although some prior works have presented characterization on BlueField SmartNICs [61, 114, 121], SG-IOV imposes several unique requirements not fully addressed by prior works:

- Consistent low latency for fast signaling such that synchronization can be quickly done between the host and the SmartNIC SoC;
- Interference-conscious communication, where inter-host (cross-server) traffic should observe no performance drop while competing with intra-host traffic;
- SoC-delegated zero-copy communication, allowing data moves directly through the NIC without being routed through the SoC, while the SoC retains the control on behalf of the host. Based on the requirements, we present the following findings from characterizing the BF3:

Tests	Unloaded	100% Loaded
Device Init., Host Bounce	RTT=1.7 ± 0.1 μs	RTT=1.9 ± 0.2 μs
Host Init., Device Bounce	RTT=2.3 ± 0.2 μs	RTT=2.4 ± 0.2 μs
Device Send, Host Receive	RPS=0.73 ± 0.02 M	RPS=0.71 ± 0.02 M
Host Send, Device Receive	RPS=3.32 ± 0.04 M	RPS=3.26 ± 0.04 M

Table 5. Microbenchmark of the HW-FIFO.

Finding #1: HW-assisted FIFO queues deliver deterministic and consistently low latency. BF3 provides a pair of hardware FIFO queues (HW-FIFO) between the host and the SoC, which can implement the Cross-FIFO data structure described in §5.1. BF3 originally uses it to realize the management interface (RShim) for SSH access. We repurpose it to implement Cross-FIFO. We measure the latency and throughput of the HW FIFO queue using a ping-pong test between the host and the SoC. As shown in Table 5, it achieves consistently low latency and high RPS regardless of the network load. When transferring an 8-byte word, one HW-FIFO takes sub-microsecond level one-way latency, more than 3× faster than the best-configured DMA command. The Host-to-Device and Device-to-Host deliver 3.32 and 0.73 MRPS (million requests per second), respectively. We observed negligible performance degradation even when the network port is fully loaded (100 Gbps). Thus, the HW-FIFO is good at transferring small and latency-sensitive messages, manifesting it as a suitable option for realizing signaling.

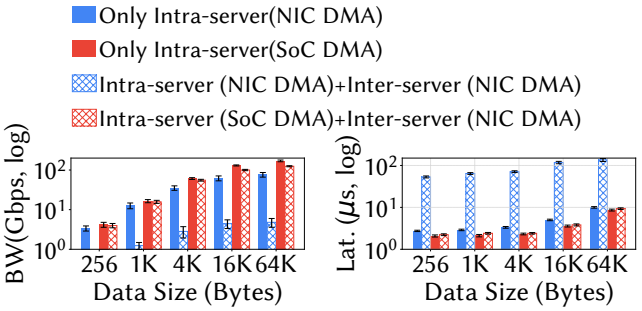
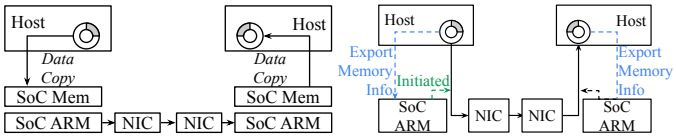


Figure 20. Interference from Out-facing Traffic, NIC DMA vs. SoC DMA at 100Gbps Inter-Server Network Load.

Finding #2: The SoC DMA incurs little interference with the NIC DMA. BF3 employs a *dual-DMA-engines* architecture that provides two DMA data transfer paths [41, 114]. The NIC ASIC DMA, with restricted programmability, facilitates traditional NIC-Host networking functions with RDMA support. The SoC DMA, favoring SoC-Host offloading, allows



(a) Typical Data Copy **(b) X-GVMI**

Figure 21. Comparing Typical Data Copy and X-GVMI.

Primitives	Low Lat. (1KB msg)	Max BW (1MB msg)
Host-Init. RDMA	4.13 ± 0.11 μs	89.95 ± 2.4 Gbps
X-GVMI RDMA	4.33 ± 0.14 μs	87.47 ± 2.1 Gbps

Table 6. Microbenchmark of the X-GVMI Mechanisms.

applications to explicitly manipulate DMA instruction words and control the submission/completion procedure. SG-IOV facilitates both inter- and intra-server data transfer, which ideally should cause no interference to each other. To explore which communication primitives satisfy this requirement, we perform a characterization experiment that co-locates the inter-server and intra-server traffic and examines the performance degradation. Specifically, the inter-server traffic uses RDMA READ under 16 concurrent QPs (which can max out the port bandwidth). We then launch intra-server data transfers using the NIC DMA or SoC DMA.

Figure 20 presents the throughput and latency in different cases. When using the NIC DMA engine for both traffic, there is a drastic performance reduction, whose throughput drops to almost zero and latency increases significantly. However, when using the SoC DMA engine, since it is physically isolated from the NIC DMA, we observe considerably less interference (within 1%) compared with the case when running inter-server traffic exclusively. Further, we find out that the SoC DMA on the BlueField-3 sustains at more than 150Gbps, outperforming the NIC DMA.

Finding #3: The SoC cores can initiate Host-to-Host RDMA on behalf of the host to enable efficient zero-copy. Utilizing the advanced X-GVMI (Cross-Guest Virtual Machine Index) capability of BlueField-3, the SoC ARM cores can initiate data transfers on behalf of the host—a feature originally designed for HPC scenarios [107, 112].

Figure 21 illustrates the X-GVMI mechanisms. In a typical RDMA workflow (Figure 21a), access to a Memory Region (MR) is exclusive to the MR creator. Consequently, a typical offloading procedure involves transferring data from the host memory to the SmartNIC SoC memory. Then, the SmartNIC initiates network operations from its own memory. This process involves an additional hop. As Figure 21b shows, X-GVMI allows access to MR across host and SoC domains without copying data. The main procedure consists of three main steps: 1) the host allows X-GVMI access to a registered MR; 2) the host exports a memory key (MKey); 3) the SoC attaches the MR indicated by the Mkey. Crucially, it allows for data to remain on the host, aligning with the zero-copy mode requirements by avoiding unnecessary data movement. We call it the Delegator-Initialized RDMA in §6.2.

Table 6 shows the performance achieved by X-GVMI compared to traditional host-initiated RDMA. The results show that X-GVMI incurs minimal performance overhead.

A.3 DRF Policy under Equal-Bandwidth Demand

Setup:

- K accelerators with processing bandwidths $C_1, \ldots, C_K$.
- Tenant t uses a subset $S_t \subseteq \{1, \ldots, K\}$, consuming the *same* bandwidth on each resource in S_t (*equal-bandwidth demand*).
- Per-tenant rate: a_t denotes the bandwidth allocated to tenant t on each resource in S_t.
- Dominant share:

$$\sigma_t \triangleq \frac{a_t}{Z_t}$$

is the tenant t's usage normalized to its bottleneck capacity.

Derivation of the closed form. Starting from the optimization problem of DRF [70], we first transform the problem of maximizing allocation to maximizing the dominant share.

$$\begin{aligned} \max_i \quad & a_i \\ \text{s.t.} \quad & \sum_{t:j\in S_t} a_t \le C_j, && \forall j \quad \text{(Resource Constrain)} \\ & \max_{j\in S_{t_x}} \frac{a_{t_x}}{C_j} = \max_{j\in S_{t_y}} \frac{a_{t_y}}{C_j} && \forall t_x, t_y \text{(Dominate Share)} \end{aligned} \tag{1}$$

The above optimization problem can be converted to a Linear Program with an additional auxiliary variable z_t that mitigates the max constraint with some $\epsilon \ge \max_j C_j$:

$$\begin{aligned} \max_i \quad & a_i - \epsilon \sum_t z_t \\ \text{s.t.} \quad & \sum_{t:j\in S_t} a_t \le C_j, && \forall j \\ & a_t \le z_t C_j && \forall t \\ & z_{t_x} = z_{t_y} && \forall t_x, t_y \\ & z_t \ge 0 \forall t \end{aligned}$$

The constraint set is always feasible. In reality, we always have positive solutions because $C_i > 0$.

Thus Program (1) always has a solution σ where for all possible tenants t, $a_t > 0$, which means we can rewrite

$$\begin{aligned} \sigma &= \max_{j\in S_i} \frac{a_t}{C_j} \\ \frac{1}{a_t} &= \frac{1}{\sigma} \max_{j\in S_i} \frac{1}{C_j} \\ a_i &= \sigma \min_{j\in S_i} C_j = \sigma Z_t \end{aligned}$$

Therefore we can rewrite the original optimization problem as the following:

$$\max_{\sigma,\{a_t\}} \quad \sigma$$
$$\text{s.t.} \quad \sum_{t:\, j\in S_t} a_t \le C_j, \quad \forall j.$$

Then we derive the closed form expression.

Step 1 (Normalization). Since $\sigma_t = a_t/Z_{S_t}$, $a_t = \sigma_t Z_{S_t}$. The capacity constraint on resource j becomes:

$$\sum_{t:\, j\in S_t} \sigma_t Z_t \le C_j.$$

Step 2 (Equalization). At the DRF optimum, all tenants have the same share: $\sigma_t = s$ for all t. Substituting into the capacity constraints:

$$\sigma \sum_{t:\, j\in S_t} Z_{S_t} \le C_j \quad \Rightarrow \quad \sigma \le \frac{C_j}{\sum_t Z_t}.$$

Step 3 (Closed form). The maximum common share is:

$$\sigma^* = \min_j \frac{C_j}{\sum_t Z_t}.$$

Step 4 (Allocation). Each tenant receives:

$$a_t = \sigma^* Z_t, \quad \frac{a_t}{Z_t} = \sigma^* \quad (\forall t),$$

which satisfies capacities and maximizes resource allocation.

Extension to Size-Varying Workloads: Our DRF analysis naturally generalizes to size-varying workloads where a tenant t consumes multiple accelerators at fixed ratios $\{r_{t,j}\}$ instead of at equal (1:1) bandwidth.

DRF under Ratio-Bandwidth Demand: *Consider K accelerators with processing bandwidths $C_1, \ldots, C_K$. Tenant t uses a subset $S_t \subseteq \{1, \ldots, K\}$ and consumes resource $j \in S_t$ at a fixed* ratio $r_{t,j}$ *relative to its per-tenant allocation scalar a_t. That is, if tenant t is allocated a_t, it consumes $a_t \cdot r_{t,j}$ bandwidth on resource j.*

Ratio-aware bottleneck (per tenant):

$$Z_t \triangleq \min_{j\in S_t} \frac{C_j}{r_{t,j}}.$$

Per-tenant allocation (scalar):

$$a_t = s^* \cdot Z_t \quad \text{where} \quad s^* = \min_j \frac{C_j}{\sum_{t:\, j\in S_t} Z_t\, r_{t,j}}.$$

Here $\sum_{t:\, j\in S_t} Z_t r_{t,j}$ sums the normalized load *that all tenants impose on resource j under the bottleneck-normalized scaling Z_t.*

Reduction to equal-bandwidth: When $r_{t,j} = 1$ for all t, j, we recover $Z_t = \min_{j\in S_t} C_j$ and $s^* = \min_j \frac{C_j}{\sum_{t:j\in S_t} Z_t}$.

A.4 SG-IOV Control and Auxiliary Functionalities

For SG-IOV design, we focused on I/O-intensive data APIs, but offloading also needs auxiliary features like socket control APIs. In SG-IOV, we support socket control APIs (*e.g.,* *connect()*). Between the host and the device, we dedicate two warp pipes: one for host-to-device communication and the other for device-to-host, forming the control channel. For example, in the case of connection setup syscalls like TCP *connect()*, we modify the syscall implementation to redirect connection requests from the host to the offloaded network stack. Requests are transferred via the control channel. Once the offloaded stack establishes the connection, the offloading device (*e.g.,* SoC cores) responds back to the host's syscall via the control channel. With this mechanism, we implement control APIs such as *listen()*/*accept()*, and others.

Helper Functionalities and Semantics: Beyond socket control APIs, we identified helper functionalities to enhance host-device interaction in Table 7.

Functionality Description	API
Basic Data Transfer	Read / Write SetHead / SetTail GetHead / GetTail SetCopyBatchSize
TCP Proxy	ConnectRelay / AcceptRelay
Cross-pipe Orderliness	Rd/Wr (cond. OrderCrossPipe)
Object Atomic Transfer	Rd/Wr (bool AtomicToRemote)
Query across Host-Device	GetHead/GetTail (bool Sync)
Event Notification Relay	EventSet / EventPoll

Table 7. Auxiliary Functionality Set of SG-IOV.

A.5 Implementation Detail

Optimized Cross-FIFO for BF3: We leverage the BF3 hardware-assisted RShim FIFO to transfer signals, as indicated by Finding #1 from our tests (§A.2). RShim FIFO was originally used for SSH access from the host to the SoC, but we surprisingly found that its low latency makes it well-suited for implementing Cross-FIFO for intra-node warp pipes. To support cross-node warp pipes, we implement Cross-FIFO using RDMA to transfer signals.

Virtio Compliance: We reference the Virtio Specification to ensure compatibility of SG-IOV devices. Specifically, warp pipes conform to the Virtio Socket Device model, enabled in part by Cross-FIFO's compatibility with virtqueues. We emphasize Virtio compliance for two key reasons: first, Virtio standardizes the host front-end driver interface; second, modern SmartNICs widely support Virtio acceleration technologies such as vDPA [50], which enhance hardware virtqueue performance over PCIe.

Memory Management: Each node maintains two main data buffer regions: one on the host side and one on the SmartNIC SoC side. We manage both regions using the unified DOCA mmap library, which enables two key capabilities. First, accelerators from other DOCA libraries (*e.g.,* crypto) can directly operate on these regions. Second, programs running on the SoC ARM cores can initiate RDMA operations using these buffers. Even when both the source and destination buffers reside outside the SoC ARM side, the ARM cores can still delegate RDMA initialization via the X-GVMI mechanism described in §A.2, thereby avoiding staging copies.

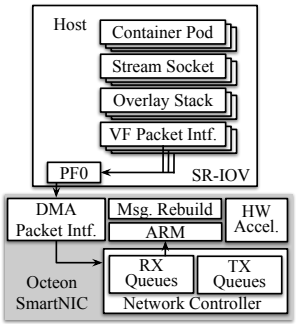


Figure 22. Offloading Container Networks through SR-IOV Paradigms for On-Path SmartNICs.

Module	Main Sub-modules	Lines of Code
Signal Plane	Job Generator / Memory View Sync	~1,600
Data Plane	Job Scheduler / Executor Setup	~4,100
PCIe DevEmu	Doorbell / MSI-X Handler	~1,300
Core States	Job Queues / Global Contexts	~1200
Memory View	Data Buffer Management	~400
Cross-FIFO	HW-FIFO / RDMA-FIFO	~900
Auxiliary	Lock-free Ring Buffer/ Eventfd	~800
Total		~10,300 C/C++

Table 8. Modules of SG-IOV.

DMA Isolation and Security: In SG-IOV, all DMA jobs are initiated by the data plane on the SmartNIC back-end. The SmartNIC SoC is physically isolated from the host CPU, offering a stronger trust model. This centralized control enables the SmartNIC back-end to enforce isolation across all warp pipes; for example, through scheduling policies discussed in §6.1. Additionally, hardware features such as PASID-based isolation [45] of the IOMMU are applicable for SG-IOV.

Accelerators Integration: On the SmartNIC back-end, we dedicate ARM cores to the data plane—one handling TX flow tasks and the other for RX direction tasks. Notably, these ARM cores are responsible only for submitting jobs to accelerators and polling for completions; they do not perform heavy computation themselves. To support a wide range of accelerators, the data plane adopts a general submission and polling workflow compatible with most domain-specific accelerators, including those provided by the DOCA library.

SGIOV-CNI for Traditional Containers: Although we focus on secure containers, we also explore integrating SGIOV-CNI with traditional host Linux kernel-based container runtimes. Within the host kernel, we use the eBPF program [23] to redirect containerized application sockets to SG-IOV sockets. This method is inspired by how Cilium [18] redirects application sockets to Envoy Proxy listeners. We hook the eBPF program to socket operations (*i.e.,* sockops and sockmap), so that when syscalls like *sendmsg()* trigger, we redirect the traffic to SG-IOV sockets. Beyond eBPF-based redirection, another approach is to expose warp pipe pass-through devices directly to the container namespace, similar to how SRIOV-CNI exposes VFs [46]. This method requires a warp pipe driver in the host Linux kernel, which can be implemented using the vDPA driver framework [30]. For prototyping with generality, we adopt the eBPF-based method.

SG-IOV Internal Modules. During SG-IOV implementation, we emphasize modularity as Table 8 outlines.

A.6 Generality to Other Hardware

On-path SmartNICs: Figure 22 illustrates on-path SmartNIC such as Marvell Octeon [36] and Netronome Agilio [37]. Here, we use Octeon as one example. On the host side, Octeon supports SR-IOV VFs, which operate as packetized interfaces similar to the BlueField case. On the SmartNIC side, Octeon includes a specialized hardware unit called *System DMA Packet Interface Unit* (SDP), which acts as a bridging interface between the Host and Octeon. Notably, Octeon is viewed as an "On-Path" SmartNIC. It does not expose a normal Ethernet Device to host, but only the SDP interface. Instead, the Ethernet devices are operated on the Octeon side. However, similar to the Bluefield case, the SDP devices are still operating as packetized interfaces. If the Octeon wants to perform message-level accelerations over hardware engines or software libraries, it must first reconstruct the original messages. Thus, Octeon faces similar limitations with *cyclic transformation* as BF3, which SG-IOV helps address.

Porting Cross-FIFO to Marvell Octeon SmartNIC: To further assess SG-IOV's feasibility, we port Cross-FIFO to the Marvell Octeon SmartNIC. The testbed directly connects two 100GbE servers: one with an Octeon 10 DPU (CN106-2P100) and the other with an NVIDIA ConnectX-7. As described in §5, one end of Cross-FIFO runs on Octeon's ARM Neoverse N2 cores, while the other runs on the remote host CPU. Since Octeon lacks RDMA support, we use DPDK for cross-node transfers. Cross-FIFO on Octeon achieves up to 20M requests/sec, sufficient for signal plane operations and even exceeding BlueField-3's hardware FIFO (§A.2).

Operations	NVIDIA BlueField-3	Intel IPU	Xeon Chipset
Data Transfer	SoC DMA	SoC DMA	DSA
	NIC DMA	NIC DMA	-
Signalling	RShim FIFO	MMIO	-
Executors	In-line	In-line	QAT
	Look-aside	Look-aside	-
	SoC ARM	SoC ARM	-

Table 9. Accelerators of BF3, Intel IPU, and Xeon Chipset.

Other Hardware: Besides the BlueField-3 and Octeon used in this project, we also consider accommodations with other hardware devices. For instance, we discuss the implementation feasibility of the Intel Mount Evans IPU based on its specifications [29] and Intel Xeon Chipset Built-in Accelerators, which are not SmartNIC-based. As Table 9 shows, we can find suitable primitives from the Intel IPU to implement the SG-IOV operations. One notable difference between the BF3 and the IPU is that the IPU supports Memory-Mapped IO (MMIO), which we can utilize for the signaling mechanism, whereas for the BF3, we use HW-FIFO. For Xeon chipset accelerators, they provide DSA to accelerate data copies like DMA, data compression [81], and QAT to accelerate transformations like AES.