Network Policy Enforcement With Commodity Multiqueue NICs for Multitenant Data Centers

Gyuyeong Kim, *Member, IEEE*, and Wonjun Lee, *Fellow, IEEE*

***Abstract*—Data centers are the fundamental component in the Internet of Things (IoT) system architecture. Data center servers where IoT services are co-located require hierarchical network policy enforcement to ensure fair bandwidth sharing among tenants and to prioritize latency-sensitive traffic within a tenant simultaneously. Meanwhile, emerging network interface cards (NICs) in servers make use of multiple hardware queues to drive increasing line rates. Unfortunately, multiqueue NICs make it hard to enforce hierarchical policies because the NIC packet scheduler dequeues packets in a static round-robin (RR) fashion for per-flow fairness. In this article, we enable hierarchical network policy enforcement with existing commodity multiqueue NICs. We design TONIC, a multiqueue NIC packet scheduling solution that approximates hierarchical packet scheduling by manipulating the packet dequeueing sequence of the NIC scheduler through dynamic packet enqueueing decisions. Specifically, TONIC leverages multiple hardware queues and the double-ended queue structure of qdiscs to express different tenant weights and application priorities without hardware modifications. We implement a TONIC prototype as a Linux kernel module and evaluate it on a testbed with commodity multiqueue NICs. Our experiment results show that TONIC can enforce hierarchical policies consisting of weighted fair sharing and traffic prioritization while maintaining robustness to various network conditions.**

***Index Terms*—Data center networks, Internet of Things (IoT) system architecture, network interface cards (NICs).**

I. Introduction

DATA centers in edges and clouds are the fundamental infrastructure for modern Internet of Things (IoT) services to deal with the deluge of data that IoT devices generate [1], [2]. To accommodate many IoT services with high utilization, data center servers are generally shared by multiple tenants, which run various applications. Each tenant has different network performance requirements depending on application characteristics. For example, a tenant who runs Web services may desire to prioritize latency-sensitive small flows ($\leq$1 kB) [3]–[6] of key-value stores (KVSs) over throughput-sensitive large flows of batch analytics while maintaining fair bandwidth sharing among tenants. To satisfy the requirements, operators apply hierarchical network policies, which specify not only the weights among tenants but also the priorities among tenant applications. To enforce network policy, the operator commonly combines widely used packet schedulers, such as weighted round-robin (WRR) and strict priority queueing (SPQ), in a hierarchical manner.

Meanwhile, the line speed of network interface cards (NICs) has been increased to deal with tens of thousands of flows. 10-GbE NICs have increasingly replaced 1-GbE NICs, and 100-GbE NICs are already commercially available. These emerging high-speed NICs provide multiple hardware queues for parallel packet processing across central processing unit (CPU) cores because a single core is not enough to achieve line-rate throughput of the high-speed NICs.

Unfortunately, the use of multiqueues makes it hard to enforce hierarchical network policies. The traditional policy enforcement point is the qdisc layer in the operating system (OS) network stack where the operator can enforce network policy in software. With multiqueues, the NIC is ultimately responsible for policy enforcement because the NIC performs packet scheduling across hardware queues. However, commodity NICs provide only a static round-robin (RR) scheduler [7]–[9], and this invalidates the OS-level scheduling. This is because the multiqueue NICs are designed to cooperate with queue assignment mechanisms in the OS. For example, a per-flow hashing scheme tries to spread flows into queues uniformly, and the NIC scheduler dequeues packets from the queues with RR, achieving per-flow fairness. The current Linux kernel uses transmit packet steering (XPS) [10], which matches CPU cores and NIC queues one-by-one for intercore performance isolation.

The primary hurdle to solve the problem is that we cannot program NIC application-specific-integrated circuits (ASICs). We may design NIC hardware that adopts recent programmable scheduling models, such as push-in-first-out (PIFO) [8], [11], which can express hierarchical packet scheduling by grouping multiple scheduling blocks. However, the model can process only a limited number of flows at line rate (i.e., 2K flows [11], [12]) due to increased processing delay. Furthermore, this requires a long time to deploy, and the average hardware refresh cycle is five years [13]. Operators also upgrade hardware incrementally since it is not feasible to make a flag day.

In this context, we take one step back and ask the following question: can we enforce hierarchical network policies with existing commodity multiqueue NICs? We answer the question optimistically by presenting TONIC, a multiqueue NIC packet scheduling solution that enables hierarchical policy

Manuscript received May 25, 2021; revised August 19, 2021; accepted September 4, 2021. Date of publication September 7, 2021; date of current version April 7, 2022. This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT under Grant 2019R1A2C2088812. *(Corresponding author: Wonjun Lee.)*

The authors are with the Network and Security Research Lab, School of Cybersecurity, Korea University, Seoul 02841, South Korea (e-mail: gykim08@korea.ac.kr; wlee@korea.ac.kr).

Digital Object Identifier 10.1109/JIOT.2021.3110843

enforcement in multitenant servers without hardware modifications. Our key insight is that although we cannot program NIC hardware, packet enqueueing decisions are done in the OS. Based on this insight, TONIC enqueues packets into hardware queues dynamically to manipulate the packet dequeueing sequence of the static NIC scheduler. This enables us to approximate widely used work-conserving packet schedulers, including WRR and SPQ, in a hierarchical manner.

TONIC consists of two stages: 1) tenant-aware queue assignment and 2) priority-aware packet insertion. The first stage strictly preserves fair sharing among tenants for intertenant policy enforcement. TONIC reserves a disjoint pool of hardware queues for each tenant to avoid the traffic collision with other tenants in the same queue. We express tenant weights indirectly as the number of assigned hardware queues. This is based on the observations that commodity NICs provide abundant number of queues for per-flow fairness and the number of tenant per server is much less than the number of queues. TONIC enqueues packets to the assigned queues evenly to ensure that the NIC scheduler drains packets as much as weights every round. In the second stage, to preserve traffic prioritization between applications of a tenant for intratenant policy enforcement, TONIC leverages the double-ended queue structure of qdiscs where we can freely enqueue/dequeue packets to/from the head and tail. TONIC buffers high priority packets in the head of queues to make the packets jump over buffered packets. TONIC preserves policy hierarchies by not modifying the queue index in the second stage.

We implement a TONIC prototype as a Linux kernel module and evaluate the performance of TONIC on a testbed with Intel 82599 X520-DA2 multiqueue NICs. We compare TONIC with XPS, and make the following observations from our experiment results. First, TONIC can achieve equal fair sharing among tenants regardless of the number of competing flows. Second, TONIC can preserve weighted fair sharing among tenants with different weights and is robust to tenant configurations. Third, TONIC can prioritize latency-sensitive applications, such as `memcached` [14], within a tenant while maintaining fair sharing among tenants.

In summary, we make the following contributions.

1) We enable hierarchical network policy enforcement with commodity multiqueue NICs for edge/cloud data centers in the IoT architecture.
2) We propose TONIC, a new multiqueue packet scheduling solution that approximates hierarchical scheduling through dynamic packet enqueueing decisions to overcome the inflexibility of NIC hardware without hardware modifications.
3) We comprehensively evaluate TONIC using commodity multiqueue NICs, and demonstrate that TONIC can enforce both intertenant and intratenant network policies in a hierarchical manner across a variety of network conditions.

The remainder of this article is organized as follows. In Section II, we describe the background and motivation of this work. Section III provides the detailed design of TONIC. We present the implementation of TONIC and performance evaluation results in Sections IV and V, respectively. We discuss related work in Section VI. Finally, we conclude our work in Section VII.

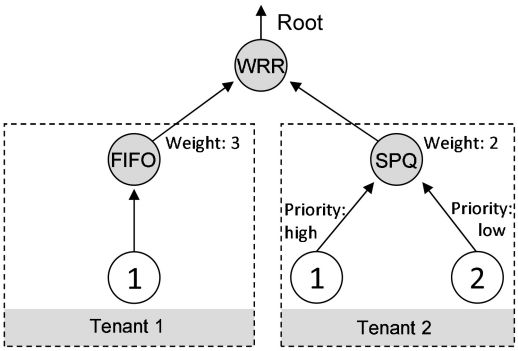

Fig. 1. Example of a hierarchical network policy.

II. Background and Motivation

A. Hierarchical Network Policies

In a server of multitenant data centers, tens of containers or virtual machines (VMs) are co-located to run IoT applications. Specifically, it is not rare to run more than 18 containers [15] and 40–60 VMs [16] per server. The number of tenants per server is generally much smaller than the number of containers/VMs because a single tenant commonly owns multiple containers/VMs to run applications.

Operators apply hierarchical network policies to satisfy tenant requirements. Network policy can be expressed as a restricted directed acyclic graph (DAG) [8]. A policy DAG specifies: 1) intertenant network policy that generally clarifies how tenants should share the link capacity and 2) intratenant network policy that specifies the priorities among applications within a tenant. It is important to ensure intertenant policy since the end host is the first place where tenants collide in the network. Preserving intratenant policy is also important since many tenants run latency-sensitive applications and throughput-sensitive applications simultaneously [11], [17], [18].

The tenants often want to prioritize latency-sensitive small flows over throughput-sensitive large flows for better user experience. Specifically, flow size distributions in data centers are heavily skewed. 90% of flows are less than 10 kB [19], [20] and the typical flow size of Web search workloads such as KVS is less than 1 kB [3]–[6], [21], [22], which can be transmitted with a single packet.

Fig. 1 shows an example of a hierarchical network policy. In the example, tenant 1 and tenant 2 should share link bandwidth with weights of 3:2. Meanwhile, unlike tenant 1 who runs a single application, tenant 2 runs two applications and desires to prioritize the traffic of application 1 over application 2. Operators should provide hierarchical packet scheduling to preserve the hierarchy of the specified policy. We need the WRR scheduler to preserve the intertenant network policy. To enforce the intratenant policies of the tenants, we need the first-in–first-out (FIFO) for tenant 1 and SPQ for tenant 2, respectively. This is because tenant 1 has only one application whereas tenant 2 has two applications with different priorities.

TABLE I
SPECIFICATIONS OF SEVERAL COMMODITY MULTIQUEUE NICS (BW = BANDWIDTH)

NIC Model	Vendor	BW (GbE)	# of queues
82599 [25]	Intel	10	128
X/XXV/XL710 [26]	Intel	10/25/40	1536
ConnectX-4/5/6 [27]	Mellanox	25/40/50/100	512

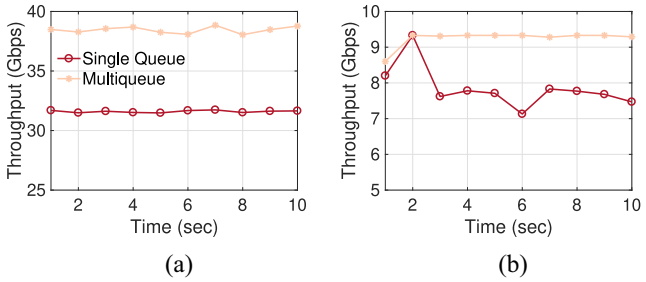

Fig. 2. Aggregate throughput of (a) 40- and (b) 10-GbE NICs with single queue and multiqueue configurations.

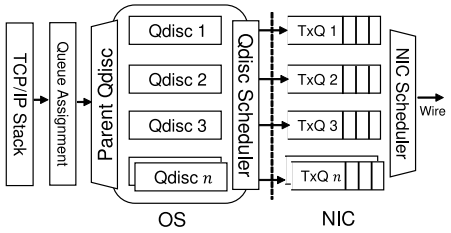

Fig. 3. Packet transmission with multiqueue NICs.

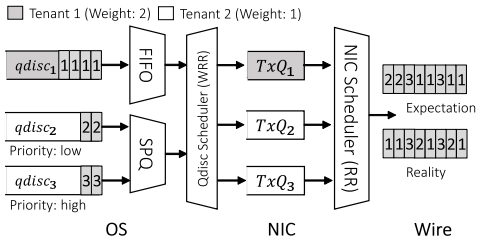

Fig. 4. Network policy violation with multiqueues.

FIFO is a simple scheduling algorithm that dequeues packets in the order that they arrive in the queue. SPQ is a scheduling algorithm that dequeues packets in strict priority.

B. Packet Processing With Multiqueues

The line rate of NICs has been scaled to deal with the increasing network resource demand of modern applications. 10-GbE NICs are today's commodity, and they have rapidly replaced 1GbE NICs [21], [23], [24]. In addition, 100-GbE NICs such as Mellanox ConnectX-5 are already on the market. The key feature of these high-speed NICs is multiple hardware transmit (Tx) and receive (Rx) queues.[1] Table I lists the specifications of several commodity multiqueue NICs. We can find that commodity NICs generally support hundreds to thousands of hardware queues. For example, Mellanox ConnectX-5 supports 512 hardware queues.

Multiqueues enable servers to drive line rates of 10 Gb/s and beyond by processing packets in parallel across multiple CPU cores without intercore communication overhead. For example, a single CPU core cannot achieve the line rate of 40-GbE NICs even with segmentation offload techniques such as TCP segmentation offload (TSO), which reduces the CPU resource usage for packet processing. Fig. 2 shows the results of testbed experiments that measure the aggregate throughput of multiqueue NICs with different configurations. To saturate the link between two servers, we generate six flows using `iperf` [28]. For the 10-GbE NIC experiment, we disable TSO to show the impact of multiqueues in detail. In Fig. 2(a), we can see that the aggregate throughput of the single queue configuration is limited to 32 Gb/s approximately. From Fig. 2(b), we observe that a single core cannot drive even 10 Gb/s of line rate when TSO is disabled.

Fig. 3 shows how packets are transmitted in Linux kernel with multiqueue NICs. After a packet has passed through the TCP/IP stack, the OS assigns a Tx queue index to the packet by updating `queue_mapping` that denotes the queue index metadata in the socket buffer structure (SKB). Today's Linux kernel supports two queue assignment mechanisms. When cores are shared by multiple tenants, we can use a per-flow hash scheme that assigns packets to Tx queues uniformly as much as possible in flow level by calculating the queue index through a simple hash algorithm. For servers where each tenant uses dedicated CPU cores, we can use XPS [10] that pins Tx queues to cores one-to-one. In this regime, packets are enqueued into the matched Tx queue of the CPU core that processed the packet. XPS is the default queue assignment mechanism in the current Linux kernel because it provides strong performance isolation between cores.

All of the packets are transmitted to the qdisc layer after Tx queue assignment. The OS creates one child qdisc per Tx queue under a parent qdisc to synchronize the qdisc layer and the NIC. Therefore, when n Tx queues exist, n children qdiscs exist. Each qdisc is a FIFO queue by default. The parent qdisc enqueues the packet into child qdisc i where i is the index in `queue_mapping`. The qdisc scheduler performs RR scheduling across the children qdiscs and the packets are buffered to the corresponding Tx queue i in the NIC. Each Tx queue is a ring buffer containing packet descriptors that point to the SKB. To dequeue packets, the NIC sends hardware interrupt requests (IRQs) to the CPU by performing per-queue RR scheduling. After processing IRQs, the packet is finally transmitted to the wire.

C. Policy Violation by Multiqueue NICs

The packet scheduler of multiqueue NICs performs flat RR scheduling across hardware queues, and this makes it hard to enforce hierarchical network policies. The primary reason behind the static NIC scheduler is that the multiqueue NIC is designed without awareness to multitenancy. Basically, the NIC strives to achieve per-flow fairness by cooperating with the queue assignment mechanisms.

Fig. 4 illustrates an example of policy violation caused by the NIC scheduler with XPS. We suppose that the qdisc layer scheduling is properly configured in the OS. While tenant 1

[1]In this work, we focus on transmit-side networking. Therefore, NIC hardware queues in this article indicate Tx queues unless specified.

requires only FIFO, tenant 2 needs SPQ due to competing applications with different priorities. Meanwhile, the packets of the two tenants should be scheduled by the WRR scheduler with weights of 2:1. However, we can see that the actual packet dequeueing sequence into the wire is different from the expectation of OS-level packet scheduling. This is because, as we have mentioned, the NIC scheduler drains bytes from competing Tx queues with equal weight regardless of the qdisc scheduling. A few NICs provide data center bridging (DCB) [29], which provides QoS configuration between eight DCB classes. However, DCB supports only plain policies for the limited number of classes (i.e., tenants).

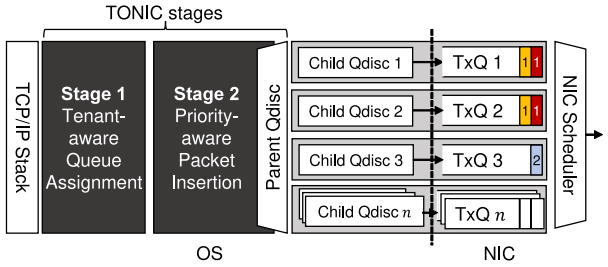

Fig. 5. Overview of TONIC. TONIC manipulates the packet dequeueing sequence of the static NIC scheduler in the OS, approximating hierarchical packet scheduling.

III. TONIC Design

A. Design Goals

Our goal is to design a solution that enables hierarchical network policy enforcement in servers with existing commodity multiqueue NICs. We believe that a good solution should satisfy the following requirements simultaneously.

1) *Intertenant Policy Enforcement:* A solution should strictly preserve fair sharing among tenants any time regardless of traffic dynamics.
2) *Intratenant Policy Enforcement:* A solution should be able to prioritize the traffic of preferred applications over the other applications within a tenant.
3) *Hierarchical Policy Enforcement:* A solution should preserve a policy hierarchy while enforcing inter-tenant/intratenant network policy.
4) *Practicality:* A solution should support existing commodity multiqueue NICs and should not require hardware modifications.

B. Overview and Design Rationale

Packet scheduling requires two decisions: 1) the packet enqueueing decision and 2) the packet dequeueing decision. The key challenge to achieve our goal is that we cannot program the packet scheduler of commodity NICs even though the root cause of the policy violation problem is the hardware packet scheduler. To address the challenge, we pay attention to that the NIC packet enqueueing decision happens in the OS, not in the NIC. This implies that although we cannot modify the packet scheduler of the NIC directly, we may be able to approximate the behavior of hierarchical packet schedulers by designing a new NIC packet enqueueing decision mechanism. Based on this insight, we design TONIC, a multiqueue NIC packet scheduling solution that dynamically performs packet enqueueing decisions to manipulate the NIC packet dequeueing sequence, approximating hierarchical packet scheduling without hardware modifications.

Fig. 5 shows the overview of TONIC. TONIC resides between the TCP/IP stack and the qdisc layer as a shim layer. In the Linux architecture with multiqueue qdiscs such as `multiq`, the qdiscs are completely synchronized with hardware queues since the packets in the qdisc layer are scheduled with RR as same as in the NIC. This means that the qdisc layer and the NIC have the same packet enqueueing/dequeueing sequence. The parent qdisc buffers a packet into child qdisc *i* where *i* is the index of assigned Tx queue in `queue_mapping` metadata, which is determined by TONIC stages. TONIC consists of two stages as follows.

Stage 1 (Tenant-Aware Queue Assignment): In the first stage, TONIC ensures fair bandwidth sharing in tenant level for intertenant policy enforcement. To avoid interference in the same Tx queue, TONIC reserves a disjoint set of Tx queues to each tenant. This ensures fairness among tenants when they share the bottleneck. However, another challenge is how to express different tenant weights for WRR using a simple RR NIC scheduler, which drains the equal bytes-per-round (BpR) from all nonempty queues every round.

TONIC addresses the above challenge by expressing tenant weights as the number of Tx queues. As Table I shows, commodity multiqueue NICs provide abundant number of hardware queues for per-flow fairness. Meanwhile, the number of tenants per server is much less than the number of supported queues because a single server runs generally tens of VMs/containers [15], [16] and a single tenant owns multiple VMs/containers to run applications. This mismatch provides us opportunities to express tenant weights indirectly. Specifically, TONIC assigns Tx queues to packets evenly within the Tx queue pool reserved for a tenant. In Fig. 5, TxQ $\{1, 2\}$ and TxQ 3 are assigned to tenant 1 and tenant 2, respectively. The NIC scheduler dequeues packets from the queues with equal BpR, but the tenants share the link capacity with weights of 2:1 since the aggregate BpR of tenant 1 is $2\times$BpR.

Stage 2 (Priority-Aware Packet Insertion): In the second stage, TONIC enables traffic prioritization within a single tenant for intratenant policy enforcement. A challenge is how to prioritize the packets of high priority applications while preserving a scheduling hierarchy.

Our idea to address the challenge is leveraging the double-ended queue structure of qdiscs. The current Linux kernel implements a qdisc as a double-ended queue where we can enqueue and dequeue packets to/from the head and tail freely. Therefore, TONIC buffers high priority packets into the head of qdiscs instead of the tail. Our head buffering makes the high priority packets jump the buffered low priority packets without experiencing queueing delay. To preserve a scheduling hierarchy, the second stage does not change the Tx queue index, which is determined in the first stage. The second stage only determines whether packets will be buffered in the head or the tail of the queue. In Fig. 5, we can find that high priority packets with darker color of tenant 1 are buffered in front

of the other packets in TxQ 1 and TxQ 2, not in TxQ 3 of tenant 2.

C. Detailed Mechanisms

In the following, we show the detailed design of TONIC. We consider a dedicated resource sharing model where tenants do not share assigned CPU cores with the other tenants for strong performance isolation. We suppose that the operator knows essential information for network policy enforcement, such as the number of CPU cores owned by each tenant, weights for each tenant, and port numbers of applications to be prioritized. TONIC initializes the following four arrays with the information.

1) *Tenant Mapping Array:* This array enables us to classify the traffic of different tenants. Specifically, we classify tenant traffic based on the CPU core index. For example, when tenant 1 and tenant 2 have 2 and 4 cores with a 6-core CPU, the mapping array is expressed as {1, 1, 2, 2, 2, 2}. We can obtain the tenant ID of a packet by referring the mapping array and `sender_cpu` metadata in the SKB, which stores the index of CPU core that processes the packet.
2) *Tenant Weight Array and Start Index Array:* TONIC maintains the tenant weight array (e.g., {2, 1} for two tenants with 2:1 weights). Based on the tenant weights, TONIC calculates the index range of tenant queues in ascending order and stores the index of the first queue in the start index array. Suppose that we have two tenants with weights of 2 and 1. In this case, Tx queue {0, 1} and Tx queue 2 are assigned to each of the tenants, respectively. The corresponding elements in the start index array are {0, 2}.
3) *Port Number Array:* TONIC maintains the port number array, which stores the port number of applications specified in intratenant policy to prioritize the traffic of a preferred application within a tenant.

1) Tenant-Aware Queue Assignment: To approximate WRR, TONIC expresses tenant weights by leveraging multiple Tx queues available in commodity NICs. In particular, TONIC assigns Tx queues to tenant i as much as tenant weight w_i. For example, when there exist two tenants, tenant 1 and tenant 2 with weights of 1 and 2, we assign 1 and 2 Tx queues for each tenant, respectively. The multiqueue NIC scheduler dequeues packets from nonempty queues with equal BpR every round whose unit BpR is an MTU-byte. With TONIC, the aggregate drained bytes of tenant i in a round can be given by

$$B_i = q_i \times BpR_c \tag{1}$$

where $q_i = w_i$ is the number of assigned Tx queues to tenant i and BpR_c is the unit BpR. In the original WRR, tenant weights are expressed as different BpRs. Therefore, the drained bytes of tenant i in a round are

$$B_i^{\text{Origin}} = BpR_i \tag{2}$$

where $BpR_i = w_i \times BpR_c$. We can find that $B_i = B_i^{\text{Origin}}$ easily.

To ensure the aggregate BpR as much as the weight, TONIC evenly enqueues packets into the assigned Tx queues by updating `queue_mapping` in the SKB for each packet. For instance, when a tenant has two Tx queues of queue 0 and queue 1, the assigned Tx queue indices for each packet are {0, 1, 0, 1, 0, 1, ...}.

Algorithm 1 Tx Queue Assignment in TONIC

```
1: Inputs:
2: i: An integer indicating tenant ID
3: w_i: The weight of tenant i
4: s: The index of the first Tx queue assigned to tenant i
5: c_i: The current Tx queue index of tenant i
6:
7: i ← TenantMappingArray[skb->sender_cpu]
8: w_i ← WeightArray[i]
9: s ← StartIdxArray[i]
10:
11: if c_i ≥ w_i then                    ▷ c_i exceeds the index range of tenant i
12:     c_i ← 0  ▷ Initialize the index to circulate Tx queue index based on the assigned index range for tenant i
13: end if
14: skb->queue_mapping ← s + c_i         ▷ Assign Tx queue index
15: c_i ← c_i +1                         ▷ Update current index
```

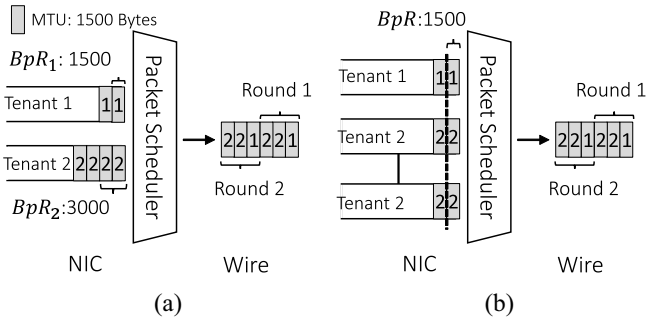

Fig. 6. Comparison of TONIC with the original WRR. TONIC results in the same packet dequeueing sequence with WRR by leveraging multiple queues with equal BpR. (a) WRR. (b) TONIC.

Algorithm 1 describes the pseudocode of Tx queue assignment in TONIC. To derive the index of Tx queue, we need four input parameters that include tenant ID, tenant weight, index of the first queue assigned to the tenant, and the index to track the current Tx queue index (lines 1–5). The required inputs can be obtained when the arrays are initialized (lines 7 and 8). Note that c_i is initialized as 0 at the beginning. To spread packets to the assigned Tx queues evenly, we initialize c_i again when it exceeds w_i (lines 11–13). Recall that TONIC reserves Tx queues to tenant i as much as weight w_i. TONIC determines the queue index for the current packet by updating `queue_mapping` and increases the current index for the next packet (lines 14 and 15).

Fig. 6 shows an example where two tenants, tenant 1 and tenant 2, share the 1500-byte MTU link. Fig. 6(a) depicts the original WRR where weights are expressed as different BpRs in a single queue. For each round, the NIC scheduler drains 1500 bytes and 3000 bytes from the queues of tenant 1 and tenant 2, respectively. Fig. 6(b) shows the approach of TONIC where weights are expressed as multiple queues with equal BpR. In this approach, the NIC drains 1500 bytes of equal data from the queues every round. We can find that draining 1500 bytes from two queues results in the identical packet dequeueing sequence to the original WRR that drains

Algorithm 2 Priority Assignment in TONIC

```
1: Inputs:
2: p_s: The source port number in TCP/UDP header
3: p_d: The destination port number in TCP/UDP header
4:
5: if p_s ∈ PortNumberArray or p_d ∈ PortNumberArray then
6:     skb->priority ← 1        ▷ Update metadata to indicate that the packet belongs to the high priority application
7: else
8:     skb->priority ← 0        ▷ Tag low priority in metadata for the other packets
9: end if
```

Algorithm 3 Enqueueing Decision in the Parent qdisc

```
1: Inputs:
2: i: An integer indicating Tx queue index
3: j: An integer denoting packet priority
4:
5: i ← skb->queue_mapping          ▷ Determined in Algorithm 1
6: j ← skb->priority               ▷ Determined in Algorithm 2
7: if j = 1 then                   ▷ High priority?
8:     Enqueue(head,i)             ▷ Insert the packet into the head of queue i
9: else
10:    Enqueue(tail,i)             ▷ Insert the packet into the tail of queue i
11: end if
```

$2 \times 1500 = 3000$ bytes from a single queue. We can also see that the same bytes are transmitted per round in both WRR and TONIC.

2) Priority-Aware Packet Insertion: To approximate SPQ for traffic prioritization between applications in a tenant, TONIC leverages the double-ended queue structure of the qdisc. With the double-ended queue, we can enqueue to or dequeue from either the head or tail of the queue. Our objective is to make high priority packets be dequeued earlier than the other buffered packets. To do this, as shown in Algorithm 2, TONIC sets `priority` metadata in the SKB to 1 if source/destination port numbers are specified in the port number array (lines 5 and 6). Otherwise, `priority` is set to 0 denoting a low priority (lines 7 and 8).

With the socket metadata, the parent qdisc buffers the packet to the head of the child qdisc if `priority` is 1. Otherwise, TONIC enqueues the packet into the tail as usual. Algorithm 3 describes how the parent qdisc performs the packet enqueueing decision. The parent qdisc first gets the Tx queue index determined in Algorithm 1 by referring `queue_mapping` metadata (line 5). The qdisc also obtains the priority value with `priority` determined in Algorithm 2 (line 6). TONIC buffers the packet to the head of the corresponding child qdisc if the packet has high priority or into the tail of child qdisc if otherwise (lines 5–8). Since the children qdiscs and Tx queues are synchronized, the NIC observes the identical packet enqueueing sequence with the qdisc.

Fig. 7 compares the original SPQ with TONIC to show how TONIC enforces traffic prioritization between two applications in a single tenant with three Tx queues. Unlike SPQ where high priority packets are dequeued first using the concept of priority, TONIC schedules the high priority packets indirectly by the RR packet scheduler. Nevertheless, we can see that TONIC results in the same packet dequeueing sequence with SPQ without priority scheduling.

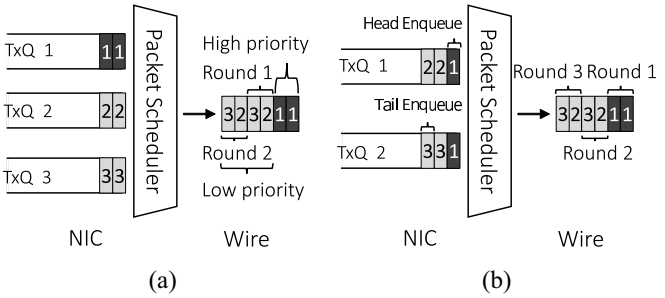

Fig. 7. Comparison of TONIC with the original SPQ. To express SPQ without the concept of priority, TONIC buffers high priority packets in the head of queue. (a) SPQ. (b) TONIC.

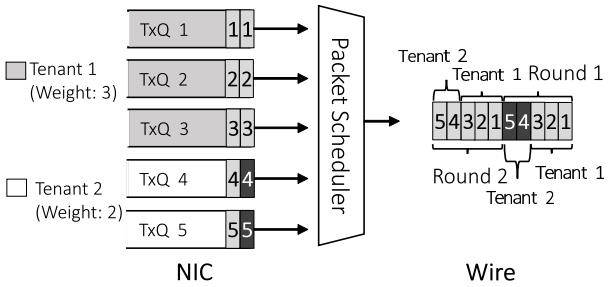

Fig. 8. Preservation of a policy hierarchy. TONIC naturally maintains the policy hierarchy because the traffic prioritization of a tenant does not affect the weighted fair sharing between the tenants.

3) Put Everything Together: We now explain how TONIC preserves policy hierarchies. Our queue assignment mechanism in the first stage strictly isolates hardware queues among different tenants, and this is the key for intertenant policy enforcement. If the second stage breaks queue isolation for traffic prioritization, the policy hierarchy may be violated. However, in the second stage, TONIC never changes the index of Tx queues to be buffered but only determines the direction of buffering. Therefore, the policy hierarchy can be maintained naturally.

Fig. 8 depicts an example that shows that our approach preserves a policy hierarchy. We consider tenant 1 and tenant 2 with different weights of 3:2. We suppose that tenant 2 runs two applications with different priorities. In the figure, packets with darker color are high priority packets. To dequeue all the packets of the tenants, the NIC scheduler requires two scheduling rounds. We can see that although the scheduler dequeues the packets with RR scheduling, the packet dequeueing sequence is approximate to the result of hierarchical packet scheduling using WRR, SPQ, and FIFO. The enqueueing sequence for each queue is regardless of the number of tenants since we assign a set of dedicated queues for each tenant.

D. Discussion

We now discuss several design issues and a few limitations of TONIC.

Virtualized Environments: We have implicitly considered containerized environments where qdiscs and hardware queues are not virtualized. However, the hypervisor-based

virtualization is still widely deployed. In the virtualized environment, the guest OS in VMs sees virtualized NICs (vNICs). To use TONIC, we should transform the queue index of the vNIC determined in VMs into the queue index of the physical NIC. This translation mechanism can be implemented as a reference table in the hypervisor, who bridges the VMs and physical servers as a shim layer. Since all the packets of VMs go through the physical server, we do not need to modify TONIC itself.

Rate Limiters: While this article focuses on work-conserving schedulers for high link utilization, there exist nonwork conserving packet schedulers such as token bucket to express rate limiting and packet pacing [7], [30]. TONIC does not limit the functionality of rate limiting, which means that rate limiters can be configured in the qdiscs or NICs. Commodity multiqueue NICs cannot enforce tenant-level rate limiting even with per-queue limiters due to uneven traffic distribution across Tx queues [8]. However, TONIC can enforce tenant-level rate-limiting with per-queue rate limiters because TONIC spreads packets evenly across Tx queues regardless of the CPU core processed the packets. It is also possible to combine per-queue rate limiters and TONIC to let a tenant utilize idle bandwidth without exceeding the maximum rate.

Efficiency in Overcrowded Servers: TONIC may not be effective for extreme cases. For example, one might consider 128 tenants in a single server with an Intel 82599 NIC, which supports 128 hardware queues. In this case, it is hard to preserve weighted fair sharing between the tenants because a 128-queue NIC is not enough to represent the diverse weights of the tenants. This also can be a problem if we consider a multitenancy model where multiple tenants share the same CPU core without performance isolation. However, we believe that such an overcrowded server is rare in practice. In addition, the number of tenants per server depends on the tenant placement policy. Therefore, the operator can modify the tenant placement policy to avoid the lack of queues when TONIC should be deployed.

Limited Expressiveness: Compared to hardware solutions, TONIC has limited expressiveness. For instance, our work does not support complex schedulers (e.g., least slack-time first (LSTF) [31] and service-curve earliest deadline first (SC-EDF) [32]). In addition, TONIC supports only two priorities between applications in a tenant. It is also complicated to use weighted fair sharing within a tenant because the number of queues can be partitioned is limited to the number of assigned queues for the tenant. It may not be possible to use weights consisting of prime numbers, which require many hardware queues. However, we believe FIFO, WRR, and SPQ can satisfy the majority of network policy because most hierarchical policies generally consist of fair sharing and traffic prioritization. Basically, TONIC sacrifices expressiveness at the expense of supporting existing commodity NICs. This is the distinguished contribution from new NIC hardware designs that easily eliminate the root cause of the problem by replacing hardware.

Packet Reordering: The head buffering for traffic prioritization may cause packet reordering by the last-in–first-out (LIFO)-like behavior, which may harm latency for flows consisting of many packets. However, latency-sensitive applications such as KVS have small flow sizes less than 1 kB [3], [5], [6]. Measurement studies for production workloads report that 90% of flows are less than 10 kB [19] and 75% of the flows consist of a single packet [21], [22]. In addition, since the buffered packet is dequeued almost instantly, no serious performance degradation occurs. We demonstrate this in Section V through testbed experiments. An alternative way to avoid packet reordering is using priority scheduling in the qdisc layer, but this may increase packet processing delay slightly.

IV. Implementation

In this section, we present the implementation of a TONIC prototype and discuss userspace implementation. We also analyze the processing overhead of TONIC.

Prototype Implementation: A TONIC prototype is implemented as a `NETFILTER` module in Linux kernel 4.4. The module is located as a shim layer between the TCP/IP stack and the qdisc layer. Every outgoing packet is intercepted by `NETFILTER` hook at `INET_POST_ROUTING` and TONIC updates `queue_mapping` and `priority` in the SKB data structure if the conditions of Algorithms 1 and 2 are met. The queue mapping information and tenant weights can be configured using `sysctl` without recompiling the module.

Our current implementation requires a few lines of codes in the Linux kernel. First, we disable Tx queue assignment operations in `netdev_pick_tx()` because the current kernel implements queue assignment mechanisms as built-in functions, not modules. Therefore, unless disabling the operations, the kernel eventually overwrites `queue_mapping` metadata, which is determined by TONIC. Second, we make the kernel store the value of `smp_processor_id()`, the CPU core index, to `sender_cpu` in the SKB when the OS begins packet processing. Finally, we modify the `multiq` qdisc module to enqueue the packet in the head when `priority` is 1.

Userspace Implementation: TONIC can be implemented in userspace network stacks [33]–[35], which provide higher packet processing performance compared to the OS network stack. To support userspace networking, we should implement TONIC as a userspace module since our current design targets the OS network stack with the SKB metadata and qdiscs. Implementing TONIC in userspace is not problematic because we only need the metadata to determine the queue index and the direction of enqueueing, which means that the metadata is not delivered to the NIC driver. The SKB metadata, such as `queue_mapping` and `priority`, can be replaced as metadata in userspace network stacks easily. The qdisc also can be implemented as software queues in userspace.

Processing Overhead: We now analyze the processing overhead of TONIC to demonstrate the scalability of TONIC. Specifically, TONIC can support line-rate operations with emerging high-speed NICs (e.g., 200 GbE) for the following reasons. First, there exist no added processing delays in the NIC caused by extra enqueue/dequeue operations since TONIC does not modify NIC hardware. Second, TONIC does not increase enqueue/dequeue operations in the network

stack as well. TONIC only updates the socket metadata `queue_mapping` and `priority`, which requires negligible processing delay. Third, to determine the value of metadata, TONIC does not use loop operations, which can result in unpredictable processing delay. This also implies that TONIC can work with many tenants since TONIC has only a little processing overhead.

V. Performance Evaluation

In this section, we evaluate TONIC through testbed experiments. The questions and key results are as follows.

1) Can TONIC ensure equal fair sharing regardless of the different number of flows?
2) Can TONIC preserve weighted fair sharing with multiple tenants?
3) How robust is TONIC to different configurations?
4) Can TONIC enforce traffic prioritization while preserving a policy hierarchy?

A. Experimental Setup

Testbed: To evaluate TONIC, we build a testbed consisting of two servers connected to a switch. Each server is with Intel Core i5-8400 6-core 2.8-GHz CPU, 16 GB of memory, and Intel 82599 X520-DA2 10GbE NIC. We have enabled TSO and large receive offload (LRO) to reduce CPU overhead so that a single core can drive the line rate of 10 Gb/s. Tenants in our experiments are managed through Linux containers (LXC). Therefore, each tenant is isolated by Linux cgroups and owns a number of isolated CPU cores. The applications of the tenants are computed across their assigned cores. We synchronize IRQ affinity with CPU affinity to avoid the resource contention caused by overlapped IRQs among tenants.

Compared Scheme: We compare TONIC against XPS [10], the default queue assignment mechanism in Linux kernel. The Linux kernel also supports the per-flow hash scheme. However, we omit the results of the scheme since the results are very similar to those of XPS. This is because both XPS and the per-flow hash scheme do not isolate Tx queues completely. Specifically, active flows are not uniformly distributed across the cores in XPS. With the per-flow hash scheme, the flows of different tenants are contended in the same Tx queue.

Performance Metrics: We use the following performance metrics.

1) *Aggregate Throughput:* Our throughput denotes the aggregate throughput in tenant level to show the bandwidth share of each tenant.
2) *Fairness Index:* We use Jain's fairness index [36], which is widely used to evaluate fairness when the bandwidth should be shared with equal weight.
3) *Relative Error Ratio:* To evaluate weighted fair sharing, we use relative error ratio that shows the difference between the actual bandwidth share and the ideal bandwidth share in percentage. The relative error ratio of tenant i is defined as

$$\text{Relative error}_i = \frac{|B^i_{\text{actual}} - B^i_{\text{ideal}}|}{B^i_{\text{ideal}}} \tag{3}$$

where $B^i_{\text{ideal}} = \sum B_{\text{actual}} \cdot (w_i / \sum w)$ and $(w_i / \sum w)$ is the normalized weight of tenant i.

4) *Query Completion Time:* To evaluate the impact of TONIC on the performance of `memcached`, we use the query completion time (QCT), which denotes the required time to request a query and receive a response.

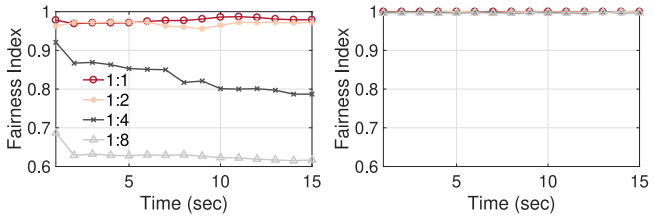

Fig. 9. Time series of fairness index in equal fair sharing. TONIC provides nearly perfect fairness even with a high asymmetry ratio over time. (a) XPS. (b) TONIC.

B. Equal Fair Sharing

Methodology: We first inspect whether TONIC can share bandwidth equally regardless of a different number of flows per tenant. In this experiment, we have two tenants T1 and T2 with equal weight. Each tenant owns 3 CPU cores and 3 Tx queues. The tenants generate traffic for 15 s using `iperf` [28], a multithreaded network-intensive application. In our scenario, T1 has always eight flows. However, for T2, we vary the number of flows to $\{8, 16, 32, 64\}$. Therefore, we have the asymmetry ratios of 1:1 to 1:8 in the number of flows. We measure the aggregate throughput of each tenant every 0.5 s.

Results: Our results demonstrate that TONIC can provide fair bandwidth sharing. Fig. 9 shows the time series of Jain's fairness index between the throughput of the two tenants. We find that TONIC isolates the two tenants regardless of asymmetry ratios. With TONIC, although the fairness index fluctuates over time and the average fairness index decreases as the asymmetry ratio increases, the lowest index is only 0.996 when T2 has 64 flows. However, XPS does not provide enough fairness. When the asymmetry ratio is 1:8, the lowest fairness index is 0.630 and the average fairness index is only 0.665. Even when the tenants have an equal number of flows, XPS only provides 0.978 of the fairness index while that of TONIC is always 1.000.

Fig. 10 compares the average throughput between the tenants. It is easy to see that XPS results in unfair bandwidth sharing as the asymmetry ratio increases while TONIC generally shares bandwidth equally between the tenants. One notable point is that XPS does not share bandwidth equally even when the asymmetry ratio is 1:1. This is because active flows of applications are not distributed across the CPU cores of a tenant uniformly, and this lead to uneven packet distribution across the hardware queues. Unlike XPS, TONIC determines the queue index regardless of the CPU cores. Therefore, this results in fair packet distribution across the queues, causing no issues like in XPS.

C. Weighted Fair Sharing

Methodology: We conduct experiments with three tenants to inspect whether TONIC can preserve weighted fair sharing

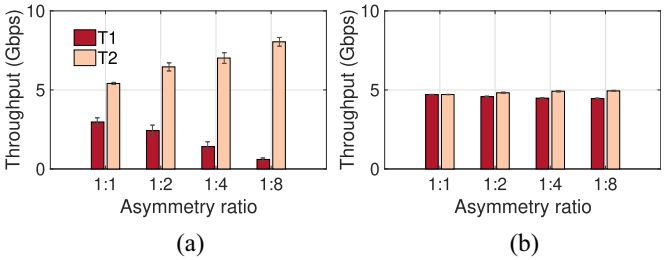

Fig. 10. Average throughput of tenants with different number of flow ratios. XPS results in unfair bandwidth sharing even with the asymmetry ratio of 1:1 due to uneven traffic distribution across queues. Unlike (a) XPS and (b) TONIC shares bandwidth almost equally by distributing traffic evenly.

TABLE II
USED TENANT CONFIGURATIONS

		# of CPU cores	Weight	# of flows
Case 1	T1	1	1	8
	T2	2	2	16
	T3	3	3	32
Case 2	T1	3	2	8
	T2	2	3	16
	T3	1	5	32

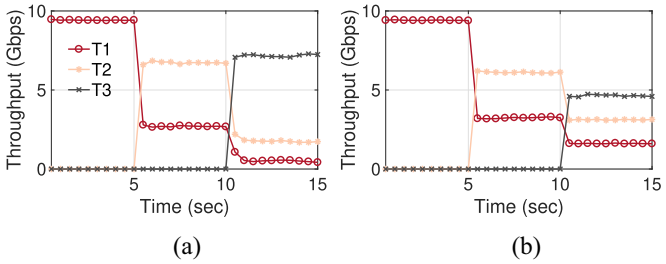

Fig. 11. Time series of the aggregate throughput of three tenants with weights of 1:2:3. XPS cannot enforce weighted fair sharing because a tenant with more flows occupies more bandwidth. Unlike (a) XPS and (b) TONIC respects the assigned weights regardless of different number of flows.

among tenants with different weights. Like in the previous experiment, we use `iperf` to generate traffic. To understand the impact of tenant configurations comprehensively, we use two different settings described in Table II. While case 1 assigns the same number of cores and weights, case 2 assigns a mismatched number of cores and weights. Ideally, TONIC should partition bandwidth using only tenant weights and should not be influenced by other conditions, such as the number of CPU cores and flows. In case 1, T1 first starts eight flows for 15 s. At 5 s, T2 generates 16 flows for 10 s. Finally, at 10 s, T3 generates 32 flows for 5 s. We measure the aggregate throughput of each tenant every 0.5 s.

Results: Fig. 11 shows the results for case 1. We observe that TONIC can share bandwidth in a weighted fair manner while XPS allows a tenant with many flows to occupy more bandwidth. For example, T3 in XPS occupies excessive bandwidth because of the large number of flows. Meanwhile, except the first 5 s, T1 in XPS cannot enjoy enough throughput due to the other tenants. To examine the bandwidth share between the tenants in detail, we calculate the relative error ratios of the tenants. Fig. 12 reports the time series of relative error ratios. We can find that compared to XPS, the error ratio of TONIC

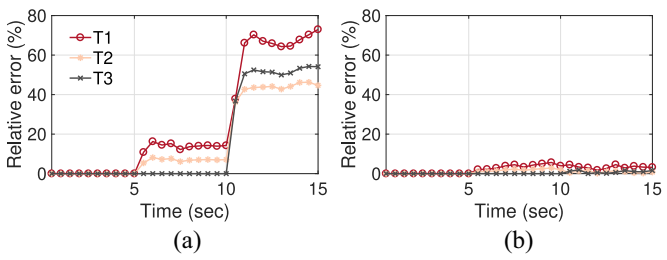

Fig. 12. Time series of the relative error ratio of three tenants with weights of 1:2:3. TONIC causes lower error ratios than XPS because it can enforce weighted fair sharing by assigning different number of Tx queues to each of tenants. (a) XPS. (b) TONIC.

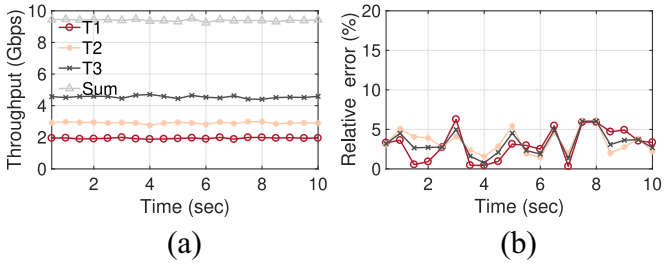

Fig. 13. Time series of the aggregate (a) throughput and relative (b) error ratio when weights and the number of CPU cores are different in case 2. This suggests that TONIC can enforce weighted fair sharing among tenants well without regard to tenant configurations.

is much lower. For example, when all the tenants are active for the last 5 s, XPS violates tenant weights significantly whereas TONIC achieves weighted fairness. Overall, the error ratios of TONIC are within 0.02%−5.65% whereas those of XPS are within 5.41%–72.96%.

For case 2, all the tenants start the corresponding number of flows in Table II at the same time and last transmission for 10 s. Fig. 13 shows the time series of throughput and relative error ratio of TONIC. We omit the results of XPS because the measured throughput of case 2 is very similar to the result of case 1. In Fig. 13(a), we find that TONIC achieves weighted fair sharing regardless of the tenant configuration. We also observe that TONIC maintains line-rate throughput. The error ratios shown in Fig. 13(a) are similar to those in Fig. 12(b) that the ranges are within 0.03%−6.27% and its average is 3.2%.

D. Traffic Prioritization With Hierarchical Policies

We conduct a series of experiments to demonstrate that TONIC can prioritize the traffic of high priority applications within a tenant without harming the policy hierarchy. We consider two tenants, T1 and T2. The tenants have 3 CPU cores and the equal weight of 3. T1 runs a throughput-sensitive application only. T2 runs two applications where one is latency sensitive and the other is throughput sensitive. The two tenants should share bandwidth equally and the latency-sensitive application should be prioritized over the throughput-sensitive application of T2. We use `iperf` for throughput-sensitive applications. For latency-sensitive applications, we use two different applications: 1) `sockperf` [37] and 2) `memcached` [14]. `sockperf` [37] is a network benchmarking tool that can measure latency in fine granularity.

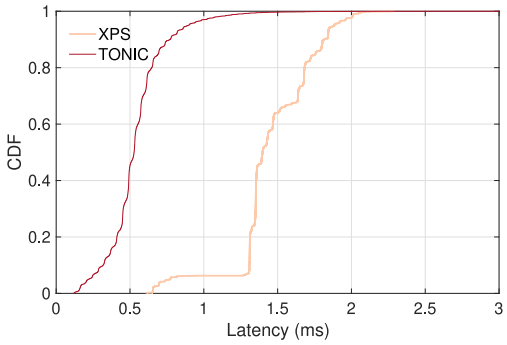

Fig. 14. CDF of measured latency using `sockperf` for T2.

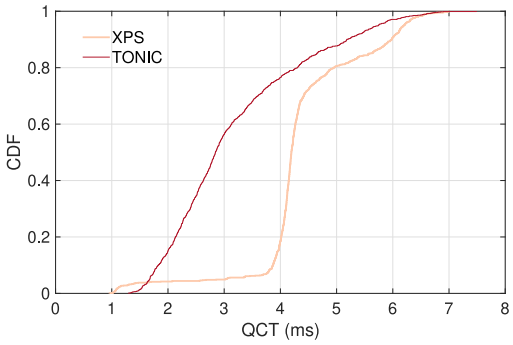

Fig. 16. CDF of the QCT in `memcached` with 1-kB items.

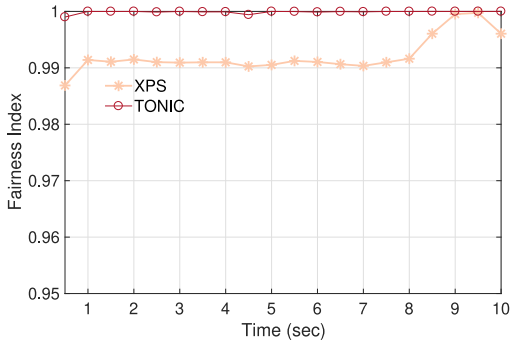

Fig. 15. Time series of Jain's fairness index between the two tenants.

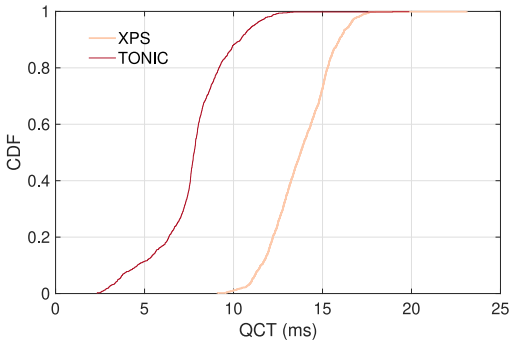

Fig. 17. CDF of the QCT in `memcached` with 16-kB items.

`Memcached` is a widely used KVS for modern online services, such as Web search and video streaming [6].

Methodology and Results With Sockperf: We first use `sockperf` as the latency-sensitive application of T2. In this experiment, we measure the end-to-end latency for 10 s with `sockperf`. `sockperf` continuously sends next messages when the response of the current messages is received. Overall, 8816 and 3269 messages are generated for TONIC and XPS during 10 s, respectively. The difference in number of messages is due to the difference of per-message latency. For throughput-sensitive applications, the two tenants start eight `iperf` flows at the beginning to saturate the link capacity. We measure the aggregate throughput of each tenant and calculate Jain's fairness index between the tenants every 0.5 s.

Figs. 14 and 15 suggest that TONIC can enforce hierarchical network policies without hardware modifications. Fig. 14 shows the latency results with `sockperf`. We can clearly observe that TONIC shows a lower latency than XPS. Compared to XPS, TONIC speeds up the average latency by 2.94×. For the 99th percentile latency, TONIC beats XPS by 2.24×. This is because TONIC makes the packets of `sockperf` pass over the buffered packets of `iperf`. Meanwhile, XPS cannot prioritize `sockperf` traffic due to the static RR scheduler in the NIC, resulting in poor latency. Fig. 15 shows the time series of Jain's fairness index between T1 and T2 over time. We can see that TONIC maintains Jain's fairness index close to 1 across time. Unlike TONIC, XPS provides imperfect intertenant isolation, which stems from that XPS does not distribute traffic across queues uniformly.

Methodology and Results With `Memcached`: We now use `memcached` [14] for the latency-sensitive application of T2. Since the services are user-facing, the QCT is crucial to user experience. We prepopulate a server instance by varying the item sizes using `PUT` operation. We use 1 kB and 16 kB items. The 1 kB item represents a typical item value size in KVS [3], [5], [6]. The 16 kB item is to examine the impact of item size on the performance. The client sends a `GET` query to the server and the server responses with the item matched with the requested key. We generate 1 K queries and measure the QCT for each of the queries. Like the `sockperf` experiment, we let each tenant generate 8 `iperf` flows at the beginning for throughput-sensitive applications. In these experiments, we omit the throughput fairness results because they are very similar to the results of the `sockperf` experiment.

Fig. 16 compares XPS and TONIC in the QCT. We find that TONIC generally outperforms XPS. TONIC is better than XPS in the average QCT by 1.37×. However, TONIC underperforms XPS for the 10th percentile QCT. We suspect that this is the impact of delay caused by packet reordering. For the 99th percentile tail QCT, which is the most important for user experience, TONIC is worse than XPS by 0.99×. The reason why TONIC does not beat XPS in the tail QCT is that the retransmission delay due to packet reordering increases the overall QCT although head buffering reduces the queueing delay. The above results suggest that TONIC can provide better performance for KVS through traffic prioritization while preserving policy hierarchies.

For deep dive, we inspect the impact of item size on the performance. Note that 90% of flows in modern workloads are smaller than 10 kB [19]. Fig. 17 shows results with 16 kB items. It is easy to see that TONIC outperforms XPS. By

comparing this result with the 1-kB result, we can know that larger items bring more performance gains to TONIC. For example, we observe that TONIC significantly outperforms XPS for the 10th percentile QCT. For 99th percentile QCT, TONIC achieves better performance than XPS by 1.40×. We suspect that this is because 16-kB flows are less sensitive to the queueing delay and packet loss than 1-kB flows.

VI. Related Work

Programmable Scheduling: PIFO [11] is a programmable scheduling model that can express various packet scheduling algorithms using rank-based enqueueing. Owing to its rich expressiveness, there exist several works that employ PIFO for packet scheduling. Loom [8] is a NIC hardware design that uses multiple PIFO blocks to express policy hierarchies. Eiffel [38] is a software packet scheduling mechanism that also adopts PIFO. Unfortunately, PIFO has a limited capacity that provides only 2K flows at line rate due to increased packet processing overhead [11], [12]. Motivated by the limitation of the PIFO model, push-in-extract-out (PIEO) [12] scales PIFO to 30K flows by dequeueing packets from arbitrary positions in packet queues. Similar to PIFO, PIEO can express hierarchical scheduling using multiple PIEO scheduling blocks. However, this also degrades the scalability of PIEO by increasing scheduling overhead. Unlike the above scheduling models, TONIC does not degrade scalability since our solution space is limited to the existing packet processing layers in the network stack. In addition, while the models require hardware modifications and long deployment time, TONIC can approximate hierarchical scheduling with existing commodity multiqueue NICs.

Meanwhile, motivated by that PIFO requires new hardware, SP-PIFO [39] approximates PIFO on existing programmable hardware at line rate by rotating queue priorities dynamically. However, it inherits the drawbacks of PIFO including insufficient scalability. In addition, since SP-PIFO basically targets programmable switches of a specific switch ASIC vendor and a specific programming language [40], it is unclear whether SP-PIFO can be implemented on programmable NICs at line rate, which have different hardware architectures and less resources compared to the switch.

Per-Flow NIC Scheduling: There exist NIC scheduling solutions that aim at ensuring per-flow fairness. Unlike TONIC, these works do not support multitenancy and cannot enforce hierarchical network policies. SENIC [7] is a NIC design that provides tens of thousands of hardware queues for scalable per-flow rate limiting. Titan [9] provides per-flow fairness with by assigning flows to hardware queues fairer than the per-flow hash scheme. Titan shows that we can configure queue weights by modifying NIC drivers. Specifically, it tracks the number of flows per queue as queue weights. However, it is limited to a specific NIC model. In addition, as a side effect, it limits the available number of Rx queues to only four queues. This increases packet loss rate and degrades throughput because of the limited number of queues. It also harms the performance of receive side scaling (RSS), which has significant impacts on application performance [41]. Furthermore, direct weight exposure jeopardizes packet latency because weight update causes a PCIe write, which can take up to hundreds of microseconds. During this time, it is unavailable to transmit and receive packets. TONIC leverages multiple hardware queues to express tenant weights, and this is possible with commodity NICs without performance degradation.

Multiqueue SSD Scheduling: Multiqueue support is not a trend limited to NICs. Recent NVMe SSDs also support multiqueues for parallel storage I/O processing. Motivated by this trend, there exist several works that address multiqueue SDDs. FLIN [42] solves unfairness among concurrently executing applications by designing an interference-aware I/O request scheduler. MQFQ [43] also solves the unfairness problem in multiqueue SSDs by designing a fair scheduler. Compared to FLIN, MQFQ is more generic solution for various multiqueue I/O devices including multiqueue NICs. However, MQFQ cannot enforce hierarchical network policies and is unaware to multitenancy.

VII. Conclusion

This work addressed multiqueue NICs in multitenant data centers, which are the key infrastructure in the IoT system architecture. While multiqueue support is essential to drive line rates of modern NICs, commodity multiqueue NICs cannot enforce hierarchical network policies. Accordingly, we presented TONIC, a multiqueue NIC packet scheduling solution that can enforce hierarchical network policies with existing commodity multiqueue NICs by approximating hierarchical packet scheduling. TONIC reserves a number of hardware queues for each tenant to express tenant weights indirectly, and enqueues packets into the head of qdisc to prioritize latency-sensitive traffic within a tenant without violating hierarchies. We implemented a TONIC prototype and evaluated the performance on a testbed. The experimental results showed that TONIC can enforce hierarchical network policies consisting of weighted fair sharing and traffic prioritization with robustness to various conditions.

References

[1] A. Libri, A. Bartolini, and L. Benini, "pAELLA: Edge AI-based real-time malware detection in data centers," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9589–9599, Oct. 2020.

[2] J. Xie, L. Lyu, Y. Deng, and L. T. Yang, "Improving routing performance via dynamic programming in large-scale data centers," *IEEE Internet Things J.*, vol. 2, no. 4, pp. 321–328, Aug. 2015.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. ACM SIGMETRICS*, New York, NY, USA, 2012, pp. 53–64.

[4] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores," in *Proc. USENIX NSDI*, Boston, MA, USA, 2019, pp. 79–94.

[5] M. Blott, L. Liu, K. Karras, and K. Vissers, "Scaling out to a single-node 80gbps memcached server with 40terabytes of memory," in *Proc. USENIX HotStorage*, 2015, pp. 8–12.

[6] R. Nishtala *et al.*, "Scaling memcache at facebook," in *Proc. USENIX NSDI*, Berkeley, CA, USA, 2013, pp. 385–398.

[7] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: Scalable NIC for end-host rate limiting," in *Proc. USENIX NSDI*, Berkeley, CA, USA, 2014, pp. 475–488.

[8] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and efficient NIC packet scheduling," in *Proc. USENIX NSDI*, Boston, MA, USA, 2019, pp. 33–46.

[9] B. Stephens, A. Singhvi, A. Akella, and M. Swift, "Titan: Fair packet scheduling for commodity multiqueue NICs," in *Proc. USENIX ATC*, 2017, pp. 431–444.
[10] (2010). *Scaling in the Linux Networking Stack*. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt
[11] A. Sivaraman *et al.*, "Programmable packet scheduling at line rate," in *Proc. ACM SIGCOMM*, 2016, pp. 44–57.
[12] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proc. ACM SIGCOMM*, New York, NY, USA, 2019, pp. 367–379.
[13] D. Firestone *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *Proc. USENIX NSDI*, 2018, pp. 51–66.
[14] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, p. 5, Aug. 2004.
[15] (2018). *8 Surprising Facts About Real Docker Adoption*. [Online]. Available: https://www.datadoghq.com/docker-adoption/
[16] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. J. Shenker, "Extending networking into the virtualization layer," in *Proc. ACM HotNets*, New York, NY, USA, 2009, pp. 1–6.
[17] J. C. R. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *IEEE/ACM Trans. Netw.*, vol. 5, no. 5, pp. 675–689, Oct. 1997.
[18] G. Kim and W. Lee, "Protocol-independent service queue isolation for multi-queue data centers," in *Proc. IEEE ICDCS*, 2020, pp. 355–365.
[19] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM IMC*, 2010, pp. 267–280.
[20] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM*, 2010, pp. 63–74.
[21] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM SIGCOMM*, 2015, pp. 123–137.
[22] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proc. ACM IMC*, 2017, pp. 78–85.
[23] A. Greenberg *et al.*, "Vl2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009, pp. 51–62.
[24] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, 2008, pp. 63–74.
[25] (2011). *Intel 82599 10GbE Controller*. [Online]. Available: https://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html
[26] (2015). *Intel X710/XXV710/XL710 Controller*. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/332464
[27] (May 2020). *Mlnx_en for Linux User Manual*. [Online]. Available: https://docs.mellanox.com/display/MLNXENv490170/
[28] *iPerf2: A Tool for Measuring TCP and UDP Network Performance*, 2020. [Online]. Available: https://sourceforge.net/projects/iperf2/
[29] (2011). *Data Center Bridging Task Group*. [Online]. Available: http://www.ieee802.org/1/pages/dcbridges.html
[30] A. Saeed, N. Dukkipati, V. Valancius, V. T. Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end hosts," in *Proc. ACM SIGCOMM*, New York, NY, USA, 2017, pp. 404–417.
[31] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proc. USENIX NSDI*, Berkeley, CA, USA, 2016, pp. 501–521.
[32] H. Sariowan, R. L. Cruz, and G. C. Polyzos, "SCED: A generalized scheduling policy for guaranteeing quality-of-service," *IEEE/ACM Trans. Netw.*, vol. 7, no. 5, pp. 669–684, Oct. 1999.
[33] C. Guo *et al.*, "RDMA over commodity Ethernet at scale," in *Proc. ACM SIGCOMM*, 2016, pp. 202–215.
[34] A. Langley *et al.*, "The QUIC transport protocol: Design and internet-scale deployment," in *Proc. ACM SIGCOMM*, 2017, pp. 183–196.
[35] Q. D. Coninck *et al.*, "Pluginizing QUIC," in *Proc. ACM SIGCOMM*, New York, NY, USA, 2019, pp. 59–74.
[36] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," Digit. Equip. Corp., Maynard, MA, USA, Rep. DEC-TR-301, Sep. 1984.
[37] (2020). *Sockperf: Network Benchmarking Utility*. [Online]. Available: https://github.com/Mellanox/sockperf/
[38] A. Saeed *et al.*, "Eiffel: Efficient and flexible software packet scheduling," in *Proc. USENIX NSDI*, 2019, pp. 17–32.
[39] A. G. Alcoz, A. Dietmüller, and L. Vanbeve, "SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues," in *Proc. USENIX NSDI*, Santa Clara, CA, USA, Feb. 2020, pp. 59–76.
[40] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.
[41] T. Barbette, G. P. Katsikas, G. Q. Maguire, and D. Kostić, "RSS++: Load and state-aware receive side scaling," in *Proc. ACM CoNEXT*, New York, NY, USA, 2019, pp. 318–333.
[42] A. Tavakkol *et al.*, "Flin: Enabling fairness and enhancing performance in modern NVME solid state drives," in *Proc. ISCA*, Jun. 2018, pp. 397–410.
[43] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-queue fair queuing," in *Proc. USENIX ATC*, 2019, pp. 301–314.

Gyuyeong Kim (Member, IEEE) received the B.S. and Ph.D. degrees in computer science from Korea University, Seoul, South Korea, in 2012 and 2020, respectively.

He is currently a Research Professor with the Future Network Center, Korea University. His research interests include networked systems, cloud computing systems, and programmable hardware.

Wonjun Lee (Fellow, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1989 and 1991, respectively, the M.S. degree in computer science from the University of Maryland, College Park, MD, USA, in 1996, and the Ph.D. degree in computer science and engineering from the University of Minnesota, Minneapolis, MN, USA, in 1999.

In 2002, he joined the Faculty of Korea University, Seoul, where he is currently a Professor with the School of Cybersecurity. He has authored or coauthored over 220 papers in refereed international journals and conferences. His research interests include communication and network protocols, optimization techniques in wireless communication and networking, security and privacy in mobile computing, and RF-powered computing and networking.

Prof. Lee has served on the TPC and/or Organizing Committee Member of IEEE INFOCOM from 2008 to 2021, a PC Vice Chair of IEEE ICDCS 2019, and the ACM MobiHoc from 2008 to 2009, and over 130 international conferences.