SENIC: Scalable NIC for End-Host Rate Limiting

Sivasankar Radhakrishnan*, Yilong Geng+, Vimalkumar Jeyakumar+,
Abdul Kabbani†, George Porter*, Amin Vahdat†*

* University of California, San Diego + Stanford University † Google Inc.

{sivasankar, gmporter, vahdat}@cs.ucsd.edu {gengyl08, jvimal}@stanford.edu akabbani@google.com

Abstract

Rate limiting is an important primitive for managing server network resources. Unfortunately, software-based rate limiting suffers from limited accuracy and high CPU overhead, and modern NICs only support a handful of rate limiters. We present SENIC, a NIC design that can natively support 10s of thousands of rate limiters—100x to 1000x the number available in NICs today. The key idea is that the host CPU only classifies packets, enqueues them in per-class queues in host memory, and specifies rate limits for each traffic class. On the NIC, SENIC maintains class metadata, computes the transmit schedule, and only pulls packets from host memory when they are ready to be transmitted (on a real time basis). We implemented SENIC on NetFPGA, with 1000 rate limiters requiring just 30KB SRAM, and it was able to accurately pace packets. Further, in a memcached benchmark against software rate limiters, SENIC is able to sustain up to 250% higher load, while simultaneously keeping tail latency under 4ms at 90% network utilization.

1 Introduction

Today's trend towards consolidating servers in dense data centers necessitates careful resource management. It is hence unsurprising that there have been several recent proposals to manage and allocate network bandwidth to different services, tenants and traffic flows in data centers. This can be a challenge given the bursty and unpredictable nature of data center traffic, which has necessitated new designs for congestion control [29].

Many of these recent proposals can be realized on top of a simple substrate of *programmable rate limiters*. For example, Seawall [38], Oktopus [4], EyeQ [12] and Gatekeeper [35] use rate limiters between pairs of communicating virtual machines to provide tenant rate guarantees. QCN [1] and D^3 [40] use explicit network feedback to rate limit traffic sources. Such systems need to support thousands of rate limited flows or traffic classes, especially in virtual machine deployments.

Unfortunately these new ideas have been hamstrung by the inability of current NIC hardware to support more than a handful of rate limiters (e.g., 8–128) [11, 18]. This has resulted in delegating packet scheduling functionality to software, which is unable to keep up with line rates, while diverting CPU resources away from application processing. As networks get faster, this problem will only get worse since the capabilities of individual cores will likely not increase. We are left with a compromise between precise hardware rate limiters that are few in number [14, 38] and software rate limiters that support more flows but suffer from high CPU overhead and burstiness (see Table 1). Software rate limiters also preclude VMs from bypassing the hypervisor for better performance [20, 24].

Property	Hardware	Software
Scales to many classes	×	✓
Works at high link speeds	✓	×
Low CPU overhead	✓	×
Precise rate enforcement	✓	×
Supports hypervisor bypass	✓	×

Table 1: Pros and cons of current hardware and software based approaches to rate limiting.

The NIC is an ideal place to offload common case or repetitive network functions. Features such as segmentation offload (TSO), and checksum offload are widely used to improve CPU performance as we scale communication rates. However, a key missing functionality is scalable rate limiting.

In this work, we present SENIC, a NIC architecture that combines the scalability of software rate limiters with the precision and low overhead of hardware rate limiters. Specifically, in hardware, SENIC supports 10s of thousands of rate limiters, 100–1000x the number available in today's NICs. The key insight in SENIC is to invert the current duties of the host and the NIC: the OS stores packet queues in host memory, and classifies packets into them. The NIC handles packet scheduling and proactively pulls packets via host memory DMA for transmission. This late-binding enables SENIC to maintain transmit queues for many classes in host memory, while the NIC enforces precise rate limits in real-time.

This paper's contributions are: (1) identifying the limitations of current operating system and NIC capabilities, (2) the SENIC design that provides scalable rate limiting with low CPU overhead, and supports hypervisor bypass, (3) a unified scheduling algorithm that enforces strict rate limits and gracefully falls back to weighted sharing if the link is oversubscribed, and (4) evaluating SENIC through implementation of a software prototype and a hardware 10G-NetFPGA prototype. Our evalua-

tion shows that SENIC can pull packets on-demand and achieve (nearly) perfect packet pacing. SENIC sustains 43–250% higher memcached load than current software rate limiters, and achieves low tail latency under 4ms even at high loads. SENIC isolates memcached from bandwidth intensive tenants, and sustains the configured rate limits for all tenants even at high loads (9Gb/s), unlike current approaches.

2 Motivation

We motivate SENIC by describing two capabilities which rely on scalable rate limiting, then describe the limitations of current NICs which prevent these capabilities from being realized.[1]

2.1 The Need For Scalable Rate Limiting

Scalable rate limiting is required for network virtualization as well as new approaches for data center congestion control, as we now describe.

Network Virtualization: Sharing network bandwidth often relies on hierarchical rate limiting and weighted bandwidth sharing. For example, Gatekeeper [35], and EyeQ [12] both rate limit traffic between every communicating source-destination VM pair, as well as use weighted sharing across source VMs on a single machine. With greater server consolidation and increasing number of cores per server, the number of rate limiters needed is only expected to increase.

To quantify the number of rate limiters required for network virtualization, we observe that Moshref et al. [22] cite the need for 10s of thousands of flow rules per server to support VM-to-VM rules in a cluster with 10s of thousands of servers. Extending these to support rate limits would thus necessitate an equal number of rate limiters. For example, if there are 50 VMs/server, each communicating with a modest 50 other VMs, we need 2500 rate limiters to provide bandwidth isolation. Furthermore, supporting native hardware rate limiting is necessary, since VMs with latency sensitive applications may want to bypass the hypervisor entirely [20, 24].

Data Center Congestion Control: Congestion control has typically been an end-host responsibility, as exemplified by TCP. Bursty correlated traffic at high link speeds, coupled with small buffers in commodity switches can result in poor application performance [29]. This has led to the development of QCN [1], DCTCP [2], HULL [3], and D^3 [40] to demonstrate how explicit network feedback can be used to pace or rate limit traffic sources and reduce congestion. In the limit, each flow (potentially thousands [7]) needs its own rate limiter.

[1]The motivation for this work appeared in an earlier workshop paper [30].

2.2 Limitations of Current Systems

Today, rate limiting is performed either (1) in hardware in the NIC, or (2) in software in the OS or VM hypervisor. We consider these alternatives in detail.

2.2.1 Hardware Rate Limiting

Modern NICs support a few hardware transmit queues (8-128) that can be rate limited. When the OS transmits a packet, it sends a doorbell request[2] to the NIC notifying it of the packet and the NIC Tx ring buffer to use. The NIC DMA's the packet descriptor from host RAM to its internal SRAM memory. The NIC uses an arbiter to compute the order in which to fetch packets from different Tx ring buffers. It looks up the physical address of the packet in the descriptor, and initiates a DMA transfer of the packet contents to its internal packet buffer. Eventually a scheduler decides when different packets are transmitted.

A straightforward approach of storing per-class packet queues on the NIC does not scale well. For instance, even storing 15KB packet data per queue for 10,000 queues requires around 150MB of SRAM, which is too expensive for commodity NICs. Likewise, storing large packet descriptor ring buffers for each queue is also expensive.

2.2.2 Software Rate Limiting

Operating systems and VM hypervisors support rate limiting and per-class prioritization; for example, Linux offers a configurable queueing discipline (QDisc) layer for enforcing packet transmission policies. The QDisc can be configured with traffic classes from which packets are transmitted by the operating system.

In general, handling individual packets in software imposes high CPU overhead due to lock contention and frequent interrupts for computing and enforcing the schedule. To reduce CPU load, the OS transfers packets to the NIC in batches, leveraging features like TSO. Once these batches of packets are in the NIC, the operating system loses control over packet schedules; packets may end up being transmitted at unpredictable times on the wire, frequently in large bursts (e.g., 64KB with 10Gb/s NICs) of back-to-back MTU-sized packets transmitted at the full line rate.

Quantifying Software Overheads: Accurate rate limiting is challenging at 10Gb/s and higher. For instance, at 40Gb/s, accurately pacing 1500B packets means sending a packet approximately every 300ns. Such accuracy is difficult to achieve even with Linux's high resolution timers, as servicing an interrupt can easily cost thousands of nanoseconds. To quantify the overhead of software rate limiting, we benchmarked Linux's

[2]A doorbell request is a mechanism whereby the network driver notifies the NIC that packet(s) are ready to be transmitted.

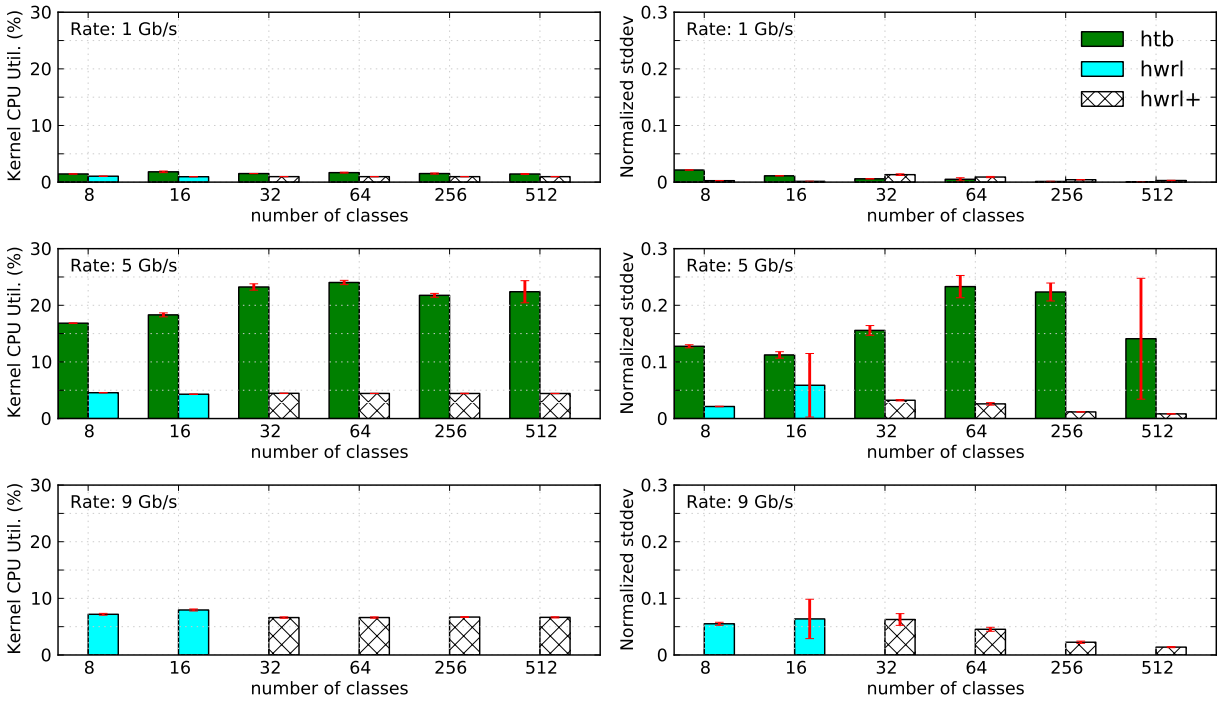

Figure 1: Comparison of CPU overhead and accuracy of software (Linux htb) and hardware (hwrl, hwrl+) rate limiting. At high rates (5Gb/s and 9Gb/s), hwrl ensures low CPU overhead and high accuracy, while htb is unable to drive more than 6.5Gb/s of aggregate throughput. Accuracy is measured as the ratio between the standard deviation of successive packet departure time differences, to the ideal. For instance, at 0.5Gb/s, 1500B packets should depart at times roughly 24us apart, but a "normalized stddev" of 0.2 means the observed deviation from 24us was as much as ∼4.8us.

Hierarchical Token Bucket (htb), and compared it to the hardware rate limiter (hwrl) on an Intel 82599 NIC. The tests were conducted on a dual 4-core, 2-way hyperthreaded Intel Xeon E5520 2.27GHz server running Linux 3.6.6.

We use userspace UDP traffic generators to send 1500B packets, and compare htb and hwrl on two metrics—OS overhead and accuracy—for varying number of classes. Each class is allocated an equal rate (total rate is 1Gb/s, 5Gb/s, or 9Gb/s). When the number of classes exceeds the available hardware rate limiters (16 in our setup), we assign classes to them in a round robin fashion (shown as hwrl+). OS overhead is the total fraction of CPU time spent in the kernel across all cores, and includes overheads in the network stack, packet scheduling, and servicing interrupts. To measure how well traffic is paced, we use a hardware packet sniffer at the receiver, which records timestamps with a 500ns precision. These metrics are plotted in Figure 1; the shaded bars indicate that many classes are mapped to one hardware rate limiter (hwrl+).

These experiments show that implementations of rate limiting in hardware are promising and deliver accurate rate limiting at low CPU overheads. However, they only offer few rate limiters, in part due to limited buffering on the NIC. Figure 1 shows that htb, while scalable in terms of the number of queues supported, is unable to pace packets at 9Gb/s, resulting in inaccurate rates.

3 Design

In the previous section, we described limitations of today's software and hardware approaches to rate limiting. The primary limitation in hardware today is scalability on the transmit path; we do not modify the receive path. In light of this, we now describe the design of the basic features in SENIC, and defer more advanced NIC features to §5. We begin with the service model abstraction.

3.1 Service Model

SENIC has a simple service model. The NIC exposes multiple transmit queues (classes), each with an associated rate limit. When the sum of rate limits of active classes does not exceed link capacity, each class is restricted to its rate limit. When it exceeds link capacity (i.e., the link is oversubscribed), SENIC gracefully shares the capacity in the ratio of class rate limits.

Entry	Bytes	Description
Queue management		
`ring_buffer`	4	Aligned address of the head of the ring buffer
`buffer_size`	2	Size of ring buffer (entries)
`head_index`	2	Index of first packet
`tail_index`	2	Index of last packet
Head packet descriptor		
`head_paddr`	8	Address of the first packet
`head_plen`	2	Length of the first packet (B)
`pkt_offset`	2	Next segment offset into the packet (for TSO)
Scheduler state (say for token bucket scheduler)		
`rate_mbps`	2	Rate limit for the queue
`tokens_bytes`	2	Number of bytes that can be sent from the queue without violating rate limit
`timestamp`	4	Last timestamp at which tokens were refreshed

Table 2: Per-class metadata in NIC SRAM. Total size=30B

3.2 CPU and NIC Responsibilities

To enforce a service model, we need a packet scheduler, and must store state for all classes. The state and functionality are spread across the CPU/host and the NIC.

State: Memory on the NIC (typically SRAM) is expensive, and we therefore use it to only store metadata about the classes. To store packet queues, SENIC leverages the large amount of host memory. Table 2 shows an example class metadata structure; the total size for storing 10,000 classes is about 300kB of SRAM. Note that the Myricom 10Gb/s NIC has 2MB SRAM [23].

Functionality: At a high level, the CPU classifies and enqueues packets in transmit queues, while the NIC computes a schedule that obeys the rate limits, pulls packets from queues in host memory using DMA, and transmits them on to the wire. The NIC handles all real time per-packet operations and transmit scheduling of packets from different classes based on their rate limits. This frees up the CPU to batch network processing, which reduces overall CPU utilization. This architecture is illustrated in Figure 2, which we now describe in detail.

3.2.1 CPU Functionality

As in current systems, the OS manages the NIC and initializes the device, creates/deletes classes, and configures rate limits. The OS is also in charge of classifying and enqueueing packets in appropriate queues. In both cases, the OS communicates with the NIC through memory-mapped IO. For instance, when the OS enqueues a packet (or a burst of packets) into a queue, it notifies the NIC through a special doorbell request that it writes to a device-specific memory address.

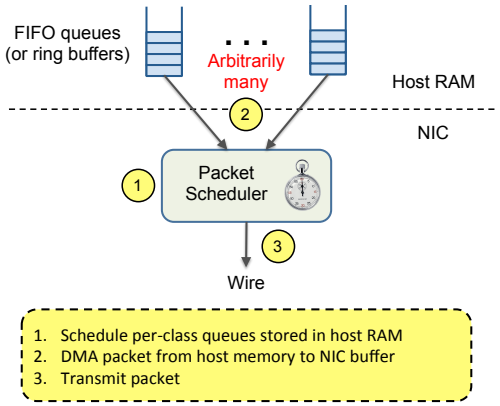

Figure 2: SENIC — "Schedule and Pull" model.

3.2.2 NIC Functionality

The NIC is responsible for all per-packet real time operations on transmit queues. Since it has limited hardware buffer resources, the NIC first computes the transmit schedule based on the rate limits. It then chooses the next packet that should be transmitted, and DMAs the packet from the per-class queue in host memory to a small internal NIC buffer for transmitting on to the wire. Figure 3 shows a schematic of the SENIC hardware and related interfaces from software.

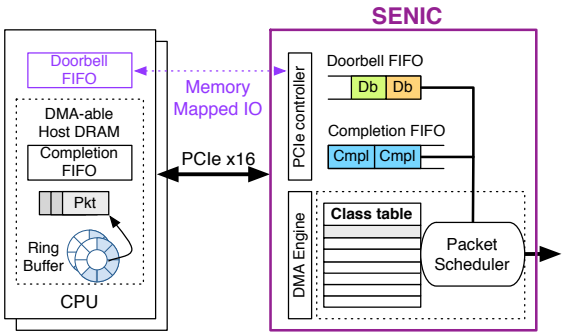

Figure 3: SENIC hardware design. Once the NIC DMAs a packet from host memory, there is further processing (e.g. checksum offloads) before the packet is transmitted on the wire. This paper focuses on the scheduler and the NIC interaction with the software stack.

Metadata: The NIC maintains state about traffic classes to enforce rate limits. In the case of a token bucket scheduler, each class maintains metadata on the number of tokens, and the global state is a list of active classes with enough tokens to transmit the next packet. The memory footprint is small, easily supporting 10,000 or more traffic classes with a few 100kB of metadata.

Scheduling: The NIC schedules and pulls packets from host memory on demand at link speed. Packets are not pulled faster, even though PCIe bandwidth between the NIC and CPU is much higher. This late bind-

ing reduces the size of NIC hardware buffers required for storing packets. It also avoids head-of-line blocking, and allows the NIC to quickly schedule newly active classes or use updated rate limits. This offloading of scheduling and real time work to the NIC is what enables SENIC to accurately enforce rate limits even at high link speeds.

Other Functionality: The NIC does more tasks than just state management and rate limiting. After the packet is DMA'd onto NIC memory, there is a standard pipeline of operations that we leave unmodified. For instance, NICs support TCP and IP checksum offloading, VLAN encapsulation, and send completions to notify the CPU when it can reclaim packet memory.

4 Packet Scheduling in SENIC

SENIC employs an internal scheduler to rate limit traffic classes. The task of packet scheduling can be realized using a number of algorithms such as Deficit Round Robin (DRR) [39], Weighted Fair Queueing (WFQ) [9], Worst-case Fair weighted Fair Queueing (WF^{2}Q) [5], or simple token buckets. The choice of algorithm impacts the sharing model, and packet delay bounds. For instance, token buckets support rate limits, but DRR is work-conserving; simply arbitrating across token buckets in a DRR-like fashion can result in bursty transmissions [6].

In this section, we start with our main requirements to pick the appropriate scheduling algorithm. We desire hierarchical rate limits, so the above work-conserving algorithms (DRR, WFQ, etc.) do not directly suit our needs. We now describe a unified scheduling algorithm that supports hierarchies and rate limits.

4.1 SENIC Packet Scheduling Algorithm

Recall that the service model exposed by SENIC is rate limits on classes, with fallback to weighted sharing proportional to the class rates. We begin by describing a scheduling algorithm which can enforce this service model. We leverage a virtual time based weighted sharing algorithm, WF^{2}Q+ [6], and modify its *system virtual time (V)* computation to support strict rate limiting with a fallback to weighted sharing. The algorithm computes a *start (S)* and *finish (F)* time for every packet based on the class rate w_i. Packets with $S \leq V$ are considered *eligible*, and the algorithm transmits eligible packets in increasing order of their finish times.

Computing Start and Finish Time: Since each class is a FIFO, the start and finish times are maintained only for the packets at the head of each transmit queue. The start time S_i of a class C_i is only updated when a packet is dequeued from that class or a packet is enqueued into a previously empty class. The finish time F_i is updated whenever S_i is updated. S_i and F_i for each flow C_i are computed in the same way as in WF^{2}Q+, as follows:

$$S_i = \begin{cases} \max(F_i, V_{enq}) & \text{on enqueue into empty queue} \\ F_i & \text{on dequeue} \end{cases}$$

$$F_i = S_i + \frac{L}{w_i}$$

where V_{enq} is the *system time V* (described below) when the packet is enqueued, and L is the head packet's length.

System Time Computation: WF^{2}Q+ computes a work-conserving schedule where at least one class is always eligible to transmit data. To enforce strict rate limits, SENIC incorporates the notions of real time and the link drain rate (R) to compute the transmit schedule. The system time is increased by 1 unit (*bytetime*), in the time it takes to transmit 1B of data at link speed, and thus incorporates the link's known drain rate R (e.g. 10Gb/s).

SENIC supports graceful fallback to weighted sharing when the link is oversubscribed. When the link is oversubscribed, we slow the system time V down to reflect the marginal rate at which the active flows are serviced. Without loss of generality, let the rate limits of flows C_i be represented as fractions w_i of the link speed R. We define the *rate oversubscription factor* ϕ to be the sum of rate limits (weights) of currently backlogged classes or flows in the system; $\phi = 0$ when no flows are active. The scheduler modifies the system time V to slow down by the rate oversubscription factor and proceed at most as fast as the link speed. V is computed as:

$$V(0) = 0$$
$$V(t+\tau) = V(t) + R\tau \times \max(1, \phi)$$

where τ is a single packet transmission period, or contiguous link idle period, or the period between successive updates to ϕ. Given the system time, the start and finish times of all classes, we schedule packets in the same order as WF^{2}Q+, i.e. in order of increasing finish times among all eligible classes at the time of dequeueing.

Example: We now look at an example transmit schedule computed using these time functions. Assume a 10Gb/s link with two continuously backlogged classes C_1 and C_2 (with rate limits 4Gb/s and 2Gb/s respectively). The transmit schedule is shown in Figure 4. The values of S_i and F_i are computed using rate limits as a fraction of link speed (so w_1 = 0.4 and w_2 = 0.2). All packets are 1500B in length.

If we consider a single iteration (7500 bytetimes), C_1 transmits 3000B, C_2 transmits 1500B, and the link is idle for (750 + 2250 = 3000 bytetimes). Thus C_1 achieves 3000 / 7500 = 0.4 of link capacity and C_2 achieves 1500 / 7500 = 0.2 of link capacity. The link remains idle for 40% of the time in each iteration, thereby enforcing strict rate limits. Notice also that the packets are appropriately interleaved and accurately paced.

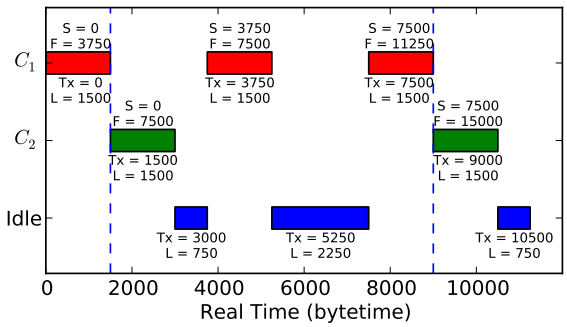

Figure 4: Transmit schedule example. Link is not oversubscribed. The interval between the vertical dashed lines indicates repeating sequence in the transmit schedule (only 1 repetition shown for clarity). S, F, L as defined in the text. T_x is the time when a particular packet transmission or idle period starts.

Delay Guarantees: The advantage of using virtual time based scheduling algorithms is that they offer strong per-packet delay guarantees. Specifically, WF^{2}Q+ guarantees that the finish time of a packet in the discretized system is no more than a bounded delay from an ideal fluid model system. SENIC's unified scheduling algorithm offers similar strong guarantees. Algorithms such as DRR do not have such strong guarantees [6].

4.2 Hierarchical bandwidth sharing

So far we discussed a flat rate limiting scheme. In practice, it may be desirable to group classes and enforce another rate limit on the group. For example, an approach useful in multi-tenant environments is a two level hierarchy where the first level implements strict rate limits for each VM on the server, and the second level provides weighted sharing between the flows originating from each VM.

It is possible to enforce any hierarchical allocation by modulating the rate limits of hardware traffic classes. Control logic in the hypervisor can measure demands and hardware counters, and adjust the rates based on preconfigured limits. We instead now describe an extension to the virtual time based scheduler described above to support a simple two level hierarchy.

Sharing Model: We define an L_1 (level 1) class as one which is directly attached to the root of the hierarchy. An L_2 (level 2) class is attached to an L_1 class. Each class is configured with a rate limit. The L_1 classes only support strict rate limits, i.e. sum of rate limits of active L_1 classes should not exceed link capacity. L_2 classes support strict rate limiting, but fallback to weighted sharing in the ratio of their rate limits when the active L_2 classes within an L_1 class oversubscribe the rate limit of that L_1 class. An L_1 class might be a leaf or an internal class while L_2 classes can only be leaves.

Start and Finish Time Computation: SENIC only computes time variables for leaf classes as packets are "enqueued" and "dequeued" only at the leaves. For L_1 leaf classes, the scheduler computes start and finish times as usual, using the rate limits of the respective classes. For each L_1 class, it maintains a rate oversubscription factor ϕ_{L_1}, of active L_2 classes within the L_1 class. For L_2 classes, to compute finish time, the scheduler scales the rate limits and uses the minimum of (1) the configured rate limit w_i of the L_2 class, and (2) the scaled rate limit of the parent L_1 class based on L_2's share, given as:

$$w_{i_{scaled}} = \min\left(w_i, w_{L_1} \times \frac{w_i}{\phi_{L_1}}\right)$$

System Time Computation: System time is purely based on real time and link drain rate R, as the L_1 classes are configured such that they never oversubscribe the link. This condition can be easily met even if weighted sharing is required at level 1 of the hierarchy, by simply having the host driver periodically measure demand and adjust the rate limits of the L_1 classes.

Summary: Driven by requirements to support rate limits, we described a scheduling algorithm incorporating both weighted sharing and rate limiting into one coherent algorithm. We also extended the algorithm to support two-level rate limits across classes and groups of classes. We realized the unified scheduling algorithm on top of QFQ [8], which in turn implements WF^{2}Q+ efficiently. The metadata structure for this QFQ based scheduler is around 40B per class, and it needs only 10kB of global state, thereby scaling easily to 10,000 classes.

5 Advanced NIC features

This section touches upon advanced features in today's NICs that are impacted by SENIC's design, and how we achieve similar functionality with SENIC.

5.1 OS and Hypervisor Bypass

Many applications benefit from bypassing the OS network stack to meet their stringent latency and performance requirements [13, 25]. Further, high-performance virtualized workloads benefit from bypassing the hypervisor entirely, and directly access the NIC [20, 24]. To support such requirements, modern NICs expose queues directly to user-space, and include features that virtualize the device state (ring buffers, etc.) through technologies like Single-Root IO Virtualization (SR-IOV [28]). We now describe how SENIC provides these features.

Configurable SR-IOV Slices or VNICs: SENIC leverages SR-IOV to expose multiple VNICs. Each

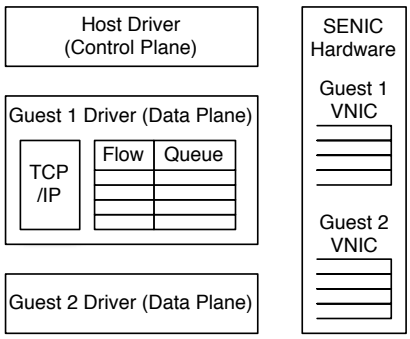

Figure 5: The SENIC architecture, with each guest given a virtualized slice of the NIC (VNIC) using SR-IOV.

VNIC is allocated a configurable number of queues, and guest VMs directly transmit and receive packets through the VNICs, as shown in Figure 5. Guest VMs are only aware of queues for their respective VNICs (which is standard SR-IOV functionality), thereby ensuring isolation between transmit queues of different guest VMs. A simple lookup table on the NIC translates VNIC queue IDs to actual queue IDs. A host SENIC driver provides the interface for the hypervisor to configure VNICs, allocate queues, and configure rate limits. A guest driver running in the VM provides a standard interface to enqueue packets into different queues on the VNIC.

Classifying Packets: SENIC relies on the operating system to classify and enqueue packets in the right traffic classes or queues. The host driver residing in the hypervisor maintains the packet classification table. It exports an OpenFlow [26] like API to configure traffic classes and rate limits. When SR-IOV is enabled, the hypervisor is bypassed in the datapath. SENIC therefore relies on the guest VM to perform packet classification.

The guest driver maintains a cached copy of the packet classification table. When the guest driver receives a packet from the network stack for transmission, it looks up its guest packet classification table for a match. If no match is found, it makes a hypercall to the hypervisor for a lookup and caches the matching rule. The actual mapping to the appropriate queue is also cached in the socket data structure to avoid repeated lookups for each packet of a flow. The hypervisor can also proactively setup rules in guest classification tables. Once the rules are cached in the guest, the hypervisor is completely bypassed during packet transmission.

Untrusted Guests: It may be unwise to trust guests to classify packets correctly. However, we argue this is not an issue. Even though SR-IOV ensures that a VM can only place packets in queues for its own VNIC, the guest may ignore the hypervisor-specified classification among its queues. We adopt a *trust-but-verify* approach to ensure that guest VMs do not cheat by directing packets to queues with higher rate limits. The key idea is that the hypervisor need not look at every packet to ensure rate limits are not violated, but instead only look at a sampled subset of packets. Since classification is used to provide QoS, sampling packet headers and verifying their classification is sufficient to identify violations. The administrator can be alerted to misbehaving guests, or they can be halted, or forced to give up SR-IOV, and rely on the hypervisor for future packet transmissions.

5.2 Other features

Below we describe few other features that are affected by SENIC's design.

Segmentation Offload: TCP Segmentation Offload (TSO) is a widely available NIC feature to reduce CPU load by transferring large (upto 64KB) TCP segments to the NIC, which are then divided into MTU sized segments and transmitted with appropriately updated checksums and sequence numbers. SENIC only pulls MTU sized portions of the packet on demand from host memory queues before transmission. This avoids long bursts from a single class, and enables better interleaving and pacing. SENIC augments per-queue metadata with a *TSO-offset* field that indicates which portion of the packet at the head of the queue remains to be transmitted. When interleaving packets, SENIC does not cache packet headers for each class on the NIC, thereby keeping NIC SRAM requirements low. When transmitting TSO packets, SENIC issues two DMA requests: one for the packet header, and another for the MTU sized payload based on TSO-offset.

Scatter-Gather: A related optimization is scatter-gather, where the NIC can fetch packet data spread across multiple memory regions, e.g., the header separately from the payload. In such cases, SENIC stores the location of the next segment to be transmitted for each queue and fetches descriptors and data on demand.

Handling Concurrency: The design assumed each transmit queue corresponds to one traffic class. To allow multiple CPU cores to concurrently enqueue packets to a class, the SENIC design is extended to support some number of queues (say 8) for each class. Round robin ordering is used among queues within a class, whenever the class gets its turn to transmit. This is easily accomplished by separately storing head and tail indices for each queue in the class metadata table, an active queue bitmap and round robin counter for each class.

Priority Scheduling: SENIC can easily also support strict priority scheduling between transmit queues of a class instead of round-robin scheduling. In this case, a priority encoder picks the highest priority active class. One use case is for applications to prioritize their traffic within a given rate limit.

6 Implementation

We have implemented two SENIC prototypes:

1. A software prototype using a dedicated CPU core to perform custom NIC processing. This implements the unified QFQ-based rate limiting and weighted sharing scheduler described in §4.1.
2. A NetFPGA-based hardware prototype designed to run microbenchmarks and evaluate the feasibility of pulling packets on demand from host memory for transmission. For engineering expediency, this prototype relies on a simpler, token bucket scheduler (without hierarchies).

We now describe both prototypes in detail. Both prototypes are available for download at `http://sivasankar.me/senic/`.

6.1 Software Prototype

The software prototype is implemented as a Linux kernel module with modest changes to the kernel. The scheduler is implemented in a new Linux queueing discipline (QDisc) kernel module. We also modified the Linux `tc` utility to enable us to configure the new QDisc module. As described in §4, SENIC's packet scheduling algorithm is implemented on top of the Quick Fair Queueing (QFQ) scheduler available in Linux.

Transmit Queues and Rate Limits: The SENIC QDisc maintains per-class FIFO transmit queues in host memory as linked lists. We configure classification rules via `tc`, and also set a rate limit for each class.

Enqueueing Packets: In Linux, when the transport layer wants to transmit a packet, it hands it down to the IP layer, which in turn hands it to the QDisc layer. When the QDisc receives a packet from IP, it first classifies the packet, then enqueues it in the corresponding queue, marking the class as active.

Dedicated CPU Core for Packet Scheduling: In today's kernel, the dequeue operation starts right after enqueue. However, to mimic NIC functionality, we modified the kernel so the enqueue call immediately returns to the caller, and dedicate a CPU core to perform all NIC scheduling (i.e. dequeueing). The dedicated CPU core runs a kernel thread that computes the schedule based on configured rate limits, and pulls packets from the active transmit queues when they should be transmitted. Packets are transferred to the physical NIC using the standard NIC driver. We disabled TSO to control the transmit schedule at a fine granularity and avoid traffic bursts.

6.2 NetFPGA Prototype

We now describe our SENIC hardware implementation on a NetFPGA [16]. The primary hardware components of SENIC are (a) the packet scheduler with the class table, (b) doorbell FIFOs to process notifications from the host, and (c) completion FIFOs to send notifications to the host. Each component maintains its own independent state machine and executes in parallel. Figure 6 below zooms into the operation of the packet scheduler. We now describe each component in detail.

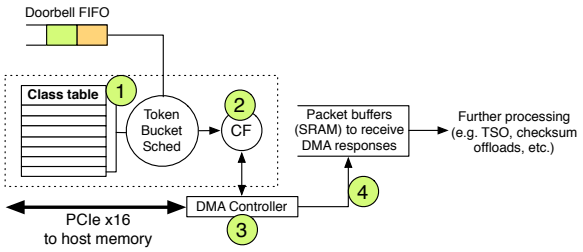

Figure 6: The 4 stages of scheduling a packet: (1) pick a class for dequeueing, (2) submit work-request to the class-fetch (CF) module, (3) DMA descriptors and packet payload from the class, (4) handoff packet payload for further processing.

6.2.1 Packet Scheduler

The scheduler operates on the class metadata table (SRAM block), and performs the following operations:

- It cycles through all active classes (i.e., classes with at least one enqueued packet), and determines if a class has enough tokens to transmit a packet (i.e., whether it is *eligible*). If not, the scheduler refreshes the class's tokens and continues with other classes.
- If the class is eligible, the scheduler submits a work-request to a 'class-fetch' (CF) module and disables the class. Each CF module has a small FIFO to accept requests from the scheduler.
- If the CF module's FIFO is full, the scheduler stalls and waits for feedback from the CF module.
- In parallel, the scheduler processes any pending doorbell requests that modify the class metadata table. For instance, if the doorbell request is an enqueue operation, the scheduler parses the class ID in the request and updates the class table.

6.2.2 Class-Fetch Module

The class-fetch (CF) module is given a class entry, and its task is to dequeue as many packets as possible until limited by (a) the tokens available for the class, or (b) the burst size of the class. The class entry only stores the descriptor for the first packet. Therefore the CF dispatches DMA requests to (a) fetch the descriptor of the next packet in the ring buffer, and (b) fetch the packet payload of the first descriptor stored in the class entry. The module then synchronously waits for the first DMA

to complete, and repeats the process until it exhausts the class tokens, or burst size. Finally, it issues (a) feedback to the scheduler with the new class entry state (updated tokens, tail pointer, and the first packet descriptor), and (b) a completion notification for the class.

The latency to make a scheduling decision, and the DMA fetch latency determine the maximum achievable throughput. We evaluate this in detail in §7.1.3.

6.2.3 Host Notifications

SENIC uses standard notification mechanisms to synchronize state between the NIC and the host: doorbell requests and completions. Doorbells update class state on the NIC (e.g., new packets and new rates), and completions notify the host about transmitted packets and processed doorbells. Doorbells and completions are stored in FIFO ring buffers, on the NIC and host respectively.

Doorbells: The doorbell is a 16B message written by the host to the memory mapped doorbell FIFO on the NIC. The FIFO is a circular buffer—the host enqueues at the tail while the NIC dequeues at the head. The host synchronizes the head index when it receives completions from the NIC, thereby freeing FIFO entries.

Completions: The NIC issues completions by DMA'ing an entry into the completion FIFO in host memory and interrupting the CPU. Each entry indicates (1) the class and number of packets transmitted from the class, or (2) the number of doorbell requests processed. This information is used by the host to reclaim packet memory, and doorbell FIFO entries. These event notifications are similar to BSD's kqueue mechanism [15].

Avoiding Write Conflicts: Note that the CF module's feedback, and host notifications both modify the class entry state. However, the feedback only modifies tokens, the first packet's length and address; the host notification only modifies the tail index. If the class's rate changes while it is being serviced, the new rate takes effect only the next iteration when the scheduler refreshes tokens.

7 Evaluation

This section dissects SENIC to answer the following aspects of the system:

- How scalable and accurate are the hardware rate limiters? We synthesized our hardware prototype with 1000 rate limiters. At 1Gb/s, we found the mean inter-packet timing was within 10ns of ideal, and the standard deviation was 191ns (less than 1.6% of the mean).
- How many packets should be pipelined for achieving line rate at various link speeds? This value depends on the scheduling and DMA latency, and the dominant factor is the DMA latency across the PCIe bus.
- How effective is SENIC at supporting high loads and delivering low latency compared to state of the art software rate limiters? We compare SENIC against Linux HTB and a Parallel Token Bucket (PTB) implementation in software (used in EyeQ [12]). We found that at very low load, all approaches have comparable latencies. But SENIC sustains 55% higher load compared to PTB, and 250% higher than HTB while keeping memcached 99.9th percentile latency under 3ms.
- How effectively can SENIC isolate different tenants—memcached latency sensitive tenants and a background bandwidth intensive UDP tenant? We found that SENIC could comfortably sustain the configured 3Gb/s of UDP traffic and nearly 6Gb/s of memcached traffic with tail latency under 4ms. However, HTB and PTB had trouble sustaining more than 1.4Gb/s of UDP traffic. SENIC sustains 233% higher memcached load compared to HTB and 43% higher than PTB.

7.1 Hardware Microbenchmarks

7.1.1 Scalability and Accuracy

Due to limitations on the number of outstanding DMA requests[3], and pipeline datawidth, we were unable to sustain more than 3Gb/s packet transmission rate, and we restrict our tests to rates less than 3Gb/s.

N	Rate	$\mu \pm \sigma$	Rel. error in μ
500	1Mb/s	12ms $\pm$ 7.1us	3.1×10^{-6}
1	10Mb/s	1.2ms $\pm$ 233ns	1.5×10^{-6}
10		1.2ms $\pm$ 240ns	1.5×10^{-6}
100		1.2ms $\pm$ 1.3μs	2.3×10^{-5}
1	100Mb/s	120μs $\pm$ 87ns	1.7×10^{-7}
10		120μs $\pm$ 173ns	1.6×10^{-6}
1	1Gb/s	11.25μs† $\pm$ 161ns	3.5×10^{-4}
3		11.25μs† $\pm$ 191ns	3.8×10^{-4}

Table 3: Rate limit accuracy as we vary the number of rate limiters N, and the rate per class. We see that SENIC is within 10^{-2}% of ideal even as we approach the maximum throughput we could push through the NetFPGA (3Gb/s).

Table 3 shows the rate limiting accuracy of one of the classes, as we vary the number of eligible classes on the NIC. We measure accuracy by timestamping every packet with a clock resolution of 10ns, and retrieving the inter-packet timestamp difference for packets of that one class. We compute the mean (μ) and standard deviation (σ), and also the relative error in μ as $|\mu_{\text{empirical}} - \mu_{\text{ideal}}|/\mu_{\text{ideal}}$. We see that SENIC very accurately enforces the configured rate even with 500 classes each operating at 1Mb/s.

[3] Our NetFPGA stalls the processor if it has more than 2 outstanding DMA requests. Others have reported a similar issue with the Virtex5 FPGA [41].

†**Note**: NetFPGA supports rates that are of form 12.8Gb/s/K, where K is an integer. Therefore, though we set the rate limit to 1Gb/s, the output will 12.8/12 Gb/s (1.067Gb/s), for which the inter-packet time is 11.25μs.

7.1.2 Scheduler Latency

We dig deeper into how long it takes for a scheduling operation in hardware. On the NetFPGA, the SRAM has a datawidth of 512 bits (64B), an access latency of 1 cycle, and enough bandwidth to support one operation (either a read or a write) every cycle. In the *worst case*, each scheduler iteration takes at most 5 cycles:

- 1 for reading the class metadata from SRAM.
- 1 for refreshing the tokens and CF-enqueue.
- 1 SRAM write for processing CF-feedback.
- 2 for processing a doorbell: 1 for reading the class metadata from SRAM, and 1 for updating class metadata and writing it back.

We synthesized our NetFPGA prototype at 100MHz (10ns per clock cycle), and therefore, it takes no more than 50ns to make a scheduling decision. We expect a production-quality NIC to have a higher clock rate, and thus a faster scheduler. For instance, the ASIC in Myricom 10Gb/s Ethernet NIC runs at a clock rate of 364.6MHz [23]. The QFQ based scheduler takes about twice as many cycles as simple token buckets [8], so with a higher clock rate, it can still complete in 50ns.

7.1.3 Maximum Per-Class Throughput

In this experiment, we first analyze the DMA latency which affects the achievable throughput per-class. We measure the time interval between sending a DMA request from the CF-module to fetch 16B from host memory, and receiving the response. We find that the average latency is $L = 1.25\mu$s ($\sigma = 40$ns) with the NetFPGA platform (using a second generation PCIe x8 bus). However, the number is often better with a production-quality NIC. For instance, the DMA latency on an Intel NIC was found to be close to 200ns [31].

Recall that the CF-module processes each class by issuing a DMA request for the class's second packet descriptor, followed by the request for the class's first packet payload. With a burst size of 1 packet per class, the maximum achievable throughput per class depends on the sum of DMA latency and scheduler latency. For instance, if the scheduler takes 50ns to dispatch a class to the CF module, the DMA latency to fetch a packet descriptor is 1250ns, and burst size is 1 packet, the maximum achievable throughput per-class is about 1500B (MTU) every 1300ns. Therefore, to achieve line rate

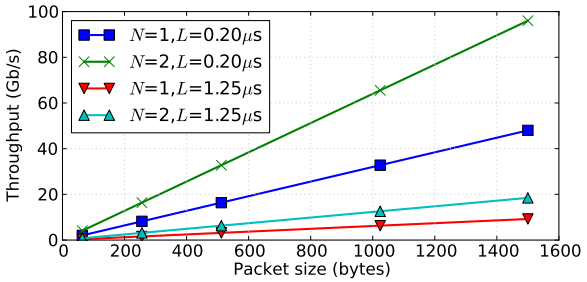

Figure 7: Maximum throughput per class as a function of the packet size, and the number N of CF modules operating in parallel, and the DMA latency L. We see that the achievable throughput on the NetFPGA ($L = 1.25\mu$s) with $N = 1$ is 9.23Gb/s with 1500B packets (if not for the DMA request constraints described earlier).

we can instantiate multiple CF modules, and the scheduler dispatches classes to them in parallel. Further, using TSO, or multiple queues per class enables higher throughput per class. Figure 7 shows the trend.

7.2 Software Macrobenchmarks

We ran experiments with our software based SENIC prototype to evaluate the application level performance when SENIC is used for rate limiting traffic.

7.2.1 Memcached

We conducted an experiment with several memcached tenants sharing a cluster—10 tenants on each machine in an 8 node cluster. Each node is a dual 4-core, 2-way hyperthreaded Intel Xeon E5520 2.27GHz server, with 24GB of RAM, a 10Gb/s NIC (Intel or Myricom), and running Linux 3.9.0. Each tenant was allocated 1 CPU hyperthread on each machine, and 2GB of RAM. One machine (M_{srv}) had 10 memcached server instances—1 for each tenant. We pre-populated them with 12B-key, 2KB-value pairs. Each of the other 7 machines (M_{cli}) ran 10 memcached client processes that sent `GET` requests to the respective tenant's memcached server instance.

Rate limits were configured for each memcached client-server pair. The total rate limit was 9.5Gb/s on M_{srv}, and 6Gb/s on M_{cli} machines. Each tenant got an equal share of the total rate, divided equally among its own destinations. These limits were chosen to be large enough that memcached would not be bandwidth limited. We ran experiments using HTB, PTB, and the SENIC software prototype.

We define the unit *rpstc*, requests per second per tenant per client, to denote the load on the system. For instance, 2,000 rpstc means each of the 7 client instances of each tenant generates a load of 2,000 req/s, resulting in a total load on M_{srv} of 140,000 req/s.

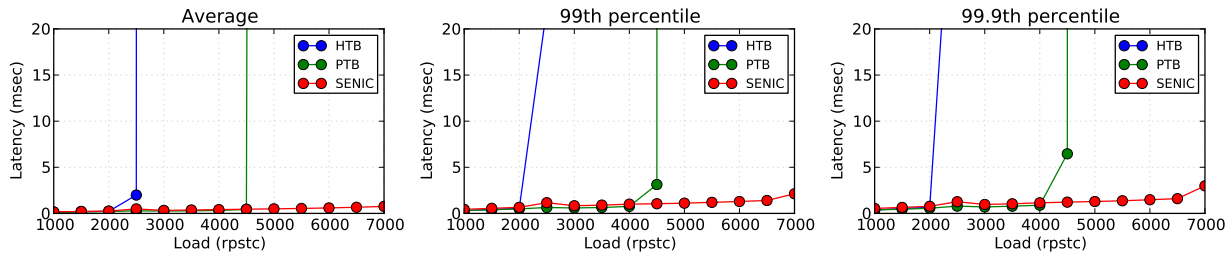

Figure 9: Memcached response latency at different loads. We see that SENIC easily sustains 7,000 rpstc (which was also the maximum load the cluster sustained without any rate limiting). However HTB and PTB latencies spike up at much lower loads.

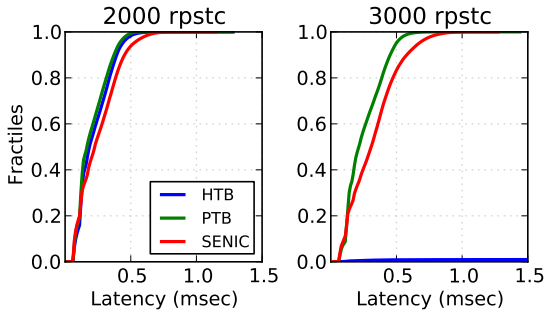

Figure 8: CDF of memcached response latency at different loads. SENIC, HTB and PTB have similar latency at 2,000 rpstc, but HTB latency shoots up at 3,000 rpstc.

Latency: We varied the client load (2000, 3000 rpstc) and observed the latency distribution of memcached responses (Figure 8). The total egress bandwidth utilization on M_{srv} is quite low at 2.3Gb/s and 2.9Gb/s respectively at the two loads. At 2,000 rpstc, we observed that HTB, PTB and SENIC perform similarly. But at 3,000 rpstc, HTB's latency suffers a drastic hit, whereas PTB and SENIC are able to keep up. With HTB, requests keep getting backlogged as the scheduler is the bottleneck and is unable to push packets out of the server fast enough. At the fairly low load of 3,000 rpstc, PTB has marginally lower latency than the SENIC software prototype due to the cache misses incurred for pulling and transmitting all packets from a single CPU core. A hardware SENIC implementation would not have this penalty.

Throughput: We varied the memcached load and measured the average, 99th, and 99.9th percentile latency in each case. Figure 9 shows that SENIC could comfortably handle 7,000 rpstc, sustaining 55% higher load compared to PTB, and 250% higher than HTB. We stopped at 7,000 rpstc as that was the maximum load the cluster could sustain even without any rate limiters (with the default Linux multi queue QDisc).

While the SENIC software prototype is much better than HTB and PTB, a hardware SENIC implementation would perform even better as there would not be cache misses for each transmit operation. Further, if hypervisor bypass is used by VMs to communicate directly with SENIC hardware, the relative latency and throughput benefits of the hardware solution would be even more.

7.2.2 Memcached and UDP Tenant Isolation

To evaluate how effectively SENIC can isolate different tenants, we repeated the above experiments with 1 co-located UDP tenant on each machine, that generates all-to-all UDP traffic as fast as it can. The total rate limit was set at 3Gb/s for UDP traffic, and 6Gb/s for memcached on each machine—divided equally among respective tenants and destinations. The maximum memcached bandwidth utilization we tested was around 5.75Gb/s on M_{srv}, so memcached was again not bandwidth limited.

Memcached Latency and Throughput: As shown in Figure 10, SENIC was able to sustain 5,000 rpstc memcached throughput (5.75Gb/s) with 99.9th percentile latency around 4ms while simultaneously delivering very close to the configured 3Gb/s of total UDP tenant traffic on the memcached server machine. On the other hand, HTB was only able to sustain 1,500 rpstc, while PTB sustained 3,500 rpstc.

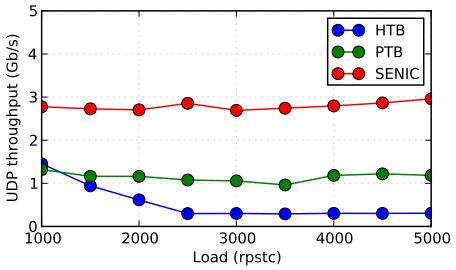

Figure 11: Throughput achieved by UDP background traffic. The configured rate limit was 3Gb/s. We found that SENIC could sustain very close to the configured 3Gb/s throughput, but HTB and PTB had trouble delivering more than 1.3Gb/s.

UDP Tenant Throughput: We measured the total throughput the UDP tenant achieved on M_{srv} as it was the primary machine under heavy overall load. Figure 11 shows that while SENIC sustained the configured 3Gb/s of throughput for the bandwidth intensive UDP tenant,

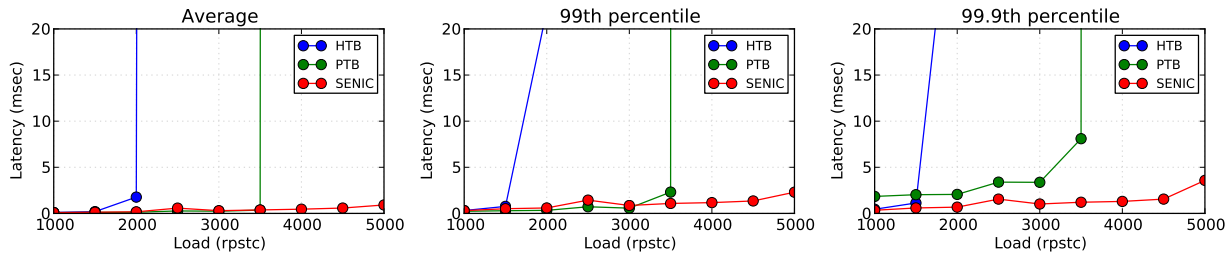

Figure 10: Memcached latency at different loads, with configured background all-to-all UDP traffic of 3Gb/s from each server. We see that SENIC could sustain 5,000 rpstc (network throughput was roughly equal to the configured limit of 6Gb/s). HTB and PTB on the other hand, fell over at lower loads.

HTB and PTB had difficulty keeping up. Even at lower memcached loads, HTB and PTB had trouble delivering more than 1.3Gb/s UDP throughput. Measurements showed that the CPU cores allocated to the UDP tenant were highly loaded, indicating that current software approaches suffer when CPU load increases and the tenants with high CPU load might notice degraded performance as the rate limiter is unable to keep up.

8 Practical Considerations

SENIC's design goals expose a tension in its implementation. Its on-board packet scheduler must be able to transfer sequences of individual packets from a potentially large number of traffic classes for fine-grained rate control. Yet, to drive high line rates, it must support a high overall DMA transfer rate to transfer packets from host memory to the wire. Thus, the performance of SENIC is upper-bounded by the performance of the host's underlying DMA subsystem.

Today's NICs rely on a number of optimizations to drive high link rates, while lowering their impact on the DMA subsystem. For example, when TSO is enabled, they can transfer the packet header just once from memory and cache it on the NIC. The NIC can then pull in the rest of the payload (issuing the appropriate DMA operations), combine it with the cached header and transmit MTU-sized segments. SENIC's design supports interleaving MTU-sized segments from different traffic classes, depending on their configured rates and burst sizes. Because the number of such classes can be quite large, SENIC does not cache packet headers on the NIC for each class. Thus, SENIC's impact on the underlying DMA subsystem is going to be greater than a traditional NIC with TSO. We now briefly examine this impact.

In the absence of TSO, SENIC requires the same number of DMA transfers from host memory as current NICs—one for each packet, in addition to the packet descriptors. However when TSO is active, SENIC issues a DMA operation for the header in addition to one for the payload, for each MTU-sized segment. Note that NICs today are capable of processing many more DMA transfers per second than required for handling MTU-sized frames at line rate. This headroom allows SENIC to drive high line rates even when TSO is enabled, despite the larger number of DMA transfers it requires.

To ground this claim experimentally, we examined the DMA subsystem performance of both 10Gb/s and 40Gb/s commercial NICs. Using a Myricom 10Gb/s NIC, we were able to sustain 13–14 million 64 byte packets per second (pps). Since packets were randomly spread across host memory, each packet required at least one DMA transfer, and thus the NIC can sustain roughly the same number of DMA transfers per second.

For 40Gb/s, we used a Mellanox Connect-X3 NIC [18] to transmit 64 byte packets. We observed that it could only support about 13.1 Mpps, which is less than the rate required to sustain 40Gb/s with 64 byte packets. However, using MTU-sized frames, and TSO disabled, it was able to drive 3.25 Mpps, which was sufficient to sustain 40Gb/s.

The above reference points allow us to gauge the performance of SENIC at both 10Gb/s and 40Gb/s. For instance, at 40Gb/s, SENIC would require $3.25 \times 2 = 6.5$ million DMA transfers per second (to DMA both payloads and headers) to achieve line rate. This is well under the 13.1 million transfers per second we were able to sustain on the same NIC. Hence, we believe that SENIC should be able to support line rate performance with TSO enabled for MTU-sized segments. Since SENIC does not introduce additional DMA requests for non-TSO packets, it should perform comparably to today's commercial NICs.

9 Related Work

We classify related work into two parts: (1) hardware improvements, and (2) software improvements, some of which try to work around limited hardware capabilities. The NIC hardware datapath has only recently received attention from the research community in light of the requirements listed in §2.

Hardware Efforts: Commercial NICs support transport offloading to support millions of connection endpoints, such as 'queue pairs' in InfiniBand [32], or TCP sockets in case of TCP offload engines [19]. The SENIC design is simpler as we only offload rate limiting, and leave the task of reliable delivery to software.

Recent work [20, 36] calls for changes in the NIC architecture in light of low-latency applications (e.g. RAMCloud [10]), and virtualized environments (e.g. public clouds). Such efforts are complementary to SENIC, which focuses only on scaling transmit scheduling. ServerSwitch [17] presented a programmable NIC to support packet classification and configurable congestion management. ServerSwitch can directly benefit from the large number of rate limiters in SENIC.

A number of efforts have focused on scalable packet schedulers in switches [21, 33]. A NIC is conceptually no different from a switch; however, switch schedulers have to deal with additional complexity due to limited on-chip SRAM, and the fact that they cannot control the exogenous traffic arrival rate. Thus, commercial switches often resort to simpler approaches like AFD [27] which can scale to 1000s of policers, but can only *drop* packets (instead of accurate pacing). On the other hand, the NIC being the first hop is in a unique position—its design can be made considerably simpler by leveraging host DRAM to store all packets. This approach enables SENIC to simultaneously scale to, and accurately pace, a large number of traffic classes.

Software Efforts: An alternate approach to deal with limited NIC rate limiters is to share them in some fashion, which has been explored by approaches like vShaper [14] and FasTrak [24]. SENIC eases the burden on such approaches, as we believe the NIC is particularly amenable to large-scale rate limiting by taking advantage of host DRAM. However, if unforeseen applications require more rate limiters than SENIC can offer, such techniques come in handy.

IsoStack [37] proposed offloading the entire TCP/IP network stack to dedicated cores. Our SENIC software prototype mimics this approach (offloading only the scheduler to a dedicated core), which explains the performance benefits in our evaluation. Architectures for fast packet IO such as Netmap [34] are orthogonal to SENIC, and they only stand to benefit from scalable rate limiting in the NIC.

10 Conclusion

Historically, the NIC has been an ideal place to offload common network tasks such as packet segmentation, VLAN encapsulation, checksumming, and rate limiting is no exception. Today's NICs offer only a handful of rate limiters, however new requirements such as performance isolation and OS-bypass for low-latency transport demand more rate limiters. We argued why it makes sense to pursue a hardware offload approach to rate limiting: at data center scale, a custom ASIC is cheaper than dedicating CPU resources for a task that requires real time packet processing. We implemented a proof-of-concept NIC on the NetFPGA to demonstrate the feasibility of scaling hardware rate limiters to thousands of queues. We believe the NIC hardware is *the* cost-effective place to implement rate limiting, especially as we scale the bandwidth per-server to 40Gb/s and beyond.

Acknowledgments

This research was supported in part by the NSF through grants CNS-1314921 and CNS-1040190. Additional funding was provided by a Google Focused Research Award. We would like to thank our shepherd Saikat Guha and the anonymous NSDI reviewers.

References

[1] ALIZADEH, M., ATIKOGLU, B., KABBANI, A., LAKSHMIKANTHA, A., PAN, R., PRABHAKAR, B., AND SEAMAN, M. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *46th Annual Allerton Conference on Communication, Control, and Computing* (2008).

[2] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[3] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI* (2012).

[4] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards Predictable Datacenter Networks. In *SIGCOMM* (2011).

[5] BENNETT, J. C., AND ZHANG, H. WF^2Q : Worst-case Fair Weighted Fair Queueing. In *INFOCOM* (1996).

[6] BENNETT, J. C. R., AND ZHANG, H. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM* (1996).

[7] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* (2010).

[8] CHECCONI, F., RIZZO, L., AND VALENTE, P. QFQ: Efficient Packet Scheduling With Tight Guarantees. In *IEEE/ACM Transactions on Networking* (June 2013).

[9] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM* (1989).

[10] FLAJSLIK, M., AND ROSENBLUM, M. Network Interface Design for Low Latency Request-Response Protocols. In *USENIX ATC* (2013).

[11] Intel 82599 10GbE Controller. `http://www.intel.com/content/dam/doc/datasheet/82599-10-gbe-controller-datasheet.pdf`.

[12] JEYAKUMAR, V., ALIZADEH, M., MAZIÈRES, D., PRABHAKAR, B., KIM, C., AND GREENBERG, A. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI* (2013).

[13] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: Predictable Low Latency for Data Center Applications. In *SOCC* (2012).

[14] KUMAR, G., KANDULA, S., BODIK, P., AND MENACHE, I. Virtualizing Traffic Shapers for Practical Resource Allocation. In *HotCloud* (2013).

[15] LEMON, J. Kqueue - A Generic and Scalable Event Notification Facility. In *USENIX ATC* (2001).

[16] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education* (2007).

[17] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *NSDI* (2011).

[18] Mellanox Connect-X3. `http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf`.

[19] MOGUL, J. C. TCP Offload Is a Dumb Idea Whose Time Has Come. In *HotOS* (2003).

[20] MOGUL, J. C., MUDIGONDA, J., SANTOS, J. R., AND TURNER, Y. The NIC Is the Hypervisor: Bare-Metal Guests in IaaS Clouds. In *HotCloud* (2013).

[21] MOON, S., REXFORD, J., AND SHIN, K. G. Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches. *IEEE Transactions on Computers* (Nov. 2000).

[22] MOSHREF, M., YU, M., SHARMA, A., AND GOVINDAN, R. Scalable Rule Management for Data Centers. In *NSDI* (2013).

[23] Myri-10G PCI Express Network Adapter. `https://www.myricom.com/products/network-adapters/10g-pcie-8b-2s.html`, Retrieved 25 September 2013.

[24] MYSORE, R. N., PORTER, G., AND VAHDAT, A. FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers. In *CoNEXT* (2013).

[25] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast Crash Recovery in RAMCloud. In *SOSP* (2011).

[26] OpenFlow Consortium. `http::://www.openflow.org`.

[27] PAN, R., BRESLAU, L., PRABHAKAR, B., AND SHENKER, S. Approximate Fairness Through Differential Dropping. *SIGCOMM CCR* (Apr. 2003).

[28] PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. `http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html`, Retrieved 25 September 2013.

[29] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *USENIX FAST* (2008).

[30] RADHAKRISHNAN, S., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. NicPic: Scalable and Accurate End-Host Rate Limiting. In *HotCloud* (2013).

[31] RAMCloud RPC Performance Numbers. `https://ramcloud.stanford.edu/wiki/display/ramcloud/RPC+Performance+Numbers`, Retrieved 25 September 2013.

[32] RDMA Aware Networks Programming User Manual. `http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`, Retrieved 25 September 2013.

[33] REXFORD, J., BONOMI, F., GREENBERG, A., AND WONG, A. A Scalable Architecture for Fair Leaky-Bucket Shaping. In *INFOCOM* (1997).

[34] RIZZO, L. netmap: a novel framework for fast packet I/O. In *USENIX ATC* (2012).

[35] RODRIGUES, H., SANTOS, J. R., TURNER, Y., SOARES, P., AND GUEDES, D. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *WIOV* (2011).

[36] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's Time for Low Latency. In *HotOS* (2011).

[37] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *USENIX ATC* (2010).

[38] SHIEH, A., KANDULA, S., GREENBERG, A., AND KIM, C. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud* (2010).

[39] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queueing Using Deficit Round Robin. In *SIGCOMM* (1995).

[40] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM* (2011).

[41] Xilinx User Community Forums: ML506 board: Why my DMA IP hangs OS? `http://forums.xilinx.com/t5/PCI-Express/ML506-board-Why-my-DMA-IP-hangs-OS/td-p/94298`, Retrieved 25 September 2013.