SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud

Stewart Grant*, Anil Yelam*, Maxwell Bland† and Alex C. Snoeren

UC San Diego †University of Illinois-Urbana Champaign

ABSTRACT

Multiple vendors have recently released SmartNICs that provide both special-purpose accelerators and programmable processing cores that allow increasingly sophisticated packet processing tasks to be offloaded from general-purpose CPUs. Indeed, leading datacenter operators have designed and deployed SmartNICs at scale to support both network virtualization and application-specific tasks. Unfortunately, cloud providers have not yet opened up the full power of these devices to tenants, as current runtimes do not provide adequate isolation between individual applications running on the SmartNICs themselves.

We introduce FairNIC, a system to provide performance isolation between tenants utilizing the full capabilities of a commodity SoC SmartNIC. We implement FairNIC on Cavium LiquidIO 2360s and show that we are able to isolate not only typical packet processing, but also prevent MIPS-core cache pollution and fairly share access to fixed-function hardware accelerators. We use FairNIC to implement NIC-accelerated OVS and key/value store applications and show that they both can cohabitate on a single NIC using the same port, where the performance of each is unimpacted by other tenants. We argue that our results demonstrate the feasibility of sharing SmartNICs among virtual tenants, and motivate the development of appropriate security isolation mechanisms.

CCS CONCEPTS

• **Networks → Network adapters**;

KEYWORDS

Network adapters, cloud hosting, performance isolation

ACM Reference Format:
Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. 2020. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3387514.3405895

*These authors contributed equally.

SIGCOMM '20, August 10–14, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7955-7/20/08.
https://doi.org/10.1145/3387514.3405895

1 INTRODUCTION

Cloud providers have determined that it is inefficient to implement network processing tasks on host cores, and are deploying custom-designed SmartNICs at scale to support traffic scheduling, security, and network virtualization, among others [2, 16]. Enterprises and service providers have reached similar conclusions, and a host of manufacturers have introduced commodity, programmable SmartNICs to accelerate a large variety of tasks [4, 6, 23]. Unfortunately, while the efficiency benefits of SmartNICs are contributing to cloud providers' bottom lines, tenants are barred from sharing in these gains because providers do not allow them to download their own applications onto NIC hardware in virtualized environments.

We explore the potential of opening up the network acceleration benefits of commodity SmartNICs to cohabitating tenants in cloud environments. In particular, we seek to enable individual tenants to run their own, custom on-NIC programs that make use of shared hardware resources to improve performance [33, 42], decrease host CPU utilization [16, 36], or both [11, 35]. The key challenge to running applications from different tenants on shared SmartNIC hardware is ensuring isolation. In particular, we design isolation techniques that work within the confines of an existing manufacturer's SDK and do not require SmartNIC programmers to learn a new language or application framework.

We recognize that production-grade isolation is a very high bar. In particular, cloud platforms have been shown to exhibit numerous side-channel security vulnerabilities [44]. Given the complexity and performance overheads inherent in enforcing truly secure isolation [39], we start by considering whether those costs are even potentially worth incurring. As the first effort to share SmartNICs between tenants, we defer consideration of deliberate malfeasance, potential side-channel or other privacy attacks to future work, and focus exclusively on achieving performance isolation.

In this work, we consider system-on-a-chip (SoC) SmartNICs due to their relative ease of programmability. While several previous studies [16, 33] consider FPGA-based SmartNICs, various aspects of the FPGA design ecosystem (such as the need to globally synthesize, place and route functionality) complicate use in a multi-tenant environment [28]. Yet the design of today's' SoC SmartNICs also frustrate our task, as they lack much of the hardware support found in modern host processors for virtualization and multi-tenancy.

We illustrate the challenge of cross-tenant performance isolation by studying the behavior of a Cavium LiquidIO 2360 SmartNIC when running multiple applications concurrently. We demonstrate that the NIC processing cores, shared caches, packet processing units, and special-purpose coprocessors all serve as potential points of contention and performance crosstalk between tenants. Our

experiments show that in certain instances, co-location can decrease tenant performance by one-to-two orders of magnitude. In response, we develop isolation mechanisms that enable fair sharing of each of the contended resources. Our solutions balance the need to maintain 25-Gbps line-rate processing while leaving as many hardware resources as possible for tenant use.

We prototype our isolation mechanisms in FairNIC, an extension of the Cavium Simple Executive for SmartNIC applications. FairNIC provides strict core partitioning, cache and memory striping, DWRR packet scheduling, and distributed token-based rate limiting of access to the fixed-function hardware accelerator units. We evaluate FairNIC in both micro-benchmarks and realistic multi-tenant environments. We demonstrate that each of our isolation mechanisms can not only enforce fairness, but even defend against tenants that would otherwise exhaust shared NIC resources. We implement two popular SmartNIC-accelerated applications—Open vSwitch (OVS) and a key/value store—and show that both applications can coexist on the same SmartNIC while preserving performance isolation. Hence, we conclude that it is indeed worthwhile from a performance point of view to support SmartNICs in a multi-tenant environment, and discuss potential next steps toward our vision of virtualizing commodity SmartNICs in commercial clouds.

2 BACKGROUND

SmartNICs are network interface cards that allow applications to run offload functionality directly in the data path. Our goal is to enable safe SmartNIC access for multiple tenant applications, which necessitates the development of isolation enforcement mechanisms. The requirements for these isolation mechanisms are influenced by both the service and deployment models chosen by the datacenter operator and the capabilities of the SmartNIC hardware itself.

2.1 Service models

We consider the requirements of three common cloud service models, each in the context of a public cloud (i.e., we assume strict isolation requirements between tenants), and adapt these models to the context of SmartNIC multiplexing. This paper addresses performance isolation issues common to all three service models, but does not fully address the security isolation requirements of any; we discuss these shortcomings further in Section 7.2.

SaaS: In a software-as-a-service model, we envision SmartNIC applications are written, compiled and deployed by datacenter operators. Tenants pay for a selection of these applications to be offloaded onto SmartNICs. Security isolation mechanisms are required mainly to address potential provider errors, but performance isolation is necessary for quality-of-service guarantees in multi-application deployments. These assumptions are similar to those made by the authors of NIC-A which provides tenant isolation on FPGA-based SmartNICs [14].

PaaS: In a platform-as-a-service model developers could write custom SmartNIC applications and submit them to the datacenter operator for approval and deployment onto SmartNICs. This model might restrict tenants' code to allow for easier static checking or software-based access restrictions to hardware such as coprocessors. Runtime isolation mechanisms are necessary to the extent they are not enforced by the platform API and static checking.

	ASIC	FPGA	SoC
Speed	Fastest	Fast	Moderate
Programmability	Limited	Difficult	Straightforward

Table 1: SmartNIC technology trade-offs

IaaS: As with virtual machines, a SmartNIC infrastructure-as-a-service would provide "bare-metal" SmartNIC ABIs against which tenants could run SmartNIC programs unmodified. In this deployment model, performance isolation requires either full hardware virtualization in software or proper hardware support for isolation like Intel VT-x. Security isolation is necessary if the tenants are distrusting, or vulnerable to a malicious third party.

2.2 Types of SoC SmartNICs

SmartNICs are built out of a variety of different technologies including ASIC, FPGA, and SoC. Traditional NICs are ASIC-based, with predefined network semantics baked into hardware. While these offer the best price/performance, they are generally not programmable. Some vendors have shipped high-core-count, programmable ASIC-based SmartNICs [23], but they are famously challenging to program [8] and have seen limited deployment.

Table 1 overviews the trade-offs between different SmartNIC technologies. FPGAs provide a flexible alternative with near-ASIC performance and some hyperscalers already utilize FPGAs in their datacenters [16]. While FPGAs have the advantage of hardware-like performance, they are expensive and power-hungry, and programming them requires expert knowledge of the hardware and application timing requirements. System-on-a-chip (SoC) SmartNICs represent a middle ground by combining traditional ASICs with a modest number of cache-coherent general-purpose cores for much easier programming and fixed-function coprocessors for custom workload acceleration. As a result, SoC SmartNICs seem the most appropriate for tenant-authored applications.

SoC SmartNICs are not homogeneous in design. A key distinction revolves around how the NIC moves packets between the network ports and host memory [35]. On one hand, the "on-path" approach passes all packets through (a subset of) cores on the NIC on the way to or from the network [6]. In contrast, the "off-path" design pattern uses an on-NIC switch to route traffic between the network and NIC and host cores [4]. The variation in designs has trade-offs for packet throughput, with the former requiring more cores to scale to higher line rates, and the latter incurring additional latency before reaching a computing resource. Recently, researchers proposed switching as a general mechanism for routing between SmartNIC resources in a hybrid of both architectures [49].

2.3 Cavium architecture

In this paper, we work with "on-path" LiquidIO SoC SmartNICs from Cavium (now owned by Marvell) [6]. In addition to traditional packet-processing engines for ingress and egress, the OCTEON processor employed by the SmartNIC provides a set of embedded cores with cache and memory subsystems for general-purpose programmability and a number of special-purpose coprocessors for accelerating certain popular networking tasks.

Cavium CN2360s have 16 1.5-GHz MIPS64 cores connected to a shared 4-MB L2 cache and 16 GB of main memory connected via a

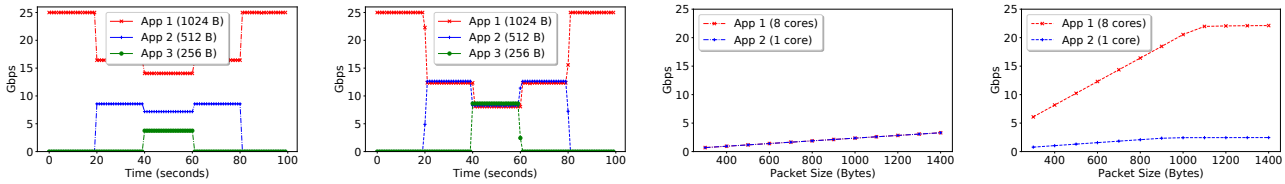

Figure 1: The leftmost plot shows unfair bandwidth allocation between applications due to varying packet sizes. Deficit Round Robin scheduling addresses the issue in the second plot. The third plot shows a case of head-of-line blocking where two applications get roughly the same throughput despite disparate core allocations. The rightmost plot shows that by decoupling ingress queues and buffer pools, application performance is decoupled.

fast consistent memory bus. In its most straightforward use case, each core runs firmware—known as the Cavium Simple Executive—written in C that is executed when packets are delivered to it. While the cores themselves are relatively under-powered, the cards are equipped with a multitude of coprocessors to assist in packet processing. These coprocessors range from accelerating common functions like synchronization primitives and buffer allocation to application-specific functions such as random-number generation, compression, encryption, and regular expression matching.

Packet ingress and egress are handled by dedicated processing units that provide software-configurable QoS options for flow classification, packet scheduling and shaping. To avoid unnecessary processing overheads, there is no traditional kernel and the cores run in a simple execution environment with each non-preemptable core running a binary to launch its own process, and there is no context switching. In a model familiar to DPDK programmers, the cores continually poll for packets to avoid the overhead of interrupts.

End-to-end packet processing involves a chain of hardware components. A typical packet coming in from the host or network goes through the packet ingress engine that tags the packet based upon flow attributes and puts it into pre-configured packet pools in memory. The packet is then pulled off the queue by a core associated with that particular pool, which executes user-provided C code. The cores may call other coprocessors such as the compression unit to accelerate common packet-processing routines. After finishing processing, the packet is dispatched to the egress engine where it may undergo traffic scheduling before it is sent out on the wire or the PCIe bus to be delivered to the host.

3 MOTIVATION & CHALLENGES

The key challenge to enabling tenant access to the programmable features of SmartNICs is the fact that these resources lie outside the traditional boundaries of cloud isolation. Almost all of the virtualization mechanisms deployed by today's cloud providers focus on applications that run on host processors.[1] Indeed, network virtualization is a key focus of many providers, but existing solutions arbitrate access on a packet-by-packet basis. When employing programmable SmartNICs, even "fair" access to the NIC may result in disproportionate network utilization due to the differing ways in which tenants may program the SmartNIC. In this section, we demonstrate the myriad ways in which allowing tenants to deploy applications on a SmartNIC can lead to performance crosstalk.

[1]Some providers do provide access to GPU and TPU accelerators, but that is orthogonal to a tenant's network usage.

3.1 Traffic scheduling

Link bandwidth is the main resource that is typically taken into consideration for network isolation when working with traditional ASIC-based fixed-function NICs. Bandwidth isolation for tenant traffic is usually enforced by some form of virtual switch employing a combination of packet scheduling and rate-limiting techniques [24, 30]. Because per-packet processing on host CPUs is not feasible at high link rates, modern cloud providers are increasingly moving traffic-scheduling tasks to the NIC itself [2, 16].

While this approach remains applicable in the case of SmartNICs, one of the key features of programmable NICs is the wealth of hierarchical traffic-scheduling functionality. Hence, care must be taken to ensure that a tenant's internal traffic-scheduling desires do not conflict with—or override—the provider's inter-tenant mechanisms. Moreover, because tenants can now install on-NIC logic that can create and drop packets at will, host/NIC-bus (i.e., PCIe) utilization and network-link utilization are no longer tightly coupled, necessitating separate isolation mechanisms for host/NIC and network traffic.

3.1.1 Packet egress. Bandwidth isolation requires accounting for the different packet sizes of different NIC applications—which may differ from the original packet size when sent by the tenant's host-based application. The leftmost portion of Figure 1 shows the default behavior when we run three on-NIC applications that generate different packet sizes. Despite equal core and host/NIC traffic allocations, outgoing packets from the NIC cores to the network are scheduled on a round-robin basis resulting in unfair link bandwidth allocation (applications with larger packet sizes consume a larger share). The second plot shows that fair allocation can be restored by enforcing appropriate traffic scheduling (deficit round robin in this case) at the NIC egress—after on-NIC tenant application processing.

3.1.2 Packet ingress. Ingress link bandwidth cannot be isolated by the NIC itself as hosts are not in control of incoming traffic. Datacenters usually use some form of sender-side admission control that is out of scope for this paper. Once traffic arrives at the NIC, however, ingress hardware parses the packets and determines how to handle them. Traditional NICs generally DMA the packet directly into tenant host memory, but packets in SmartNICs are likely destined to on-NIC cores for processing. While the processing rate of SmartNIC ingress hardware is sufficient to demultiplex incoming traffic, the effective service rate of the pipeline is gated by how fast packets are consumed by later stages (i.e., tenant application cores)

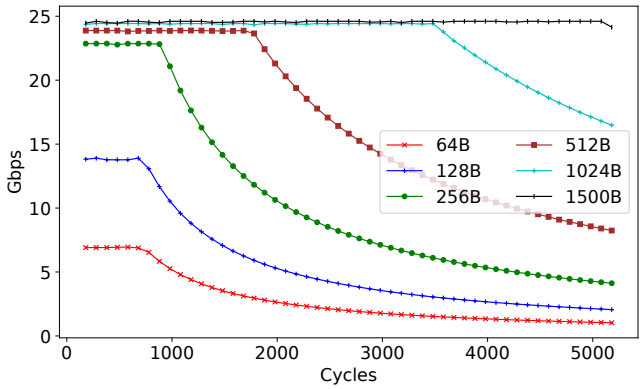

Figure 2: Maximum throughput of seven NIC cores for various packet sizes as a function of processing cycles per packet

of the pipeline. Because tenants process their packets at different rates, it is important to separate traffic as soon as possible.

To demonstrate this issue, we generate two distinct traffic flows destined to different tenants, each one of which is running an on-NIC OVS application. One tenant's application is allocated eight NIC cores while the other uses just one, effectively limiting the latter flow's throughput to one-eighth of the first. However, as shown in the third portion of Figure 1, both flows are processed at the same rate—namely that achievable by one core—due to head-of-line blocking: The ingress engine uses a single buffer pool per port and does not differentiate between tenants. By allocating separate buffer pools and separating the traffic immediately upon arrival (see Section 4.2.1), we are able to restore proportional allocation as shown in the rightmost portion of the figure.

3.2 Core cycles

As discussed in Section 2 general-purpose cores provide the programmability at the heart of SoC SmartNICs. While other execution models exist, usually cores perform end-to-end packet processing wherein each core processes a batch of packets at a time and runs to completion before moving on. The amount of time each core spends on a batch determines the effective throughput of the application. In particular, the more complicated an application's logic, the lower throughput an individual core can deliver. Critically, individual cores on today's SoC-based SmartNICs are unable to keep up with commodity (e.g., 25-Gbps) link rates, and even generous core allocations may fall short when tenant application processing is particularly involved.

To characterize the packet processing capabilities of our SmartNIC, we measure the throughput of a simple NIC program running on seven cores that redirects incoming packets back to the network, but incurs a specified amount of artificial overhead for each packet. We repeat this experiment for various packet sizes and plot throughput as a function of cycles spent per packet in Figure 2. (Figures 2–4 of Liu et al. [35] show similar trends for other commodity SmartNICs.) The plot shows that within a couple of thousand cycles (instructions), core processing replaces link bandwidth as the limiting resource, even for packet sizes as large as 1 KB. Hence, appropriate core allocation is critical for application performance, even with relatively simple processing tasks.

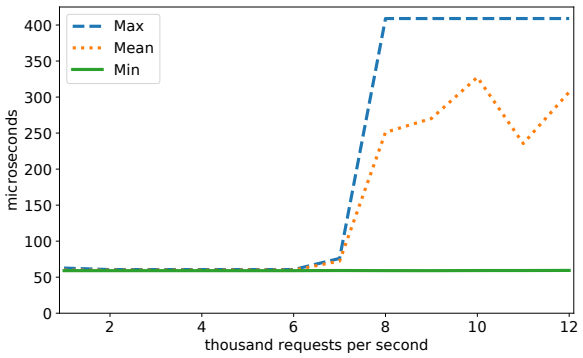

Figure 3: ZIP latency as a function of offered load

3.3 Memory access latency

Programs that process packets at link rate must meet tight timing requirements which are frustrated by the memory access latencies of typical SmartNICs (≈60-ns penalty for an L2-cache miss in our case; see, e.g., Table 2 of Liu et al. [35]). Cavium's programmers' guide extensively documents techniques for packet processing in primarily L1 cache and stresses the criticality of working within the limits of L2. Even if individual tenant applications are diligent in their memory locality, however, the L2 cache is typically shared across cores on a SmartNIC, meaning applications are likely to evict the cache lines of their neighbors. On the Cavium CN2360, all 16 cores share a single L2 cache, making the issue particularly acute.

The performance degradation from cache interference can clearly be seen when running a key/value store (KVS) program (described in Section 6.2.2) alongside a program with poor data locality that issues a large number of memory accesses, resulting in high cache pressure. Our KVS program runs on eight cores sharing a total of 5 MB of RAM. The high-cache-pressure application runs on the other eight cores, stepping over a large allocation of memory at 128-byte intervals to maximize cache-line evictions. As detailed in Table 2, the throughput of the KVS program drops by over an order of magnitude (from 23.55 to 3.2 Gbps) in the presence of the cache-thrashing application, while transaction latency increases by more than two orders of magnitude (from 65 to over 6700 microseconds).

3.4 Coprocessors

SoC SmartNICs like Cavium's LiquidIO come with a rich ecosystem of hardware coprocessors. There are a variety of hardware accelerators that implement common networking tasks such as random number generation, secret key storage and access, RAID, ZIP, and deep packet inspection (regular-expression matching). Each coprocessor has different performance characteristics and physical location which can affect a given core's access latency to the offload.

Each accelerator has a roughly fixed rate at which it can compute; until requests over-saturate that rate no queuing occurs. If all requests were synchronous and took the same amount of time, core isolation would imply fair accelerator access. Some operations, however, incur latencies that are proportional to the size of the input data so applications issuing larger requests can gain disproportional access, leading to starvation for coexisting applications.

Figure 3 shows the latency spike that results when the ZIP (de)compression accelerator is overloaded with very large (16-MB

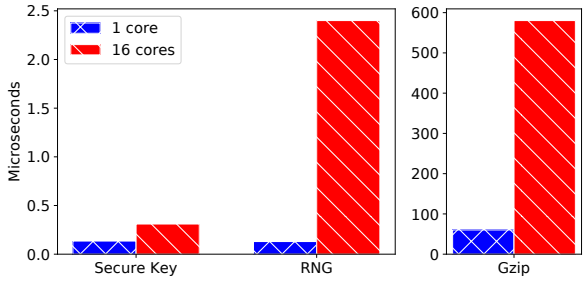

Figure 4: Coprocessor latency with(out) contention

in this example) requests. Until around 3,000 requests per second, the ZIP accelerator is able to meet demand; afterward, requests are queued. The flat maximum latency is the point at which the eight cores in the experiment all block on ZIP requests; had more cores participated in the experiment the latency would be even greater.

We find that the majority of accelerators have similar latency-response curves, with the exception of accelerators that have (seemingly) non-deterministic execution times such as the regular-expression accelerator. Figure 4 shows the minimum and maximum latency observed for three different accelerators; latency increases under contention are approximately an order of magnitude.

3.5 Bus arbitration

Others have shown that PCIe bandwidth arbitration can become a shared bottleneck [27, 34, 40] and propose solutions [40]. We do not encounter that limit in our current configuration (we use one 25-Gbps PCIe 3.0 ×8 SmartNIC per host), but commercial clouds may need to employ appropriate mechanisms to address the contention.

4 ISOLATION TECHNIQUES

FairNIC provides a set of per-resource isolation techniques to ensure that each resource is partitioned (wherever possible) or multiplexed according to tenant service-level objectives. In this section, we introduce our isolation techniques by demonstrating how they solve the issues discussed in the previous section and then discuss their costs in the context of our targeted SmartNIC platform.

In our current implementation, SLOs are expressed in terms of per-resource weights (e.g., fraction of cores or DWRR shares). By expressly allocating every resource in a packet's path that could become a shared point of contention, FairNIC effectively dedicates a portion of the SmartNIC's end-to-end packet processing pipeline to each tenant as shown in Figure 5. Note that individual resources may be allocated in different proportions depending on the needs of each tenant. Moreover, we presume that tenants provide their offload applications to the cloud operator for verification before installation (i.e., the PaaS model from Section 2.1). The cloud provider may choose to test the application or employ static analysis to ensure benign behavior in the common case. In particular, we assume that the applications are written using our framework, and do not attempt to circumvent our isolation mechanisms.[2] We discuss the limitations of these assumptions in Section 7.

[2]Cavium's Simple Executive does not employ hardware memory protection; we leave support for such traditional isolation mechanisms to future work.

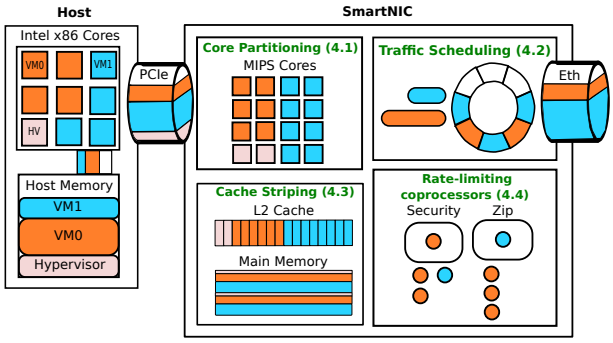

Figure 5: FairNIC sharing resources between two tenants, shown in orange and blue. (Some resources are consumed by FairNIC itself, depicted in pink.)

4.1 Core partitioning

The cornerstone of FairNIC's isolation is a static partitioning of cores across tenant applications: each tenant application is assigned a set of cores that process all of that tenant's traffic in a non-work-conserving fashion. Static application-core mappings allow tenant applications to benefit from instruction locality in the L1 cache and simplify packet processing. We configure the ingress engine to group packets by tenant MAC address and directly steer packets to the cores upon which that tenant's application is running.

Costs. While time-sharing cores across applications could, in principle, result in more efficient use of resources, the required context switches would likely add significant delay to packet processing. Our approach fundamentally limits the number of tenant applications a given SmartNIC can support, but we argue that today's SmartNICs provide an adequate number of cores (e.g., 16–48 for the LiquidIO boards we consider) for the handful of tenants sharing a single machine. Moreover, hosts shared by a significant number of network-hungry tenants are likely to be provisioned with more than one NIC to deliver adequate link bandwidth.

4.2 Traffic scheduling

Cavium NICs come with highly configurable packet ingress/egress engines that support a variety of quality-of-service features in hardware. For example, the egress engine provides multiple layers of packet schedulers and shaper units that can be configured in software to build hierarchical packet schedulers [48]. Each of these units provides a set of scheduling algorithms (typically, a combination of deficit weighted round robin (DWRR) [47], strict priority scheduling, and traffic shaping) and schedule packets at line rate.

4.2.1 Ingress. To isolate incoming traffic, we program the ingress hardware to differentiate between packets from different tenants and direct each tenant's traffic to its own separate buffer pool where it waits for cores to process it. Once these pools fill up, the ingress hardware starts dropping packets, but only those belonging to tenants with full buffers.

4.2.2 Egress. We use deficit weighted round robin to ensure bandwidth isolation for tenant applications. FairNIC implements a hierarchical scheduler where each tenant gets an independent subtree of packet schedulers (which they can configure in whatever

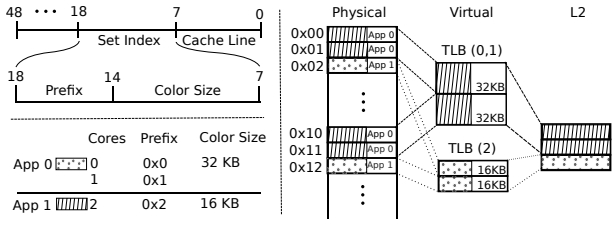

Figure 6: FairNIC inserts TLB entries of various sizes depending on the number of cores assigned.

manner they like), each of which falls into a DWRR scheduler at the root, with one input queue for each application and weights (DRR quanta) on the input queues proportional to their SLOs. One downside of DWRR is that it does not provide strong latency guarantees and may exacerbate tail-latency issues. While there are "better" schedulers in the literature, DWRR is available in hardware and runs at line rate; latency is not an issue in our case as we only need one input queue per application.

Costs. FairNIC's tenant isolation consumes one of the layers of packet schedulers/shapers in the egress engine, leaving fewer layers for other intra-tenant traffic scheduling purposes. Moreover, FairNIC does not at present interpose upon dynamic modifications to the packet-processing engines, passing the burden of ensuring that tenant applications do not attempt to reconfigure the scheduling hardware to circumvent pre-assigned policies to the operator.

4.3 Cache striping

Our solution to L2-cache isolation is to provide each application an isolated region of physical memory. FairNIC implements this isolation by explicitly constructing each application's virtual address space so that its (contiguous) virtual heap is mapped to a (striped) set of physical memory regions that occupy distinct L2 cache lines. Cavium CN2360 NICs have a 16-way, 2048-set-associative L2 cache, with 128-byte cache lines. Cache lines are indexed with address bits 7 to 17. Assuming each core gets an equal-sized memory region (to which we refer to as a color), with 16 cores, the upper-4 bits (14–17) of the cache index are a color prefix corresponding to the color of each core. Each core's colored memory consists of non-contiguous stripes of 16-KB chunks.

16-KB chunks are ideal for paging, but the latency induced by walking a page table is far too great when serving packets at 25 Gbps. Rather than page we statically set TLB entries at calls to `malloc`.[3] Our `malloc` allocates a single color of 16-KB chunks of physical memory on a per-core basis. TLB entries are written to the calling core which stitches the non-contiguous physical allocation into a single contiguous virtual allocation. Each core has 255 TLB entries so FairNIC can support up to 4 MB of colored allocation per core with a total of 64 MB of isolated memory in the system.

MIPS TLBs allow for variable-sized TLB entries. We utilize this feature to provide applications running on multiple cores with proportionally larger stripes of isolated L2 cache. An application with n cores is allocated $(n \cdot 16384)$-byte contiguous stripes of physical memory. As shown in Figure 6, identical expanded TLB entries of size $n \cdot 16384$ are mapped to each of the application's cores.

[3]Our implementation does not currently stripe code or stack segments.

Coloring memory in this way fundamentally limits NIC programs to a small portion of the 16-GB available physical memory. However, any program wishing to access larger regions of memory is free to implement its own pager.

Costs. Applications using the Cavium Simple Executive have their text, stack and fixed-sized heap loaded into a 256-MB contiguous region of physical memory which is mapped into their virtual address space using three TLB entries. Accesses outside this pre-configured region (e.g., packet buffers, coprocessors, etc.) use physical addressing. FairNIC employs virtual addressing for all references to enforce isolation. Hence, memory references incur up to one additional cycle of latency for TLB translation that was previously not needed. On the plus side, FairNIC's virtual addressing provides wild-write [9] protection for buggy application code.

4.4 Rate-limiting coprocessor access

FairNIC delivers accelerator performance isolation by rate-limiting requests to each accelerator to an aggregate rate the unit can sustain without queuing, similar in nature to the queue minimization strategy of DCTCP [1]. In so doing, FairNIC ensures that any allowed request to an accelerator observes the minimum possible latency. If an accelerator is over-subscribed, each tenant exceeding their fair share will be throttled, but the remaining tenants are not impacted. Because not all tenant applications use each accelerator, FairNIC also provides a form of work conservation that allows tenants to divvy up accelerator resources allocated to cores not currently accessing them.

An ideal implementation might employ a centralized DRR queue for each accelerator. Unfortunately, there is no hardware support for such a construct, and our best software implementation of a shared queue increases the latency of accessing offloads by at least 300 ns. In some cases—such as the random number generator and secret key accelerator—that delay dominates accelerator access times.

Instead, FairNIC implements a distributed algorithm, similar in spirit to distributed rate-limiting (DRL) [43] and sloppy counters [3]. When a core first requests the use of an offload, a token rate-limiter is instantiated with a static number of tokens. This base rate is the minimum guarantee per core. When a call to an accelerator is made, the calling core decrements its local token count. When its tokens are exhausted, it checks if sufficient time (based on its predefined limit) has passed for it to replenish its token count. This mechanism allows for cores to rate-limit accesses without directly communicating with one another and incurring the additional 100-ns latency of cross-core communication. Using distributed tokens in place of a centralized queue has the downside that requests can be bursty for short periods. The maximum burst of requests is double a core's maximum number of tokens. Hence, the burst size can be adjusted by setting how often tokens are replenished.

Static token allocation is not work conserving: There may be additional accelerator bandwidth which could be allocated to a core with no remaining tokens in its given window. To attain work conservation we allow a core to steal tokens from the non-allocated pool when they run out. Stolen tokens are counted separately from statically allocated tokens and are subject to a fair-sharing policy. Specifically, we implement additive increase, multiplicative decrease as it allows for cores to eventually reach stability and it is

adaptive to changing loads [10]. To reduce the overhead of sharing, cores only check the global counter when they run out of tokens and have consumed them at a rate above their limiter.

We measure the maximum effective throughput of each accelerator empirically (see Section 3.4) and set token allocations accordingly. Unfortunately, not all accelerators run in constant time. Accelerators such as ZIP and RAID execute as a function of the size of their input. For these accelerators we dynamically calculate the number of tokens based on the size of the request to retain the desired rate of usage. We leave (seemingly) non-deterministic accelerators such as the regular-expression parser to future work.

Costs. Rate-limiting incurs the overhead of subtracting tokens from a core's local cache in the common case. When accessing the global token cache cores must access a global lock and increment their local counters (≈100 ns), which is a substantial delay in the case of the fastest accelerators (e.g., random number generation). Moreover, applications are limited to an aggregate rate that ensures the lowest-possible coprocessor access latency, which, due to imprecision in calibration, may under-utilize the coprocessor. It is possible unrestricted access might lead to higher overall throughput.

5 IMPLEMENTATION

Cavium LiquidIO CN2360s come with a driver/firmware package where the host driver communicates with the SmartNIC firmware over PCIe. It provides support for traditional NIC functions such as hardware queues, SR-IOV [12] and tools like `ifconfig` and `ethtool`. FairNIC extends the firmware by adding a shim layer that operates between core firmware and the applications. The shim includes an `application` abstraction which can execute multiple NIC applications. FairNIC includes an isolation library that implements core partitioning, virtual memory mapping and allocation, and coprocessor rate-limiting. The shim provides a `syscall`-like interface for applications to access shared resources.

5.1 Programming model

Each NIC application must register itself as an `application` object. FairNIC maintains a `struct` (portions shown in the top part of Figure 7) that tracks state and resources (like memory partitions and output queues) associated with each application, along with a set of callback functions for initialization and packet processing. At tenant provisioning time, the cloud provider assigns each tenant application a `weight` that is used in cache partitioning and token allocation, a `coremask` that explicitly assigns NIC cores, and an (`sso_group`) ID which is used to tag all of the application's packets. FairNIC maintains set of host queues (`host_vfs`) for interacting with the tenant VMs on the host, output queues (`pko_ports`) to send packets on the wire and memory regions (`memory_stripes`) assigned to it. The tenant provides callbacks for traffic from the host and wire which FairNIC invokes when packets arrive.

5.2 Isolation library

We implement our isolation mechanisms discussed in Section 4 as a C library and expose methods (shown in the bottom portion of Figure 7) that applications call to allocate memory, send packets or access coprocessors per the isolation policy. None of these interfaces prevent applications from bypassing FairNIC and directly accessing

```
typedef struct application {
    char *name;
    uint16_t weight;
    coremask_t cores;
    uint16_t sso_group;
    uint16_t host_vfs[];
    uint16_t pko_ports[];
    uint64_t dest_mac;
    void *memory_stripes[];
    int (*global_init) (struct application* app);
    int (*per_core_init) (struct application* app);
    int (*from_host_packet_cb) (struct application* app,
         packet* work);
    int (*from_wire_packet_cb) (struct application* app,
         packet* work, int *port);
}
```

```
void* memory_allocate(application, size);
void memory_free(void* p);
int send_pkt_to_host(application, packet, queue);
int send_pkt_to_wire(application, packet, queue);
int call_coprocessor(application, type, params);
```

Figure 7: FairNIC provides an `application` abstraction (top) and an isolation library which exposes an API for applications to access NIC resources (bottom).

NIC resources. Moreover, all code runs in the same protection domain and we do not make any claims of security isolation. We assume that the application code is not malicious and uses the provided library for all resource access.

Cache striping. Based on the `weight` property, each application is allocated regions of memory during initialization, which are made accessible through memory stripes. Applications use our memory API to allocate or free memory, which also inserts the necessary TLB entries for address translation.

Packet processing. Applications register callbacks for when they receive packets and send packets to both host and wire using our provided API. As a proof of concept, we use SR-IOV virtual functions (VFs) to classify host traffic and Ethernet destination addresses to classify wire traffic. Using the `host_vfs` property, we dedicate a set of VFs for each application and tag packets on these VFs with group ID `sso_group`. This labeling also allows for allocating separate buffer pools and sending back-pressure to only certain VFs (and tenants) as our isolation mechanisms kick in and constrain their traffic, while other tenants can keep sending.

Coprocessor access. Applications invoke coprocessors via wrapped calls (not shown) to existing Cavium APIs. Each call has blocking and non-blocking variants. The wrapped calls first check the core local token counter for the coprocessor being called. On the first call, tokens are initialized by setting their value to the guaranteed rate specified in the application's context. If the core has available tokens it decrements its local count and makes a direct call to the coprocessor. If a core has no tokens, it checks its local rate-limiter. If enough time has passed since its last invocation, the local tokens are replenished. Otherwise, the

global token cache is accessed. If available, global tokens are then allocated to the local cache of a core. Global overflow tokens are single-use, and can only be reclaimed by re-checking the global cache.

6 EVALUATION

In this section we demonstrate FairNIC's ability to run multiple tenant applications simultaneously and evaluate the effectiveness of core partitioning, cache striping, and coprocessor rate-limiting. We use our own implementations of Open vSwitch and a custom key/value store that downloads functionality to the SmartNIC.

6.1 Experimental setup

Our testbed consists of two Intel servers, one equipped with a Cavium 2360 SmartNIC and the other with a regular 25-Gbps NIC, connected to each other via a point-to-point SFP+ cable. Each of the servers sports forty 3.6-GHz x86 cores running Ubuntu 18.04. The Cavium NIC that hosts our NIC applications features 16 1.5-GHz MIPS cores, 4 MB of shared L2 cache and 16 GB of DRAM. The server with the SmartNIC hosts tenant applications while the second server generates workloads of various sizes and distributions using DPDK `pktgen` [18]. We emulate a cloud environment by instantiating tenants in virtual machines (VMs) using KVM [22] and employ SR-IOV between the VMs and SmartNIC.

6.2 Applications

We implement two applications that are frequently (c.f. Table 3) employed in the literature to showcase SmartNIC technology: virtual switching and a key/value store.

6.2.1 Open vSwitch datapath. Open vSwitch (OVS) is an open-source implementation of a software switch [19] that offers a rich set of features, including OpenFlow for SDN. OVS has three components: `vswitchd` that contains the control logic, a database `ovsdb` to store configuration and a `datapath` that handles most of the traffic using a set of match-action rules installed by the `vswitchd` component. OVS datapath runs in the kernel in the original implementation and is usually the only component offloaded to hardware.

We start with Cavium's port of OVS [7] and strip away the control components while keeping the datapath intact. For our experiments, the control behavior is limited to installing a set of pre-configured rules so that all flows readily find a match in the datapath. Unless specified otherwise, each rule simply swaps Ethernet and IP addresses and sends the packet back out the arriving interface.

6.2.2 Key/value store. We implement a key/value store (KVS) which has its key state partitioned between the host's main memory and the on-NIC storage. The NIC hosts the top-5% most-popular keys, while the remaining 95% are resident only in host memory. Due to the complexities involved in porting an existing key/value store such as Memcached [17] or Redis [5] we developed our own streamlined implementation that supports the standard put, get, insert, and delete operations. We modify the open-source version of MemC3 [15] to run in both user-space and on the SmartNIC. MemC3 implements a concurrent hash table with constant-time worst-case look-ups.

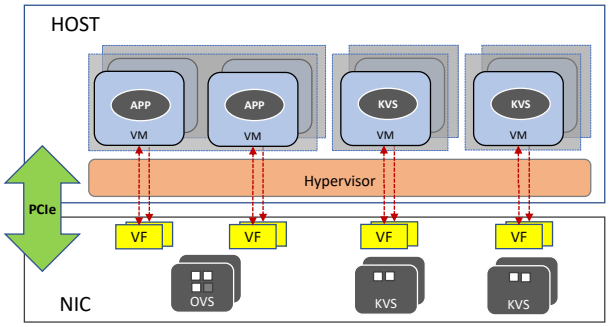

Figure 8: Tenants (shown in gray) are deployed in KVS VMs (blue), which can communicate with FairNIC applications (dark gray) through SR-IOV.

To drive our key/value store, we extend DPDK's packet generator to generate and track key/value requests with variable-sized keys. The workload requests keys using a Zipf distribution.

6.3 Cohabitation

We start by demonstrating FairNIC's ability to multiplex SmartNIC resources across a representative set of tenants each offloading application logic to the SmartNIC. In the configuration shown in Figure 8 we deploy six tenants across eight virtual machines and all sixteen NIC cores. Four tenants run our KVS application in one VM paired with a corresponding SmartNIC application. Two other tenants run two VMs each and use our OVS SmartNIC application to route traffic between them. The OVS applications are assigned three or four NIC cores each, while the KVS applications each run on two. (The FairNIC runtime executes on the remaining core). We send traffic from a client machine at line rate (25 Gbps) and segregate the traffic such that each tenant (app) gets one-sixth of the total offered load (≈4 Gbps).

As shown in Figure 9, each of the (identically provisioned) KVS tenants serve the same throughput, while the two tenants employing OVS obtain differing performance due to their disparate core allocations. The KVS tenants all deliver relatively higher throughput and low per-packet latency because most requests are served entirely by their on-NIC applications, while the OVS tenants' responses are much slower as packets are processed through the VMs on the hosts. Note that whenever an app is not able to service the ≈4-Gbps offered load, it means that the cores are saturated due to high packet rate which happens at lower packet sizes for all the tenants. The right-hand plot shows a CDF of the per-packet latencies experienced by each of the tenants at 1000-B packet size. OVS tenants experience higher latencies due to queue buildup as they are overloaded (which is worse for OVS 2 with fewer cores) while the KVS tenants comfortably service all offered load at this packet size.

6.4 Performance isolation

We now evaluate the performance crosstalk between two tenants each running an application on the NIC. The first tenant runs a well-behaved application that runs in its normal operation mode.

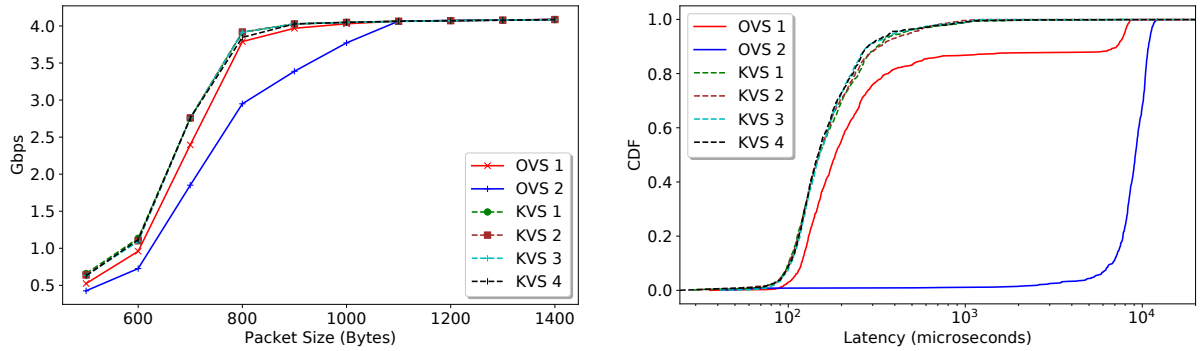

Figure 9: Throughput (left) and per-packet latency (right) for six cohabitating tenants using FairNIC: two tenants running a four and three-core OVS application switching traffic between two VMs each, and four tenants running one VM with a corresponding KVS application across two cores.

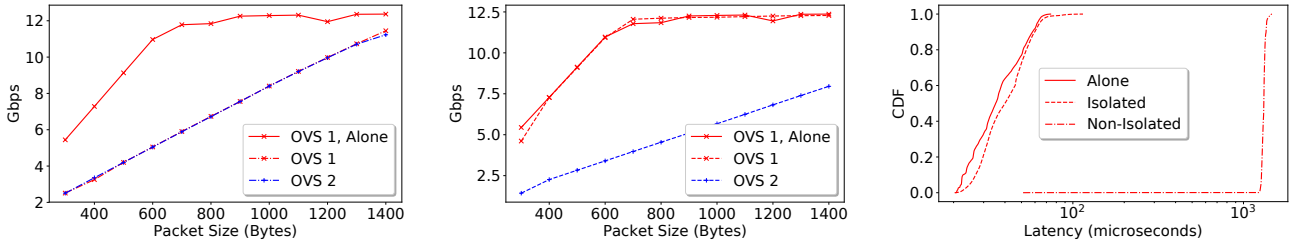

Figure 10: The left two plots show the throughput of two cohabitating OVS applications without and with FairNIC isolation, respectively. The rightmost plot shows per-packet latencies for OVS 1 with 1-KB packets.

The other tenant runs a second application in a deliberately antagonistic fashion that exhibits various traffic or resource usage/access patterns in order to impact the performance of the first one.

6.4.1 Traffic scheduling and core isolation. For this experiment, we run two instances of Open vSwitch, OVS 1 and OVS 2. Both run the same implementations of our Open vSwitch offload with similar sets of flow table rules, except for one difference. While OVS 1 has three actions for each flow rule: `swap_mac`, `swap_ip` (that swap Ethernet and IP source and destination addresses, respectively) and `output` (send the packet out on the same port)—actions that effectively turn the packet around—OVS 2 has more core-intensive packet-processing rules with an extra 100 `swap` actions per packet (representative of a complex action). This extra processing reduces the throughput of OVS 2 compared to OVS 1 (given same number of cores for each). We send 50/50% OVS 1/2 traffic on the wire, of which only a portion is returned based on the the effective capacity of each application which we use to measure throughput and latencies.

We consider three different configurations: *alone*, where OVS 1 is run by itself on seven cores; *non-isolated*, where OVS 1 and OVS 2 are run together across 14 cores with each core servicing either instance depending on the packet it receives; and finally *isolated*, where OVS 1 and OVS 2 are each assigned to a distinct set of seven cores and packets are forwarded directly to the appropriate cores.

The first two plots in Figure 10 show the throughput of OVS 1 and OVS 2 in the non-isolated and isolated scenarios, respectively; in both cases we plot the performance of OVS 1 alone for reference. In the non-isolated scenario on the left, the sharing of core cycles causes OVS 1 and OVS 2 to have same throughput due to head-of-line blocking on the slower OVS 2 packet processing. Ideally OVS 1 would perform at the throughput level shown in the alone case. This unfairness is corrected in the core-isolated scenario shown in the middle graph that decouples the throughput of the two applications and lets them process packets at their own rates.

Similar effects can be seen for latencies as well. As a baseline, round-trip latencies for packets that are bounced off the NIC over the wire fall in the 10–100 μs range. These latencies are amplified by an order of magnitude the moment receive throughput goes above what the application can handle and queues build up. While applications can choose to stay within their maximum throughput limit, it does not help in the non-isolated case as the throughputs of both applications are strictly coupled. This effect is demonstrated in the rightmost plot of Figure 10 which shows OVS 1 latencies in the alone, non-isolated and isolated cases; it suffers a significant latency hit in the non-isolated case.

6.4.2 Cache striping. We demonstrate the effectiveness of FairNIC's cache isolation by running KVS alongside a cache-thrashing program. The KVS application component is allocated 5 MB of NIC memory and services requests generated at 23 Gbps (4-byte keys and 1024-byte values) according to a YCSB-B (95/5% read/write ratio) distribution: 5 percent of keys are "hot" and requested 95 percent of the time. The experiment has three configurations: *alone*, where KVS runs by itself on eight cores, *isolated*, where we use FairNIC to run KVS alongside the cache thrasher (assigned to the other eight cores), and *non-isolated*, where we turn off FairNIC's

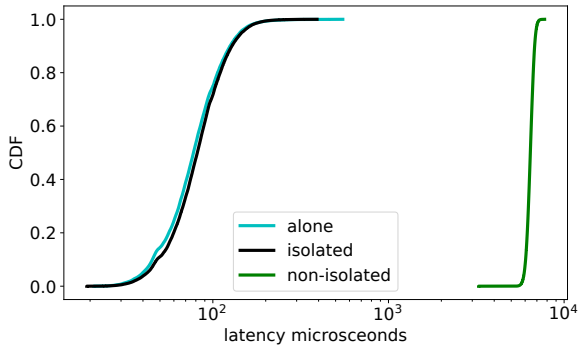

Figure 11: Key/value store response latencies alongside an antagonistic cache-thrashing program, with and without cache striping.

Experiment	Mean Latency	Gbps
Alone	65.69	23.55
Isolated	100.52	23.55
Non-Isolated	6764.20	3.2

Table 2: Mean response latency and average bandwidth of KVS with and without cache coloring.

cache striping. The duration of each experiment is roughly five minutes or approximately 100M packets. Figure 11 plots CDFs of the per-request response latency for each of the configurations. Table 2 reports mean latencies and bandwidths.

Running KVS against the cache thrasher without isolation results in over a 100× increase in response latency and a bandwidth reduction of 86.5%. The increase in latency is the result of multiple factors. First, the vast majority of memory accesses result in an L2 miss and severely impact writes into the Cuckoo hash which can require many memory accesses when collisions occur and hash values are pushed to different locations. These delays cause both queuing and packet loss resulting in poor latency and throughput. With cache striping turned on response latency increases by only 50% on average, which is appropriate given its resource allocation: While running alone without FairNIC isolation KVS has free access to the entire L2 cache. In the isolated case KVS is only allocated half the L2 cache space (in proportion to its core count).

6.4.3 Coprocessor rate-limiting. We demonstrate the effectiveness of our distributed coprocessor-rate limiting using the ZIP coprocessor. We extend our OVS implementation to support IP compression [45] by implementing `compress` and `decompress` actions. We run two instances of OVS: a benign OVS 1 on eight cores and an antagonistic OVS 2 on seven cores as in Section 6.4.1. OVS 1 is configured with flow rules that compresses all incoming packets, while OVS 2 is artificially modified to compresses 10× the data in each packet to emulate a compression-intensive co-tenant.

We plot throughput as a function of packet size for three different isolation configurations in Figure 12. To provide a baseline, OVS 1 *alone* shows the throughput of OVS 1 in the absence of OVS 2 (but still restricted to eight cores). The *non-isolated* lines show the performance of both OVS instances when cohabitating with

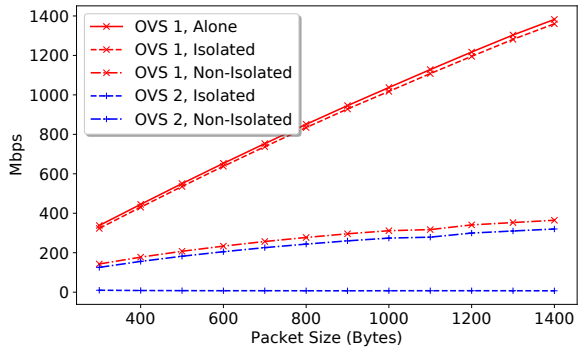

Figure 12: Throughput of two OVS instances using the ZIP coprocessor with and without rate-limiting.

FairNIC's traffic, core, and heap isolation enabled. Without coprocessor rate-limiting, each instance issues eight parallel requests to the coprocessor, and OVS 2's large requests restrict overall coprocessor throughput, bringing the performance of OVS 1 down with it. Coprocessor rate-limiting is enabled for the *isolated* runs, which restore the original OVS 1 performance while limiting OVS 2 to its fair share (half) of the coprocessor's throughput, or approximately (coprocessor performance is not quite linear; we leave finetuning to future work) one-tenth of the performance obtained by OVS 1.

7 DISCUSSION

In this section we address the practicality of FairNIC by outlining the challenges in selecting—and implementing—appropriate fairness and security policies for a deployable service model. We also consider the relevance of the challenges we address, and our proposed solutions, to other flavors of SmartNIC hardware.

7.1 Fairness policies

Various definitions of fairness have been proposed in the literature for multi-resource settings like SmartNICs. These include per-resource fairness (PRF) that extends traditional fair queuing mechanisms to every resource separately, bottleneck fairness [13] that implements fairness based on sharing the bottlenecked resource, and, more recently, dominant resource fairness (DRF) [20] that compares and shares resources based on proportional usage across different flows.

Robustness. While all these policies are "fair" in their own sense, only DRF is strategy proof. At first glance, FairNIC's fairness model may seem like PRF but, strictly speaking, it is not. While PRF traditionally refers to work-conserving fair queuing implemented independently at every resource, some of FairNIC's resources are allocated statically (i.e., in a non-work conserving fashion) as we favor low-overhead/low-latency mechanisms. That said, static allocations are strategy proof by definition as flows cannot do anything (such as inflating their demands) to change their allocation at run time. It is true, however, that the subset of resources with work-conserving schedulers in FairNIC may benefit from strategy-proof fairness models like DRF.

Complexity. DRF, in at least its current implementation [20], requires centralized scheduling based on usage information gathered

from all the resources involved. Apart from the non-trivial challenges involved in collecting accurate resource usage in real time (which may be approximated with help of models), a centralized scheduler implementation would require expensive inter-core communication for scheduling every packet. Recall that our primary motivation for employing distributed token rate-limiters for accelerators instead of fine-grained implementations like DRR was precisely to avoid this cost. Furthermore, a complex centralized scheduler like DRF may be less amenable to hardware implementation. For example, FairNIC already makes use of a hardware packet scheduler to fairly share egress bandwidth and readily allows different accelerators to implement in-house fair queuing without any dependencies on other resources.

In general, the appropriate policy should be guided by what fairness means from an end-user and business perspective. A fairness policy that adapts usage at all resources to fairly allocate one aggregate metric like end-to-end throughput or latency, while not strategy proof, might be more suitable for inclusion in a business/SLO model. Moreover, implementing a sophisticated fairness model like DRF on the SmartNIC that services traffic that likely subsequently experience some sort of packet scheduling in the network may be superfluous. Fairness needs to be considered in context, relative to its end-to-end impact on tenant applications. Our initial prototype employs a per-resource model as it provides the flexibility to make individual implementation choices that are low-overhead and fit naturally with the NIC's overall design.

7.2 Security isolation

SmartNIC applications propose a potential security threat to co-tenants, themselves, and the host in which the SmartNIC resides. This section highlights four general problems of SmartNIC security. Where appropriate, risks are related to the service model adopted (i.e., SaaS, PaaS, and IaaS; see Section 2.1). We end with a path forward for ensuring security under SmartNIC multi-tenancy.

Shared address space. FairNIC's programming model requires tenant applications to respect address-space bounds (i.e., not to inspect or write to addresses allocated to other applications). Datacenters providing SaaS can attempt to ensure address-space safety with rigorous testing and review. However, a wide range of memory attacks have been demonstrated against even highly tested software [50]. In PaaS, code provided by tenants can be inspected to check for address-space safety using static and dynamic analysis [21]. However, code obfuscation, imperfect analysis, and unintentional bugs make ensuring this property is difficult. IaaS only exasperates the issues of protection, as SmartNIC resources typically have fewer abstractions than their on-host counterparts. Hardware and software (compiler) mechanisms for enforcing address-space safety would likely correct many of these issues. For example, hardware rings of protection could be leveraged to implement some kind of hypervisor for tenant SmartNIC applications.

Shared hardware. Due to limited resources, applications are often forced to share hardware, creating numerous architecture-dependent security concerns. For example, in the case of Cavium CN2360, coprocessor memory is allocated by a shared free pool allocator (FPA). Thus, a buffer used to ZIP a file can be returned to the FPA by one application, and then allocated to another, while the buffer still contains private information. This poses the highest risk to PaaS services as the buffers may be intentionally read to violate the privacy of co-tenants, but also in SaaS where programming bugs can lead to information leaks. It is trivial to zero buffers in hardware or software; however, there are likely many such security problems spanning a number of different SmartNIC models. Further work is needed to understand the extent of this problem.

End-host protection. In the PaaS and IaaS models, the end-host/SmartNIC interface is another challenge. The concerns are again too numerous to state, so we provide an example. A SmartNIC has access to the IOMMU and the ability to DMA into the address space of any registered VM through SR-IOV virtual functions. In PaaS and IaaS, VFs should be guarded using capabilities to protect their access from unprivileged applications. In general, NICs have unfettered access to the PCIe bus. A first step towards host-SmartNIC isolation would be the protection of access to all SmartNIC I/O controllers, however, this is likely but one of many open problems relating to the end-host/SmartNIC interface.

Side channels. In both SaaS and PaaS, application code may contain privacy-violating side channels. Although enumerating all side channels is an open problem, existing channels may be mitigated by diligent code review and software/hardware patches. Current side channel research will need to be expanded to study SmartNIC virtualization platforms, should these platforms be developed. The well-acquainted reader will appreciate performance isolation mechanisms such as FairNIC can help address some side channels. FairNIC's static cache partitioning, in particular, has been shown to prevent a class of timing side channels [41].

We do not claim to know what combination of hardware and software security mechanisms would be optimal to support SmartNIC multi-tenancy. The first step toward address space, end-host, and shared hardware security is likely a comprehensive model for enforcing protection domains. These protections will require careful design to address performance requirements and will need to account for security challenges unique to SmartNICs. Protections must also be broad, including infrastructure for quickly deploying patches. Finally, we note that many SmartNICs—including the Cavium CN2360—do have mechanisms for enforcing rings of protection. However, it is unclear whether these mechanisms are acceptably performant and secure along the dimensions noted above. We leave such a study to future work.

7.3 Generality of FairNIC

FairNIC's isolation mechanisms are implemented on the specific architecture of Cavium LiquidIO CN2360. However, there are other (SoC) SmartNICs from various vendors [4, 38] that exhibit some architectural differences from the Cavium's. At a high level, all these SmartNICs have processor cores and a memory subsystem to support programmability, coprocessors and traffic management hardware; as such, they face similar isolation challenges.

In general, SoC SmartNICs can be categorized as either *on-path* or *off-path*, with either *wimpy* or *beefy* cores [35]. Our Cavium cards employ a large number of wimpy cores allowing for high levels of packet processing parallelism, and perform on-path processing

which requires every packet sent or received to be delivered to a core. Other cards like BlueField [38] and Stingray [4], in contrast, employ fewer but more powerful (ARM) cores with much higher clock speeds. They also perform off-path processing where packets first go through a packet switch on the card. Packets with headers which require NIC processing are matched against forwarding rules before routing to one of the NIC cores. Those that are not matched are directly routed across PCIe to the end host CPU. Other components like the memory subsystem and the coprocessors do not show any significant architectural differences.

Traffic scheduling. For tenant traffic isolation, FairNIC performs ingress buffer separation to avoid head-of-line blocking and fair queuing on egress to account for different packet sizes. On Cavium NICs, we achieve this with the support of dedicated (programmable) ingress and egress hardware that sits before and after the cores on the data path. In off-path designs, the on-chip switch encapsulates both ingress and egress, and can be programmed to achieve similar functionality in hardware. A key difference though is that since host traffic can bypass the NIC cores in off-path designs, packets coming from both host and NIC cores must be aggregated for each tenant prior to applying any egress fair queuing among the tenants.

Core/Cache isolation. Figure 10 demonstrates performance interference in the absence of core isolation, which FairNIC achieves using static core partitioning. However, this may not be suitable for NICs with few, powerful cores where traditional time-shared process schedulers (Linux/DPDK) may be preferable. These schedulers represent a trade-off as they use the processor efficiently but incur the processing overhead associated with scheduling and the latency cost of context switches. Prior work has explored the CPU efficiency and latency trade-offs of statically partitioning and dynamically spreading network traffic on beefy cores [37]. Furthermore, cache striping for memory isolation may be unnecessary on SmartNICs with more sophisticated CPUs that support hardware-based cache isolation techniques such as cache tagging.

Coprocessor isolation. FairNIC's rate-limiting approach to coprocessor isolation is designed to be architecturally independent as it models coprocessors based solely on their throughput characteristics. While SmartNICs differ in the set of coprocessors they support, each of these coprocessors can employ a rate-limiter customized to its performance properties for isolating its tenant workloads. Moreover, rate-limiting can be turned on or off separately for each coprocessor, allowing for hardware designs where a subset of coprocessors employ their own scheduling in hardware while the rest of them continue to use rate-limiting.

8 RELATED WORK

Cloud datacenter virtualization stacks have increasingly focused on network performance isolation; the work is far too voluminous to catalog here. We observe, however, that some operators—like Google—seem focused on software mechanisms. PicNIC [30] utilizes user-specified service-level agreements as criteria for sharing, and CPU-enforced fair queuing for rate limiting. Others, like Amazon and Microsoft, implement network virtualization functionality in custom SmartNIC hardware [2, 16]. In all cases, however, they do not currently address performance isolation of programmable NIC resources found in SoC SmartNICs.

Paper	Program	Hardware
Approx FairQ [46]	Flow monitor	Switch
KV-Direct [32]	KV store	FPGA NIC
Floem [42]	Top-N ranker	SoC NIC
ClickNP [33]	Rate limiter	FPGA NIC
ClickNP [33]	Firewall	FPGA NIC
AccelNet [16]	SDN stack	FPGA NIC
NBA [29]	Router	GPU
E3 [36]	Microservices	SoC NIC
λ-NIC [11]	Microservices	ASIC NIC
iPipe [35]	KV store	SoC NIC
iPipe [35]	Lock server	SoC NIC
iPipe [35]	Analytics	SoC NIC
NetChain [25]	Chain replication	Switch
NetCache [26]	KV store	Switch

Table 3: Some hardware-accelerated projects

Recent research has shown significant benefits from offloading certain functions from host CPUs to more targeted hardware. Table 3 showcases the variety of these efforts. Most applications are purpose-built for the particular platform under consideration, and are not amenable to use in a multi-tenant environment. Moreover, developers require intimate knowledge of the hardware [16].

In contrast, several recent efforts have focused on developing programmer-friendly frameworks to facilitate offloading general-purpose applications to SmartNICs. The authors of Floem [42] note that state migration between host and SmartNIC is difficult for developers to reason about and provide an automatic framework for state migration. Ipipe [35] authors further point out that reasoning about performance is difficult, and propose automatic scheduling and migration of tasks between host and NIC based on runtime performance. Uno [31] identifies that chains of network functions could behave poorly due to repeated PCIe crossings and provided automatic support for network function placement. It would be interesting to explore implementing FairNIC's isolation mechanisms in the context of one of them.

9 CONCLUSION

We take a first step towards enabling SoC SmartNIC use in multi-tenant cloud environments. We identify key points of performance contention such as packet ingress and egress, core assignment, memory access, and coprocessor usage, and implement low-cost isolation mechanisms. We show the effectiveness of our Cavium prototype for two representative cloud applications. FairNIC is limited to performance isolation; security is beyond our scope. Yet, our results suggest it may indeed be possible to maintain the performance benefits of SmartNICs in a multi-tenant cloud environment. Hence, complete SmartNIC virtualization remains an important topic for future study; our work does not raise any ethical issues.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation (CNS-1564185) and the Advanced Research Projects Agency-Energy. We are indebted to Shay Gal-On, Weishan Sun, Ugendreshwar Kudupudi, Jim Ballingal, and others at Cavium/Marvell for their generous support and assistance, and to Geoff Voelker, the anonymous reviewers, and our shepherd Anirudh Sivaraman for feedback on earlier drafts of this manuscript. Ming Liu and Dave Andersen provided extensive help with their respective codebases.

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Pate, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM.*

[2] Amazon. 2020. AWS Nitro System. https://aws.amazon.com/ec2/nitro/. (2020).

[3] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proc. USENIX OSDI.*

[4] Broadcom. 2019. Broadcom Stringray SmartNIC. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225. (2019).

[5] Josiah L. Carlson. 2013. *Redis in Action.* Manning Publications Co., USA.

[6] Cavium. 2017. Liquid IO II 10/25G Smart NIC Family. (2017).

[7] Cavium. 2017. LiquidIO OVS Software Architecture. (Dec. 2017). https://www.marvell.com/documents/ocwqbcxlc2ir4o7n16rn/.

[8] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. 2005. Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[9] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. 1996. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[10] Dah-Ming Chiu and Raj Jain. 1989. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Journal of Computer Networks and ISDN Systems* 17, 1 (June 1989).

[11] Sam Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2019. λ-NIC: Interactive Serverless Compute on Programmable SmartNICs. (Sept. 2019). http://arxiv.org/abs/1909.11958v1.

[12] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. 2010. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 1–10.

[13] Norbert Egi, Adam Greenhalgh, Mark Handley, Gianluca Iannaccone, Maziar Manesh, Laurent Mathy, and Sylvia Ratnasamy. 2009. Improved Forwarding Architecture and Resource Management for Multi-Core Software Routers. *Network and Parallel Computing Workshops, IFIP International Conference on* 0, 117–124.

[14] 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).* USENIX Association, Renton, WA, 345–362.

[15] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI).* Lombard, IL, 371–384.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[17] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5.

[18] Linux Foundation. 2015. Data Plane Development Kit (DPDK). (2015). http://www.dpdk.org

[19] Linux Foundation. 2020. Open vSwitch. https://www.openvswitch.org/. (2020). Accessed: 2020-01-31.

[20] 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 (SIGCOMM '12).* Association for Computing Machinery, New York, NY, USA, 1–12.

[21] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 575–589.

[22] Irfan Habib. 2008. Virtualization with KVM. *Linux J.* 2008, 166, Article Article 8 (Feb. 2008), 1 pages.

[23] Intel. 2019. Netronome-Agilio-SmartNICs. (2019). Accessed: 2019-03-22.

[24] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13).* USENIX, Lombard, IL, 297–311.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *Proc. USENIX NSDI.*

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proc. ACM SOSP.*

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC).*

[28] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[29] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proc. ACM EuroSys.*

[30] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. 2019. PicNIC: Predictable Virtualized NIC. In *Proc. ACM SIGCOMM.*

[31] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T. V. Lakshman. 2017. UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing. In *Proc. ACM SoCC.*

[32] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proc. ACM SOSP.*

[33] Bojie Li, Kun Tan, Larry Luo, Renqian Luo, Yanqing Peng, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. 2016. ClickNP: Highly Flexible and High-performance Network Processing with Reconfigurable Hardware. In *Proceedings of the ACM SIGCOMM Conference.*

[34] Jiuxing Liu, Amith Mamidala, Abhinav Vishnn, and Dhabaleswar K. Panda. 2004. Performance evaluation of InfiniBand with PCI Express. In *Proceedings of Symposium on High Performance Interconnects.*

[35] 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 Conference.*

[36] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-efficient Microservices on SmartNIC-accelerated Servers. In *Proceedings of the USENIX Annual Technical Conference.*

[37] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19).* Association for Computing Machinery, New York, NY, USA, 399–413.

[38] Mellanox. 2018. Mellanox BuleField SmartNIC. (Dec. 2018). http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic.

[39] Soo-Jin Moon, Vyas Sekar, and Michael K. Reiter. 2015. Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[40] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the ACM SIGCOMM Conference.*

[41] D Page. 2005. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. (2005).

[42] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI).*

[43] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. 2007. Cloud Control with Distributed Rate Limiting. In *Proceedings of the ACM SIGCOMM Conference.* Kyoto, Japan. Best student paper.

[44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the ACM Conference on Computer and Communications Security.* Chicago, IL. Test of Time Award.

[45] Abraham Shacham, Bob Monsour, Roy Pereira, and Matt Thomas. 2001. *IP Payload Compression Protocol (IPComp).* RFC 3173. Internet Engineering Task Force.

[46] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *Proc. USENIX NSDI.*

[47] M. Shreedhar and G. Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking* 4, 3 (June 1996), 375–385.

[48] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).* USENIX Association, Boston, MA, 33–46.

[49] Brent Stephens, Aditya Akella, and Michael M. Swift. 2018. Your Programmable NIC Should Be a Programmable Switch. In *Proc. ACM HotNets.*

[50] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 48–62.