OSMOSIS: Enabling High-Performance, Multi-Tenant SmartNICs in Datacenter Systems

Mikhail Khalilov[1], Marcin Chrapek[1], Siyuan Shen[1], Alessandro Vezzu[1], Thomas Benz[2], Salvatore Di Girolamo[1], Timo Schneider[1], Daniele Di Sensi[1,3], Luca Benini[2], and Torsten Hoefler[1]

[1]SPCL, D-INFK, ETH Zurich
[2]IIS, D-ITET, ETH Zurich
[3]Department of Computer Science, Sapienza University of Rome

Abstract

SmartNICs play a crucial role in driving innovation in datacenter networking. SmartNICs effectively reduce latency and jitter associated with packet delivery and host CPU processing by placing energy-efficient programmable cores close to the network link. Deploying SmartNICs in a datacenter context with support for multiple tenants is essential for unleashing their computing capabilities. Yet, as our systematic analysis in this work shows, the resource multiplexing limitations of existing on-path SmartNIC designs lead to the lack of multi-tenancy capabilities such as performance isolation and QoS provisioning of compute and IO resources. SmartNIC unpredictable kernel execution times, compared to standard predictable NIC datapaths, make conventional multi-tenant approaches untenable. We address these limitations by proposing OSMOSIS, a control plane co-design for datacenter SmartNICs. OSMOSIS extends existing OS mechanisms to enable SmartNIC-aware QoS management on top of the scalable packet processing hardware data plane. We implement OSMOSIS within an open-source RISC-V-based on-path SmartNIC. Our performance results demonstrate that OSMOSIS fully supports multi-tenancy and enables broader adoption of SmartNICs in cloud datacenters with low overhead.

1 Introduction

Network data plane design has undergone two decades of exciting research, leading to the achievement of sub-microsecond packet processing host latency [7, 18, 26, 29, 34, 35, 51, 53]. SmartNICs (sNICs) have further improved processing times by enabling direct in-network packet processing, thereby reducing data movement [32]. sNICs started a trend in datacenter networking acceleration [49, 63] similar to the GPU trend in high-performance computing [64].

sNICs enable running *kernels* on programmable, energy-efficient cores tailored for packet processing and integrated within the host network interface card (NIC) System-on-Chip (SoC). These cores are attached directly (i.e., *on-path*) to the datacenter Ethernet or InfiniBand link [5, 39]. Such a design reduces the latency of some applications since the sNIC can process the packets in the network [42] and reply directly without moving the packets to/from the host OS networking stack [1, 23]. For example, machine learning gradients can be summed on the sNIC without the involvement of the host [64]. sNICs can also be used to accelerate datacenter disaggregation [20, 45], storage [22, 45, 46], Key-Value Stores (KVS) [52, 62], compute [41], client Remote Procedure Calls (RPCs) [9, 45, 68], the network stack [9, 47, 68, 69] and telemetry [30].

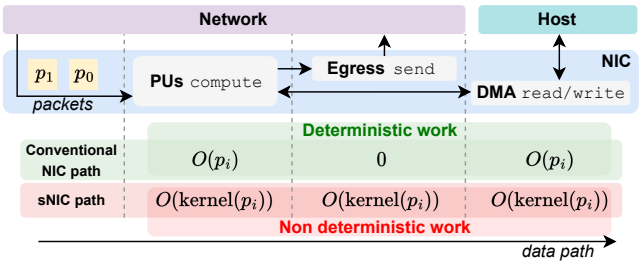

Figure 1: An overview of a typical NIC datapath with a predictable amount of work compared with the unpredictable nature of kernel sNIC evaluation.

Resources in a modern datacenter are flexibly multiplexed between multiple tenants. However, user code processing in the network enabled by sNICs brings a set of considerable resource management issues. As Figure 1 shows, NICs have three resources that must be mutiplexed: compute, DMA, and egress. The traditional NIC datapath only forwards packets to host memory and executes simple operations with a *predictable* and *bounded* complexity. Typically, the number of incoming bytes equals the number of outcoming bytes, and NIC does not conduct any elaborate processing on them. In contrast, sNICs are executing *unpredictably* complex kernels dependent on the received packets and application needs. For example, Allreduce, heavily used in machine learning, is compute-bound as it requires operations on the provided data,

while key-value store (KVS) is DMA bound by accesses to the host memory. sNICs need to operate on *uncoordinated*, *non-deterministic*, and *concurrent* data streams while meeting Service Level Objective (SLO) policies set by the administrator.

Achieving a fair multiplexing of resources for sNICs is challenging. sNICs combine an accelerator, such as a GPU, and a traditional NIC. While this provides the aforementioned benefits, the resource management of neither is directly applicable due to the unique requirements of sNICs (Section 3). Conventional RDMA NICs (rNICs) which have bounded and predictable workloads (e.g., atomics, scatter-gather RDMA reads/writes) usually use link bandwidth allocation as a *"just enough"* mechanism for resource isolation and Quality-of-Service (QoS) between tenants. While bounded and predictable, unlike in the sNIC case, even for rNICs, fairness is hard to obtain [66]. Compute accelerators are external host devices like sNICs. However, they are controlled entirely by the host OS, which manages all running kernels [36, 37] and does not generate or receive events other than commands from accelerated applications. sNICs can execute arbitrary kernels without the involvement of the host.

Furthermore, for sNICs to sustain the sub-*nano*second packet arrival intervals at fully utilized 400Gbit/s link (Section 3, [19]), resource multiplexing must be conducted fast. On-path sNICs have much stricter compute and buffering constraints than traditional NICs and accelerators due to the packet rate and the three multiplexed resources (compute, DMA, and egress). This issue is even more critical as network rates constantly increase, and are expected to exceed Terabit per second speeds by 2025 [10, 16].

A common approach to effectively manage processing at high packet arrival rates would be implementing resource management in hardware [2, 4, 19]. This is usually accomplished through scheduling policies such as Weighted Round Robin (WRR), which divide link bandwidth among tenants [13, 14, 66]. However, because sNICs have varying application kernel requirements, incorporating WRR for compute resource allocation can lead to unfairness. For example, as we show in Section 3, if one application like Allreduce is compute-bound and takes twice as much compute time than a non-compute bound application like KVS, the compute-bound application will be able to process twice as many bytes. Other recently proposed methods for compute isolation in sNICs are not optimal for all scenarios as they either are non-work conserving [21] or rely on the host CPU as a fallback mechanism [41].

We tackle these by introducing OSMOSIS (Operating System Support for Streaming In-Network Processing) (Section 4), a light-weight sNIC management layer that separates the performance-critical data-plane implemented in hardware and the non-critical management tasks implemented in flexible software control-plane. OSMOSIS is a fair, work-conserving sNIC resource manager that requires minimal hardware footprint and employs expressive yet simple Service Level Objective (SLO) semantics.

In OSMOSIS, the sNIC is exposed to a tenant as Single-Root Input/Output Virtualization (SR-IOV) Virtual Function (VF). This allows the administrator to allocate proportionally more *interconnect*, *compute*, and *memory* resources to VFs associated with high-priority tenants. OSMOSIS achieves fair resource allocation and tenant isolation.

We implement (Section 5) and evaluate (Section 6) OSMOSIS on top of one of the available open-source on-path sNIC architectures, PsPIN [12, 24], based on energy-efficient silicon-proven RISC-V cores. In our setup, PsPIN is the hardware backbone for packet processing using kernels written in C. Our performance evaluation focuses on typical datacenter workloads, such as Key-Value Store (KVS), storage IO and in-network Allreduce. OSMOSIS provides comprehensive support for multi-tenancy without sacrificing performance.

In summary, we make the following contributions.

1. *sNIC multi-tenancy:* We show the typical multi-tenancy sNIC problems in detail and define requirements for high-performance sNICs serving as a guideline for developing sNICs that can meet the needs of diverse workloads and tenant environments (Section 3).

2. *OSMOSIS:* We introduce OSMOSIS, a lightweight sNIC resource manager based on a set of fair and work-conserving scheduling policies. OSMOSIS is a minimal hardware footprint solution to the aforementioned problem of fair and fast resource multiplexing within sNICs in a multi-tenant environment with varying application requirements (Section 4).

3. *Evaluation:* We implement OSMOSIS within a RISC-V-based on-path sNIC architecture that supports packet processing at 400Gbit/s and extend it by the necessary schedulers and the prototype of a control path infrastructure (Section 5). We use this implementation to verify and evaluate OSMOSIS. We show how it solves the defined sNIC problems and how it fairly and with minimal tail latency overhead handles multi-tenant applications with varying resource requirements (Section 6).

2 Background and Related Work

From the system's perspective, we abstract out the sNIC as a packet processing accelerator located between the network fabric and the host CPU, GPU, or FPGA. It can handle various operations, including communication, gradient reduction offloading for DNN training (in-network Allreduce [64]), and storage (IO reads/writes [45]). Existing sNICs can be classified broadly into two categories: *off-path* and *on-path* [41].

Off-path sNICs add a full CPU complex to the network card, often running a full operating system (e.g., Linux). This design enables a management plane based on receive side

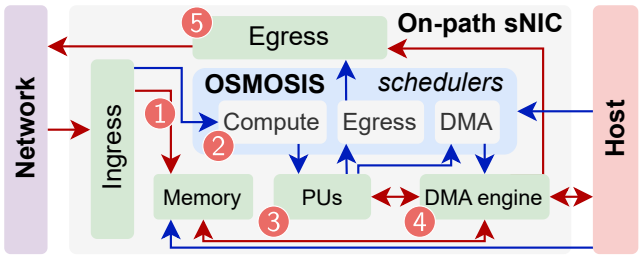

Figure 2: Schematic overview of on-path sNIC architectures.

scaling (RSS) to be conveniently implemented [7, 44, 53]. However, they often suffer from lower performance in terms of latency, bandwidth, and packet processing rates due to their system design, which closely resembles the host architecture (e.g., Broadcom Stingray and Nvidia Bluefield both feature CPU SoCs with PCIe and DRAM) (Table 1).

On-path sNICs share packet input buffers with *processing units* (PUs) tailored for packet processing (e.g., LiquidIO [43], Netronome [49], Bluefield-3 DPA [50], PsPIN [12]). On-path sNICs typically provide programming API for writing *kernels* that process traffic on PUs, either on per-packet (sPIN) or per-message granularity (Bluefield-3 FlexIO API [50], nanoPU [29]). PUs typically feature three layers of the memory hierarchy, e.g., L1 single-cycle access scratchpad, L2 memory with access latency of 15-50 cycles, and host side memory (either off-path SoC or host CPU memory). L1 and L2 memories could be organized as multi-level caches (e.g., LiquidIO) or be explicitly managed by the user (e.g., PsPIN).

OSMOSIS provides a solution to a fair resource multiplexing for sNICs in a multi-tenant context and is not specific to any system. However, to showcase the identified issues, and verify and evaluate the overhead of OSMOSIS, we selected one of the possible open-source sNIC implementations available in the literature. We decided to use an on-path sNIC as our experiments (Table 1) show that only such sNICs can sustain the processing at the emerging line rates. We selected PsPIN as the underlying sNIC. PsPIN is open-source, based on energy-efficient silicon-proven RISC-V cores, and allows the users much more granular memory access than other implementations stretching the resource management more. PsPIN is also unique because it can be synthesized to real hardware running at the frequency of 1GHz. OSMOSIS could have been equivalently implemented in any other sNIC framework [43, 49, 50].

2.1 Challenges of Resource Isolation

We generalize on-path sNIC architecture in Figure 2. Packets arrive at the sNIC inbound engine (1) and are initially stored at the L2 packet buffer with a per-application first-in-first-out (FIFO) queue semantics. Next (2), packets are scheduled for processing on available PUs where kernel execution is initiated (3). Kernels execute using three resources, PUs, DMA, and Egress engines. Depending on the application needs, these may be used more or less (e.g., compute- or IO-bound). In general, these resources can be used, for example, as follows:

(3) PUs: computing (e.g, packet header hashes or summing values in an Allreduce reduction);

(4) Direct Memory Access (DMA) engine: transferring data through one of the interconnects to read/write in sNIC memory (e.g., KVS-cache in sNIC L2 memory) or host memory (e.g., KVS cold storage);

(5) sNIC egress engine: sending packet replies (e.g., reply with data from IO read or value from KVS cache).

Metrics to measure the quality of resource multiplexing by datacenter tenants, known as Service-Level Objectives (SLOs), are typically tied to the conventional NIC path displayed in Figure 2 by considering tail latency [11] and throughput [48, 59]. However, these SLOs do not consider the sNIC datapath with its unique resource multiplexing. Tail latency of DMA over host interconnect, PU time, and buffer space should be considered. Existing proposals have only partially addressed this issue through the support of performance isolation mechanisms, such as multi-level packet scheduling [19, 41, 61] and static resource allocation [21] within the shared components of a system such as processing cores and DMA engines. Yet, because of the dynamic and unpredictable nature of the kernels and packet flows, static assignments do not solve the problem. OSMOSIS addresses this gap by *providing bounded guarantees for the availability of sNIC resources to tenants using dynamic resource multiplexing*.

3 Multi-Tenant sNICs

We illustrate how mechanisms found in existing sNIC stacks are insufficient for fair resource management. Applications differ in their resource requirements, thus, leading to different resource multiplexing bottlenecks. We provide quantitative analysis on how these problems display in a multi-tenant environment, resulting in specific requirements for sNICs. These insights directly led to the microarchitectural and software choices for OSMOSIS. For all experiments, we use a 400 Gbit/s link speed together with the workloads and experimental setup as described in Section 6.

Per-packet time budget (PPB): While studies of datacenter traffic show that only a fraction of the established connections actively exchange data at any given time [8, 58, 67], they saturate the link bandwidth. To analyze the implications of this for sNICs we use PPB. We define PPB using PU count N, packet size P, and link bandwidth B as $PPB(N,P,B) = N \times (P/B)$. PPB shows how long the sNIC can process a packet until the next one arrives, assuming a fully utilized link.

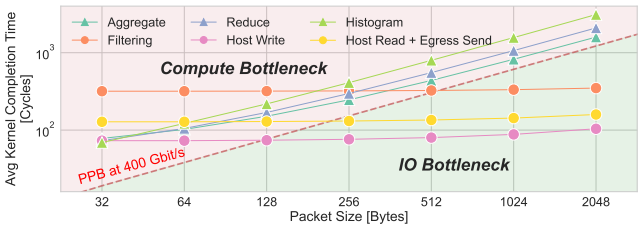

Figure 3: sNIC core (PU) processing time needed to serve 1 packet for common sNIC kernels. Workloads with triangle markers are compute-bound, circular markers are IO-bound. All workloads with $\leq$ 64B packet size (including 28 byte IPv4/UDP-header) exceed PPB showing congestion at PUs when link bandwidth is fully utilized.

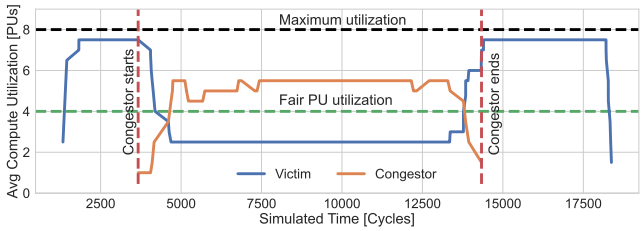

Figure 4: *Congestor* and *Victim* tenants' flows with equal priorities are mapped to two different SR-IOV VFs with equal share of Ingress bandwidth. With the round robin scheduling policy of per-flow queues, the *Congestor* tenant with 2× higher compute cost per packet occupies a proportionally larger number of cores than the *Victim* tenant.

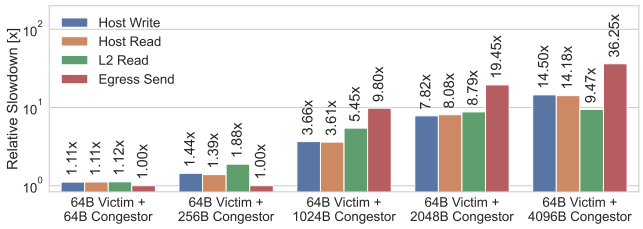

Figure 5: Slow-down of various IO operations (e.g., DMA and sending packets to Egress) initiated by the tenant's kernel results in HoL-blocking small requests due to underlying IO path contention.

If PPB is exceeded, the corresponding sNIC per-application ingress queue will eventually fill up during transient bursts leading to packet drops or falling back to priority-based flow control (PFC) [70] and a possible violation of per-VF SLO policy.

Figure 3 compares service times of IO– and compute-bound workloads with theoretical PPB assuming that tenant workloads fit one packet and that the sNIC has only one tenant. We observe that all workloads with packet size $\leq$ 256 Bytes fail to fit in PPB. Compute-bound workloads whose execution time scales linearly with the packet payload length (i.e., Reduce, Aggregate, Histogram) exceed the theoretical limit for all packet sizes congesting the PUs that create a bottleneck. Notably, IO-bound kernels above 256 Bytes (e.g., DMA writes/reads or packet sends to Egress) are bottlenecked by the link bandwidth and fit PPB as they do not congest the PUs but the IO engines. However, as we will demonstrate, *IO-bound workloads tend to be sensitive to a multi-tenant contention on the host interconnect*.

PU contention: While a single tenant can cause pressure on the ingress queue and contention of PUs, multiple tenants can lead to unfairness. For example, consider two compute-bound tenants with different requirements. One of them, the *Congestor*, has twice as large compute cost per packet as the other, the *Victim*, leading to twice as many cycles on PU to finish the kernel. During the burst, *Congestor* and *Victim* push packets at the corresponding per-application (per-VF) queues at the same ingress rate. As Figure 4 shows, using the conventional round robin (RR) scheduling of per-application queues across 8 sNIC PUs provides the *Congestor* unfairly with 2× higher occupation of PUs than the *Victim*.

R1 *sNIC manager should fairly allocate compute components (e.g., PUs, cryptographic accelerators) while serving tenants with different compute cost per packet.*

Egress and DMA engines contention: Similarly, as the compute-bound kernels cause contention on PUs, IO-bound kernels can lead to contention on the appropriate DMA or egress engines. IO-bound kernels running on different PUs can simultaneously initiate IO requests through the same sNIC engines, e.g., DMA requests from a KVS application. In case the underlying interconnect (e.g., PCIe or AXI [55]) is blocking and lacks the support of QoS provisioning, *the issue of multiple concurrent requests may result in Head-of-Line (HoL) blocking* [1].

For example, consider two IO-bound tenants with different IO requirements. The *Victim*, has constant 64B packets, while the *Congestor*, increases its packet size from 64B to 4096B. As Figure 5 shows, the contention on the IO engine leads to an order of magnitude higher latency of the *Victim*'s messages without considerably affecting the *Congestor*'s flow. This unfairly increases the latency of one of the tenants by 4-15×.

R2 *sNIC manager should fairly allocate DMA and egress bandwidth (e.g., using AXI and PCIe) between running kernels and be resilient to HoL-blocking.*

Memory management: Applications impose different runtime requirements on the memory. For example, some applications could allocate memory dynamically, leading to an unknown *a priori* memory consumption. In the extreme, it is possible that one tenant can consume all sNIC memory, including packet buffers, causing others to be HoL-blocked.

PU	Frequency	ISA	Linux	Caladan	RTOS
Host Ryzen 7 5700	3.8GHz	x86	28576	211	–
BF-2 DPU A72	2.5GHz	ARMv8	13250	192	–
PULP cores [6] (used in PsPIN)	1GHz	RISC-V	–	–	121

Table 1: Average latency of context switching between 2 processes. Measurements shown in PU cycles scaled to 1 GHz (i.e., 1 ns/cycle).

Implementing virtual memory (i.e., paging) semantics would likely cause high memory access overheads, given that each page fault increases memory access latency by orders of magnitude [28].

R3 *sNIC manager should fairly allocate memory using lightweight allocation strategies defined in the control plane.*

Scheduling overhead: Existing packet processing user-defined datapaths were designed for off-path sNICs or conventional host processing. As recent studies show *effectiveness* of kernel execution scheduling in terms of achieved maximum utilization while running on off-path sNICs supported by OS's like Linux is driven by the latency of context switching [18, 34]. PU cycles are wasted during context switching to transition between the kernel states. We benchmark context-switching of Linux running on host and off-path sNIC (Bluefield-2 ARM SoC). We compare these to the state-of-the-art Caladan scheduler we ported to the ARM ISA [18]. For reference, we also show the latency of PULP cores as implemented in PsPIN used to evaluate OSMOSIS. Notably, we observe that the context switching latencies we report in Table 1 are higher or of the same order of magnitude as the PPB from the analysis presented in Figure 3.

R4 *Data path performance should not be impacted by overheads stemming from scheduling policies, providing low-latency scheduling of kernel execution.*

Control path priority: In case of issues such as exceeding compute or time budgets by a tenant on the sNIC, *control traffic* may require an immediate response from a control plane running on the host. However, sNIC to host communication involves the system interconnect (e.g., PCIe) that typically introduces 0.5 – 2 usec overhead per each read/write request. As the system interconnect is sensitive to congestion (Figure 5), that may result in HoL-blocking of the control traffic and unpredictable packet processing behavior without the ability of the host to react. Resolving this by moving the execution of requests to the host (e.g., iPipe [41]) introduces additional latency overheads to the processing of each packet.

R5 *sNIC accelerated packet processing should prioritize control-path traffic and not rely on latency-introducing host CPU as a fallback path.*

QoS API: NIC capabilities are exposed to tenants through a virtualization layer (OS hypervisor) that provides an illusion of full resource ownership. SR-IOV is a conventional way to implement NIC virtualization. In SR-IOV, each NIC physical function (PF) (such as TX and RX capabilities) is mapped to several virtual functions (VFs). Each VF is exposed to OS hypervisor as a stand-alone PCIe NIC. To our knowledge, existing production rNICs and sNICs support only Ingress and Egress bandwidth allocation on the coarse basis of VFs and not compute or DMA resources.

R6 *sNIC management plane should support conventional QoS provisioning mechanisms for all types of resources.*

4 OSMOSIS

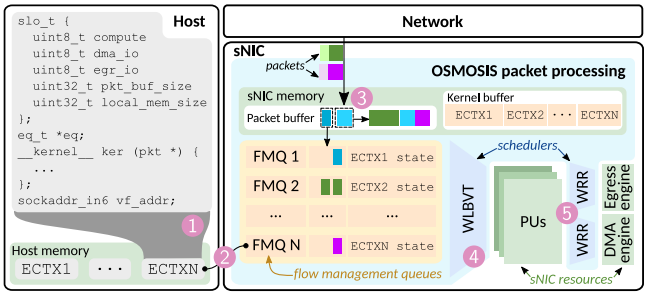

Figure 6: Abstract model of OSMOSIS-enabled sNIC. Packets are mapped by Matching Engine to FMQs and dispatched for execution by scheduler.

We present OSMOSIS in Figure 6. We begin with a high-level overview of how OSMOSIS manages the three competing sNIC resources and satisfies the multi-tenancy requirements outlined in the previous section. We then show how this is achieved by separating the system into two parts: a non-critical flexible software control plane dealing with management tasks running on the host and a performance-critical hardware data plane supporting SLO policy enforcement. We discuss each part in detail.

	PUs	DMA	Egress	Memory
Scheduler	WLBVT	WRR	WRR	Static
SLO knob	Priority Kernel cycle limit	Priority	Priority	Allocation size
Fulfilled requirements	R1 R4 R6	R2 R4 R5 R6	R2 R4 R6	R3 R4 R6

Table 2: OSMOSIS resource management principles with all six fulfilled multi-tenancy requirements.

4.1 High-level Overview

❶ **Flow execution context creation:** To utilize sNIC packet processing, tenants create a flow *execution context* (ECTX). ECTX contains seven elements, such as the SLO policy and the packet processing *kernel*, a piece of code compiled for the target PU architecture and describing the actions to be performed for each packet destined for the tenant.

❷ **ECTX initialization:** After the tenant provides the basic elements of an ECTX, OSMOSIS instantiates it. It allocates a virtualized sNIC interface through the host OS hypervisor and associates it with a tenant IP address and SLO policy. It also sets up the IOMMU to allow kernel access to specific host pages, *statically* allocates on sNIC memory and loads the kernel binary into sNIC memory.

❸ **Matching packets to flow management queue:** The sNIC matching engine filters packets that require sNIC processing. All incoming packets are matched against the three-tuple (in case of UDP) or five-tuple (in case of TCP) of active sNIC ECTXs. Once matched, *packet descriptors* (e.g., pointer to packets in sNIC memory) are stored at one of the *flow management queues* (FMQs). FMQs are used to store all information regarding an active flow ECTX on the sNIC hardware. FMQs are organized as FIFO queues of packet descriptors with an additional memory state to store running execution information (e.g., BVT metric).

❹ **PU scheduling:** Once a PU becomes available, OSMOSIS schedules the packet at the head of one of the FMQs. To achieve fair PU allocation, OSMOSIS implements a centralized, non-preemptive scheduler inspired by the Borrowed Virtual Time (BVT) policy [15, 34]. BVT aims to allow each tenant to obtain the same amount of access time to the scheduled resource by keeping track of their past usage. OSMOSIS FMQ scheduler *allocates sNIC PUs to FMQs with the smallest priority-adjusted past PU usage measured in cycles*, while maintaining the SLO policy specified by the sNIC administrator, such as the upper per-FMQ PU limit.

❺ **Kernel execution and IO management:** Once the packet is loaded into local PU memory, the PU can initiate processing by executing the appropriate kernel. As exemplified in Section 3, the DMA and egress data transfers originating at parallel kernels executing on different PUs may result in head-of-line blocking (HoL-blocking) and unpredictable tail latency. OSMOSIS ensures fair arbitration across IO data paths by splitting large DMA requests into smaller transactions and scheduling these transactions using the weighted round-robin (WRR) policy. This gives each tenant a priority-adjusted fair chunk of the bandwidth.

4.2 Flexible software control plane

OSMOSIS provides a host OS API for managing the sNIC packet processing, creating the execution context, and offloading the handling of specific flows to the NIC. When a tenant offloads processing of a flow to the sNIC, they create a flow ECTX that describes the offloading state. ECTX allows the host to specify the necessary control details using the following components.

SLO policy: The SLO policy specifies the compute, DMA and egress priorities, per-kernel cycle budget, size of packet buffer, and on-sNIC memory size available to the executing kernels. OSMOSIS provides transparent SLO management semantics indicated in Table 2 as SLO knobs. By default, all tenants' execution contexts have the same priority. To achieve perfect fairness in such a scenario, all flows should get the same portion of PUs and IO bandwidth at any point in time. Increasing the priority of the execution context leads to *proportionally* more resources (PUs, bandwidth) allocated to the execution context. To stop kernels that are using the PUs for too long, we also introduce per kernel cycle limit, which can be set as a limit for the sum of all parallel kernel execution times or a limit for the execution time of a single kernel. We evaluate the impact of priorities on the fairness of resource allocation in Section 6.

Kernel binary: kernel binary compiled by the tenant is loaded into sNIC memory by the control plane and is later executed on the flow packets. The kernel binary can compute and schedule DMA and egress requests according to the tenant requirements.

A virtualized sNIC device: A virtualized device is allocated for the tenant by OSMOSIS, e.g., SR-IOV VF. OSMOSIS associates an IP address with the virtualized device and uses it later for matching. The virtualized device is connected internally with a single FMQ.

A matching rule: The matching rule is used to match packets from the sNIC inbound stream to the execution context and manage their processing within the same FMQ. A matching rule allows the tenants to open multiple ports on the same virtualized device. The matching engine can match packets based on their UDP/TCP header contents. For example, it can match the IP address and the destination port of the application.

sNIC memory segments: The sNIC memory segments are allocated statically to each kernel depending on the requested memory size. The kernels can store the application state in sNIC local memory, e.g., KVS-cache or packet filter table. The minimum allocation a valid ECTX has is the size of the kernel binary loaded into the sNIC memory by the control plane. An error is returned if the tenant uses too much memory or the kernel binary is larger than the SLO policy limits.

Host memory pages: The ECTX specifies which host pages

can be accessed from the specific kernel via DMA. Accessing the host memory can be useful, for example, with accessing cold KVS data. The DMA engine on the sNIC interfaces the host memory with an IOMMU, translating host virtual addresses to physical addresses. The IOMMU also checks whether the sNIC is accessing an allowed memory region. The control plane initializes the IOMMU with appropriate page tables during execution context creation.

Event queue (EQ): An event queue is used to keep track of events such as errors that happened during kernel execution. Whenever an error occurs during kernel runtime (e.g., illegal memory access from kernel, or kernel execution time exceeding quota), OSMOSIS notifies the host by writing an event in the EQ of the kernel's ECTX. OSMOSIS provides a corresponding host API call to check the presence of error messages in that queue, which then needs to be managed by the application. EQ could be implemented as a contiguous memory region of sNIC memory, mapped to host address space, e.g., as in the RDMA Verbs API [31]. Control path traffic traverses the same sNIC DMA datapath as normal kernel execution traffic. However, as the control events usually require immediate action from tenants, we reserve the highest IO priority control path traffic.

4.3 Hardware data plane

OSMOSIS aims to provide the lowest possible traffic management overhead with a minimal hardware footprint. In this section, we discuss two key mechanisms that help us to achieve this goal: a hardware flow abstraction (FMQs), and scheduling algorithms suitable for hardware implementation (WLBVT and DWRR).

Flow management queues: FMQ abstraction generalizes a packet flow similarly to how a hardware thread generalizes a process. FMQs store matched packet descriptors in a FIFO queue and monitor the flow processing performance. The scheduler then uses these measures to fairly allocate compute resources and enforce per-flow priorities. The processing of a FIFO queue results in a sequence of kernel executions on the sNIC PUs, which is similar to a sequence of program instruction executions of a conventional OS process execution flow. FMQs additionally store a part of the flow execution context state, such as the matching rule, pointers to the kernel binary, and the SLO policy definition. FMQs are initialized and controlled by the aforementioned host-side control plane. The control plane maps the FMQs to the host as a set of MMIO registers within the SR-IOV VF address space. FMQs are highly extensible. For example, the OSMOSIS priority model is compatible with Ethernet DCB. In case of congestion on the FMQ FIFO queue, the packets can be marked with the appropriate Ethernet ECN congestion flag.

FMQ Scheduling: The goal of the FMQ scheduler is to allocate PUs across flows with different DMA, egress, and compute cost-per-packet that is not known a priori. Thus, to achieve fair compute utilization, the FMQ arbitration policy needs to be *invariant to the cost-per-byte of the packet*. Conventional round-robin lacks this desirable property (see Figure 4). OSMOSIS implements a scheduler as simple and scalable as the deficit-weighted round-robin (DWRR) but with a minimal additional hardware area footprint (see Section 5).

OSMOSIS utilizes a greedy *Weight Limited Borrowed Virtual Time* (WLBVT) arbitration policy, a hybrid of Weighted Fair Queuing (WFQ) like model of FMQ weights and Borrowed Virtual Time (BVT) scheduler. We adopt the BVT algorithm to suit specific sNIC hardware implementation constraints [15, 34] and present our scheduler in pseudo-code Listing 1. Intuitively, our modified BVT scheduler aims to allocate each tenant the same amount of PU processing time normalized by priority while ensuring that each tenant is served fairly during resource contention.

```
1  def pu_limit(ActiveFMQs, fmq):
2    prio_sum = 0
3    for fmq in FMQs:
4      if not fmq.empty:
5        prio_sum += fmq.prio
6    return ceil(len(FMQs) * fmq.prio / prio_sum)
7
8  def update_tput(FMQs): #called at each clock cycle
9    for fmq in FMQs:
10     fmq.total_pu_occup += fmq.cur_pu_occup
11     if not fmq.empty or fmq.cur_pu_occup > 0:
12       fmq.bvt += 1 # update only in active state
13     fmq.tput = fmq.total_pu_occup / fmq.bvt
14
15 def get_fmq_idx(): #called once PU core is free
16   min_tput = MAX_INT
17   for fmq in ActiveFMQs:
18     if fmq.pu_occup < pu_limit(activeFMQs, fmq):
19       if fmq.tput / fmp.prio < min_tput:
20         min_tput = fmq.tput / fmq.prio
21         fmq_idx = fmq.idx
22   return fmq_idx
```

Listing 1: WLBVT scheduler procedural pseudocode.

An FMQ is in an active state if it contains packet descriptors in the FIFO queue or if its packets are currently being processed on any PU. Flow throughput is updated (`update_tput`) at each sNIC clock cycle only if the corresponding FMQ is active. The scheduler (`get_fmq_idx`) returns the index of the non-empty FMQ that fits the upper limit of weighted PU occupation (`pu_limit` called in line 21) and has the lowest current throughput normalized by FMQ priority (lines 22, 23).

The upper limit of weighted PU occupation ensures tenants are served fairly by occupying the number of PUs proportional to their priority. `pu_limit` is set using a ceil function to ensure that each tenant obtains processing time in the case of a larger number of active FMQs than the number of PUs priorities or when the priorities divide the number of PUs into a non-integer value. The lowest priority normalized throughput ensures that each tenant has the same access to the oversub-

scribed PU over time, and tenants with lower overall resource usage are prioritized. Our policy could be easily extended to account for the total virtual time spent by each tenant on PU (i.e., line 21), thus making it suitable for tenant billing.

Kernel execution: After scheduling on a PU, the corresponding kernel binary is loaded from local memory, and kernel execution starts. Kernel execution is a short-lived event as each execution only processes one packet. In OSMOSIS, we run kernels to completion [7, 53]. We avoid context-switching for several reasons. As shown in Table 1, context switching can introduce significant overhead. It also increases the complexity of the hardware datapath and requires additional states per each active kernel.

If a kernel does not terminate within a configurable time limit, for example, because of a bug in the code, the kernel is killed, and the host application is notified through the corresponding EQ. We believe that run-to-completion semantics enable sNIC programming model that, together with OSMOSIS fair priority adjusted schedulers, enforces predictable packet processing tail latency. Run-to-completion also prohibits offloading compute-intensive workloads better suited to run on GPUs or FPGAs rather than sNICs [7, 53].

Kernel IO Scheduling: Throughout execution, per packet kernels can issue concurrent DMA transfers to the sNIC/host memory and data transfers to the sNIC egress engine. For example, the kernel can implement pipelining of large storage IO read operations by overlapping asynchronous DMA reads of packet-sized payloads with sending packets to the sNIC egress engine. The DMA and egress engines obtain from the FMQs the corresponding IO priorities of each tenant that initiated IO requests within the kernels. OSMOSIS employs a fragmentation of DMA and egress transfers into smaller chunks to enable fair IO scheduling. These smaller chunks are arbitrated with a WRR policy and can achieve near-perfect fairness.

4.4 Discussion

IO security: Host memory is protected against unauthorized DMA transfers using an IOMMU setup by OSMOSIS when the host creates the flow context. Similarly, local sNIC memory accesses need to be protected. This can be achieved, for example, by a *Physical Memory Protection* unit (PMP) [65] as shown in Section 5.1.

Kernels can also interact with the outbound engine to inject packets into the network. Injecting arbitrary packets might be used to perform security attacks such as spoofing [57]. To avoid that, OSMOSIS installs filtering rules in the outbound engine. These rules are similar to the ones in the packet matching engine, which associate packets with FMQs. The filtering rules are specific to the system, and protocol. They are installed when the ECTX enters the sNIC and removed when the ECTX is evicted from the sNIC.

Cryptographic accelerators: The sNIC not only moves the data but might also need to access it for processing. Hence, the sNIC should be able to decrypt packets if the flow is encrypted, for example, in protocols like QUIC [69]. The sNIC can either have a dedicated cryptographic accelerator per PU accessible through ISA extensions (e.g., Intel AES-NI [25]) or, for resource/space efficiency reasons, one shared accelerator per multiple PUs (e.g., like in Marvell LiquidIO [43]). In the latter scenario, the cryptographic accelerator is a compute device similar to the PUs. Thus, the WLBVT scheduler could be used for scheduling access to the cryptographic accelerators.

Transport protocols: While this work does not focus on sNIC transport protocols, OSMOSIS, by design, is compatible with conventional congestion signaling (e.g., ECN) and lossless flow control mechanisms (e.g., Ethernet DCB). It can also be deployed with DCQCN [70] and DCTCP [3]. From the transport protocol perspective, the packet queueing delay within the FMQs and the corresponding execution of the packet kernel is just another source of latency. For example, the FMQ abstraction deployed with Ethernet can support RED/ECN marking [17, 31]. Another mechanism that FMQ can easily support is supplying the P4 INT-MD telemetry information [2] to enable the HPCC protocol [40].

Off-path processing: While OSMOSIS is designed with on-path packet processing in mind, OSMOSIS design principles are applicable to off-path processing and could be implemented fully in software (e.g., using receive side scaling (RSS) [44]).

5 Implementation

We describe how we implement OSMOSIS on top of PsPIN [12, 24], an open-source on-path sNIC that supports portable C API for packet processing offloading. We also show how we adopt the PsPIN implementation for performance-critical operations within OSMOSIS. For that purpose, we extended the host-side PsPIN API to support multiple execution contexts and specify tenant SLOs using 335 lines of code (LOCs). We also implemented a cycle-accurate simulation of OSMOSIS using functional C++ implementations of the matching engine, FMQ scheduling, WLBVT, and AXI and DMA request fragmentation with 1216 LOCs. In addition, we implemented all these components as synthesizable SystemVerilog IP blocks for hardware area estimations. These blocks can serve as a future prototype for full hardware ASIC or FPGA-based implementation of OSMOSIS.

5.1 Implementing OSMOSIS on top of PsPIN

Packet processing units: OSMOSIS PsPIN architecture is based on scalable silicon-proven RISC-V PULP SoC [12, 38, 56]. The PUs are RI5CY 32-bit cores organized in clusters.

Each PsPIN cluster contains 8 PUs clocked at 1GHz and coupled with a low-latency (1 cycle), multi-banked local memory called L1 scratchpad. For our experiments, we use the default configuration of the PsPIN PU cluster with a 1 MiB L1 data, and 4 KiB L1 instruction caches. Clusters share a global 4 MiB L2 packet buffer and a 4 MiB L2 kernel buffer, which can be used for local data storage.

Portable Programming API: OSMOSIS utilizes PsPIN infrastructure to offload the processing packets to the PUs. The user writes a C kernel cross-compiled on the host for the RISC-V ISA architecture. The kernels are then loaded and executed on the flow packets according to the sPIN API [24].

Kernel IO: The sPIN API permits the programmer to use blocking and non-blocking calls within the kernel code to initiate DMA and sNIC egress packet transfers. Each PsPIN cluster has a 512-bit AXI DMA interconnect engine that connects cluster scratchpad memories to the global sNIC L2 kernel buffer, host DMA engine buffer, and sNIC egress engine buffer. Such an interface allows kernels to issue `read` and `write` data transfers between these buffers. PUs in one cluster can access the local memory of other clusters and the shared L2 kernel sNIC memory in 10 to 30 cycles.

Such design also allows transparently supporting NIC egress packet `send` by internally issuing DMA `write` of the packet from kernel scratchpad memory to the NIC egress engine buffer. Specifically, the PU core L1 scratchpad interfaces a functional implementation of an Ethernet link on top of the AXI bus protocol. PsPIN IO-calls write a DMA command, including source and destination addresses, transfer length, and a pointer to the completion handle to the registers of the PU core. This involves a sequence of 7 `sw` instructions followed by one `lw` instruction. Outstanding IO commands are queued in the cluster command FIFO. The WRR policy arbitrates the command FIFOs for admission to the DMA engines.

Memory management: Our implementation allows to specify the size of the L2 and L1 memories allocatable to tenants. We implement memory isolation using Physical Memory Protection (PMP) PsPIN PU unit. When the kernel accesses L1 and L2 memories, the virtual memory addresses are translated to physical addresses with relocation registers. The PMP then checks that the addresses are within the valid segment range. Like the relocation registers, the PMP unit does not increase the memory access latency [12].

5.2 OSMOSIS Schedulers

FMQ scheduling implementation: FMQ consists of a FIFO queue, the execution context, and the state necessary for scheduling. FIFO queue stores packet descriptors. Each packet descriptor contains a 32-bit pointer to the packet. The execution context has been discussed in depth in Section 4. The scheduling state contains a BVT counter that tracks the past resource utilization of tenants and a priority. We implemented the counter as a 64-bit register selected to avoid overflow[1]. We implemented the priority as a 16-bit register. Using the BVT counter and the priority, the WLBVT scheduler selects FMQs. We implemented WLBVT and the aforementioned per FMQ registers in SystemVerilog with 128 FMQs. Our implementation synthesizes at 1 GHz with a scheduling decision taking five cycles. Most of the latency stems from the weight-limiting part of the WLBVT algorithm, which needs a division that creates issues within a fast hardware implementation. To hide the associated latency, we employ pipelining. We overlap the FMQ arbitration with the DMA transfer of the packet from the L2 packet buffer to the cluster scratchpad taking at least 13 cycles for a 64-byte packet.

Enhanced DMA engine: To avoid HoL-blocking of small requests behind large DMA transfers, OSMOSIS employs transfer fragmentation on the DMA engine interfacing the host and on the egress engine. We implement two modes of fragmentation: a runtime executed *software* fragmentation implemented within the kernel call for a DMA transfer, and a *hardware* fragmentation within the DMA engine.

We implement the software approach as a function wrapper around `pspin_dma_read/write` and `pspin_send_packet` calls. Our middleware splits larger requests into smaller chunks and issues multiple non-blocking DMA requests of smaller sizes. Internally, the OSMOSIS holds the state for each smaller transfer and ensures they execute entirely. As we demonstrate in Section 6, fragmentation helps to avoid HoL blocking, yet the software implementation limits the improvement, which comes at the cost of increased DMA request completion time. We extend the AXI interconnects functional part to reduce software overheads and enable hardware fragmentation. Our extension includes a state for multiple outstanding AXI write requests arbitrated using a WRR scheduler.

6 Evaluation

We study how OSMOSIS allocates sNIC resources under different traffic conditions and workload requirements. We investigate the following research questions:

1. How does the hardware area of critical components of OSMOSIS (e.g., PU clusters and schedulers) scale up with the ingress link rates and the number of tenants?
2. What are the overheads of OSMOSIS compared to the reference PsPIN implementation?
3. What is the maximum load that OSMOSIS can sustain?
4. How fair are OSMOSIS resource allocations?

[1]The overflow of the 64-bit per-FMQ BVT counter at 1 GHz frequency with increments done every cycle will happen in $2^{64} \div 10^{-9}$ sec/op $\div 60$ sec $\div 60$ min $\div 24$ hr $\div 365.25$ days ≈ 584 years.

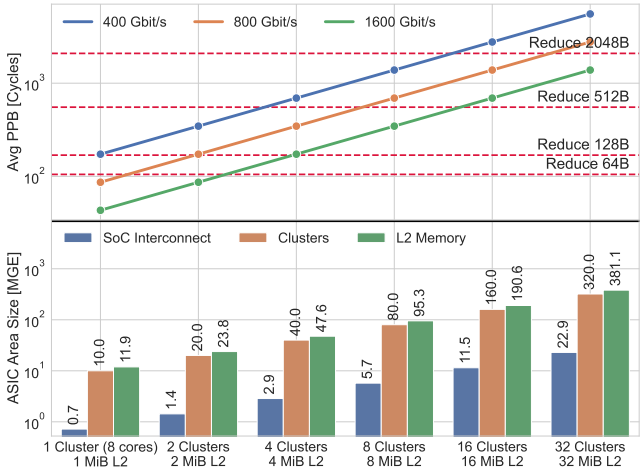

Figure 7: The cost model of PUs, L2 memory and SoC interconnect area scaling synthesized in 22nm GF process, compared to the theoretical per packet budget (averaged for different packet sizes at 64 – 4096 byte interval) achieved with 400, 800 and 1600 Gbit/s ingress link rates.

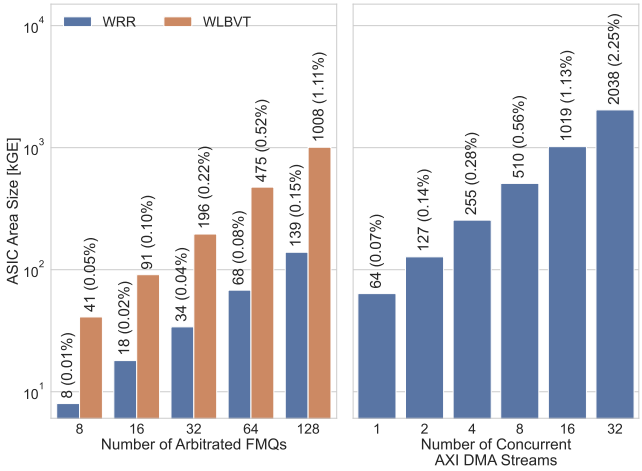

Figure 8: The linear area scaling of WLBVT and WRR schedulers synthesized in GF 22nm process. The captions within the bars show the absolute number of gates and relative area compared to 4 PU clusters with 4 MiB L2.

6.1 Hardware Scaling

To estimate hardware area costs, we synthesize OSMOSIS and PsPIN SystemVerilog IP blocks at 1GHz frequency in GlobalFoundries 22nm node process using Synopsys Design Compiler NXT in topographic mode.

sNIC area scaling with compute capacity: PsPIN clusters utilize a hierarchical SoC-interconnect similar to the Manticore scale-out study [38]. We group four clusters in a *quadrant* that shares a local interconnect. Each quadrant is connected to the L2 memory allowing all cores to access the shared packet buffer. Synthesis studies [12, 38] have shown that area increases and timing overhead of adding more ports to L2 are negligible compared to the overall size of L2. Figure 7 shows our synthesis results, indicating that PsPIN achieves linear scaling of compute capacity with respect to the core area. We also provide an analysis of the cluster requirements for a Reduce workload. For example, 4 PU clusters achieve an average per-packet budget (PPB) (see Section 3) that is enough to sustain Reduce with packets of up to 512 bytes.

OSMOSIS Schedulers Scaling: Figure 8 shows the hardware area consumption of OSMOSIS schedulers. We observe a linear scaling of the FMQ and AXI DMA engine schedulers with the number of inputs. Compared to RR, WLBVT needs 7× more gates, yet with 128 FMQs (e.g., 128 tenants), WLBVT area consumption takes only 1% of PsPIN cluster and L2 memory area.

6.2 Experimental Methodology

We evaluate OSMOSIS runtime performance using cycle-accurate simulation with Verilator v4.228 SystemVerilog simulator [60]. Our experimental testbed features two setups: a *Reference (baseline) PsPIN implementation*, which is a conventional on-path sNIC without multi-tenant OS and a *PsPIN implementation enhanced with OSMOSIS management*.

We configure both setups with 4 PsPIN clusters and 32 cores clocked at 1GHz. Ingress and egress data paths are configured to achieve 400 Gbit/s bandwidth. L2 and host memory are configured with 512 Gbit/s peak theoretical bandwidth. We run experiments using randomly pre-generated packet traces that fully utilize ingress link bandwidth. We sample the sequences of packet arrivals from a uniform distribution, and packet sizes from a lognormal distribution. For fairness measurements, we use Jain's fairness metric [27]. Jain's metric is always scaled between 1 and 1/number of tenants. If Jain's metric is y, then y% of users are treated fairly, and $(100-y)$% are starved. Fair treatment is defined as each tenant having the same priority-adjusted access to resources.

6.3 Synthetic Benchmarks

We evaluate OSMOSIS on a set of synthetic benchmarks to assess its overheads in a low-complexity environment.

R1 R5 Fair HPU allocation: We evaluate the WLBVT scheduler and compare it to the traditional RR. We run two applications, one with a larger *compute cost per byte*, the *Congestor*, and the other with a smaller one, the *Victim*. Both spin in a `for` loop to simulate a compute-bound task. Figure 9 shows how RR over-allocates PUs to the *Congestor*, leading

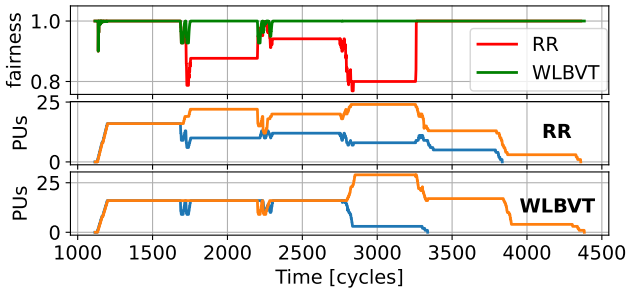

Figure 9: The fairness of WLBVT and RR with two tenants of different compute cost per byte.

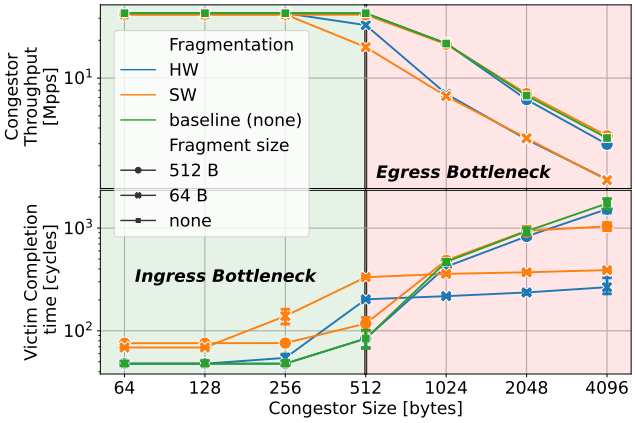

Figure 10: The impact on the *Congestor* throughput and the *Victim* kernel completion time as a function of the *Congestor* size and various fragment sizes.

to lower fairness as shown by Jain's metric. WLBVT consistently splits all the resources equally between tenants. When the *Victim* has no outstanding packets, WLBVT allows the *Congestor* to overtake more PUs. WLBVT enables fair compute resource allocation within OSMOSIS and does not cause slowdowns within the benchmarks.

R2 R5 **Resolving HoL-blocking:** We evaluate the scaling of throughput of the *Congestor* and the kernel completion time of the *Victim* while conducting only Egress transfers that involve AXI writes. Figure 10 presents how OSMOSIS resolves HoL-blocking. Depending on the fragmentation method, the *Victim*'s kernel completion time can be reduced by an order of magnitude while preserving a relative slowdown of only around 2×. The throughput reduction stems from control traffic overhead related to fragmentation, which can be resolved through a custom implementation of the AXI protocol, allowing for parallel transfer states [33, 54]. We also observed two bottlenecks: ingress and egress. In the ingress bottleneck, the incoming link bandwidth is the limit, while in the egress one, the AXI bus congestion causes slowdowns. While the overheads come from the interconnect, OSMOSIS scheduling does not introduce overheads, as evident for low *Congestor* sizes.

6.4 Datacenter Workloads

Additionally, we port a set of real datacenter workloads such as aggregation, histogramming, filtering, and KVS reads and writes to evaluate the performance of OSMOSIS. Our evaluation begins by measuring the impact of OSMOSIS on the packet throughput of standalone applications. We then proceed to evaluate application mixtures to showcase more realistic operational scenarios.

Management overheads: To assess the influence of OSMOSIS management on applications' performance, we start by running them in isolation. Figure 11 displays how OSMOSIS does not introduce considerable overheads for compute workloads. These oscillate within ±3% of the baseline PsPIN implementation and reach the maximum of 310Mpps for the

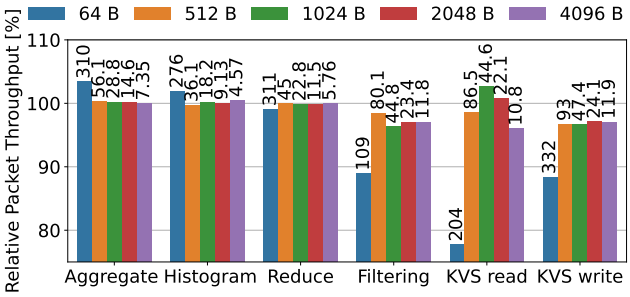

Figure 11: The relative packet throughput of common datacenter workloads run in a standalone mode as a function of fragment size with their raw performance in million packets per second (Mpps) at the top of the bars.

Aggregation workloads. For IO-related workloads, OSMOSIS introduces overheads stemming from the fragmentation, which have been discussed in Section 6.3. This can be resolved by introducing an extension to the AXI bus protocol [33, 54]. While the overheads reach from 23% to 2% and represent the cost of introducing fair and efficient multitenancy, the workloads still achieve 332Mpps in the *KVS write* case.

Application mixtures: Evaluating applications in isolation is not representative of real workloads which occur in multitenant datacenters for which OSMOSIS was designed and where multiple users contend for resources. We consider two application sets: a *compute-bound set* and an *IO-bound* set, each resulting in resource contention between tenants.

The compute-bound set consists of the *Reduce* and *Histogram* workloads. Each is introduced as a *Victim* (64B packets for *Reduce* and 64-128 packets for *Histogram*) and *Congestor* (4KB packets for *Reduce* and 3072-4096 byte packets for *Histogram*). As Figure 12 shows, these workloads saturate

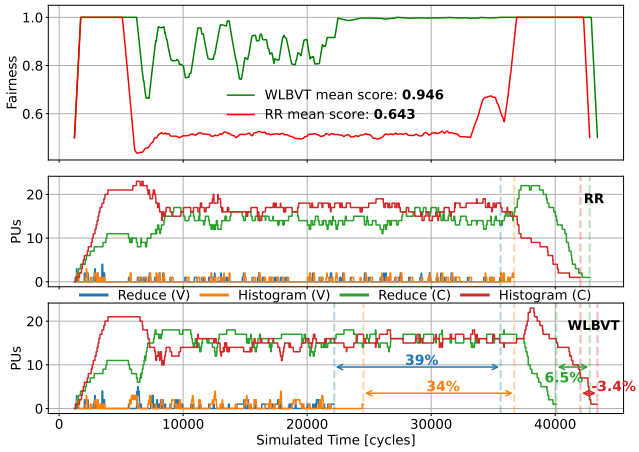

Figure 12: The evolution of tenant PU occupation and time average fairness against the simulated time in cycles. The percentages indicate the reduction in flow completion time for each tenant.

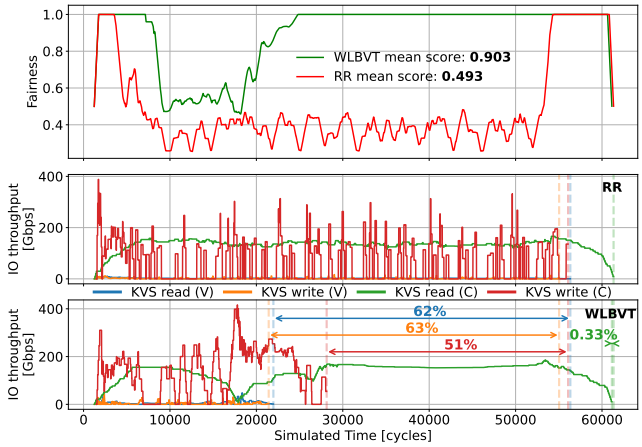

Figure 13: The evolution of tenant IO throughput and time average fairness against the simulated time in cycles. The percentages indicate the reduction in flow completion time for each tenant.

the PUs of the sNIC within the first couple thousand cycles and introduce compute congestion. Using OSMOSIS WLBVT scheduling, each tenant obtains an allocation that is, on average, 47% fairer than that of the typical RR implementation as measured using Jain's metric. Such allocations ensure SLO fulfillment and result in 39-7% faster flow completion times because of lower average contention, while only sacrificing 3% of the *Histogram Congestor*. OSMOSIS achieves a fair and efficient resource allocation.

The IO-bound set consists of *KVS read* and *write* workloads which are again introduced as both a *Victim* and *Congestor* with the same packet size parameters as the *Histogram* workload. For the IO-bound workloads, we focus on the average throughput of each workload. Figure 13 shows that, similarly to the compute case, OSMOSIS obtains a consistently fairer allocation than a traditional RR scheduler by 83% as measured by the average Jain's fairness metric. OSMOSIS also manages to reduce flow completion times for all of the tenants by up to 63%. Such large improvement comes from solving the HoL-blocking problem and obtaining a more efficient allocation. The *KVS read Congestor* is initially suppressed to let other tenants fairly finish their workloads and then obtains full exclusive utilization, eliminating contention and allowing it to regain the lost performance. On the other hand, the other tenants are fairly allocated and, as Figure 14 shows, they do not suffer from HoL-blocking.

Figure 14 also displays the true cost of the aforementioned gains. While the overall flow completion time is reduced for all tenants, the single kernel completion time shows a different story. The HoL-blocking is resolved for the *Victim* tenants, for which the kernel completion time is reduced more than fivefold. However, the other *Congestor* tenants display an up to eightfold increased median kernel completion time.

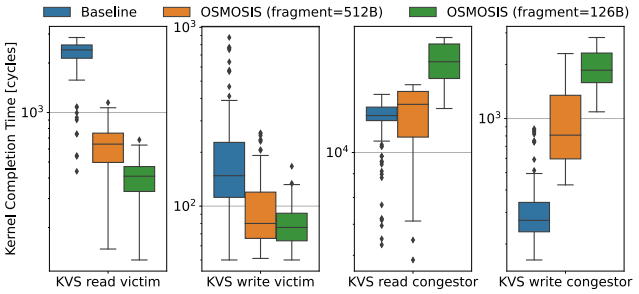

Figure 14: The completion time distribution for IO-bound applications for two fragment sizes.

OSMOSIS achieves overall flow completion time gains for the IO set by allocating the resources fairly, more efficiently, and by parallelizing the packets appropriately. However, it also increases the median single-packet processing time.

7 Conclusions

Enabling user-level in-network processing in modern multi-tenant datacenters brings resource multiplexing challenges. OSMOSIS solves sNIC multi-tenancy by distributing resources such as the egress engine, DMA engine, and processing units across flows with different priorities, input bandwidth, and computational requirements. To achieve fair distribution of resources, OSMOSIS relies on sNIC-specific principles, such as work-conservative dynamic allocation of resources, and NIC-to-host rate limiting. The evaluation shows that OSMOSIS efficiently and fairly redistributes resources, enabling performance isolation and prioritization between flows. It can improve the flow completion times by up to 60% and is fairer by up to 83% than the typical schedulers. We

believe that OSMOSIS could enable wider adoption of sNICs in cloud datacenters with low overhead.

References

[1] AGARWAL, S., AGARWAL, R., MONTAZERI, B., MOSHREF, M., ELMELEEGY, K., RIZZO, L., DE KRUIJF, M. A., KUMAR, G., RATNASAMY, S., CULLER, D., ET AL. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (2022), pp. 198–204.

[2] AGRAWAL, A., AND KIM, C. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)* (2020), IEEE Computer Society, pp. 1–32.

[3] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference* (2010), pp. 63–74.

[4] ANDERSON, T. E., OWICKI, S. S., SAXE, J. B., AND THACKER, C. P. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems (TOCS) 11*, 4 (1993), 319–352.

[5] ATTIG, M., AND BREBNER, G. 400 gb/s programmable packet parsing on a single fpga. In *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems* (2011), IEEE, pp. 12–23.

[6] BALAS, R., AND BENINI, L. Risc-v for real-time mcus - software optimization and microarchitectural gap analysis. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2021), pp. 874–877.

[7] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 49–65.

[8] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2010), IMC '10, Association for Computing Machinery, p. 267–280.

[9] BORROMEO, J. C., KONDEPU, K., ANDRIOLLI, N., AND VALCARENGHI, L. Fpga-accelerated smartnic for supporting 5g virtualized radio access network. *Computer Networks 210* (2022), 108931.

[10] CAI, Q., VUPPALAPATI, M., HWANG, J., KOZYRAKIS, C., AND AGARWAL, R. Towards μ s tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 767–779.

[11] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[12] DI GIROLAMO, S., KURTH, A., CALOTOIU, A., BENZ, T., SCHNEIDER, T., BERANEK, J., BENINI, L., AND HOEFLER, T. A risc-v in-network accelerator for flexible high-performance low-power packet processing. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), IEEE, pp. 958–971.

[13] DONG, Y., YANG, X., LI, J., LIAO, G., TIAN, K., AND GUAN, H. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing 72*, 11 (2012), 1471–1480.

[14] DONG, Y., YU, Z., AND ROSE, G. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O virtualization* (2008), vol. 2.

[15] DUDA, K. J., AND CHERITON, D. R. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *ACM SIGOPS Operating Systems Review 33*, 5 (1999), 261–276.

[16] ETHERNET ALLIANCE. Ethernet Roadmap 2022. https://ethernetalliance.org/technology/ethernet-roadmap/.

[17] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking 1*, 4 (1993), 397–413.

[18] FRIED, J., RUAN, Z., OUSTERHOUT, A., AND BELAY, A. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (2020), pp. 281–297.

[19] GAO, P., DALLEGGIO, A., XU, Y., AND CHAO, H. J. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 551–565.

[20] GAO, P. X., NARAYAN, A., KARANDIKAR, S., CARREIRA, J., HAN, S., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)* (2016), pp. 249–264.

[21] GRANT, S., YELAM, A., BLAND, M., AND SNOEREN, A. C. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 681–693.

[22] GUO, Z., SHAN, Y., LUO, X., HUANG, Y., AND ZHANG, Y. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), pp. 417–433.

[23] HAECKI, R., MYSORE, R. N., SURESH, L., ZELLWEGER, G., GAN, B., MERRIFIELD, T., BANERJEE, S., AND ROSCOE, T. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 861–877.

[24] HOEFLER, T., DI GIROLAMO, S., TARANOV, K., GRANT, R. E., AND BRIGHTWELL, R. spin: High-performance streaming processing in the network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), pp. 1–16.

[25] HOFEMEIER, G., AND CHESEBROUGH, R. Introduction to intel aes-ni and intel secure key instructions. *Intel, White Paper 62* (2012).

[26] HØILAND-JØRGENSEN, T., BROUER, J. D., BORKMANN, D., FASTABEND, J., HERBERT, T., AHERN, D., AND MILLER, D. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies* (2018), pp. 54–66.

[27] HOSSFELD, T., SKORIN-KAPOV, L., HEEGAARD, P. E., AND VARELA, M. Definition of qoe fairness in shared systems. *IEEE Communications Letters 21*, 1 (2016), 184–187.

[28] HUNTER, A., KENNELLY, C., TURNER, P., GOVE, D., MOSELEY, T., AND RANGANATHAN, P. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association, pp. 257–273.

[29] IBANEZ, S., MALLERY, A., ARSLAN, S., JEPSEN, T., SHAHBAZ, M., KIM, C., AND MCKEOWN, N. The nanopu: A nanosecond network stack for datacenters. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)* (2021), pp. 239–256.

[30] IBANEZ, S., MALLERY, A., ARSLAN, S., JEPSEN, T., SHAHBAZ, M., KIM, C., AND MCKEOWN, N. Enabling the reflex plane with the nanopu. *arXiv preprint arXiv:2212.06658* (2022).

[31] INFINIBAND TRADE ASSOCIATION. InfiniBand Specification. https://www.infinibandta.org.

[32] IVANOV, A., DRYDEN, N., BEN-NUN, T., LI, S., AND HOEFLER, T. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems 3* (2021), 711–732.

[33] JIANG, Z., YANG, K., FISHER, N., GRAY, I., AUDSLEY, N. C., AND DONG, Z. Axi-ic rt: Towards a real-time axi-interconnect for highly integrated socs. *IEEE Transactions on Computers 72*, 3 (2022), 786–799.

[34] KAFFES, K., CHONG, T., HUMPHRIES, J. T., BELAY, A., MAZIÈRES, D., AND KOZYRAKIS, C. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 345–360.

[35] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Datacenter rpcs can be general and fast. *arXiv preprint arXiv:1806.00680* (2018).

[36] KHAWAJA, A., LANDGRAF, J., PRAKASH, R., WEI, M., SCHKUFZA, E., AND ROSSBACH, C. J. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 107–127.

[37] KOROLIJA, D., ROSCOE, T., AND ALONSO, G. Do os abstractions make sense on fpgas? In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (2020), pp. 991–1010.

[38] KURTH, A., RÖNNINGER, W., BENZ, T., CAVALCANTE, M., SCHUIKI, F., ZARUBA, F., AND BENINI, L. An open-source platform for high-performance non-coherent on-chip communication. *IEEE Transactions on Computers 71*, 8 (2021), 1794–1809.

[39] LE, Y., CHANG, H., MUKHERJEE, S., WANG, L., AKELLA, A., SWIFT, M. M., AND LAKSHMAN, T. Uno: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), pp. 506–519.

[40] LI, Y., MIAO, R., LIU, H. H., ZHUANG, Y., FENG, F., TANG, L., CAO, Z., ZHANG, M., KELLY, F., ALIZADEH, M., ET AL. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 44–58.

[41] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 318–333.

[42] LIU, M., PETER, S., KRISHNAMURTHY, A., AND PHOTHILIMTHANA, P. M. E3: Energy-efficient microservices on smartnic-accelerated servers. In *USENIX annual technical conference* (2019), pp. 363–378.

[43] MARVELL. LiquidIO-III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf.

[44] MICROSOFT. Introduction to Receive Side Scaling. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[45] MIN, J., LIU, M., CHUGH, T., ZHAO, C., WEI, A., DOH, I. H., AND KRISHNAMURTHY, A. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 106–122.

[46] MINTURN, D. Nvm express over fabrics. In *11th Annual OpenFabrics International OFS Developers' Workshop* (2015).

[47] MOGUL, J. C. Tcp offload is a dumb idea whose time has come. In *HotOS* (2003), pp. 25–30.

[48] NAMYAR, P., SUPITTAYAPORNPONG, S., ZHANG, M., YU, M., AND GOVINDAN, R. A throughput-centric view of the performance of datacenter topologies. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 349–369.

[49] NETRONOME. Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/.

[50] NVIDIA. Bluefield-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.

[51] PETER, S., LI, J., ZHANG, I., PORTS, D. R., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS) 33*, 4 (2015), 1–30.

[52] POURHABIBI, A., SUTHERLAND, M., DAGLIS, A., AND FALSAFI, B. Cerebros: Evading the rpc tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (2021), pp. 407–420.

[53] PREKAS, G., KOGIAS, M., AND BUGNION, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 325–341.

[54] RESTUCCIA, F., BIONDI, A., MARINONI, M., CICERO, G., AND BUTTAZZO, G. Axi hyperconnect: A predictable, hypervisor-level interconnect for hardware accelerators in fpga soc. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (2020), IEEE, pp. 1–6.

[55] RESTUCCIA, F., PAGANI, M., BIONDI, A., MARINONI, M., AND BUTTAZZO, G. Is your bus arbiter really fair? restoring fairness in axi interconnects for fpga socs. *ACM Transactions on Embedded Computing Systems (TECS) 18*, 5s (2019), 1–22.

[56] ROSSI, D., CONTI, F., MARONGIU, A., PULLINI, A., LOI, I., GAUTSCHI, M., TAGLIAVINI, G., CAPOTONDI, A., FLATRESSE, P., AND BENINI, L. Pulp: A parallel ultra low power platform for next generation iot applications. In *2015 IEEE Hot Chips 27 Symposium (HCS)* (2015), IEEE Computer Society, pp. 1–39.

[57] ROTHENBERGER, B., TARANOV, K., PERRIG, A., AND HOEFLER, T. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 4277–4292.

[58] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev. 45*, 4 (Aug. 2015), 123–137.

[59] SINGLA, A., GODFREY, P. B., AND KOLLA, A. High throughput data center topology design. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 29–41.

[60] SNYDER, W. Verilator: Open simulation-growing up. *DVClub Bristol* (2013).

[61] STEPHENS, B. E., AKELLA, A., AND SWIFT, M. M. Loom: Flexible and efficient nic packet scheduling. In *NSDI* (2019), vol. 19, pp. 33–46.

[62] SUN, S., ZHANG, R., YAN, M., AND WU, J. Skv: A smartnic-offloaded distributed key-value store. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)* (2022), IEEE, pp. 1–11.

[63] VAHDAT, A., AND MILOJICIC, D. The next wave in cloud systems architecture. *Computer 54*, 10 (2021), 116–120.

[64] WANG, Z., HUANG, H., ZHANG, J., WU, F., AND ALONSO, G. {FpgaNIC}: An {FPGA-based} versatile 100gb {SmartNIC} for {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), pp. 967–986.

[65] WATERMAN, A., LEE, Y., AVIZIENIS, R., PATTERSON, D. A., AND ASANOVIĆ, K. The risc-v instruction set manual volume ii: Privileged architecture version 1.9. Tech. Rep. UCB/EECS-2016-129, EECS Department, University of California, Berkeley, Jul 2016.

[66] WEIBAI, X. J., XU, Y., ELHADDAD, M., RAINDEL, S., PADHYE, J., AND ZHUO, A. R. L. D. Understanding rdma microarchitecture resources for performance isolation.

[67] WOODRUFF, J., MOORE, A. W., AND ZILBERMAN, N. Measuring burstiness in data center applications. In *Proceedings of the 2019 Workshop on Buffer Sizing* (New York, NY, USA, 2019), BS '19, Association for Computing Machinery.

[68] YAN, Y., BELDACHI, A. F., NEJABATI, R., AND SIMEONIDOU, D. P4-enabled smart nic: Enabling sliceable and service-driven optical data centres. *Journal of Lightwave Technology 38*, 9 (2020), 2688–2694.

[69] YANG, X., EGGERT, L., OTT, J., UHLIG, S., SUN, Z., AND ANTICHI, G. Making quic quicker with nic offload. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (2020), pp. 21–27.

[70] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review 45*, 4 (2015), 523–536.