Yama: Providing Performance Isolation for Black-Box Offloads

Tao Ji
UT Austin

Divyanshu Saxena
UT Austin

Brent E. Stephens
University of Utah

Aditya Akella
UT Austin

ABSTRACT

The sharing of clusters with various on-NIC offloads by high-level entities (users, containers, etc.) has become increasingly common. Performance isolation across these entities is desired because the offloads can become bottlenecks due to the limited capacity of hardware. However, the existing works that provide scheduling and resource management to NIC offloads all require customization of the NIC or offloads, while commodity off-the-shelf NICs and offloads with proprietary implementation have been widely deployed in datacenters. This paper presents Yama, the first solution to enable per-entity isolation in the sharing of such black-box NIC offloads. Yama provides a generic framework that captures a common abstraction to the operation of most offloads, which allows operators to incorporate existing offloads. The framework proactively probes for the performance of the offloads with auxiliary workload and enforces isolation at the initiator side. Yama also accommodates chained offloads. Our evaluation shows that 1) Yama achieves per-entity max-min fairness for various types of offloads and in complicated offload chaining scenarios; 2) Yama quickly converges to changes in equilibrium and 3) Yama adds negligible overhead to application workload.

CCS CONCEPTS

• **Networks** → **Cloud computing**; **In-network processing**; **Programmable networks**.

KEYWORDS

In-network computing, offload, performance isolation, black-box approach

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624792

ACM Reference Format:
Tao Ji, Divyanshu Saxena, Brent E. Stephens, and Aditya Akella. 2023. Yama: Providing Performance Isolation for Black-Box Offloads. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620678.3624792

1 INTRODUCTION

NIC offloading has emerged as an important approach to continue scaling up application performance with increasing network line-rates in a post-Denard era [37, 69]. For example, RDMA can provide ultra-low latency, high bandwidth, and high message rate and these properties have enabled novel datacenter application designs that leverage RDMA to achieve higher performance than using traditional network stacks [7, 15, 22, 23, 25, 28, 40, 44, 67]. Further, other application-specific offloads have been used to accelerate important applications like ML [62], network transport [2, 46, 59], load balancing [19], key-value stores and caches [7, 23, 24, 38, 67], and distributed transactions [25, 67]. In general, NIC offloads can reduce latency, increase throughput, and save CPU cycles by performing some application computation with custom in-network processors.

However, in datacenter environments, many different entities share the network [63]. For example, there are typically many different containers and applications sharing the same infrastructure, and each back-end service is typically shared by a large number of front-end applications [3]. When different applications compete without explicit isolation policies, the performance received by an entity is an ad-hoc side-effect of the workload and how individual devices process work. Typically, most devices default to policies like first-come, first-served (FCFS), first-in, first-out (FIFO), and per-queue/per-flow fairness. As a result, it is often the case that the entity that requests for the most service gets it at the detriment of other entities. This leads to priority inversions where entities performing unimportant background work can starve important applications and significantly increase the application-level latency of entities performing important online work [73].

Unfortunately, many important NIC offloads are *black boxes* that do not provide intrinsic support for entity-level sharing, and we believe that this will continue to hold true in the future. For example, RDMA is a widely deployed NIC offload that only supports eight different priorities through DCB and otherwise defaults to per-queue pair fairness [16, 73]. The offloads used by Microsoft for inference also do not provide primitives for perfromance isolation [62]. As such, without outside intervention, priority inversions are expected anytime one of today's offloads is shared across more than eight entities.

Ideally, it would be possible to layer support for multi-entity isolation on top of existing on-NIC offloads so that they can be safely shared in datacenter environments. However, in practice, there are many challenges that arise with providing isolation support for black-box offloads. In addition to a lack of support for isolation, offloads may be one-sided and have variable throughput [16, 23, 24, 29], and be arranged in heterogeneous chains [37]. As a result, static approaches like rate-limiting are not appropriate because they lead to either under- or over-utilization. Similarly, other approaches that use TCP-like feedback loops based on signals like packet loss and delay [5, 18, 31] can take too long to converge and cause underutilization, especially for short-lived workloads and suffer from congestion from microbursts.

To address this multi-entity isolation problem, this paper introduces Yama, a new system that allows end hosts to layer multi-entity isolation on top of a network of offloads. Yama supports arbitrary black-box offloads and can provide isolation regardless of how the workload and usage pattern impacts the throughput of the offload. Yama supports general policies that are aligned with the existing resource allocation policies that are used in datacenters like providing weights that are equal to the proportion of CPUs allocated to an entity. Yama is also efficient in that it provides consistent high throughput while ensuring that offloads are not overloaded, with minimum overheads.

To provide these properties, Yama uses the following components: a generic queue pair interface, a new approach to throughput probing and rate pacing, and explicit throughput sharing through a coordinating overlay network. The generic queue pair model provides a common abstraction that can be used to pace work requests to an offload, *e.g.*, packets, RDMA verbs, KVS GETs/SETs, and more. By transparently backfilling dummy work on application workload, Yama dynamically measures the current performance of an offload. The overlay network improves the efficiency of Yama by allowing a cluster of hosts to converge to appropriate rates and to minimize the potential overhead imposed by backfilling.

Evaluation shows that:

(1) Yama achieves per-entity weighted sharing of throughput with all the representatives of network-, transport- and application-layer offloads that process different types of workload.
(2) Yama achieves per-entity weighted sharing of throughput in complicated cases when entities share offload chains with common offloads, which may or may not be the performance bottlneck.
(3) Yama converges to equilibrium changes much quicker than state-of-the-art software-based congestion control designed for low-latency RDMA.
(4) Yama incurs negligible latency to individual pieces (packets/operations/RPCs) of application workload.

In summary, this paper makes the following main contributions:

(1) Identifying the lack of entity-level isolation in network offloads and chains, as well as the black-box nature and other challenges in achieving entity-level isloation for existing offloads.
(2) Introducing a generic, end host-based approach to overload feedback and control for heterogeneous black-box offloads that converges in short time scales with mininmum application-observed performance costs.
(3) Implementing and evaluating a prototype of Yama on real testbeds, showing it is effective, fast-converging and efficient in achieving multi-entity performance isolation for invidividual and chained heterogeneous offloads.

2 MOTIVATION

Offloads implemented on SmartNICs provide an important avenue to continue scaling up the application performance in the post-Denard era. Unfortunately, many SmartNIC offloads provide poor per-entity isolation, where an entity in this paper refers to the finest granularity that an operator wants to provide isolation at, *i.e.*, the leaves in the isolation policy graph. Typically, we assume that this is an application/container. In this section, we discuss state of the art in the use of SmartNIC-based offloads in datacenters and the cloud. Then, this section discusses the per-entity isolation problem and sheds light on the major challenges that arise in tackling the problem.

2.1 Offloading in datacenters

NIC offloading ("offloading" in the rest of this paper) is a technique that delegates a part of a host's CPU computation onto the host's network interface card (NIC). Offloading can help overcome the widening gap between network bandwidth and the CPU's processing performance [27, 37, 62]. While CPU performance has stagnated since the end of Denard scaling [63, 69], offloads can continue to increase performance by leveraging their location in the network to reduce communication latency and overheads and by leveraging custom hardware to perform application-specific computation, improving efficiency and saving CPU cycles.

The space of offloads and applications that benefit from offloading is large – there are many recent systems that have shown that NIC offloading can be used to accelerate a variety of different applications [2, 7, 8, 11, 16, 19, 23, 25, 27, 33, 35, 38, 39, 42–44, 46, 51, 52, 54, 57, 59, 62, 64, 66]. For example, this includes offloads that range from simple network- and transport-layer functions such as NAT (network address translation) [11], cryptography [37], and TCP segmentation/generic receive [27, 66], to entire transport engines such as TCP [2, 46, 59] and RDMA [16, 75]. In addition, some application-level functionality can also leverage NIC offloading both for improving latency/throughput and saving CPU cycles. Examples of such higher-level offloads include caching for key-value stores [27, 38, 51] and distributed transactions [25, 52]. Additionally, offloading is particularly appealing for functionality that involves per-packet or per-message processing – the NIC is a natural and ideal location to take over some or all of such processing.

2.2 Multi-entity isolation

An important requirement in datacenters is the ability to effectively multiplex the underlying infrastructure – which now includes accelerators that can be offloaded – among multiple entities, where an entity can be an application, a user, a tenant, etc. As entities share offloads in interesting ways, it becomes important for datacenter operators to systematically support cross-entity performance isolation.

Specifically, in a datacenter, it is often the case that multiple entities may be simultaneously utilizing the same resource. In this scenario, a system that provides performance isolation will ensure that an entity that attempts to request for more of the resource cannot cause the performance experienced by another entity to deteriorate. However, we argue that it is also important to provide performance isolation such that a datacenter operator specifies a policy that determines the appropriate fair-share level of service for each entity, and each entity should be able to receive their share of service regardless of the behavior of the other entities. For example, an operator might wish to use the relative proportion of CPUs allocated per entity to define the relative share of throughput of a bottlenecked offload that each entity should receive.

Unfortunately, most offloads suffer from performance isolation problems. Because existing offloads lack intrinsic support for entity-level allocation and isolation, priority inversions can occur, where a priority inversion occurs any time one entity achieves a higher level of service than its fair share when multiple entities are bottlenecked at the same offload.

We demonstrate this by running two experiments - first using a key-value cache and then, an RDMA offload. For both experiments, we run the offload on one of the servers in our cluster, while on other servers, we set up two entities to run

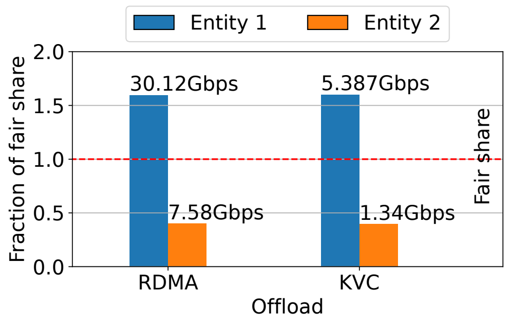


Figure 1: Mismatch between achieved throughput and assigned weight.

the workload generator for the offloads. Further, we allocate an equal amount of CPU cores to the two entities, but Entity 1 uses 4 IO queues per core while Entity 2 only uses one IO queue per core. The cluster topology, offload, and workload generator are the same as in Section 6. Ideally, the operator would like to have both entities achieve equal throughput in accordance with their CPU allocation. However, as shown in Figure 1, Entity 1 achieves 4x throughput of Entity 2, which merely achieves lower than 0.5x of its fair share of throughput. This is fundamentally due to the fact that it is much cheaper for the CPU to submit workload to offloads compared to performing all the computing.

The performance isolation problem is further exacerbated by the fact that many NIC offloads do not run at the line rate due to limited capacity of the underlying hardware. This makes offloads more susceptible to becoming bottlenecks. For instance, a well-known problem with even the state-of-the-art RDMA offload is that the NIC's on-board memory cannot cache all the connection and memory access state and has to frequently access the host's main memory, which causes degradation of operation rates [23, 24, 48]. Other below-line rate offload examples include hardware implementations of cryptography (AES-256), authentication (SHA-3), and compression (LZRW1) [37]. Therefore, datacenters need a novel design to provide entity-level performance isolation in terms of service level for bottlenecked offloads.

2.3 Challenges

Key challenges arise in systematically supporting per-entity isolation in the presence of NIC offloads in production datacenters.

Black-box Offloads: While offloads offer significant benefits, a key challenge is their *black-box* nature. In many production datacenter networks, the on-NIC hardware units that implement offloads (and in many cases, the NICs themselves) are sourced from third-party vendors. In such cases, the offload's implementation and internal details are hidden from the data center operators. This means that, in practice, it is not possible to modify these offloads to provide support for multi-entity isolation. Although some future offloads may provide mechanisms for multi-entity isolation like programmable scheduling [61, 65], it is unlikely that all offloads will implement these mechanisms, and current programmable scheduling designs have scalability limitations [65]. Further, there is the practical issue that there exist offloads like RDMA that are already widely deployed but do not provide sufficient mechanisms for isolation [16], so there is a need for a system that can be leveraged to layer isolation mechanisms on top of already deployed offloads. Additionally, local scheduling isn't sufficient to ensure isolation in chains of offloads [45] (more on chaining below).

Variable Throughput: Another challenge is that the behavior of many offloads is variable and workload dependent [16, 29, 37]. For example, RDMA NICs utilize RDMA engines and caches and access main memory and any of these components may run at less than line-rate [16, 24, 29]. As a result, even when the network is not congested, there may be periods where an offload and not the network line rate is the performance bottleneck, *e.g.*, duing a large RDMA incast [23, 24, 48]. Further, this implies that static solutions to allocate the bandwidth [53, 60] are inapplicable to the problem because they will lead to over- and under-utilization as the throughput of the offload being shared varies.

One-Sided Offloads: Offloads can work at different layers of the network stack and can have different usage patterns. Some offloads do not forward part of the workload to the host CPU. A key-value cache offload, for instance, can directly generate the response for a request that hits. So is the case with RDMA, where `read` and `write` operations to remote memory do not involve the remote host's CPU. Unfortunately, due to the black-box nature of offloads, any isolation mechanism must reside on the host CPU. As a result, approaches that rely on the cooperation between the sender/client (referred to as *initiator* for the rest of this paper) and receiver/server (*target*) for bottleneck performance measurement will not work for one-sided offloads. Examples of such approaches include BBR [5] and Swift [31], which rely on the target to reflect their measurements.

Heterogeneity and Chaining: Moreover, each type of workload can use a distinct chain of multiple offloads and different chains can share offloads at a machine (node) in common [37]. As Figure 3 illustrates, multiple offloads for different functions like cryptography, transport, NAT and caching can be composed together, depending on the application's needs. This further exacerbates the difficulty in devising an isolation solution because the location of the bottleneck in an offload chain can shift when the workload changes, and two chains may or may not share a common bottleneck from time to time. Therefore, any solution that is designed for a fixed set of offloads will not work.

2.4 Limitations of Existing Approaches

As black-box NIC offloads disallow instrumentation, canonical overload and congestion detection approaches that elicit direct feedback from the overloaded location (and modulate workload based on the feedback) are not applicable. This includes using ECN marking [1, 75] for links and credits/tokens for microservices [6, 71, 74]. Likewise, enforcing performance isolation at an offload e.g., using a custom scheduling algorithm such as priority or weighted fair queueing [14, 37] is also not possible.

Mechanisms that rely on control from a remote end or a global scheduler to schedule all of the work for an offload (e.g., Fastpass [50]) incur too high of latency overheads for many applications because contacting the scheduler is required before a piece of work can be scheduled. Additionally, these approaches require the knowledge of the offload's current performance; and since many offloads can have variable throughput depending on the working state, and no existing remote scheduler can ascertain this information. Without knowing the offload's current performance, a remote scheduler can overload or underload it. Overloading causes high queueing and packet drops which hurt the tail latency, while underloading causes underutilization and unnecessary loss of throughput [6].

Most prior works on NIC offload isolation either focus on a restricted class of NICs or require custom NIC hardware that isn't available today. For example, FairNIC [14] is based on SoC SmartNICs and runs packet processing software including part of the isolation mechanism on the general-purpose cores, while PANIC [37] proposes a novel hardware design where the isolation support is tightly coupled with the offloads. Unfortunately, neither class of approaches can be applied to the ASIC-based SmartNICs already deployed in datacenters today. Similarly, QoS (quality of service) functions like the packet schedulers that come with some SmartNICs do not suffice, since they cannot schedule workload for a remote offload (e.g., scheduling work at a remote RDMA offload). Also, techniques like DCB (datacenter bridging) that can prioritize different flows [17] generally do not scale well [54, 55].

Finally, relying on end-to-end transport based congestion control to provide multi-tenant performance isolation also cannot work out of the box. These congestion control schemes converge to per-flow fairness and not an entity-level fair share of service policy defined by an operator, therefore can still lead to priority inversions. Recent approaches to congestion control that perform explicit bandwidth allocation like XCP [26], RCP [9], and PERC [21] are promising given the need to converge faster than AIMD algorithms. However, all of these systems assume that the network throughput is known and constant, so they lack the mechanism to dynamically determine the throughput of an offload.

3 OVERVIEW

Yama is a system that supports flexible sharing and isolation of chained black-box NIC offloads across high-level entities. Yama overcomes the challenges outlined in the previous section.

In what follows, we first lay the design requirements for Yama. We then present its key ideas that help meet the requirements. Finally, we present the key components of Yama and show their functioning using a simple end-to-end example.

3.1 Design Requirements

In designing Yama, the following requirements must be met:

- Yama must support arbitrary black-box offloads, oblivious to the specific type of workload or usage pattern, and without needing custom offload or NIC implementation.
- Yama should support cross-entity service level allocation policies that align with today's existing resource allocation policies across entities, e.g., those based on assigning mainstream datacenter resources such as CPU and IO.
- Yama should have low overhead on the latency and throughput of application workload, despite operating in a complete black-box setup.

3.2 Yama's Approaches

Yama meets these requirements based on the following three main ideas.

Generic queue pair model. To accommodate offloads that work on different layers with distinct usage patterns, we make a key observation: The interface that an offload or chain of offloads expose to the host CPU for IO operations can be abstracted to a generic "queue pair" model, consisting of a *work queue* and an *event queue*. The host CPU submits work to the offload (chain) by posting descriptors to the work queue, and the NIC posts events such as a completion to the event queue, which will then be polled by the CPU.

For example, an RDMA offload engine obtains work (e.g., a `read` or `write` operation) from the CPU from a send and a receive queue, respectively, and notifies the CPU by posting to a completion queue [41]. The send and receive queues can be mapped to the work queue in our model and the completion queue can be mapped to the event queue. Many offloads that work on individual packets, such as NAT, compression and cryptography, are compatible with DPDK [12]'s interface, which requires the CPU to enqueue packets to a send/submission queue for transmission or processing, and to poll from a receive/completion queue for received and offload-processed packets.

We leverage this insight and design Yama to be agnostic to offload-specific interfaces and workload by performing op scheduling (pacing) on the abstract workload in the work queue (detailed in Section 4.2).

Weighted offload throughput sharing. We design Yama to support weighted throughput sharing policies at a bottleneck offload. That is, for all *entities* that are contending at an offload whose throughput is saturated, Yama guarantees that the ratio of per-entity throughputs adheres to the entities' relative weights. We ensure that this also holds at the common bottleneck of more than one chain of offloads.

This policy meets our requirements in two ways. First, throughput is observable end-to-end, which makes it an ideal choice of metric under the black-box assumption, especially when offloads work on different granularities of data. Some offloads handle packets, such as NAT, while the others deal with multi-packet operations, such as RDMA. Second, operator can allocate throughput by simply specifying a weight for each entity in the same way they allocate resources like CPUs and memory today [70], and each entity can further allocate its share to its constituent sub-entities (e.g., a tenant can allocate amongst its constituent applications) by specifying per-node and per-queue pair weights.

Probing and rate pacing. Yama probes for the current maximum throughput of each offload chain by producing background dummy workload to saturate the chain, eschewing the need for explicit feedbacks from the NIC or offloads, and the cooperated measurement reflection by the target CPU . In particular, Yama carefully *backfills* existing application op issuance and/or raw packet transmissions among tenant endpoints, and combines observations of the probe with the throughput observed by ongoing flows to estimate the offload's maximum throughput. Yama's lightweight *rate pacing* approach then modulates the posting of workload across different entities such that the entities' achieved throughputs quickly converge to that determined by the policy and that the sum of the throughputs is equal to the estimated total throughput of the offload.

3.3 Performance Goals and Non-Goals

The primary goal of Yama is to provide work-conserving weighted throughput sharing of an offload across competing entities. Once fairness is provided, it is not Yama's goal to provide guarantees that latency under load is within some factor of the unloaded latency of the offload. For example, in RDMA, large-op flows can cause significant increases to small op latency [73]. However, for cases where latency under load is problematic, Yama provides several primitives like shaping that operators can use to provide isolation mechanisms for latency against adversarial workloads (Section 7).

3.4 libYama

We incorporate the ideas described in Section 3.2 into `libYama`, a generic user-space library (Figure 2). Applications link `libYama`, instead of offload-specific libraries such as `libibverbs` to access offloads. Specifically, applications access `libYama` with offload-specific adaptors (e.g. RDMA and Packet IO adaptors in Figure 2), which encapsulates the offload-specific workload descriptors posted by the application into generic work-queue workload descriptors, The generic work-queue workload descriptors contain the offload-specific library hardware-access context and function that the scheduler use when scheduling the workload. Datacenter operators can implement these adaptors in a way that interposes the original hardware library, so that the application can transparently link `libYama` at run time. This enables applications to seamlessly run on `libYama` without recompiling.

In addition to the adaptors, `libYama` has two main routines: throughput prober and scheduler. Both of these work on the generic queue pairs and are application- and offload-agnostic. The throughput prober routine in `libYama` runs feedback loops to solicit the current maximum throughput of a offload chain by saturating it with dummy workload. The workload scheduling routine paces the workload in the generic queue pairs towards the chain at a rate decided by the corresponding entities' weights (provided by the operator) and the probed throughput (Section 4.2). Finally, each `libYama` instance keeps local state and consists of a 'cluster communicator' to collaborate with other `libYama` instances in the datacenter (see Section 4.3).

We place the probing and scheduling routines in the user space because kernel bypassing is the most common way that vendors provide to access the NIC and offloads for its low overhead. We target containerized and VM-based environments where only trusted images are allowed, and tenants or users cannot circumvent the scheduling mechanism by providing their own runtime libraries to access the offloads. The assumption of trusted images is common and has been adopted by prior work too [73].

Workflow Example. We demonstrate the `libYama`'s basic workflow with and RDMA transport offload as an example. Suppose the application issues an RDMA one-sided operation (`ibv_send_wr` object) to the RDMA send queue implemented by the adaptor (① in the figure). The adaptor encapsulates it into a generic work queue descriptor (②). The encapsulation procedure also sets the offload-specific functions and context (e.g., the `ibv_qp` object) for submitting the workload descriptor to hardware. The work queue is then paced by the scheduler according to the throughput of the RDMA offload determined by the prober. When the scheduler dequeues the descriptor from the work queue (③), it calls the

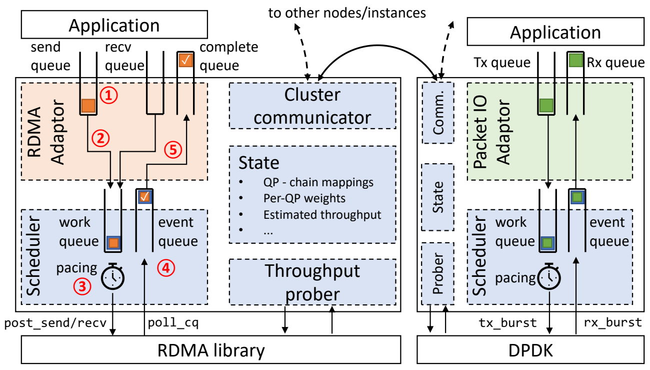


Figure 2: An overview of libYama.

submission routine that was set during encapsulation (in this case `ibv_post_send`).

To obtain the completion notification, `libYama` calls the offload-specific function (in this case `ibv_poll_cq`) provided by the adaptor, and places the polled object in an event queue descriptor (④), which is eventually decapsulated when the application polls from the RDMA completion queue of the adaptor (⑤). How `libYama` works with (DPDK-based) packet IO is largely similar, with the only difference being that the objects polled from the hardware are packets received from the network rather than completion notifications generated by the local NIC.

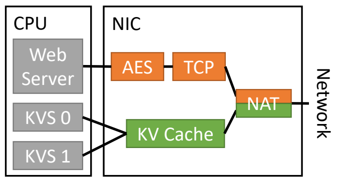
Note that `libYama` accepts per-queue pair weights, which enables an entity to adjust the relative levels of service for its constituent components on a node. `libYama` then schedules the workload in each work queue at the the probed throughput for the offload chain, discounted by the per-queue pair weight. We detail weight management and scheduling next in Section 4.

4 YAMA DESIGN

4.1 Policy

As mentioned above, Yama adopts the policy that allows weighted bottleneck throughput allocation to different entities. Yama must ensure that different entities, potentially with queue pairs across multiple nodes, achieve the throughput according to these weights. A complication to the policy is when entities use offload chains, and different chains can share a common bottleneck. In what follows, we demonstrate with an example what the policy would mean with multiple offload chains that might or might not share a common bottleneck, and shed light on how Yama manages the input weights so that they can be collectively enforced by `libYama` instances distributed across the cluster.

Example. Figure 3 shows a setup of a server running a web server and two key-value stores. The web server uses an offload chain (in orange) consisting of the NAT, TCP, and AES engines, while the key-value stores use a chain (in green)



Offload	Case 1 (Gbps)	Case 2 (Gbps)	Case 3 (Gbps)
NAT	25	25	25
TCP	10	10	20
AES	10	10	20
KV Cache	10	16	20

Figure 3: Example server and NIC offload chain setup (left) and different cases of bottleneck throughput (right).

Entity	Weight	Target Gbps in Case		
		1	2	3
0 (Web Server)	0.5	10	10	12.5
1 (KVS 0)	0.3	6	9	7.5
2 (KVS 1)	0.2	4	6	5

Table 1: Example weight assignment (normalized) and ideal throughput as per policy.

of the NAT and key-value cache engines. In this particular case, each server instance belongs to a distinct entity, whose weight is shown in Table 1. Weights are normalized to sum to 1 for simplicity. Note that the NAT engine is shared by both chains and all three entities.

We discuss three different cases in which the offloads' throughputs vary (listed on the right-hand side in Figure 3), resulting in different bottlenecks. In Case 1, both chains are bottlenecked by their own offloads at 10 Gbps, and not saturating the shared NAT engine, so the entities should achieve weighted throughput within their chains. So, the web server can use all 10 Gbps of the orange chain, and the key-value stores are allocated 6 Gbps and 4 Gbps of the green chain, respectively, according to their relative weights of 3:2. In Case 2, NAT can be saturated, and the web server should idealy share 12.5 Gbps according to its weight (0.5), but it is bottlenecked by its own chain at 10Gbps. As a result, the key-value stores should divide the rest of NAT's throughput according to their 3:2 weight ratio at 9 Gbps and 6 Gbps, respectively. In Case 3, where both chains are bottlenecked by NAT, the two chains equally divide the bottleneck throughput, and in this case allocation adheres to the weight ratio of all three entities.

Weight Management. To transform the high-level input, per-entity weights, into metadata based on which cluster-wide instances of `libYama` can collectively enforce the policy, Yama introduces a logical control plane that manages the update of weights. When an entity joins the cluster, the operator assigns it a weight that represents operator's desired level of service for the entity. This can be, for example, the hourly dollar rate that a tenants pays. The control plane then for each offload chain normalizes these weights into *per-entity weights* such that the per-entity weights of a chain sum to 1.

When an entity starts its first application on a node that uses a specific offload chain, the control plane provides a *per-node weight* that differentiates the chain's service levels for the entity initiating from different nodes. The per-entity weight is normalized such that weights of entities using the chain sum to 1. The per-node weight can be adjusted by the entity via the control plane, and an entity's per-node weights also sum up to 1. Note that contacting the control plane only for the first application is of negligible overhead, because starting an application itself can take orders of magnitude longer (e.g., > 600 ms to cold-start a container [56]) than the delay in querying a control plane (< 0.5 ms [50]).

Using the per-node weight, the entity can then create and destroy queue pairs without involving the control plane, which is friendly to short-lived applications: The weight of a queue pair is by default given by the product of the per-entity weight and offload chain's per-node weight, divided by the entity's number of queue pairs that use the same offload chain. The entity can also locally customize how to allocate the weight across these queue pairs . The scheduler eventually rate-limits the descriptors in each work queue by its per-queue pair weight multiplied by the offload chain's probed throughput (detailed next in Section 4.2).

When an entity's last application that uses an offload chain quits from a node, the `libYama` instance that executes the feedback loop and scheduling on the node notifies the control plane, which re-allocates the per-node weight to the entity's other nodes that use the same offload chain.

4.2 `libYama` Operation

Throughput Probing. With the per-queue pair weights in place, Yama needs an estimated throughput of the offload chain to calculate the rate to pace for each queue pair throughput to pace, and the offloads' black-box nature precludes any design that requires explicit feedback from the offload.

To that end, a strawman approach for Yama is to iteratively offer increasingly high throughput using a dummy background workload [1] through the chain, while monitoring the achieved throughput until the offered throughput cannot be achieved. The maximum throughput of the chain is then given by the sum of throughput of both the dummy and normal application workload. However, we observe that the dummy workload in this scheme embezzles the throughput of applications, violating the low overhead requirement.

To mitigate the impact of dummy workload, we devise a technique called backfilling. The feedback loop is described in Algorithm 1. Instead of blindly generating a dummy workload to saturate the chain, `libYama` only produces in each

[1]We assume that the method for generating dummy workload to a specific offload chain is provided by the operator who implements the corresponding adaptor and understands the performance characteristics of the deployed offloads.

iteration enough dummy workload to fill the throughput difference between the chain's estimated current maximum and the application throughput (Line 8-10). To adapt to increases in the bottleneck throughput, Yama adopts controlled oversubscription. The feedback loop provides a throughput estimation greater than the achieved value by a factor β (Line 6-7), so that `libYama` rate-paces application workload and generates dummy workload at higher throughput. In the next iteration, estimated maximum throughput is updated according to the actually achieved throughput by the dummy and application workloads combined (Line 4-6). Note that the probed throughput can be limited by applications that do not have sufficient workload to saturate the previous $\hat{\lambda}$. Therefore the feedback loop caps the dummy workload to a fraction of the application workload to avoid waste of resources and does not update the throughput estimation (Line 6-7) in such cases.

However, two challenges arise when multiple initiators need to access the same offload chain. First, the throughput of all applications using the chain must be available to the feedback loop. Second, the updated throughput estimation (produced at Line 6) must be made available to all the initiators for `libYama` to schedule workload for this chain. To tackle these challenges, we design a coordinating overlay network among the `libYama` instances, driven by the "cluster communicator" module (Figure 2) in `libYama` to exchange such informaion. We discuss this in detail in Section 4.3.

Scheduling. On each node there is a scheduler routine that is responsible for multiplexing the workload from all queue pairs towards an offload chain. The scheduler calculates the per-queue pair rate by multiplying the per-queue pair weight and the probed maximum throughput of the chain. For each queue pair, the scheduler maintains a token bucket and fills the bucket at the per-queue pair rate, up to a limit, which represents the maximum burst allowed for a queue pair. For queue pairs that do not have sufficient workload and the token bucket is full, the scheduler allocates the surplus tokens to other queue pairs towards the same offload chain for work conservation.

Execution. To execute the scheduling and probing routines, as well as the state exchanges between `libYama` instances, one approach is to allocate dedicated CPU cores on each node. But this is undesirable as it reduces the resources available to production applications. Yama instead uses the idea of cycle scavenging, where applications use their own CPU cycles to execute `libYama`'s routines: When the application polls I/O queues of the `libYama` adaptor , it executes a routine that not only polls from the hardware but also drives the scheduling routine for all other queue pairs towards the same , the feedback loop for some chains, and the communication with other `libYama` instances (detailed in Section 4.3).

Algorithm 1: Yama's Feedback Loop

```
1  while true do
       /* Update timestamps since last iter */
2      $T_{\text{last}} \leftarrow T_{\text{now}}$
3      $T_{\text{now}} \leftarrow getTime()$
       /* Count the amount of app and dummy work
          completed since last iteration */
4      $B_A^{poll} \leftarrow getAppWorkCompletedSince(T_{\text{last}})$
5      $B_D^{poll} \leftarrow pollDummyWorkCompletedSince(T_{\text{last}})$
       /* Calculate estimated maximum throughput
          with a simple sliding window */
6      $\hat{\lambda} \leftarrow slidingWindowSmooth(B_D^{poll} + B_A^{poll}, T_{\text{now}})$
       /* Oversubscription by a factor of $\beta$ */
7      $\hat{\lambda} \leftarrow (1+\beta)\hat{\lambda}$
       /* Calculate the amount of dummy work to
          backfill */
8      $B_A^{post} \leftarrow getAppWorkPostedSince(T_{\text{last}})$
9      $B_D^{post} \leftarrow min(\hat{\lambda}(T_{\text{now}} - T_{\text{last}}) - B_A^{post}, 0)$
10     $generateAndPostDummyWork(B_D^{post})$
11 end
```

This is a viable approach because, on the one hand, we observe that applications generally spend a large amount of cycles spin waiting for events such as received packets or completion notifications in kernel-bypass networking. These cycles are wasted if Yama does not utilize them. On the other hand, cycle scavenging does not incur excessive latency to individual pieces of workload because `libYama`'s routines do not involve complex computation and can be executed by different CPU cores that belong to different applications. We evaluate the overhead that `libYama` induces in Section 6.

4.3 Cluster-Wide Collaboration

4.3.1 Coordination for Feedback Loop.

For each offload chain, one single instance of the feedback loop is sufficient to provide the throughput estimation, and running multiple feedback loops for the same offload chain incurs extra compute and communication costs. This involves, for example, each initiating `libYama` instance of this chain multicasting its application throughput to more than one feedback loops, and the feedback loops reconciling on each one's dummy work throughput to post, all consuming the scavenged CPU cycles that Yama should not overuse in order to minimize the impact on application performance.

That is, Yama must: (1) minimize the compute and communication costs, and (2) keep the feedback loop running while applications start and stop.

Coordinating Overlay. We design an overlay network across `libYama` instances to meet these requirements.

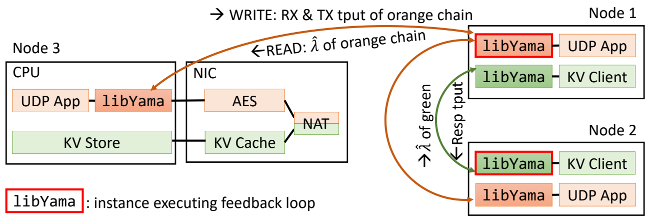

→ WRITE: RX & TX tput of orange chain
←READ: $\hat{\lambda}$ of orange chain
Node 1
Node 2
Node 3
CPU
NIC
UDP App
libYama
KV Store
AES
NAT
KV Cache
KV Client
→$\hat{\lambda}$ of green
←Resp tput
libYama : instance executing feedback loop

Figure 4: Coordinating overlay.

Figure 4 demonstrates the RDMA-based exchanges among libYama instances. The libYama instance running the feedback loop (the "leader") for an offload chain exposes a memory region that contains the estimated throughput of the chain as well as a ring buffer for each other instance that uses this chain. These other instances issue RDMA read operations to fetch the estimation and write operations to its corresponding ring buffer so as to notify the feedback loop of the amount of application workload accomplished. We set the overlay network traffic to a higher PFC (priority flow control) priority than the normal application traffic to make sure updates are delivered promptly.

This approach is scalable in terms of both memory and latency. The per-instance ring buffer is small (a few dozens of bytes), so a few 2MB hugepages are enough to support 10K+ remote instances. Moreover, each libYama registers a large memory region for RDMA when the it is loaded. This memory region is managed by the control plane, which allocates the ring buffer from that region when the entity requests to start an application. Therefore, the start of applications does not incur extra latency waiting for the leader to allocate the ring buffer just in time, which is a desired property for short-living applications.

Handover of Feedback Loop Execution. Since the feedback loop is driven by the CPU cycles of an application, it can stop when the driving application stops busy polling to execute other routines, or when the application quits either normally or accidentally. If this is not properly handled, the estimated throughput of a chain will stop being updated, and the other libYama instances will pace their workload based on a stale estimation, which can lead to over- or underloading.

To avoid this, we introduce a mechanism that transfers the feedback loop of a chain to another libYama instance whose application is active. Specifically, we accompany the throughput estimation with a timestamp field. Each time the feedback loop outputs an estimation, it also records the current timestamp. When the other libYama instances read this memory region, they compare the timestamp with the last read operation. The first libYama instance to find the timestamp being stale for over some threshold period will start an RDMA-based leader election [52] among the libYama instances whose applications use the same offload chain.

However, the leader election is also driven by CPU cycles scavenged from the application; it can happen that most participants are currently not active and so their voting routines cannot be executed for some period of time, so that a new leader might not be elected soon.

To avoid overly stale throughput estimation, the participant that proposes to be the new leader will start executing the feedback loop right away instead of waiting for votes. Those who receive the vote request will start reading the estimate from the candidate. This in effect ensures that the execution of the feedback loop is handed over quickly regardless of how long it takes to reach consensus.

4.3.2 Monitoring One- and Two-Sided Offloads.

For Yama, the significant difference between one- and two-sided offloads (or chains) is where to monitor their achieved performance for the feedback loop to calculate the amount of dummy workload to generate. As previously mentioned, one-sided offloads like RDMA and KV Cache do not always notify the CPU of individual pieces of workload. Therefore, their throughput has to be monitored on the operation initiators. In contrast, two-sided offloads, such as most network-layer ones (e.g., NAT and IP checksum) should be monitored at the target (receiver).

Yama can accommodate both cases. As shown in green in Figure 4, for applications that use one-sided offloads or chains, the collection of throughput statistics happens in the event queue paired with the work queue that issues the operations when the application polls for completion or acknowledgment. As shown in orange in Figure 4, for two-sided offloads, the applications on both sides are required to use libYama so that the achieved throughput of the offload chain can be gathered on both ends when the applications poll to receive packets, operations, or requests. In both cases the achieved throughput is delivered to the feedback loop via the coordinating overlay above.

5 IMPLEMENTATION

We implemented a prototype of Yama, which mainly focuses on the probing and scheduling functions. For applications that use the same offload chain, their libYama loads the per-node shared memory that contains the probing and scheduling state with respect to the chain. The prototype implements a lightweight coordination overlay. The leader libYama instance that executes the probing and scheduling routines is elected within a preselected node using a spin lock that applications contend for when invoking the adaptor's polling routine. Also, for two-sided offload chains, the receiver-side libYama conveys the achieved throughput to the leader in the form of tiny ACK packets. We omit the control plane and use statically assigned entity weights for evaluation, because as mentioned in Section 4.1, the interaction between libYama instances and the control plane is not performance-critical.

For evaluation purposes, we have the evaluation software NIC that runs offloads (introduced in the next section), directly interfacing libYama, but we implement an RDMA adaptor to demonstrate Yama's ability to work with different types of offloads.

The RDMA adaptor is realized by interposing the RDMA user-space library (libibverbs) so that existing RDMA applications can work over Yama without extensive adaptation. Most of the library functions are retained as-is, while the data-path ones like ibv_post_send are altered to redirect application calls to libYama. When libYama schedule the RDMA operation, it finds the function pointer in the work queue's encapsulating descriptor, which in this case is mlx5_post_send (we use Mellanox CX5 NICs). The libibverbs we base on is in package rdma-core, which is part of Mellanox OFED (Mellanox's official driver and software collection) for Linux version 5.0-2.1.8.0.

libYama and the RDMA adaptor add up to approximately 4,900 lines of C code.

6 EVALUATION

In this section, we seek to answer these questions:

(1) Can Yama achieve weighted fairness of throughput for a variety of offloads?
(2) Can Yama achieve weighted throughput fairness when offload chains share a common bottleneck?
(3) How quickly does Yama converge to changes?
(4) How much overhead does Yama incur on the application workload?

6.1 Methodology

Offloads. We evaluate Yama with four offloads: NAT, RDMA, key-value cache (KV Cache), and AES engine. This selection broadly covers the network (NAT), transport (RDMA) and application (KV Cache, AES) layers. It represents different usage patterns: NAT accepts raw packets, RDMA takes verbs like read and write, KV Cache works on RPC, and AES can handle byte chunks.

A major challenge in evaluating Yama lies in flexibly chaining them together and creating different bottleneck scenarios like the example cases shown in Section 4.1 to evaluate Yama's efficacy. While current NIC designs support chaining of offloads – to the best of our knowledge, there is no available testbed NIC design to artificially simulate different bottleneck scenarios. Our method, therefore, is to develop our own software NIC and software implementations of the offloads (except for RDMA offload which is supported by our testbed NIC). This software NIC implementation allows arbitrary chaining of different offloads and we can synthetically rate-limit each offload to emulate different bottlenecks. The software NIC performs raw packet IO via the hardware NIC using DPDK and can steer packets from/to different offload engines according to pre-configured rules. The offload engines can interface with applications with descriptor rings in shared memory.

We argue that such software implementaion of offloads shared by multiple entities is reasonable in datacenters. There exists high-level offload logic such as data analytics [36] that can only be executed by general-purpose cores hence must be implementd in software. Onloading such logic from NIC to host CPU has also been proposed for better performance [38]. Since the computation can be common in all kinds of applications, it can be desirable for datacenters to provide such computation as a shared service, so that tenants can avoid duplicate efforts.

Topology. We focus on the incast topology, where many nodes (clients) issue packets or (transport/RPC) operations to the NIC of one node (server). An incast is more prone to reach the bottleneck throughput as multiple client-side CPUs can generate more workload than that can be generated on a single node of the server. However, it should be noted that the design of Yama is orthogonal to topology and that there is no fundamental limitation on applying Yama to other topologies (such as an outcast) in case the client-side offloads become bottlenecks.

Testbed. All experiments are carried out on CloudLab [10]. Unless otherwise specified, each node has an AMD EPYC 7452 CPU with 64 hardware threads at 2.35GHz, 128GB of main memory, and a 100Gbps Mellanox CX5 RoCE NIC. A Dell Z9264F-ON non-blocking 100Gbps switch connects the partition of the cluster we use.

6.2 Fair-Share of Heterogeneous Offloads

We answer the first question by benchmarking Yama with three offloads individually: NAT, RDMA, and KV Cache. These offloads work on different network layers and represent different usage patterns. We use our software NIC-based implementation of NAT and KV Cache and the hardware RDMA engine on the testbed's NIC. To generate application workload, we develop libYama-native workload generators for NAT and KV Cache, and an RDMA-native workload generator with libYama adaptor.

The NAT generator generates IP packets with sizes subject to the bimodal distribution of data-center packets [4] that peaks at 40B and 1500B. The KV Cache workload generator produces UDP-based RPC requests of 32B keys and 64B values with 95% GET and 5% PUT, subject to a uniform distribution of 10,000 distinct keys. The RDMA workload

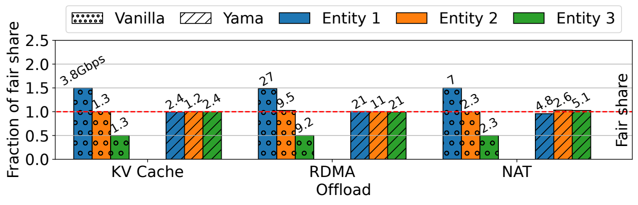


Figure 5: Per-entity fraction of fair throughput with various offloads.

generator issues a mixture of 50% `read` and 50% `write` operations of 64B.

Since the RDMA's hardware offload has a much higher maximum throughput compared to the software NAT and KV Cache, for RDMA, we use an incast from 12 nodes (each running a feedback loop) with 60 workload generator instances; and for NAT and KV Cache, we use an incast from 20 workload generator instances. Each workload generator creates a distinct `libYama` queue pair (as well as the corresponding underlying hardware queues) and is driven by a separate CPU core.

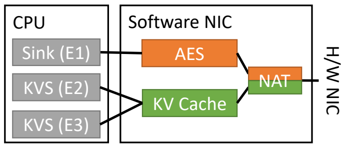
We divide the workload generator instances into three entities, where Entity 1, 2, and 3 have 60%, 20%, and 20% of the instances, but are assigned weights of 0.4, 0.2, and 0.4, respectively. Such assignment represents scenarios where low-weight entities (like Entity 1) try to gain more access to the offloads by ramping up more queues than higher-weight entities.

Figure 5 contrasts the achieved throughput of each entity with Yama and without it (called "Vanilla" in the figure). The vertical axis shows the ratio of the achieved throughput to the desired throughput as per policy, and the actual throughput is labeled above each bar. For example, the RDMA engine in this case, can process around 46Gbps. According to the weight assignment, the Entity 1 should obtain only 18.4Gbps, while it actually achieves 27.4Gbps without Yama, resulting in 1.49x times more usage than the fair share. Ideally, Yama should make all bars close to 1, which is indeed true according to the figure, and we conclude that Yama effectively achieves weighted fairness of throughput for different offloads.

Another point to note is that applying Yama does not lower the total throughput across all entities by more than 6%, and the total throughput actually improves for RDMA and NAT, which can be due to the different access patterns with Yama scheduling. This means that the dummy workload produced by the feedback loop incurs negligible overhead to the throughput of regular applications.

6.3 Fair-Share Under Offload Chaining

To answer the second question, we evaluate Yama with two offload chains on the server's software NIC configured as shown in Figure 6, where the orange chain (Chain 1) uses the



Offload	Case 1 (Gbps)	Case 2 (Gbps)	Case 3 (Gbps)
NAT	25	25	25
AES	10	10	20
KV Cache	10	16	20

Figure 6: Offload chain configuration and emulated bottlenecks.

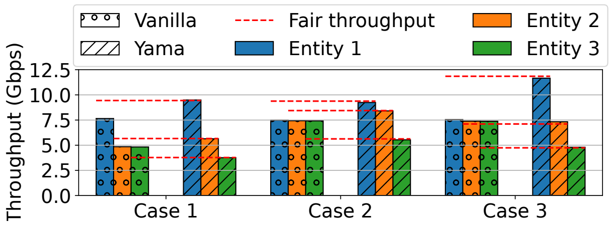


Figure 7: Per-entity throughput under a common bottleneck shared between offload chains.

NAT and AES offloads and ends up in a packet sink that receives packets and maintains statistics. Since the orange chain is two-sided (Section 4.3), the packet sink uses `libYama` on its receive routine to communicate the achieved throughput back to the `libYama` instance running this chain's feedback loop. The green chain (Chain 2) uses the NAT and KV Cache offloads and is shared by two entities, each running a key-value store instance on the server CPU. In conclusion, the three entities are the packet sink and the two key-value stores. Each entity runs an instance of the corresponding workload generator from remote with `libYama`.

The table on the right lists the combinations of rate limits that induce different bottlenecks, similar to the example in Section 4.1. We also assign the same weights to the three entities.

The results are shown in Figure 7. The red dotted line indicates the fair throughput for an entity given the total throughput achieved by Yama in each case. The figure suggests that Yama is capable of enforcing the weighted throughput policy in different chaining scenarios, even when the chains are independently bottlenecked and when they share a common bottleneck. In particular, the Yama's total throughput is no lower than the vanilla in all cases.

6.4 Convergence to Equilibrium

We answer the third question by studying the responsiveness of Yama to converge to the fair throughput in the face of applications starting and stopping workloads to the queue pairs. We use the same KV Cache experiment configuration as mentioned in Section 6.2. We model the start of the workload of each client as a Poisson process, and the time the client generates the workload for, is drawn from an exponential distribution. We regard each client as a separate entity and assign equal weights to each client to make the results simple

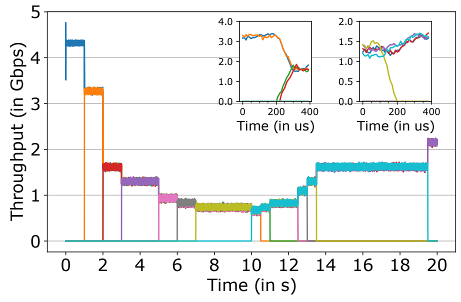


Figure 8: Throughput changes with entity joining and leaving. Each color corresponds to a separate entity.

to analyze: the achieved throughput of all active clients should be equal during the experiment.

Figure 8 shows the measured throughput of all clients aggregated every 1ms. The figure demonstrates that Yama can converge to the fair throughput share for all active entities - as entities join and leave, the throughput always converges to an equal value. The figure also zooms in to show two cases of entities joining (at around 2s) and an entity leaving (at around 13.5s), where throughput is aggregated per 20μs. For both cases, Yama converges to the new fair throughput share within 100μs, which is only single-digit RTTs (RTT evaluated below) and much faster than RoGUE [34], a congestion control scheme that does not rely on explicit feedback for low-latency RDMA operations. Hence, we claim that the Yama is responsive and can converge to fairness quickly. This is because Yama's pacing is directly based on a known throughput. Also Yama does not cause loss of throughput because the scheduler is work-conserving as weights are updated as entities join and leave.

6.5 Overheads

To study Yama's overheads on application workloads, we carry out an experiment where we use the workload generators as applications to produce workload at different throughputs and measure the end-to-end latency from when the application posts the workload to when the application is notified of the completion of the workload. For KV Cache, this is the time difference between posting the RPC and polling the reply corresponding reply. The request workload is generated as mentioned in Section 6.2 For NAT, since it is a network-layer offload that does not generate any reply packet, we measure the round-trip time of each packet by recording the send timestamp in the packet and have the receiver application on the server echo all packets back.

We run this experiment with and without Yama. The difference in the measured latencies gives the net overhead of Yama on each request or packet. Figure 9 shows the measured median and tail latencies of the two applications as the offered throughput is increased from 2 Gbps steadily until the offered

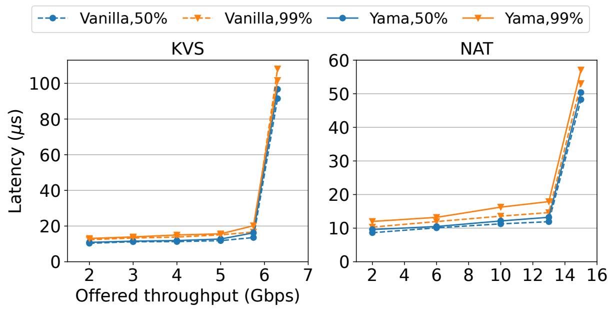


Figure 9: NAT and KVS op latency at various offered loads.

throughput can no longer be achieved. Figure 9 demonstrates that Yama adds only a few (mostly <3) μs of overhead in the tail latency. The figure also shows that for small throughputs, the overhead can be as low as less than a μs. We claim that this is a negligible overhead for the performance isolation guaranteed by Yama.

7 DISCUSSION

Broad applicability of Yama's approach. Yama's black-box approach for entity-level performance isolation can be extended beyond NIC and offloads to various types of middleboxes. On the one hand, offloading on programmable switches has become increasingly popular [20, 32, 72], and complex offload logic might not guarantee line-rate processing [72] On the other hand, high-programmability hardware such as the DPU (data processing unit) is being deployed to accelerate domain-specific high-level tasks. Both scenarios warrant a solution for performance isolation, and Yama can be easily applied to either. Specifically, Yama allows offload chains that cross network links to include switch offloads; and by treating the DPU offload as yet another black box and developing an appropriate adaptor, Yama can work seamlessly.

In addition, offloads that run on the same hardware can share some underlying resources. For example, NIC offloads can contend for the SRAM-backed cache [24, 38] and PCIe bandwidth [47]. In such cases, Yama's black-box approach can transparently provide "coarse" entity-level performance isolation in terms of average throughput by modeling the shared resource as a bottleneck "offload" shared by multiple chains. However, without knowing how these resources are multiplexed, and without local isolation mechanism such as weighted round-robin scheduling, it is hard to provide 'fine-grained' isolation over such shared hardware resources. We leave the exploration on applying Yama to shared hardware resources to future work.

Opportunities for protection against adversarial workloads. As previously mentioned, Yama does not aim to ensure that latency under load is within some factor of unloaded latency. However, the generic adaptor-based design of `libYama` enables solutions to at least a subset of these problems. For example, Justitia [73] addresses a problem in RDMA where large ops can cause head-of-line blocking for small ops from other flows, by opportunistically splitting large ops into smaller ones. This approach can be achieved in Yama by incorporating the shaping logic in the adaptor when converting RDMA ops into generic work queue descriptors, and controlling the granularity of splitting by measuring small op latency with dummy workload generation.

Another problem is offloaded functions with super-linear complexities, such as sorting, where large input can cause disproportionate throughput degradation. In such cases the operator can implement the adaptor in a way that penalizes large ops, for example, by reporting to `libYama` an op size adjusted for algorithm complexity.

Priority-based policies. In this paper, we have adopted the policy of weighted throughput allocation. However, some operators might also desire priority-based policies where high-priority entities must always be processed by the offload first.

This is a non-trivial problem without hardware support. In Yama, since the scheduling of workload does not happen locally at the offload, the nodes that schedule the workload have to perform all-to-all communication on virtually every op or RPC constituting a workload. This is needed to make sure there is no higher-priority workload on another node that needs to be scheduled. This can incur unacceptable overhead.

Nevertheless, such overhead may be mitigated by relaxing the priorities. A strawman might be able to hide the latency of all-to-all exchanges by, for example, scheduling local high-priority workload without waiting for communication. We leave a detailed exploration to future work.

Limitation of probing and backfilling. We find that, in practice, the probing feedback loop can be susceptible to jitter in collecting the completed workload statistics (Line 4 in Algorithm 1). Such jitter can be due to batch processing in the `libYama` execution routine, which must poll each queue pair to schedule pending workload and poll the hardware for events such as op completion and received packets, all while collecting completion statistics from other `libYama` instances and running the feedback loop; thus, the statistics might not be accurate by the time the feedback loop runs. As a result, instead of constantly updating the throughput estimation, the feedback loop must gather workload statistics for an extended period of time (which we empirically set to 1.25 times RTT) before the next update. As such, Yama can fall short in tracking sub-RTT changes in the offload throughput.

Also, backfilling can be limited in that, it requires the operator to know a representative distribution for the dummy workload, which involves human efforts to collect telemetry and perform analysis, or to automate this process. In particular, an open challenge is developing backfilling to represent cases where an offload's performance is related to the entities' private state in the offload.

8 RELATED WORK

Isolation in traditional clouds. A few notable systems provide performance isolation in traditional cloud networks. Seawall [60] features entity-level link bandwidth sharing in traditional datacenters. FairCloud [53] investigates the relationship among different properties of isolation policies in clouds, such as min-guarantee and network proportionality. Dominant resource fairness [13] accommodates different types of resources such as CPU, memory, and bandwidth in the sharing of datacenters. Google's BwE enforces policies hierarchically through rate-limits [30]. Nimble uses a control loop to install in-network rate limits on switches to provide per-entity fairness [68]. Since Yama focuses on offload chains, it is complementary to all of these projects. Also, since all these systems assume that the line rate of the shared resource is known, Yama addresses a key problem that arises in integrating offloads into other network-wide isolation systems by providing a low overhead mechanism for probing the current throughput of offloads.

SmartNIC primitives for isolation. There has been recent works on providing performance isolation for offloads that share the same SmartNIC. For example, PANIC [37] uses a hybrid push/pull approach that schedules packets for each offload in a chain while trying to avoid detours when possible, and SuperNIC [58] schedules packets across different chains of offloads to provide isolation. Yama is complementary to both of these systems. Even with systems that improve per-NIC local scheduling, Yama is still needed because local policy enforcement doesn't lead to global policy enforcement for chains [45, 68], and Yama moves packet buffering from in-network devices into the edge, which is desirable [49].

RDMA performance isolation for mixed op sizes. Justitia [73] uses an approach that also breaks RDMA requests into chunks to mitigate the previously mentioned head-of-line blocking problem. But this work mostly focuses on a rudimentary two-node setting without considering the isolation problem at a cluster scale. Further, Justitia makes the design choice to sacrifice some total throughput to preserve latency, and the AIMD algorithm used in Justitia can be slow to converge. In contrast, Yama provides both high throughput and predictably low latency overhead with fast convergence times.

9 CONCLUSIONS

We propose Yama, the first solution to provide per-entity isolation in the sharing of black-box NIC offloads. Unlike prior works that require customization of either the NIC or the offload, Yama enables this for commodity off-the-shelf NICs and proprietary offloads by providing a generic framework that abstracts the operations of most offloads. To ensure fair throughput sharing, Yama uses a novel performance probing approach leveraging dummy workloads, and paces the application throughput according to a policy that can provide weighted fairness of throughput for a variety of offloads as well as chains of different offloads. We demonstrate that Yama can provide per-entity fairness as per operator-defined weights for a variety of scenarios. Yama can converge to the fair throughput in a short response time and the overhead of Yama is only a few μs, which we claim to be a reasonable overhead.

ACKNOWLEDGEMENTS

We thank our shepherd, Prashant Shenoy, and the anonymous SoCC reviewers for their feedback that significantly improved the paper. We thank CloudLab [10] for providing equipment used to test and evaluate Yama. This research was supported in part by NSF awards CNS-2214015, CNS-2202649, CNS-2207317, a gift from Meta, and a Google Faculty Research Award.

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192

[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association.

[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[4] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2009. Understanding Data Center Traffic Characteristics. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking* (Barcelona, Spain) *(WREN '09)*. Association for Computing Machinery, New York, NY, USA, 65–72. https://doi.org/10.1145/1592681.1592692

[5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-Based Congestion Control. *Commun. ACM* 60, 2 (jan 2017), 58–66. https://doi.org/10.1145/3009824

[6] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for µs-scale RPCs with Breakwater. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association.

[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[8] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (ACM SOSP)*.

[9] Nandita Dukkipati. 2008. *Rate Control Protocol (Rcp): Congestion Control to Make Flows Complete Quickly*. Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) Mckeown, Nick. AAI3292347.

[10] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[11] 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 *NSDI*. USENIX Association.

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

[13] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 323–336.

[14] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. 2020. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 681–693. https://doi.org/10.1145/3387514.3405895

[15] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 649–667.

[16] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[17] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.

[18] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (jul 2008), 64–74. https://doi.org/10.1145/1400097.1400105

[19] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *ACM Workshop on Hot Topics in Networks (ACM HotNets)*.

[20] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. 2021. In-Network Support for Transaction Triaging. *Proc. VLDB Endow.* 14, 9 (may 2021), 1626–1639. https://doi.org/10.14778/3461535.3461551

[21] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. 2015. High Speed Networks Need Proactive Congestion Control. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. Association for Computing Machinery.

[22] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 1–16.

[23] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 295–306.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance {RDMA} Systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 437–450.

[25] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided ({RDMA}) Datagram RPCs. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 185–201.

[26] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pittsburgh, Pennsylvania, USA) *(SIGCOMM '02)*. Association for Computing Machinery, New York, NY, USA, 89–102. https://doi.org/10.1145/633025.633035

[27] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *ASPLOS*.

[28] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 297–312.

[29] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. 2022. Collie: Finding Performance Anomalies in RDMA Subsystems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[30] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[31] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 514–528. https://doi.org/10.1145/3387514.3406591

[32] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[33] 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 *SoCC*.

[34] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. 2018. Rogue: Rdma over generic unconverged ethernet. In *Proceedings of the ACM Symposium on Cloud Computing*. 225–236.

[35] 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 *SOSP*.

[36] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. 2023. Towards Accelerating Data Intensive Application's Shuffle Process Using SmartNICs. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 2, Article 36 (may 2023), 23 pages. https://doi.org/10.1145/3589980

[37] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*.

[38] 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 Special Interest Group on Data Communication (SIGCOMM)*.

[39] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Usenix Annual Technical Conference (ATC)*.

[40] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda. 2014. Accelerating Spark with RDMA for Big Data Processing: Early Experiences. In *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*. 9–16. https://doi.org/10.1109/HOTI.2014.15

[41] Mellanox Technologies. 2015. *RDMA Aware Networks Programming User Manual*. Retrieved July 20, 2020 from https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf

[42] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. 2020. Expanding across time to deliver bandwidth efficiency and low latency. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[43] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*.

[44] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided {RDMA} Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 103–114.

[45] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[46] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association.

[47] 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 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[48] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. 2019. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 97–108.

[49] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley,

and Costin Raiciu. 2022. An edge-queued datagram service for all datacenter traffic. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[50] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. Association for Computing Machinery.

[51] 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 *OSDI*. USENIX Association.

[52] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *HPDC*.

[53] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: Sharing the Network in Cloud Computing. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Aug. 2012), 187–198. https://doi.org/10.1145/2377677.2377717

[54] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. {SENIC}: Scalable {NIC} for End-Host Rate Limiting. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 475–488.

[55] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: efficient and flexible software packet scheduling. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 17–32.

[56] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory Deduplication for Serverless Computing with Medes. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 714–729. https://doi.org/10.1145/3492321.3524272

[57] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery.

[58] Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. 2021. Disaggregating and Consolidating Network Functionalities with SuperNIC. https://doi.org/10.48550/ARXIV.2109.07744

[59] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association.

[60] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the Data Center Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 309–322.

[61] Vishal Shrivastav. 2019. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[62] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. 2019. Direct Universal Access: Making Data Center Resources Available to FPGA. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[63] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*.

[64] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 2020. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 708–721. https://doi.org/10.1145/3387514.3405897

[65] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[66] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[67] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of Temporary Storage in the Nodekernel Architecture. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 767–781.

[68] Vineeth Sagar Thapeta, Komal Shinde, Mojtaba Malekpourshahraki, Darius Grassi, Balajee Vamanan, and Brent E. Stephens. 2021. *Nimble: Scalable TCP-Friendly Programmable In-Network Rate-Limiting*. Association for Computing Machinery.

[69] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. 2018. Dark Packets and the End of Network Scaling. In *ANCS*. ACM.

[70] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.

[71] Jiali Xing, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C. Lee. 2021. Charon: A Framework for Microservice Overload Control. In *HotNets*. ACM.

[72] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using Trio: Juniper Networks' Programmable Chipset - for Emerging in-Network Applications. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 633–648. https://doi.org/10.1145/3544216.3544262

[73] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. 2022. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1307–1326. https://www.usenix.org/conference/nsdi22/presentation/zhang-yiwen

[74] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. Association for Computing Machinery.

[75] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*.