Loom: Flexible and Efficient NIC Packet Scheduling

Brent Stephens, *UIC;* Aditya Akella and Michael Swift, *UW-Madison*

https://www.usenix.org/conference/nsdi19/presentation/stephens

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Loom: Flexible and Efficient NIC Packet Scheduling

Brent Stephens
University of Illinois at Chicago

Aditya Akella
University of Wisconsin-Madison

Michael M. Swift
University of Wisconsin-Madison

Abstract

In multi-tenant cloud data centers, operators need to ensure that competing tenants and applications are isolated from each other and fairly share limited network resources. With current NICs, operators must either 1) use a single NIC queue and enforce network policy in software, which incurs high CPU overheads and struggles to drive increasing line-rates (100Gbps), or 2) use multiple NIC queues and accept imperfect isolation and policy enforcement. These problems arise due to inflexible and static NIC packet schedulers and an inefficient OS/NIC interface.

To overcome these limitations, we present Loom, a new NIC design that moves all per-flow scheduling decisions out of the OS and into the NIC. The key aspects of Loom's design are 1) a new network policy abstraction: restricted directed acyclic graphs (DAGs), 2) a programmable hierarchical packet scheduler, and 3) a new expressive and efficient OS/NIC interface that enables the OS to precisely control how the NIC performs packet scheduling while still ensuring low CPU utilization. Loom is the only multiqueue NIC design that is able to efficiently enforce network policy. We find empirically that Loom lowers latency, increases throughput, and improves fairness for collocated applications and tenants.

1 Introduction

Many large organizations today operate data centers (DCs) with tens to hundreds of thousands of multi-core servers [54, 47, 24]. In virtualized DCs, there are many competing tenants, and operators need to ensure that these tenants are isolated from each other and share resources according to what they are allocated. With VMs and containers, it is currently possible to ensure that tenants fairly share CPU and memory. However, providing network isolation for competing tenants on a server continues to remain a problem [50, 30, 10, 32, 29, 41, 57]. Further, each tenant may run a variety of applications with different performance needs, ranging from latency-sensitive applications such as web services, search, and key-value stores, to throughput-sensitive applications such as Web indexing and batch analytics. It is similarly difficult to ensure that tenants' applications do not harm each other's network performance objectives [34].

Network isolation is hard because more functionality is moving to the network interface card (NIC), including packet scheduling. Data center operators are upgrading server NICs from 10Gbps to 100Gbps and beyond. To drive these high line-rates, NICs provide function offloading to reduce CPU load and multiple queues to enable parallel processing of packets [51, 52, 57, 43, 29]. Without these optimizations, applications struggle to drive line-rates. However, with them, it is not always possible to ensure suitable isolation among competing applications/tenants/flows. System software multiplexes applications and tenants into a small number of queues, and the NIC schedules packets from queues with coarse grain policies. As a result, the on-NIC packet scheduler, and not the OS, is now ultimately responsible for deciding which packets to send and when to send them.

The main goal of our work is to enable rich hierarchies of application-, tenant-, and DC operator-level policies to be realized on NICs while driving high line rates. This helps to simultaneously ensure that applications' network SLOs can be met, that tenants can be isolated from each other on the data center network, and that operators' network performance objectives are satisfied.

To solve this problem, we created *Loom*, a new NIC design that moves all per-flow scheduling decisions out of the OS and into the NIC. Loom provides a customizable, hierarchical on-NIC packet scheduler and an efficient OS/NIC interface with a queue per flow. This enables Loom to implement a variety of scheduling algorithms while also enabling the OS to drive line-rates (100Gbps). Loom takes inspiration from recent advances in switch design such as PIFOs and others [12, 55, 15, 56]. These switches utilize a programmable match+action pipeline and generic scheduling queues to support a variety of hierarchical scheduling algorithms. However, NICs are a fundamentally different environment than switches, and these existing approaches are not immediately applicable. Loom addresses the problems that arise when implementing programmable scheduling on NICs.

There are three key components to Loom. First, Loom introduces a new network policy abstraction: restricted directed acyclic graphs (DAGs). Existing abstractions, such as flat traffic classes or strict hierarchies cannot express a common type of end-host network policy. Specifically, hosts may want to rate limit aggregated traffic by destination, such as over intra-DC or WAN links (*e.g.*, BwE [32] or EyeQ [30]). Loom's new DAG policy abstraction allows for per-destination rate-limits to be expressed independently of the traffic classes used to determine how tenants and applications share bandwidth.

Second, Loom introduces a new programmable packet scheduling hierarchy designed for NICs. In switches, packet headers are available to make scheduling decisions. However, NICs have limited on-NIC SRAM, so they must make scheduling decisions prior to reading packet headers from main memory via DMA. In Loom, the OS enqueues scheduling metadata along with the descriptors used to notify the NIC of new packets, and the NIC only fetches packet content when it is scheduled to be sent.

Third, Loom contributes a new expressive and efficient

OS/NIC interface that utilizes batching and metadata in-lining to reduce the CPU overheads of communicating the network policy to the NIC. Specifically, Loom uses a doorbell queue per core to efficiently aggregate both multiple packets and policy changes into a single PCIe write.

We build a software Loom prototype based on BESS [1], and conduct experiments at both 10Gbps and 40Gbps. We find that Loom is able to enforce complex hierarchical network policies. Also, we show that Loom is able to enforce policies that are not expressible in existing policy abstractions. In contrast, we find that it is not possible to enforce even simple policies with existing multiqueue NICs. Further, we demonstrate that improving network isolation translates into reductions in latency, increases in throughput, and improvements in fairness for competing tenants and applications that are collocated on the same servers. Through an analysis of worst-case behavior, we argue that Loom can still operate at 100Gbps line-rate, even with minimally sized packets. Finally, we evaluate the overheads of our new OS/NIC interface and find that Loom can reduce the number of generated PCIe writes by up to 43x when compared with existing approaches [43, 35].

2 Motivation

Different DC applications and tenants have different performance requirements and service level objectives (SLOs). Ideally, DC operators would be able to ensure that competing applications and tenants are isolated according to some high-level policy, and that application- and tenant-specific SLOs are met [30, 10, 41, 11, 34]. Unfortunately, today, this is not always possible. A key part of the reason is static and inflexible packet scheduling on server NICs today. We elaborate on this issue in the rest of this section.

2.1 High-Level Network Policies

Multi-tenant DCs run different classes of applications, each with their own performance objectives. Different applications can benefit from customized scheduling algorithms [16, 18, 19, 17, 42, 59]. At the same time, cloud providers need to ensure that tenants on the same server fairly share resources, and operators want high infrastructure utilization [34]. Meeting these performance goals amounts to specifying and enforcing a policy that determines how packets from all flows, applications, and tenants on a server are scheduled.

Figure 1 shows a possible high-level network policy for a server. This policy should be enforced no matter how applications access the NIC, such as with SR-IOV that bypasses the hypervisor and/or kernel. Our example is motivated by recent work which demonstrated that there is significant potential for increasing infrastructure utilization by collocating big data applications with latency sensitive applications like key-value stores (KVS) [34]. In this policy graph, the "leaves" shown at the top are packet sources (*e.g.*, different flows). Nodes in this figure determine how packets from different flows are

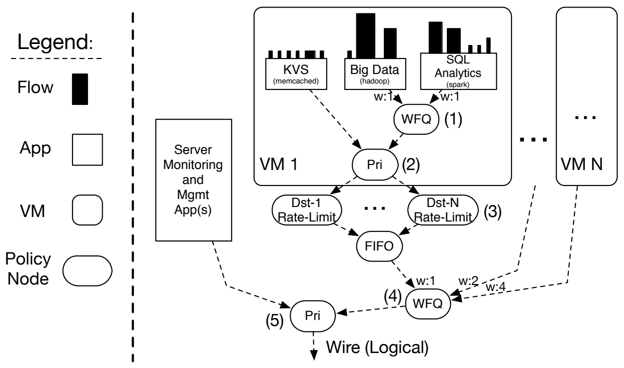


Figure 1: **A scheduling hierarchy for a server. Different parts of this hierarchy are specified by different entities (Section 3). The OS is responsible for dynamically enforcing this policy.**

scheduled. Policies at different levels of the graph come from different entities (e.g., operator, tenant, application).

In *VM 1* in Figure 1, first, all the flows from an application are aggregated. Each application can specify the scheduling algorithm used for its own flows. Next, the tenant that owns VM 1 specifies how traffic from its applications is scheduled. Node (1) specifies that the competing flows from Hadoop and Spark should use weighted fair queuing (WFQ) to fairly share bandwidth. Node (2) then specifies that traffic from the key-value store (KVS) should have strict priority over both Hadoop and Spark traffic. Together, these nodes specify a work-conserving policy for how competing traffic from VM 1 should share limited NIC bandwidth.

Additionally, some network operators may want to specify per-destination rate limits that are enforced at hosts. This is useful for ensuring network isolation across intra-DC and wide-area links (*e.g.*, BwE [32] and EyeQ [30]). In the example, an operator specifies separate per-destination rate-limits to all the traffic created by VM 1 with the Dst-1 through Dst-N nodes (3), which may be replicated for other VMs.

Finally, an operator specifies how traffic from competing tenants is scheduled. Node (4) specifies that different tenants should use weighted fair queuing to share bandwidth in proportion to the resources they are allocated (w:1, w:2, w:4). Node (5) then specifies that management and hypervisor traffic have strict priority over tenant traffic.

2.2 Issues with Multiqueue NICs

As the above illustrates, it is desirable to schedule traffic leaving a server according to a high-level policy. However, the principle challenge in doing this is in ensuring that applications with competing network objectives (*e.g.*, bandwidth-hungry vs. latency sensitive) do not impact each other's network performance. This is currently not always possible because OSes need to use many independent NIC transmit queues to drive line-rate[1]. As a result, the NIC and not the OS

[1]From our own experiments, we have found that, even with batching [7, 2] and TSO, single core throughput in Linux is limited to around 36-40 Gbps.

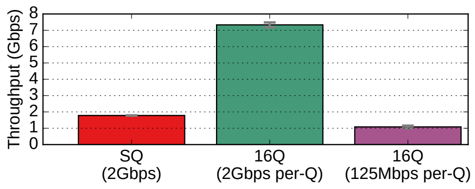


Figure 2: **Achieved rate for memcached with 16 threads when trying to enforce a 2 Gbps rate-limit.**

is now ultimately responsible for deciding which packets to send and when to send them. Unfortunately, current on-NIC packet schedulers are static, inflexible, and only support a limited number of traffic classes and scheduling algorithms [57].

2.2.1 Background

Modern NICs provide multiple transmit/receive queues and interrupt lines. This reduces overhead by allowing multiple cores to simultaneously send and receive packets. Current best practices are to configure a separate transmit queue for each core. By default, most NICs service queues using simple deficit round robin scheduling [53].

Many NICs support a few additional features for controlling the packet scheduler. NICs with DCB support [21] provide priority scheduling and partition the queues on a NIC into a different pool of traffic for each of 8 DCB priorities. Some advanced NICs allow the OS to set per-queue or per-priority rate-limits [35, 43]. A few NICs support a one-queue-per-flow (QPF) model [43, 35]. These NICs only provide rate limits, deficit round robin [53], and a small number of priorities.

While these scheduling features provide a limited ability to implement fair scheduling, today's NICs are unable to enforce many other useful, rich network policies.

2.2.2 Inflexible NIC Packet Schedulers

To illustrate the problems with enforcing network policy on multiqueue NICs, we performed a few experiments with Linux and an Intel 10 Gbps NIC [27]. For each transmit queue, we configure Linux `Qdisc` to classify and schedule packets according to a network policy. We find that this can enforce policy when only a single transmit queue (SQ) is used, but that network policy is violated when multiple NIC queues (MQ) or a queue-per-flow (QPF) are used. Although we use XPS [9] to pin transmit queues to cores in these MQ experiments, we see similar results without XPS.

First, Figure 2 illustrates the difficulty in enforcing a rate-limit for all traffic from a single multi-threaded application. In this experiment, the policy is that the sum of all traffic from a 16-threaded memcached application should be rate-limited to 2 Gbps. Because network traffic is not uniformly spread across application threads, it is not possible to configure a per-queue rate limiter. Setting a rate-limit of 2 Gbps per-queue leads to over-utilization, while setting a fair rate-limit of 125 Mbps (2 Gbps/16) leads to under-utilization.

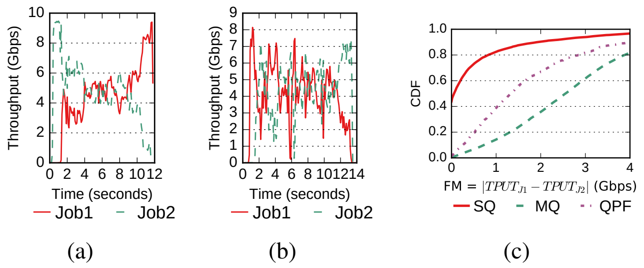


Figure 3: **Unfairness when two Spark jobs are allocated equal bandwidth shares. (a) Time series of the achieved throughput for two competing Spark jobs with the MQ configuration. (b) Throughput with the QPF configuration. (c) CDF of the difference in aggregate throughput for two competing jobs.**

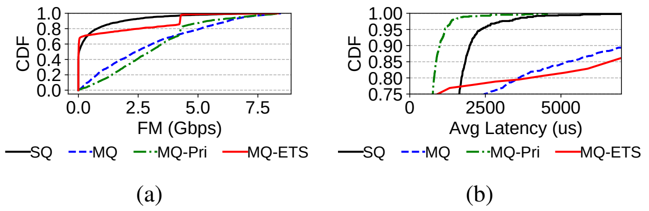


Figure 4: **Hybrid approaches that combine DCB and software to enforce a policy. (a) Difference in Spark throughput for two competing tenants. (b) Memcached latency for 32KB values.**

Next, Figure 3 shows an experiment where the network policy is that two Spark jobs (each with multiple tasks) should share network bandwidth equally, and each task is allocated its own CPU core. Figure 3a and 3b plot a time series showing unequal throughput achieved by each job for the MQ and QPS configurations, respectively, and Figure 3c plots a CDF of a fairness metric $FM = |TPUT_{J1} - TPUT_{J2}|$, i.e., the difference in achieved throughput for the two jobs when they are both active. Even though each job has the same number of queues, it is still not possible to enforce network policy in MQ because the active flows are not uniformly spread across the cores. Similarly, because the NIC performs per-queue fair scheduling and each job does not have the same number of active flows, QPF does not fairly schedule traffic (Figure 3b). In contrast, the OS can ensure a fair bandwidth distribution when using a single queue (SQ). This is shown in Figure 3c.

With DCB, it is reasonable to try to enforce part of a network policy in hardware, such as prioritizing latency sensitive traffic, and then enforce the rest in software. Figure 4 shows the results from two hybrid approaches. In this experiment, there are two tenants who should fairly share the link bandwidth. The first only runs Spark. The second runs both memcached and Spark. The second tenant's local policy is that memcached traffic should have strict priority. First, the MQ-Pri approach assigns memcached traffic to a DCB traffic class given strict priority over other traffic, while the traffic from both Spark applications share a traffic class and the cross-tenant fairness policy is enforced by software. Second, the MQ-Fair approach does the opposite and assigns traffic from each tenant to different DCB traffic classes, with each class being given a equal share of bandwidth, and enforces the priority policy for memcached traffic in software.

Figure 4a shows that MQ-Pri is unable to fairly share bandwidth between the competing tenants but achieves low memcached latency in Figure 4b. Conversely, MQ-Fair has good fairness, but very high latencies. This demonstrates that it is not possible to have an OS or hypervisor both use multiple queues and enforce only part of the network policy in software with the NIC enforcing the rest in hardware.

2.2.3 Inefficient OS/NIC Interfaces

It is possible to use a dynamic approach to enforcing network policy by collapsing it into appropriate per-queue weights, rate limits, and priorities. However, here, scheduling metadata for each queue needs to be updated as flows start and stop.

We find two performance problems to this approach. First, with current NIC interfaces drivers must write a per-queue PCIe doorbell register after adding data to a queue; for small flows across many queues, this can lead to many PCIe writes. Each update requires a separate PCIe write, taking up to 900ns [23]. During this time, the CPU is otherwise unavailable. Furthermore, past research shows that when two cores send data on different queues and write a doorbell for each packet they are unable to achieve a 40 Gbps line rate [46].

Second, updating policy also requires expensive PCIe writes. These overheads are prohibitive, especially if many different flows/queues must be updated simultaneosly, and they directly undermine the benefits of offloading packet scheduling to NICs. In this case, it may not be possible to both drive line-rate and update the NIC's packet scheduler.

To illustrate this problem, we measured the overheads of configuring per-queue rate-limiters on a Mellanox ConnectX-4 NIC [35]. Installing a new limit takes a median of 2.07ms. Setting a queue to use an existing limit takes a median of 64μs to complete. We also modified the driver to apply an existing rate asynchronously, and not wait for completion events (and errors). Even so, the median update takes 950ns!

3 DAG Policy Abstraction

Loom is a NIC design with a programmable packet scheduler that offloads the enforcement of high-level network policy. This section describes the design of the Loom policy DAG.

The key aspects of the policy DAG in Loom are: (1) what scheduling algorithms can be expressed at each node in the DAG, (2) how nodes in the DAG can be connected, (3) how different sub-policies from individual applications and tenants are composed, and (4) how traffic is classified into different leaf nodes in the hierarchy.

Node Types. There are two types of non-leaf nodes in the policy: work conserving *scheduling* nodes that determine the relative ordering of different packets, and rate-limiting *shaping* nodes that determine the timing of packets. Every node in the policy is annotated with the specific scheduling or shaping algorithm that should be used. All scheduling algorithms in

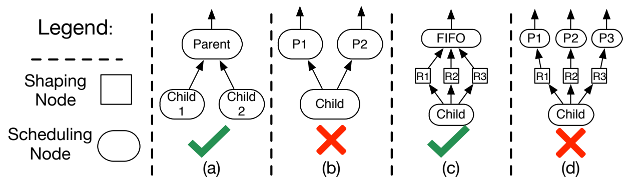


Figure 5: **An illustration of the relationships allowed between scheduling and shaping nodes in Loom. A check indicates that the relationship is allowed ((a) and (c)), while an 'x' indicates the relationship is forbidden ((b) and (d)).**

Loom are expressed by enqueue and dequeue functions that compute a priority (rank). For convenience, Loom provides default implementations of common algorithms including strict priority scheduling (Pri), rate-limits (RL), weighted fair queuing (WFQ), and least slack time first (LSTF) scheduling [38]. However, Loom also allows for installing custom scheduling algorithms at a node.

Restricted DAG Hierarchy. Network policies in Loom are expressed as a restricted DAG like what is shown in Figure 1. The restriction is that the policy graph forms a tree if all shaping nodes are removed. Each shaping node may optionally be a nested set of parallel shaping nodes. Once traffic is aggregated for scheduling, it may only be separated again for shaping. This prevents scheduling nodes from reordering packets that were already ordered by a child while still allowing for separate per-destination and per-path rate-limits.

Ideally, an operator would be able to compose a network policy that specifies both a work-conserving policy for sharing the bandwidth of the local NIC port and separate rate-limit classes to manage bandwidth in the network core and over WAN links. Unfortunately, such policies are not expressible as a tree because once flows for different destinations have been aggregated, they cannot be disaggregated and have separate rate-limits applied. *The Loom policy DAG, however, allows for policies where the traffic classes used to order competing requests (i.e., work-conserving scheduling classes) may be different than the rate-limit classes used to shape traffic.* In contrast, such policies are not expressible via Qdisc.

In more detail, the DAG is restricted in the following way: if a node N's parent is a scheduling node, then N may have at most one parent (and outgoing edge). Loom imposes the restriction that a node may have multiple parents (and outgoing edges) *only* if the parents are all shaping nodes. Further, all sibling shaping nodes must share a single FIFO parent. This ensures that scheduling nodes closer to the root do not reorder packets that were already ordered by a child node.

These DAG restrictions are illustrated in Figure 5. Figure 5(a) and Figure 5(b) show that each node may have only one parent if the parent is a scheduling node. Figure 5(c) and Figure 5(d) show that all parallel shaping nodes must share the same parent and child.

Traffic Classification. When packets are sent to the NIC, the OS tags them with the appropriate metadata, and the NIC then uses lookup tables to map the packet to the appropriate leaf nodes in the policy DAG. This happens before Loom enqueues the packet. Example metadata includes per-socket priorities, socket IDs, process IDs, users, cgroup names, and virtual interface IDs. Policies may also be expressed in terms of network addresses (*e.g.*, IP destination). For example, this is the case with per-destination rate-limits.

Composing Sub-Policies. As Figure 1 illustrates, different sub-graphs of the network policy need to come from different entities. For example, each application may have its own local scheduling algorithms. In Loom, each entity expresses its own local policy as a separate policy DAG. These sub graphs are composed at each level, such as the OS or VM, by attaching the root node of each sub-graph to a leaf node in the next level of the policy. Finally, the VMM passes the final graph to the Loom NIC. When policies are not specified (*e.g.,* when a legacy application does not specify how its flows should be scheduled), Loom uses FIFO packet scheduling.

Policy Limitations. Although our DAG policy abstraction addresses a key limitation with applying rate-limits in prior work [56], there are still some limitations to this abstraction. One limitation is that polices are local to a node. In a cloud data center, it may be desirable to express *network-wide* policies across a cluster of servers [50, 30, 10, 32, 29, 41, 44], *e.g.*, "All of the servers for Tenant A and Tenant B fairly share network bandwidth." However, such a policy is not directly expressible in our abstraction. It would have to be implemented by (dynamically) mapping this high-level policy onto a collection of per-server and possibly per-switch policies. To enforce such policies, rate-limiters need to be updated whenever either VMs [29] or flows [44] start and stop.

Another limitation is that Loom does not guarantee that the algorithms expressed as part of a policy DAG are efficient or that a policy will map well onto the underlying hardware. Providing default implementations of common algorithms helps overcome this limitation. We also allow the NIC to reject policies when it cannot efficiently compute the enqueue and dequeue functions or when the DAG is too large. This avoids poor NIC performance from inefficient policies.

4 Loom Design

There are two key components to the Loom programmable NIC design: (1) a new scheduling hierarchy, and (2) a new OS/NIC interface that enables the OS to efficiently and precisely control how NICs perform packet scheduling while still ensuring low CPU utilization. This section first provides background on programmable scheduling for switches and then discusses these two components.

4.1 Programmable Scheduling Background

Loom's design leverages recent advances in switch design for programmable and stateful match+action forwarding pipelines [12, 55, 15], and programmable hierarchical packet scheduling [56]. In these systems, lookup tables arranged in a pipeline map packet headers to logical queues and scheduling

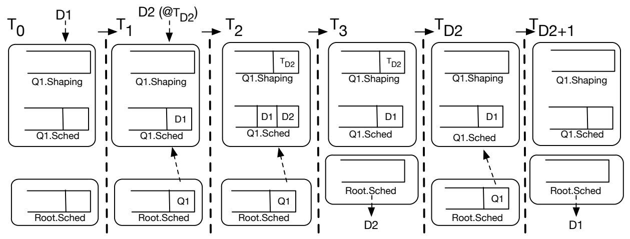


Figure 6: **An illustration how a prior PIFO-based scheduler [56] operates that also shows how rate-limiters will be incorrectly applied when only some of the packets at a scheduling node should be rate-limited. Although the packet for D2 should be rate-limited, the packet for D1 is incorrectly rate-limited because D2 has a higher rank than D1.**

metadata. For example, the NIC needs the leaf traffic class for the packet. Similarly, WFQ tracks virtual time and the number of transmitted bytes per class. Once scheduling metadata for a packet has been found, packets are enqueued into a logical scheduling hierarchy implemented by a tree of priority queues, also known as push-in first-out queues, or PIFOs.

Scheduling: Because all policies in both these systems and Loom are expressed by computing a priority (rank), many policies can be implemented with a common tree of PIFOs. Different scheduling algorithms are implemented by changing how the rank is computed (e.g., priority, deadline, virtual time, slack, etc.). In general, within a single node, this model is sufficient to emulate any scheduling algorithm [38].

A scheduling algorithm is expressed through an enqueue function, a dequeue function, and the state maintained across function calls. The enqueue function runs when packet is enqueued at a node in the DAG. Using the local state and metadata associated with a packet, this function then computes and returns a rank (priority). On dequeue, the node returns the lowest rank packet and a dequeue function updates local state. For example, an implementation of fair queuing would update a global virtual time in the dequeue function. Shaping algorithms behave similarly, but they compute a transmit time as a rank. This enables Loom to implement a wide range of scheduling and shaping algorithms.

For example, strict priority scheduling is implemented by computing a priority on enqueue. WFQ is implemented by computing a virtual time for the packet on enqueue and updating a global (to the node) virtual time on dequeue. Both strict and token bucket rate-limits can be implemented by computing a wall clock transmit time on either enqueue or dequeue. Other scheduling algorithms can also be implemented (e.g., for LSTF [38], we compute a slack time on enqueue).

Enqueuing and dequeuing from different nodes in the hierarchy operates as follows. First, a match+action pipeline finds all necessary metadata from the leaf traffic class. Then, enqueuing starts at the leaf node. The enqueue function for this node is used to compute a rank and enqueue a pointer to the packet. Traversing the hierarchy from the leaf node to the root, the local enqueue function at each node computes a rank, and the NIC enqueues a pointer to the appropriate child node at each parent.

The leftmost picture in Figure 6 illustrates this behavior. When a new packet arrives at T_0, it is first pushed into Q1, and then a pointer to Q1 is pushed into the Root PIFO. Conversely, when the transmit port is ready to transmit a packet, it starts by dequeuing the element at the head of the root of the PIFO tree, which will be a pointer to a child PIFO. After running the dequeue function, dequeuing then continues to follow this pointer chain (e.g., Q_1 and Q_2 could have other pointers enqueued) until a leaf node is reached and the original packet is ultimately dequeued.

Rate Limiting: To implement rate-limiting, each node in the prior PIFO design uses both a shaping queue for rate-limiting and a scheduling queue for ordering packets. When the enqueue function at a node determines that a packet should be rate-limited, its transmission time is pushed into the shaping queue for the node and the packet is added to the scheduling queue. However, no pointers are enqueued at subsequent parent nodes. Then, only once the computed transmit time has expired, is the packet enqueued at the rest of the parent nodes and it will eventually be scheduled.

Unfortunately, this design cannot apply rate-limits to only some of the packets sharing a PIFO node in the hierarchy (e.g., those going to specific sets of destinations).

Figure 6 illustrates this behavior and the problem that it causes. In this example, there are two queues in the hierarchy, the Root queue, and Q1. At time T_1, a packet for D1 that should not be rate-limited is enqueued in the tree. Then at T_2 a high-priority packet for D2 that should be rate-limited until time T_{D2} arrives. Because of its priority, it is ordered in Q1 ahead of D1, but it is not enqueued higher up the tree (per the PIFO rate limiting description above). Then, because there is still a reference to Q1 at the root, the next time a packet is dequeued from the root, the packet at the head of the scheduling queue at Q1 will be returned. However, in this case, the packet for D2 is dequeued because its rank in the scheduling queue at Q1 for D2 is higher than the rank for D1. At this point, there are no pointers to Q1 at the root, so D1 will not be sent. Later, at time T_{D2} a pointer to Q1 will be enqueued at the root and D1 will eventually be sent. In effect, the packet for D1 is rate-limited when it should not be, while the packet for D2 is not rate-limited when it should be!

This problem occurs because the existing design does not distinguish rate-limited from non-rate limited packets at a node. For the same reason, having different rate limits at a node can also lead to packets being sent at the wrong time.

4.2 Programmable Scheduling for NICs

NICs provide a different environment than switches, so existing approaches for programmable scheduling on switches are not immediately usable on NICs. Switches have buffer space to hold complete packet contents. In contrast, NICs perform

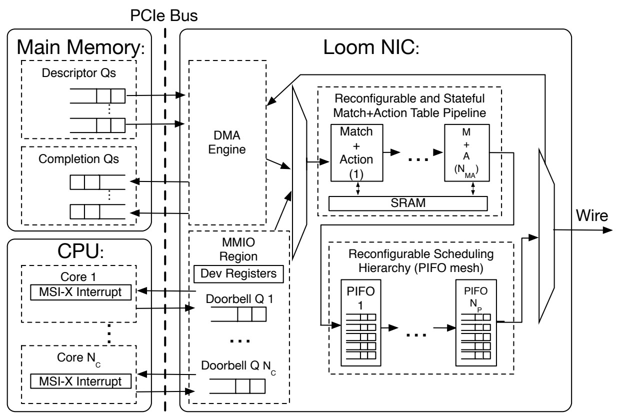


Figure 7: **A breakdown of the different components of Loom. Note that descriptor queues are per-flow while the doorbell queues are per-core.**

scheduling over packet descriptors, and defer reading packets from main memory as long as possible to keep memory requirements small. Reading just the headers is infeasible because it would require two DMA operations per packet. Instead, NICs must perform scheduling in advance of reading a packet from main memory.

4.2.1 Scheduling Operations

To overcome this challenge, Loom relies on the OS to communicate any necessary scheduling metadata to the NIC. This metadata may be explicitly set for each packet by including scheduling information in-line with transmit descriptors and doorbell writes. This metadata may also be implicitly determined by the queue the OS uses to send a packet.

Figure 7 illustrates the design of our Loom prototype. The OS assigns each flow its own descriptor queue. After writing descriptors, the OS then rings an on-NIC doorbell. Doorbells are then parsed and processed by an on-NIC stateful match+action pipeline. This pipeline is used to lookup scheduling metadata (Section 4.3.2) and per-queue DMA state (*e.g.*, descriptor ring buffer address). Next, pointers to descriptor queues are enqueued in a PIFO tree that is used to schedule competing packet transmissions. At this step, all scheduling ranks are computed using information from the OS. When the request is dequeued, it is sent to the DMA engine. Once the DMA engine has read the necessary descriptor and packet data from main memory, packets are then parsed and processed by the match+action pipeline before being transmitted. For example, packet headers are updated as needed to support features like segmentation offload (TSO) and network virtualization in the match+action pipeline.

Regardless of whether metadata is explicitly in-lined with descriptors and doorbells or implicitly associated with queues, with this division of labor, it is possible to schedule the processing of different transmit requests from different transmit queues without already having access to the packet data. The

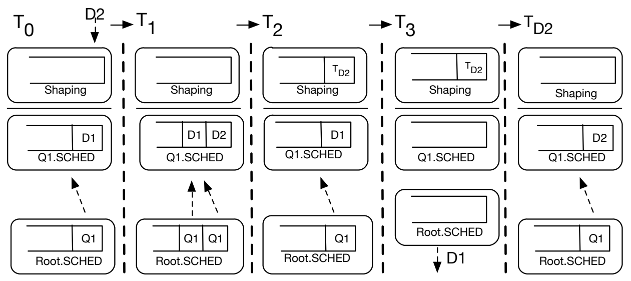


Figure 8: **An illustration how Loom enforces different rate-limits for different packets sharing the same PIFO. When a packet is dequeued before its computed transmission time, it is instead pushed into a separate global shaping queue. After its transmission time, it is re-enqueued in the hierarchy.**

trade-off between these two approaches is that including metadata in descriptors and doorbells increases the descriptor and doorbell size while using implicit per-queue metadata consumes additional on-NIC SRAM.

4.2.2 DAG Rate-Limiting

To implement the rate-limiter DAG abstraction in Loom, we created a new design for rate-limiting with PIFOs that allows multiple rate-limiting classes to be applied at the same PIFO node. Crucially, this design uses a *global* shaping queue to implement rate-limits instead of per-node shaping queues. This overcomes the limitations of previous designs that cannot support separate scheduling and shaping queues.

When no rate-limits are exceeded, packets are enqueued into the PIFO hierarchy as if no rate-limit were applied. Instead of pro-actively enforcing rate-limits, transmission times are computed and stored. Then, on dequeue, if the transmission time is in the past, dequeuing continues as normal.

However, if the transmission time is in the future on dequeue, the request is enqueued in a separate global shaping queue. Because all shaping is done with respect to wall-clock time, independent shaping queues are not needed for each node in the hierarchy. Next, after the computed (rate-limited) transmission time expires, the request is then re-enqueued according to its previous ranks in the scheduling queues, after which it will be transmitted according to the policy as normal. This ensures that subsequent high-priority traffic can bypass rate-limited traffic. In contrast with previous designs [56], Loom can correctly enforce rate-limits even when different transmission times are computed for different requests otherwise sharing the same scheduling queue in the hierarchy.

Figure 8 illustrates this design with an example. At time T0, a packet for D2 shows up that should be rate limited until time T_{D2}. However, at time T1, it is initially only enqueued in the scheduling queues. Then, when it is dequeued from the root PIFO at T2, it is re-enqueued in a global shaping queue until its reaches its transmission time of TD2. At this point, it is then re-inserted into the hierarchy.

4.2.3 Line-rate (100Gbps) Operation

An apparent trade-off of this design is that it can increase the number of enqueue and dequeue operations performed per packet. In the worst case, a packet in Loom may be enqueued and dequeued two times from the PIFO hierarchy. However, this does not prevent Loom from operating at line-rate. Although our design, like prior work [56], uses PIFOs that only support a single enqueue and dequeue operation per cycle, a Loom NIC does not need to schedule a unique packet every cycle to still be able to forward at line-rate.

Consider the following analysis: Assume a 100Gbps NIC that operates at a frequency of 1GHz. The OS sends minimum-sized 64-byte packets, and each packet requires the worst-case 2 enq/deqs per packet. Even in this case, the NIC can still schedule a packet every 2ns. Achieving line-rate only needs to send a packet every 5.12ns. Further, we note that this worst-case analysis is far from the common case. For example, packets are often bigger, and only some rate limited packets need multiple enq/deqs.

We employ other optimizations to further reduce operations. One such optimization is that we include a flag in scheduling metadata that indicates whether a rate-limit class is currently being rate-limited. This allows for packets to be immediately enqueued in the shaping queue, reducing the average number of operations per packet. Second, to bound the number of subsequent dequeue operations that do not yield a transmittable packet, we limit the number of outstanding packets from each traffic class.

However, deep policy hierarchies can require multiple average cycles per dequeue, preventing 100Gbps operation. While different stages of the hierarchy can be pipelined, if the depth of the hierarchy exceeds the number of PIFO blocks, again it may not be possible to guarantee an average rate of 1 enq/deq per cycle. We expect policies in practice to be not arbitrarily deep. For example, the graph in Figure 1 is quite rich, yet it is uses ≤ 5 PIFOs. A deeper policy graph is unlikely to need more than 10 PIFOs. Thus, we believe we can support 100Gbps for practical policies.

4.3 OS/NIC Interface

Loom introduces an efficient and expressive OS/NIC interface for notifying the NIC of new segments and communicating network policy updates. Loom minimizes the *total number of PCIe* writes needed by the OS. This is accomplished through two mechanisms: First, the OS uses batched doorbells to notify the NIC of new segments. Second, scheduling and metadata updates are passed in-line with packet descriptors to avoid generating additional PCIe writes.

4.3.1 Batched Doorbells

To efficiently support a large number of transmit queues, doorbells are separated from (per-flow) transmit descriptor queues in Loom (Figure 7). Instead of writing to a separate doorbell for each queue, in Loom, doorbells are written indirectly through *doorbell queues*. When the driver needs to notify the NIC about new segments from different flows it writes a batch of doorbell descriptors (16-bit integers) to a per-core doorbell queue. These queues are stored on the NIC, and, with write-combining, 32 doorbells (1 cacheline) can be written in a single PCIe write. This design builds on top of the fact that modern OSes already send segments to the NIC in batches [7, 2]. Unlike existing NICs that generate a PCIe write per queue to ring per-queue doorbells, Loom only generates a single PCIe write per batch.

Figure 7 illustrates this design by showing the different types of queues that are used in Loom for OS/NIC communication and where they are located in memory. The control flow when the OS needs to send a batch of packets in Loom is as follows: first, descriptors for individual packets and configuration updates (described below) are written to per-flow descriptor queues. These are fast writes to main memory. Additionally, descriptors for all of the packets in the batch that belong to the same flow are created in a single write. Next, the OS writes a batch of doorbell descriptors to a doorbell queue. The NIC detects the write, finds the queues from the doorbell descriptors, and processes the segments in the batch.

Although Loom relies on batching to reduce the number of PCIe writes, it still provides low latency. If there are fewer packets than can fit in a batch, the OS and/or application need not wait before ringing a doorbell. If new packets are generated in the interim, they will be part of the next batch. Similarly, due to per-flow queues and on-NIC scheduling, high-priority packets in later batches are not blocked by packets in earlier batches.

4.3.2 Scheduling Metadata

In Loom, the OS provides the NIC with the necessary metadata to compute scheduling ranks before the NIC has read segment data. This is accomplished in one of two ways. First, when the metadata applies to all the traffic in a flow (i.e., transmit descriptor queue), scheduling metadata updates are sent through doorbell descriptors. Second, when the policy applies to individual segments, scheduling metadata is passed in segment descriptors. By passing metadata inline, there is no extra OS overhead to communicate new scheduling information when new flows arrive. Furthermore, Loom saves per-queue metadata in on-NIC lookup tables, which allows for future segments to be sent without any additional metadata. All of the information needed by the NIC (*e.g.*, the address of the queue in main memory) is already implicitly associated with the queue the segment is enqueued in.

Because the match+action pipeline processes packets sequentially, dynamically reconfiguring the scheduling hierarchy is straightforward. When processing a segment or doorbell, each stage in the pipeline saves provided configuration values (if present) to local SRAM as the stage processes the segment or doorbell. These values can include scheduling metadata, shared memory regions, algorithms, and traffic classes. With Loom's efficient doorbells this can be accomplished without additional PCIe writes.

4.4 Discussion

There are a wide spectrum of different NIC designs, ranging from NICs with specialized ASICs [3, 35], NICs with some FPGAs [36, 33, 22], NICs built from tiled or network processors [39, 13, 14, 58, 37], and purely virtual NICs [48, 1]. We believe that Loom's design is applicable to all of these different NIC types. While the PIFO abstraction is well suited for efficient hardware implementation (ASIC) [56], recent work has proved that all local scheduling algorithms are expressible with PIFOs [38]. Thus, the Loom design can be applied even to NICs with more flexible architectures. Similarly, all NIC types benefit from an efficient OS/NIC interface. Even in a virtual NIC, the use of doorbell queues reduces the number of memory regions that need to be polled by the NIC's backend.

Self-virtualizing NICs (*e.g.*, SR-IOV) can reduce the CPU overheads of virtual networking. Supporting SR-IOV with Loom is straightforward. Instead of providing per-core doorbells, Loom provides per-VCPU doorbells. VMs are then allocated their own doorbells and events. For security, queue creation/initialization is controlled by the hypervisor. However, common case operations like updating scheduling metadata are handled directly by the guest.

Similarly, Loom is also compatible with kernel bypass frameworks like RDMA, DPDK [28], and netmap [45]. Like netmap [45], we require such applications to call into the kernel to ring doorbells and configure queues. Even with a large number of kernel-bypass queues and applications, this still allows for efficient doorbell batching and scheduling of competing doorbells from different traffic classes.

5 Implementation

There are two major aspects to our implementation. The first is a compiler for the Loom policy DAG, and the second is a prototype Loom NIC. Loom is open source and available at `https://github.com/besteph/loom`.

For the policy DAG compiler, we made modifications to an existing compiler for scheduling trees written in Domino [5]. Scheduling policies are expressed as a DAG, and each node contains both an enqueue and a dequeue function. The Loom network policy is expressed in a restricted subset of C++. The output of the compiler is C++ code that combines generic PIFOs with custom scheduling algorithms.

We created a software prototype Loom NIC using the Berkeley Extensible Software Switch (BESS [1], formerly SoftNIC [25]). To interface with the OS, BESS loads a kernel module that registers a new Ethernet adapter. The driver for this adapter communicates with a backend userspace application that emulates the functionality of the Loom NIC hardware. The driver and backend in BESS communicate through descriptor queues implemented with shared memory.

We modified the BESS kernel driver to implement Loom's OS/NIC interface. We replaced BESS's per-core descriptor queues with per-flow descriptor queues and per-core doorbell queues. Also, we modified the descriptor format and the driver so that OS and flow-level metadata is included along with transmit descriptors and doorbells. We also identified and fixed a problem in the driver that caused excessive packet loss when transmitting packets.

To implement the compiled Loom policy in the NIC backend, we extended a C++ implementation of a pipeline of PIFOs [6]. We modified the shaping queues used for rate-limiting to support our DAG abstraction (Section 4). We also modified the model to support functions called on dequeue.

6 Methodology

For a baseline in our experiments, we compare our Loom prototype against three different BESS configurations. The first uses a single transmit descriptor queue for all packets (SQ), the second uses a descriptor queue per-core (MQ), and the third uses one queue per-flow (QPF) with round-robin scheduling between competing queues. Although SQ is not able to always drive line-rate in our experiments, it provides a baseline where the OS is able to enforce network policy. In contrast, while MQ and QPF are able to drive line-rate, they provide a baseline that is not able to enforce network policy.

In both the SQ and MQ configurations, when possible we configure Qdisc to try to enforce the network policy in software (Loom policies that use DAG rate-limits are not expressible in the Qdisc tree hierarchy). Because Qdisc does not allow for packet scheduling to be configured based on container, process, or socket ids, we rely on IP addresses and ports to express the network policy when using Qdisc.

We perform two different types of experiments to evaluate Loom. First, we use network-bound applications to profile the network behavior of our implementation. Specifically, we use the iperf3 [4] program to saturate network throughput and the sockperf [8] application to measure end-to-end latency. In these experiments, applications from different tenants are isolated by placing them in their own containers (cgroups). Second, we run real data center applications. We try to capture the performance of both latency and bandwidth sensitive applications. We use `memcached` for a latency-sensitive application. As a bandwidth-sensitive application, we use Spark with the TeraSort benchmark to perform a 25 GB shuffle. We compute throughput over a 50ms window.

We evaluate both types of experiments by sending data between two servers on CloudLab [20]. In order to stress Loom's packet processing, we use a 1500 byte MTU for all experiments, and do not use large segments, either for transmit segmentation offload (TSO) or large receive offload (LRO). The small packet size increases the number of packets per second that must be scheduled by a single core.

The software NIC in every experiment uses one core for packet transmission and one core for reception. This implies that the CPU utilization of the software prototype is 200%.

We use two different experiment configurations. In the first configuration (HW1), we use two servers with Intel X520 10GbE NICs, two Intel E5-2683 v3 14-core CPUs (Haswell) and 256GB of memory. In this configuration, all servers use the Loom prototype NIC for both sending and receiving.

In the second configuration (HW2), we use two servers with 40 Gbps Mellanox ConnectX-3 NICs, two Intel E5-2650 8-core CPUs (Haswell) and 64GB of memory. These experiments are asymmetric: only the server *sending* uses the Loom prototype NIC. The BESS SoftNIC [25], upon which our prototype is based, is currently unable to *receive* at 40 Gbps with a 1500 byte MTU with a single receive core (although it can receive at 40 Gbps with 9000B jumbo frames)[2]. To demonstrate that Loom can schedule and *transmit* packets at 40 Gbps with a 1500 MTU, we perform asymmetric experiments where the receiving host uses a vanilla driver.

7 Evaluation

First, we evaluate the ability of our Loom prototype to enforce network policies. Next, we evaluate the efficiency of our new OS/NIC interface. Finally, we evaluate the performance of two different DC applications when used with Loom.

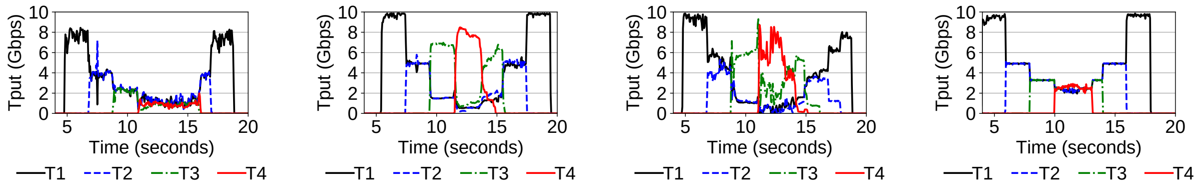
7.1 Policy Enforcement

Our first experiments demonstrate that our Loom prototype can isolate tenants. We perform the following experiment: There are five active tenants (T0-T4). Fair queuing (FQ) with equal shares is used to share bandwidth between tenants. The first tenant runs a single latency-sensitive application (sockperf). Then, in successive two-second intervals, another tenant starts running a bandwidth-hungry application. Starting with Tenant T1, each tenant *i* starts 4^i flows (4-256). Two seconds after the final tenant starts, the applications for each tenant successively finish at two-second intervals.

Figure 9 shows the throughput achieved over time by different tenants in the single queue (SQ), multi-queue (MQ), queue-per-flow (QPF), and Loom configurations. Figure 9a shows that SQ is able to approximately enforce this policy of tenant bandwidth isolation. However, because only a single transmit queue is used, SQ is not able to fully drive the 10 Gbps line-rate. Next, Figure 9b shows that MQ is not able enforce this policy. Instead, each successive tenant is able to use more than its fair share of bandwidth because it has more flows. As shown in Figure 9c QPF also leads to unfairness because the NIC's per-queue deficit round robin scheduling favors the tenant with the most flows. In contrast, Figure 9d shows that Loom is able to more precisely enforce the network policy than SQ while also driving full line-rate.

Next, Figure 10 shows the 90th percentile latency observed in each 250ms interval by tenant T0 in the same experiment

[2]The limit is not intrinsic to Loom. It arises from a BESS design choice to copy packet data in the kernel and not in the backend, and we are working with the BESS authors to address the problem.



(a) **Throughput with a single NIC queue (SQ).** (b) **Throughput with a NIC queue per-core (MQ).** (c) **Throughput with a NIC queue per-flow (QPF).** (d) **Throughput with Loom.**

Figure 9: **Aggregate throughput achieved by different tenants when each tenant should receive an equal share of bandwidth.**

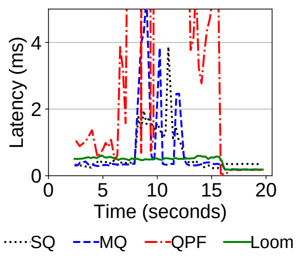


Figure 10: **90th Percentile latency over time for a latency sensitive application (T0) that is configured to fairly share bandwidth with up to four other bandwidth hungry applications (T1-T4) for SQ, MQ, and Loom.**

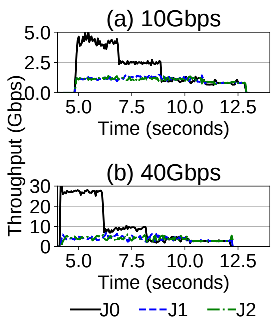


Figure 11: **Throughput for the different jobs from tenant T1 when only jobs J1 and J2 are subject to a combined 2.5 Gbps and 10 Gbps rate-limit on a 10 Gbps and 40 Gbps network, respectively.**

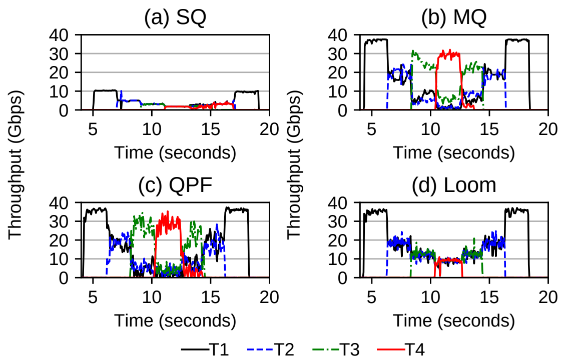


Figure 12: **Aggregate throughput achieved on a 40 Gbps network when the network policy is that each tenant should receive an equal share of bandwidth.**[4]

for the SQ, MQ, QPF, and Loom configurations. Because T0 uses less than its share of bandwidth, its packets are prioritized ahead of those from the other tenants. When the number of flows is small, Figure 10 shows that all of the approaches are able to provide similar latency. However, as the number of flows increases, only Loom is able to provide consistent low latency. QPF incurs the highest latency of any configuration because it uses more queues than SQ and MQ.

We compared the CPU utilization of Loom to existing designs: SQ, MQ, and QPF[3]. Loom performs similarly to MQ and QPF because there is little CPU scheduling work and no coordination across cores. In contrast, SQ has 25-300% higher CPU utilization due to complicated packet scheduling in software and contention for a single queue.

We study Loom's ability to enforce shaping policies where the rate-limit classes are different than the traffic classes used for scheduling. This experiment uses the same tenants as the previous one, but replaces Tenant T1. It now runs three competing jobs: J0 for Dst-1, and J1 and J2 for Dst-2. The traffic for Dst-2 is rate-limited to 10 Gbps, and for Dst-1 is not rate-limited. As before, every two seconds, an additional tenant starts flows, but in this case all stop at the same time.

The results of this experiment are shown in Figure 11. When the only active flows are from T1 (5-7 seconds), together, jobs J1 and J2 are appropriately rate-limited to 10 Gbps while J0 receives the remaining bandwidth. Further, according to the scheduling policy, jobs J1 and J2 fairly share the bandwidth available to their rate-limit class. However, after Tenant T2 and T3 have started flows at 9 seconds, tenant T1's fair share is 3.3 Gbps and the fair share of jobs J1 and J2 are less than 10 Gbps, we see that Jobs J1, J2, and J3 fairly share tenant T1's bandwidth.

From Figure 11, we see that Loom is able to enforce the policy DAG. In contrast, this policy is not expressible with the static Linux Qdisc or the scheduling tree policies used by prior work if the number of jobs is dynamic [56]. In a strict hierarchy, the rate limit must be applied before applying fair queuing across jobs, but the relative weight of the shaped and not shaped traffic changes with each new job.

Performance at 40 Gbps and Beyond: To demonstrate that our prototype scales with increasing line rates, we repeated some of the previous experiments at 40 Gbps line-rates with the small change that tenant T1 starts 16 flows instead of 4. These results are shown in Figure 12 and Figure 11(b). In both of these experiments, the trends stay the same. Figure 12 shows that SQ is only able to drive roughly ∼10 Gbps due to contention for the queue. In contrast, while MQ, QPF, and Loom are able to drive ∼37 Gbps, only Loom is able to

[3]Note that this CPU utilization is *in addition* to that used by our software prototype, which uses two cores.

[4]Our current prototype has a small throughput hit (1.3% median and 3.4% average) when compared with MQ that we are working on improving.

Number of Tenants	SENIC/QPF write/s	Loom write/s	% Reduction
32 Tenants	150K	128K	19.2%
64 Tenants	243K	205K	15.8%
96 Tenants	303K	263K	13.2%

Table 1: **The median number of PCIe writes per second generated by SENIC/QPF and Loom and the percent reduction in number of PCIe writes with Loom's batched doorbells.**

enforce network policy. Similarly, Figure 11(b) shows that Loom can rate-limit traffic while also driving a 40 Gbps link.

Based on these results, we believe that it should be possible to continue to scale Loom with increasing line-rates, *e.g.*, 100 and 200 GbE. These results demonstrate that a single CPU core can schedule 1500B packets at 40 Gbps, and we expect that a hardware NIC implementation should be able to exceed this performance. NPU-based NICs can parallelize computing ranks in enqueue and dequeue functions. Also, prior work has already demonstrated that custom hardware can enqueue/dequeue a billion packets per-second [56].

7.2 OS/NIC Interface

To evaluate Loom's batched doorbells, we estimate the difference in number of PCIe writes using Loom's batched doorbells and a NIC that has a queue-per-core with unbatched doorbells, such as SENIC [43] and the ConnectX-4. We instrument Linux to count how many doorbells Loom would use, and how many extra doorbell rings are needed without batching. With Loom, there is one PCIe write to the doorbell queue for a batch of segments added to any queue by a single core; this is approximately the same Linux behavior with a queue-per-core, which adds a batch of segments from multiple flows to a single queue. We calculate the number of PCIe doorbell writes in Loom as the number of `skbs` for which the `xmit_more` flag is false, indicating the OS has no more data to enqueue. Without batching, additional PCIe doorbell writes are needed for each different flow. We calculate the number of additional writes as the number of times a segment in a batch comes from a different flow than the preceding packet, indicating it would be sent on a different queue.

We evaluate the interface efficiency with a variable number of tenants each with 16 active flows sending traffic with iperf3. We note that this experiment is a best case scenario because it uses long-lived flows that are able to benefit from segmentation offload, which reduces the total number of segments sent to the NIC. Table 1 shows total number of writes per second and the percent reduction in number of PCIe writes with Loom when compared with current approaches. In these experiments, the total number of writes generated by the one flow per-queue approach varies from 150K–303K writes per second. Even with this benign workload, batched doorbells can reduce the number of writes by up to 19.2%. As part of future work, we are working to improve batching in Linux, which we expect can lead to even further reductions.

Next, because the benefits of batched doorbells are work-

Line-rate	64KB Batch SENIC/QPF write/s	64KB Batch Loom write/s	256KB Batch SENIC/QPF write/s	256KB Batch Loom write/s
10 Gbps	833K	19K	833K	4.8K
40 Gbps	3.3M	76K	3.3M	19K
100 Gbps	8.3M	191K	8.3M	48K

Table 2: **The number of PCIe writes per second generated by SENIC/QPF and Loom given a worst-case traffic pattern.**

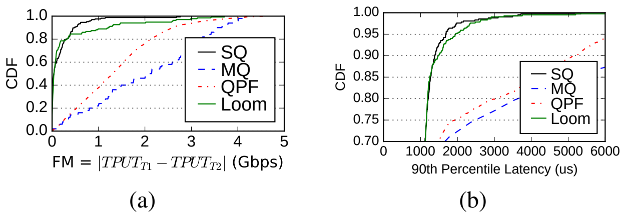


Figure 13: **Hierarchical policy performance. (a) CDF of fairness for Tenants 1 and 2. (b) CDF of memcached latency**

load dependent, we also performed a worst case analysis to estimate the benefits of Loom's approach. We approximate an RPC-style workload with many clients and assume that each 1500B packet in a batch is sent from a different flow. In this scenario, existing approaches such as SENIC that use a queue per flow generate a write per packet, while Loom generates one write per *batch*. Table 2 shows computed write rates of SENIC/QPF and Loom for different batch sizes and line-rates. At 40 Gbps with 64KB batches, SENIC generates 3.3M writes/second, while Loom causes only 76,000, a 43x reduction! For context, prior work found that the Intel XL710 40 Gbps NIC cannot drive line-rate when the OS generates more than 3.3M writes per second (*i.e.,* a single doorbell is rung per 1500B packet) [46].

This analysis demonstrates that existing one flow per-queue approaches will have difficulty driving increasing line-rates for workloads with many short flows. Additionally, this analysis shows that Loom can reduce PCIe overheads by increasing the batch size, which is not possible with existing one flow per-queue approaches under some workloads.

Finally, we also note that current approaches would generate even more PCIe writes than we estimate as flows start and stop to update scheduling metadata and the network policy. In contrast, configuration updates in Loom do not generate any additional updates as they are inlined with data.

7.3 DC Applications

We also used applications that are not always network bound to evaluate Loom. We show that Loom addresses the problems associated with using multiqueue NICs demonstrated in in Section 2. We omit most graphs due to lack of space.

Memcached rate limit. First, we found that despite using multiple queues, Loom can accurately rate-limit memcached traffic to 2 Gbps in the experiment from Figure 2.

Equal shares. When two Spark jobs are active (cf. Figure 3), Loom is able to fairly schedule traffic from the competing jobs, even as different flows start and stop. In contrast, MQ and QPF lead to job-level unfairness. SQ provides the same fairness as Loom, but is not able to drive line rate.

Priority. We evaluated Loom's ability to prioritize flows from one Spark job over those of another. We find that Loom can enforce this policy while MQ cannot. With Loom, the average and 90th percentile job completion times for the high-priority Job are 51.8s and 59.2s, respectively. In contrast, with MQ, the average and 90th percentile job completion times are 69.38s and 151.6s, respectively.

Multi-workload priority. We conducted an experiment where a single tenant is running two multi-threaded and multi-process applications with different performance requirements (memcached and Spark). The policy is that all of the traffic from memcached should have strict priority over that from Spark. We measure the impact that traffic from Spark has on memcached latency by examining the 90th percentile latency each 1 second interval. Both SQ and Loom isolate memcached from the Spark job, but MQ and QPF have 2.5X worse latencies in at least 25% of the intervals. Thus, even though each application thread has its own core in MQ and each flow has its own queue in QPF, neither is able to ensure that memcached response times are not impacted by a competing Spark job.

Hierarchical policy. Figure 13 shows that Loom can enforce hierarchical policies that cannot be enforced by DCB. In this experiment, tenants T1 and T2 are allocated equal shares of bandwidth. Tenant T1 runs memcached and Spark with the local policy that memcached has strict priority over Spark. Tenant T2 only runs Spark. 1's memcached latency should not be impacted by T1's Spark job, and each tenant should receive at most their fair share of throughput when they are both active. Figure 13a plots per-tenant unfairness, and Figure 13b plots the 90th percentile memcached request latency in each one second interval during the experiment. These figures show that Loom provides both isolation and fairness, while MQ and QPF cannot.

8 Related Work

SENIC [43] is related to Loom because it provides each flow with its own transmit descriptor queue and uses a doorbell FIFO to notify the NIC of new segments. However, SENIC only provides a single shared doorbell FIFO that requires synchronization across multiple cores. Further, SENIC writes individual 16B queue updates, and SENIC does not support programmable policies.

The ConnectX-4 [35] (CX-4) NIC is also similar to Loom. It supports many hundreds of thousands of descriptor queues and uses per-CPU registers for doorbell updates. However, individual doorbells and configuration updates need to be written to the NIC one at a time. The CX-4 also does not support hierarchical or programmable policies.

Titan [57] and MQFQ [26] can both improve fairness on multiqueue NICs. However, neither can enforce traffic priorities or hierarchical policies, and Titan incurs high scheduling update overheads with dynamic workloads.

Carousel [48] and PSPAT [46] both use multiple dedicated CPU cores to onload packet scheduling. However, both approaches still need to use multiple independent cores to drive 100 Gbps line-rates. Although they both use as few cores as possible, this does not solve the problems associated with multiqueue NICs. Further, Carousel can only express rate-limits, and, while PSPAT can express the same policies as Qdisc, it is still not able to express some policies that use per-destination rate-limits.

Like PSPAT [46], Neugebauer *et. al* [40] also benchmark PCIe performance. They find that PCIe can significantly impact the performance of end host networking. Both of these projects motivate the need for an efficient OS/NIC interface.

Eiffel [49] improves the efficiency of software packet scheduling, and, like Loom, Eiffel also addresses some shortcomings of PIFOs. Eiffel and Loom are complementary. Eiffel would benefit from Loom if it is ever not able to drive line-rate with a single core, and Eiffel can be used to enforce Loom policy sub-trees in software.

FlexNIC [31] introduces new interface for *receiving* packets on a programmable NIC. Loom's focus on packet transmission makes it complementary to FlexNIC.

Silo [29] also finds that multiqueue NICs prevent per-destination rate-limits from being enforced. However, Silo cannot ensure that competing applications share NIC bandwidth according to a high-level policy and has difficulty driving high line-rates because it only uses a single NIC queue.

9 Conclusions

DCs need to enforce a high-level hierarchical network policy. However, with today's multiqueue NICs, this is not possible. To address this deficiency, we created *Loom*, a new NIC design that moves all per-flow scheduling decisions out of the OS and into the NIC. Loom contributes 1) a new policy DAG abstraction that can express per-destination rate-limits independent of scheduling traffic classes, 2) a new flexible programmable scheduling hierarchy designed for NICs, and 3) a new expressive and efficient OS/NIC interface. Together, these aspects enable Loom to completely offload all packet scheduling to the NIC with low CPU overhead while still driving increasing line-rates (100Gbps). For collocated tenants' applications, these benefits translate into reductions in latency, increases in throughput, and improvements in fairness for competing tenants and applications.

Acknowledgments: We would like to thank the anonymous reviewers and our shepherd Alex C. Snoeren for their thoughtful feedback. Brent Stephens, Aditya Akella, and Michael Swift are supported in part by the NSF grant CNS-1717039.

References

[1] BESS: Berkeley Extensible Software Switch. https://github.com/NetSys/bess.

[2] Bulk network packet transmission. https://lwn.net/Articles/615238/.

[3] Intel ethernet switch fm10000 datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-multi-host-controller-fm10000-family-datasheet.pdf.

[4] iperf3: Documentation. http://software.es.net/iperf/.

[5] pifo-compiler: Compiler for packet scheduling programs. https://github.com/programmable-scheduling/pifo-compiler.

[6] pifo-machine: C++ reference implementation for push-in first-out queue. https://github.com/programmable-scheduling/pifo-machine.

[7] qdisc: bulk dequeue support. https://lwn.net/Articles/615240/.

[8] sockperf: Network benchmarking utility. https://github.com/Mellanox/sockperf.

[9] xps: Transmit packet steering. https://lwn.net/Articles/412062/.

[10] Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Towards predictable datacenter networks. In *SIGCOMM* (2011).

[11] Ballani, H., Jang, K., Karagiannis, T., Kim, C., Gunawardena, D., and O'Shea, G. Chatty tenants and the cloud network sharing problem. In *NSDI* (2013).

[12] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F. A., and Horowitz, M. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *SIGCOMM* (2013).

[13] Cavium Corporation. Cavium CN63XX-NIC10E. http://cavium.com/Intelligent_Network_Adapters_CN63XX_NIC10E.html.

[14] Cavium Corporation. Cavium LiquidIO. http://www.cavium.com/pdfFiles/LiquidIO_Server_Adapters_PB_Rev1.2.pdf.

[15] Chole, S., Fingerhut, A., Ma, S., Sivaraman, A., Vargaftik, S., Berger, A., Mendelson, G., Alizadeh, M., Chuang, S.-T., Keslassy, I., Orda, A., and Edsall, T. dRMT: Disaggregated programmable switching. In *SIGCOMM* (2017).

[16] Chowdhury, M., and Stoica, I. Coflow: A networking abstraction for cluster applications. In *HotNets* (2012).

[17] Chowdhury, M., and Stoica, I. Efficient coflow scheduling without prior knowledge. In *SIGCOMM* (2015).

[18] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing data transfers in computer clusters with orchestra. In *SIGCOMM* (2011), ACM.

[19] Chowdhury, M., Zhong, Y., and Stoica, I. Efficient coflow scheduling with Varys. In *SIGCOMM* (2014).

[20] Cloudlab. http://cloudlab.us/.

[21] Data Center Bridging Task Group. http://www.ieee802.org/1/pages/dcbridges.html.

[22] Exablaze. ExaNIC V5P. https://exablaze.com//exanic-v5p.

[23] Flajslik, M., and Rosenblum, M. Network interface design for low latency request-response protocols. In *USENIX ATC* (2013).

[24] Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., Lin, Z.-W., and Kurien, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM* (2015).

[25] Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. SoftNIC: A software NIC to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[26] Hedayati, M., Scott, M. L., Shen, K., , and Marty, M. Multi-queue fair queuing. Tech. Rep. UR CSD / 1005, Department of Computer Science, University of Rochester, October 2018. http://hdl.handle.net/1802/34380.

[27] Intel. Intel 82599 10 GbE controller datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[28] Intel Data Plane Development Kit. http://dpdk.org.

[29] Jang, K., Sherry, J., Ballani, H., and Moncaster, T. Silo: Predictable message latency in the cloud. In *SIGCOMM* (2015).

[30] Jeyakumar, V., Alizadeh, M., Mazieres, D., Prabhakar, B., Kim, C., and Greenberg, A. EyeQ: Practical network performance isolation at the edge. In *NSDI* (2013).

[31] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High performance packet processing with FlexNIC. In *ASPLOS* (2016).

[32] Kumar, A., Jain, S., Naik, U., Kasinadhuni, N., Zermeno, E. C., Gunn, C. S., Ai, J., Carlin, B., Amarandei-Stavila, M., Robin, M., Siganporia, A., Stuart, S., and Vahdat, A. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM* (2015).

[33] Li, J., Michael, E., Sharma, N. K., Szekeres, A., and Ports, D. R. K. Just say NO to paxos overhead: Replacing consensus with network ordering. In *OSDI* (2016).

[34] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving resource efficiency at scale. In *ISCA* (2015).

[35] Mellanox Technologies. ConnectX-4 VPI. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.

[36] Mellanox Technologies. Innova - 2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf.

[37] Mellanox Technologies. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[38] Mittal, R., Agarwal, R., Ratnasamy, S., and Shenker, S. Universal packet scheduling. In *NSDI* (2016).

[39] Netronome. NFP-6xxx flow processor. https://netronome.com/product/nfp-6xxx/.

[40] Neugebauer, R., Antichi, G., Zazo, J. F., Audzevich, Y., López-Buedo, S., and Moore, A. W. Understanding PCIe performance for end host networking. In *SIGCOMM* (2018).

[41] Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., and Stoica, I. FairCloud: Sharing the network in cloud computing. In *SIGCOMM* (2012).

[42] Qiu, Z., Stein, C., and Zhong, Y. Minimizing the total weighted completion time of coflows in datacenter networks. In *SPAA* (2015).

[43] Radhakrishnan, S., Geng, Y., Jeyakumar, V., Kabbani, A., Porter, G., and Vahdat, A. SENIC: Scalable NIC for end-host rate limiting. In *NSDI* (2014).

[44] Raghavan, B., Vishwanath, K., Ramabhadran, S., Yocum, K., and Snoeren, A. C. Cloud control with distributed rate limiting. In *SIGCOMM* (2007).

[45] Rizzo, L. netmap: A novel framework for fast packet I/O. In *USENIX ATC* (2012).

[46] Rizzo, L., Valente, P., Lettieri, G., and Maffione, V. PSPAT: Software packet scheduling at hardware speed. `http://info.iet.unipi.it/~luigi/pspat/`. Preprint; accessed May 31 2017.

[47] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[48] Saeed, A., Dukkipati, N., Valancius, V., Lam, T., Contavalli, C., and Vahdat, A. Carousel: Scalable traffic shaping at end-hosts. In *SIGCOMM* (2017).

[49] Saeed, A., Zhao, Y., Dukkipati, N., Ammar, M., Zegur, E., a Khaled Harras, and Vahdat, A. Eiffel: Efficient and flexible software packet scheduling. In *NSDI* (2019).

[50] Shieh, A., Kandula, S., Greenberg, A., and Kim, C. Sharing the data center network. In *NSDI* (2011).

[51] Shinde, P., Kaufmann, A., Kourtis, K., and Roscoe, T. Modeling NICs with Unicorn. In *PLOS* (2013).

[52] Shinde, P., Kaufmann, A., Roscoe, T., and Kaestle, S. We need to talk about NICs. In *HotOS* (2013).

[53] Shreedhar, M., and Varghese, G. Efficient fair queueing using deficit round robin. In *SIGCOMM* (1995).

[54] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hölzle, U., Stuart, S., and Vahdat, A. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *SIGCOMM* (2015).

[55] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM* (2016).

[56] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., and McKeown, N. Programmable packet scheduling at line rate. In *SIGCOMM* (2016).

[57] Stephens, B., Singhvi, A., Akella, A., and Swift, M. Titan: Fair packet scheduling for commodity multiqueue NICs. In *USENIX ATC* (2017).

[58] Tilera. Tile Processor Architecture Overview For the TILE-GX Series. `http://www.mellanox.com/repository/solutions/tile-scm/docs/UG130-ArchOverview-TILE-Gx.pdf`.

[59] Zhao, Y., Chen, K., Bai, W., Tian, C., Geng, Y., Zhang, Y., Li, D., and Wang, S. RAPIER: Integrating routing and scheduling for coflow-aware data center networks. In *INFOCOM* (2015).