Enabling Work-Conserving Bandwidth Guarantees for Multi-Tenant Datacenters via Dynamic Tenant-Queue Binding

Zhuotao Liu*†, Kai Chen*, Haitao Wu‡, Shuihai Hu*, Yih-Chun Hu†, Yi Wang§, Gong Zhang¶
*Hong Kong University of Science and Technology, †University of Illinois at Urbana-Champaign
‡Google, §Southern University of Science and Technology, ¶Huawei Future Network Theory Lab
†{zliu48, yihchun}@illinois.edu, *{kaichen,shuaa}@cse.ust.hk
‡{haitaowu}@google.com, §{yw}@ieee.org, ¶{nicholas.zhang}@huawei.com

***Abstract*— Today's cloud networks are shared among many tenants. Bandwidth guarantees and work conservation are two key properties to ensure predictable performance for tenant applications and high network utilization for providers. Despite significant efforts, very little prior work can *really* achieve both properties simultaneously even some of them claimed so.**

In this paper, we present QShare, a comprehensive in-network solution to achieve bandwidth guarantees and work conservation simultaneously. QShare leverages weighted fair queuing on commodity switches to slice network bandwidth for tenants, and solves the challenge of queue scarcity through balanced tenant placement and dynamic tenant-queue binding. We have implemented a QShare prototype and evaluated it extensively via both testbed experiments and simulations. Our results show that QShare ensures bandwidth guarantees while driving network utilization to over 91% even under unpredictable traffic demands.

I. Introduction

Sharing the network of multi-tenant datacenters has been critical for public clouds. The two primary goals, among others, are bandwidth guarantees and work conservation. Bandwidth guarantees ensure predictable lower bound network performance for tenant applications. Recent studies show that, without bandwidth guarantees, network performance may experience over 5x variations, degrading application performance [6]. Work conservation enables a tenant to use spare bandwidth beyond its minimum guarantee to further improve tenant application performance, and boost provider network utilization. Given that datacenter traffic is bursty in nature and that the average network utilization is low [7, 16, 23], work conservation can deliver over 10x additional bandwidth to a VM upon its minimum guarantee [21].

However, it is hard to enable both properties simultaneously. Oktopus [6] and SecondNet [12] achieve bandwidth guarantees, but they are not work-conserving. Seawall [24] achieves work conservation without offering bandwidth guarantees.

ElasticSwitch [21] takes the first attempt toward achieving both properties simultaneously. It is an endhost based solution that first translates per-VM hose-model bandwidth guarantees into VM-to-VM pair rate limiters (referred as Guarantee Partitioning, GP), and then dynamically allocates spare bandwidth to these VM pairs to achieve high utilization (referred as Rate Allocation, RA). However, this approach suffers from two key challenges: (i) since tenant applications are agnostic to network operators, it is hard for GP to accurately capture the real communication patterns among VMs, affecting bandwidth guarantees; (ii) to detect spare bandwidth, RA needs to probe the network by increasing rates, which causes a tradeoff between bandwidth guarantees and work conservation—a conservative RA sacrifices work conservation, while an aggressive RA affects other tenants' bandwidth guarantees.

Trinity [13] moves one step further to complement ElasticSwitch with a simple in-network support. It uses two priority queues in switches to segregate and prioritize the bandwidth guarantee traffic over work conservation traffic, so that aggressive RA of one tenant does not affect bandwidth guarantees of others. While Trinity solves the second challenge of ElasticSwitch, it still suffer from the first challenge shown above. Further, it incurs other issues such as packet reordering and starvation due to traffic segregation and priority queuing.

Due to these challenges, prior solutions, essentially, do not achieve both goals in a sufficient manner. To give some sense, our testbed experiments show that given unknown communication pattern and demands, state-of-the-art solutions cannot achieve sufficiently good work-conversation (given bandwidth guarantees are satisfied), which, for instance, causes 2x long FCTs for tenant applications compared with our solution. Motivated by this, in this paper, we propose QShare, the first comprehensive in-network solution that addresses all above challenges to achieve both goals in a sufficient manner. Unlike Trinity [13] that uses two priority queues to segregate two traffic types, QShare directly leverages weighted fair queues (WFQs)[1] in commodity switches to slice network bandwidth for tenants. As a result, this enables: (i) bandwidth guarantees are achieved via proper queue weight configuration and tenant placement rather than endhost rate limiters, thus eliminating GP; (ii) network links are driven to full utilization instantly as long as one tenant has sufficient demand; (iii) no matter how aggressively a tenant transmits, bandwidth guarantees of other tenants are not affected as they are served in separate weighted queues; (iv) no packet reordering or starvation arises.

While promising, QShare introduces a new challenge of queue scarcity: the number of queues on a commodity switch port can be less than the number of tenants served by this port (§VII-C1). To address this challenge, we make the following observation: although the total number of embedded tenants associating with a port may be large, during a short time interval (*e.g.,* a few seconds), the number of concurrent tenants whose traffic demands exceed their bandwidth guarantees is small. This is also supported by the measurement results in production datacenters, where the average link utilization is

[1]We note that while WFQ may have been considered in other contexts [17], this work is the first one to exploit WFQ to enable work-conserving bandwidth guarantees for cloud network sharing.

low [7, 16, 23]. Thus, to support more tenants with limited queues, QShare dynamically assigns dedicated queues for tenants with higher demands than their guarantees, while serving low-demanded tenants in a shared queue altogether.

QShare is mainly composed of two modules: a balanced tenant placement module and a dynamic tenant-queue binding module. The tenant placement module is responsible for allocating network resources to tenants to provide bandwidth guarantees. To facilitate the dynamic queue allocation for embedded tenants, our placement also tries to balance the usage of switch ports among tenants to avoid overwhelming certain ports. The tenant-queue binding module then takes into account the traffic demands of tenants and their payment factors to dynamically distribute queues among tenants.

We have implemented a prototype of QShare with ~2000 lines of code (C for Linux kernel space and Python for user space), and evaluated it via extensive testbed experiments and simulations. Our results suggest that:

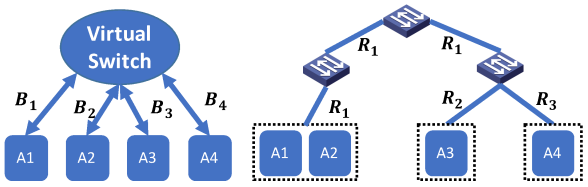
- Without affecting bandwidth guarantees, QShare achieves *perfect* work conservation given correct prediction on demand trends (not the exact demand), and over 91% link utilization given completely *unpredictable* demands.
- Given the above desirable properties, QShare significantly benefits tenant applications, for instance, by reducing their flow completion times (FCTs) by up to 50% compared with ElasticSwitch and Trinity [13, 21].
- Under production datacenter settings, QShare can assign dedicated queues to ~90% of all embedded tenants even when the datacenter is fully reserved, yielding at least 3× throughput gain over guaranteed bandwidth.

II. Background and Motivation

A. Background

Network bandwidth guarantees are preferable properties in cloud computing to offer tenants predicable performance. Typically, Hose model is used to model VM bandwidth guarantees [8]. Figure 1(a) plots a simple symmetric hose model for a tenant A's VMs, and Figure 1(b) shows the required bandwidth on each link to satisfy A's guaranteed bandwidth. Providing accurate bandwidth guarantees for VMs that can use multiple paths is an open problem since it requires a *perfect* load balancer to accurately distribute each VM's traffic over multiple paths such that the sum of guarantee on each path matches the total guaranteed bandwidth. Thus, prior works for providing bandwidth guarantees either consider tree-based network topology [6, 12, 21] or confine each tenant's traffic within a tree in multi-path network topology [14, 18]. QShare belongs to the second category since datacenters are often built redundant paths. QShare, however, can still fully utilize network bisection bandwidth via balanced tenant placement.

Work conservation is desired for achieving efficient resource utilization. In the context of multi-tenant datacenters, work conservation is defined as follows: for any link L in the network, as long as there exists at least one tenant that has packets to send along link L, L cannot have spare bandwidth [20].



(a) Hose model for B.G. (b) Bandwidth reservation

Fig. 1: Figure 1(a) shows a tenant A's bandwidth guarantees defined in a symmetric hose model. Figure 1(b) shows the reserved bandwidth on each link: $R_1 = \min\{B_1+B_2, B_3+B_4\}$; $R_2 = \min\{B_3, B_1+B_2+B_4\}$; $R_3 = \min\{B_4, B_1+B_2+B_3\}$.

B. State-of-the-Art Solutions

ElasticSwitch [21] makes the first attempt to achieve work-conserving bandwidth guarantees. It is an endhost based solution composed of two modules: a Guarantee Partitioning (GP) module that divides VM X's hose-model guarantee into guarantees to/from each other VM that X communicates with, and a Rate Allocation (RA) module that tries to assign spare bandwidth to these VM pairs to achieve high link utilization. However, it suffers from the following two key challenges.

First, since the VM traffic matrix (TM) of a tenant is typically agnostic, GP must estimate each VM-pair's demand via periodic source-destination VM coordination and throughput measurement. Whenever the TM changes, GP needs to re-estimate the TM even if per-VM demand remains the same (see detailed analysis in §10 of [19]). Given highly dynamic TM for cloud applications, it is challenging for the GP to capture the real communication pattern and estimate the TM correctly, especially considering that tens of thousands of VMs can produce billions of VM pairs.

Second, RA in ElasticSwitch [21] probes the network by increasing rates, detects congestion via packets losses or ECN, and then allocates possible spare bandwidth to VM pairs in max-min fashion following weighted TCP algorithms [24] As shown in [13, 21], it has a tradeoff between accurately providing bandwidth guarantees and being work-conserving: aggressive RA could affect other tenants' guarantees whereas conservative RA ends up with non-trivial bandwidth waste.

Trinity [13] moves one step further to complement the endhost based ElasticSwitch with simple in-network support. It exploits two priority queues in switches to segregate and prioritize the bandwidth guarantee traffic over work conservation traffic. As a result, VMs can aggressively send work conservation traffic without affecting bandwidth guarantees of others. Thus, Trinity achieves work conservation in static context, *i.e.,* the TM is a priori knowledge. However, it still inherits the challenges of performing GP in dynamic context. Further, Trinity raises packet reordering and starvation issues since network packets are served with strict priorities.

We do preform detailed experiments and analysis to quantify these limitations in §10 of our technical report [19]. Motivated by this, we propose QShare to address these challenges.

III. QShare Overview

QShare offers the first comprehensive in-network solution to address the above challenges. Instead of using two priority

queues to segregate traffic for two different types, QShare leverages multiple weighted fair queues (WFQs) to slice network bandwidth for tenants. This enables QShare to provide tenant-level bandwidth guarantees and work conservation (rather than rigid VM pair level as in [13, 21]), thus leaving tenant applications full flexibility to use its allocated bandwidth as needed. Such tenant-level bandwidth guarantees are also used in [5, 6], but they fail to achieve work conservation.

At the very high level, QShare's design has two modules: a balanced tenant placement module and a dynamic tenant-queue binding module. The placement module first seeks to provision tenant network to ensure bandwidth guarantees. Further, it *balances* the usage of switch queues among tenants to reduce the stress of performing dynamic queue allocation in the binding module. The tenant-queue binding module dynamically assigns dedicated queues to tenants whose applications tend to have higher demands than their guaranteed bandwidth, and meanwhile serves all low-demanded tenants in a shared queue. As a result, high-demanded tenants can burst their traffic in arbitrary communicate patterns without affecting other tenants. This design is the key to avoid the challenging GP and to eliminate the tradeoff between bandwidth guarantees and work conservation in the endhost based solutions [13, 21].

A challenge of the binding mechanism is how to assign dedicated queues to right tenants since traffic demand is dynamic. QShare addresses the challenge as follows. First, rather than predicting complete traffic matrix for each tenant as in [13, 21], QShare's demand prediction uses only a scalar metric for each tenant to reduce the stress of prediction. Second, to improve the worst case performance given inaccurate prediction and high-demanded tenants are mistakenly placed in the shared queue, QShare can employ ElasticSwitch for tenants in the shared queue to achieve moderate work-conserving bandwidth guarantees in the spirit of ElasticSwitch. Finally, we perform testbed experiments (§VII-A) to quantify effects of the binding mechanism: (i) the average utilization deficit caused by binding errors is less than 9% of the total capacity; (ii) to achieve good performance, it is sufficient to perform dynamic binding at more coarse time granularity (a few seconds) compared with the traffic matrix estimation performed at the granularity of milliseconds in [13, 21].

IV. Balanced Tenant Placement

The goals of tenant placement are (i) provisioning virtual networks for tenants to satisfy their computation and bandwidth guarantees and (ii) balancing the overall switch queue utilization among tenants. The prior placement algorithms proposed in [6, 18] aim to maximize the number of accepted tenant requests, which is an NP-hard problem similar to [8]. However, different from prior algorithms that make greedy embedding decisions (*i.e.,* embed a tenant immediately once a feasible option is found), our balanced tenant placement requires global topology investigation, i.e., evaluating all feasible options before making embedding decisions. Towards this end, we design our own tenant placement algorithm. Formulated in Algorithm 1, our placement algorithm contains two major parts (i) TR candidate exploration and (ii) TR candidate election.

Algorithm 1: Balanced Tenant Placement

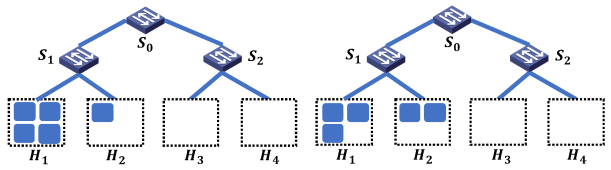
```
1  Input: A tenant request with explicit guarantees.
2  Output: The desired TR or an embedding error.
3  layer ← 1;
4  while True do
5      TRs ← get_TRs_at_layer(layer);
6      for T ∈ TRs do
7          [feasible, cost] ← evaluate_TR(T);
8          if feasible then TR_candidates.add((T, cost));
9      if TR_candidates is empty then
10         layer ← layer + 1;
11         if layer > n then return False;
12     else
13         return get_desired_TR(TR_candidates)
14 Function: evaluate_TR(T):
15     OA ← get_optimal_allocation(T);
16     if OA is feasible then return [True, (c_b, c_q)];
17     else return [False, null];
```

A. TR Candidate Exploration

We explain TR candidate exploration in the widely adopted multi-rooted tree datacenter topology [3, 10]. Given a tenant request, Algorithm 1 explores the topology from the lowest layer (hypervisor layer) towards the highest layer (core switch layer). At each layer, function get_TRs_at_layer (line 5) obtains all TR options at this layer. A layer-i TR option is a tree rooted at layer i. Its leaves are all servers reachable from the root using only downward paths. The algorithm then evaluates these TRs to produce *feasible* ones, called TR candidates (line 7). Generally speaking, a TR option is feasible if it has enough capacity to accommodate the tenant. Function evaluate_TR, detailed in §IV-B, determines such feasibility. If no TR candidates can be found, the algorithm continues exploration in the next layer (line 9). Otherwise, it stops further exploration and returns the desired TR elected from all candidates (line 13) using the criteria described in §IV-B. The early return confines tenants at the lowest possible layer to avoid unnecessary network usage at higher layers. If no TR candidates can be found after exploring the entire topology with n layers, the algorithm returns false (line 11), indicating an embedding error due to the lack of resources.

B. TR Evaluation and Candidate Election

A TR option is feasible if (i) the total available VM slots from all its servers are enough to hold the tenant's VMs and (ii) each link of the TR has enough available capacity to satisfy the tenant's bandwidth guarantees. Although evaluating the first rule is simple, the second rule requires more investigation. In particular, given a TR option, the amounts of bandwidth required on its links depend on the VM locations inside the TR. Specifically, consider a symmetric hose model where all VMs have the same guarantee $\mathbf{B}$. Given a link L of the TR, removing L breaks the TR into two disjoint components. If m VMs are in one component and n VMs are in the other one, the bandwidth required on L is $\mathbf{B} \cdot \min\{m, n\}$. Figure 2 plots a TR rooted at S_1. For the VM location in Figure 2(a), the link



(a) Requring $\mathbf{B}$ on $S_1 \leftrightarrow H_1$ (b) Requring $2\mathbf{B}$ on $S_1 \leftrightarrow H_1$

Fig. 2: Given the TR rooted at S_1, the bandwidth required on its link depends on VM locations inside the TR.

$S_1 \leftrightarrow H_1$ (and $S_1 \leftrightarrow H_2$) needs to reserve $\mathbf{B}$ ($\min\{\mathbf{B}, 4 \cdot \mathbf{B}\}$) whereas $2\mathbf{B}$ is required for the VM location in Figure 2(b).

To reduce the total network bandwidth required for embedding the tenant, function get_optimal_allocation (line 15) produces the optimal VM location that requires the least bandwidth reservation. For homogeneous hose models, the optimal allocation is produced as follows: (i) find the server H in the TR with the largest *usable VM slots*, (ii) allocate as many VMs as possible to H, (iii) update the remaining network/server capacity after allocation and (iv) repeat step one until either all VMs are allocated (indicating feasibility) or all servers in the TR have been investigated (indicating infeasibility). The usable VM slots for H in the TR is restricted by both the available VM slots in H and the available bandwidth on the path from H to the TR's root. For instance, in Figure 2, if we assume the available bandwidth on link $S_1 \leftrightarrow H_1$ is less than $\mathbf{B}$, the usable VM slots in H_1 is 0, rather than 4.

If a TR's optimal allocation is feasible, it becomes a tenant routing candidate. Then we compute its bandwidth cost c_b as the total amount of bandwidth reserved for the tenant on the TR's links, and queue allocation cost c_q as the largest number of tenants served by any of the TR's links (line 16).

Each TR candidate is associated with $cost$ combining c_b and c_q. The desired TR (line 13) is the one with least $cost$. One strategy for computing $cost$ is assigning more weight to c_q when the datacenter load is light to prefer more balanced placement whereas assigning more weight to c_b otherwise to prefer placement introducing fewer bandwidth cost.

Search Algorithm Complexity. The search complexity for embedding tenants depends on the layers at which Algorithm 1 returns. In a fattree topology [3], the worse case complexity (*i.e.,* the algorithm returns at the core switch layer) is $O(V^{\frac{5}{3}})$, where V is the number of nodes in the network. Thus, although our algorithm performs comprehensive topology search, its time complexity is polynomial rather than exponential. For topologies with higher over-subscription ratios, the complexity is reduced since the number of TR options at each layer is smaller. Further, the topology search results can be cached to achieve long-term efficiency [26].

V. Tenant-Queue Binding

To support more tenants with limited number of queues, QShare's design is inspired by how the working set of a process is often much smaller than the total memory it consumes. Similarly, only tenants whose traffic demands exceed their bandwidth guarantees need dedicated queues. Thus, there is an opportunity for QShare to dynamically allocate limited number of queues to high-demanded tenants. Specifically, QShare periodically evaluates each tenant and allocates queues among tenants based on their *scores*. Each tenant's score encapsulates its *usage factor* (§V-A) and *payment factor* (§V-B) so as to prioritize high-demanded and honest tenants.

A. Tenant Demand Trend Prediction

Since prior works [13, 21] rely on Guarantee Partitioning (GP) to achieve bandwidth guarantees, they need to predict each tenant's traffic matrix, *i.e.,* per VM-pair traffic demand. However, it is challenging to capture VMs' communication patterns and predict the traffic matrix since tenant applications are typically agnostic to cloud operators. On the contrary, QShare's tenant-queue binding module only needs to predict whether a tenant tends to have higher demands than its guaranteed bandwidth. Thus, rather than using traffic matrix, QShare proposes to use a scalar metric, usage factor (U-factor), to indicate a tenant's network utilization with respect to its guaranteed bandwidth. We do not claim that U-factor is the optimal metric for demand prediction. However, it does greatly reduce the stress of prediction by focusing on tenant-level *demand trend* rather than VM-level traffic matrix.

Each tenant's U-factor is computed per *control interval*. Specifically, in each control interval, hypervisors measure the bandwidth utilization of their hosted VMs. As VMs can have both inbound and outbound traffic, bi-directional bandwidth usage is considered. For instance, consider a hypervisor H_j hosting m VMs of a tenant $\mathbf{T}$. Then $\mathbf{T}$'s inbound (outbound) bandwidth usage U_j^i (U_j^o) measured by H_j is the sum of inbound (outbound) bandwidth usage from all these m VMs.

At the end of each control interval, QShare computes each tenant's U-factor. For tenant $\mathbf{T}$, its U-factor $\mathbf{U_T}$ is

$$\mathbf{U_T} = \min\{\max_{H_j \in \mathbf{H}} \frac{\max\{U_j^i, U_j^o\}}{B_j}, 1\}, \quad (1)$$

where $\mathbf{H}$ is the set of hypervisors managing $\mathbf{T}$'s VMs and B_j is $\mathbf{T}$'s guaranteed bandwidth on H_j's network interface. If H_j hosts m VMs from $\mathbf{T}$ (provisioned with total N VMs), then $B_j = \mathbf{B} \cdot \min\{m, N-m\}$ considering a symmetric and homogeneous model with per-VM guarantee $\mathbf{B}$.

The design rationale of Equation (1) is as follows. The innermost max is necessary as the high-demanded VMs may either send or receive large volumes of traffic. The middle max is designed to handle many-to-one traffic pattern in which many remote source VMs are communicating with a few local destination VMs. Although source hypervisors may measure small usage since source VMs are bottlenecked by destination VMs, $\mathbf{T}$'s application is actually high-demanded. Taking the largest usage among all hypervisors will capture such a traffic pattern. Finally, the outermost min sets a $\mathbf{U_T}$ cap of 1.

B. Tenant Lying Mitigation

Merely using U-factors to allocate queues has problems when handling dishonest tenants: a tenant can deliberately request smaller guaranteed bandwidth so as to have high U-factors. Note that no work-conserving allocation policies can completely prevent tenants from gaining advantages via

Algorithm 2: Queue Allocation Algorithm

```
1  Input: The set of embedded tenants 𝕊.
2  Output: Tenant-queue assignment.
3  Sort the tenants in 𝕊 decreasingly by their scores;
4  for T ∈ 𝕊 do
5      if T has a dedicated queue then continue;
6      else if T's TR has a spare queue then
7          enqueue_tenant(T);
8      else opportunistically_enqueue(T);
9      Update queue allocation state;
10 Queue weight computation;
11 Function: enqueue_tenant(T):
12 for L ∈ T's TR do
13     reserved_bandwidth ← B · min{m, N − m};
14 Function: opportunistically_enqueue(T):
15 for L ∈ T's TR do
16     get_opportunistic_queues_from_LSTs(L);
17 if T's TR has an opportunistic queue then
18     enqueue_tenant(T);
```

lying [20]. To mitigate the problem caused by lying, QShare proposes to consider payment factors, along with U-factors, when scoring tenants. Each tenant's payment factor and its guaranteed bandwidth are positively correlated such that deliberately requesting lower guarantees reduces a tenant's score whereas exaggerating guarantees requires higher payment. For simplicity, QShare assumes that a tenant's payment factor is proportional to the total guaranteed bandwidth required by its hose model.[2] Thus, given tenant $\mathbf{T}$ with N VMs and each VM requests guaranteed bandwidth $\mathbf{B}$, its payment factor is $kN\mathbf{B}$, where k is a constant depending on the pricing model.

Simplifying $\mathbf{U_T}$ in Equation (1) as $\min\{\frac{U^*}{B^*}, 1\}$, then tenant $\mathbf{T}$'s score $S_\mathbf{T}$ is computed as follows

$$S_\mathbf{T} = kN\mathbf{B} \cdot \mathbf{U_T} = \begin{cases} \tilde{k}U^*, & \text{if } \mathbf{U_T} < 1 \\ kN\mathbf{B}, & \text{otherwise} \end{cases} \tag{2}$$

$\tilde{k} = kN\mathbf{B}/B^*$, where B^* is determined by maximizing the inner max operation of Equation (1).

Using $S_\mathbf{T}$ as the criterion for queue allocation can mitigate problems caused by lying. On the one hand, as $S_\mathbf{T}$ is bounded by $kN\mathbf{B}$, deliberately requesting smaller $\mathbf{B}$ would result in a lower cap of $S_\mathbf{T}$, which is disadvantageous when competing with other tenants. On the other hand, deliberately requesting higher $\mathbf{B}$ also has problems since (i) tenant $\mathbf{T}$ has to pay more and (ii) its $S_\mathbf{T}$ is determined by $\mathbf{T}$'s real usage rather than its claimed guarantees if $\mathbf{U_T} < 1$. Generally, high-demanded tenants are preferred since $S_\mathbf{T}$ is non-decreasing as bandwidth usage increases, which is desirable for queue allocation.

C. Dynamic Queue Allocation

We present the queue allocation logic in Algorithm 2. A tenant is assigned a dedicated queue only if it is assigned a dedicated queue on each link of its TR. Otherwise, the tenant will be served in the shared queue on each link of its TR. To prioritize tenants with higher scores, Algorithm 2 starts queue allocation from the tenant with the highest score (line 3).

If a tenant T currently has a dedicated queue, it continues to hold the queue for the next control interval (line 5). Otherwise, Algorithm 2 determines whether allocating T a dedicated queue is possible. To satisfy the condition on line 6, each link of T's TR needs to have at least one spare queue. If positive, function enqueue_tenant assigns T a queue on each link L of its TR (line 12). Otherwise, function opportunistically_enqueue (line 8) *opportunistically* finds queues for T by preempting queues from low-scored tenants (LSTs). Specifically, on link L without spare queues, the algorithm obtains an opportunistic queue from a tenant $\tilde{T}$ such that (i) $\tilde{T}$'s score is less than T's score and (ii) $\tilde{T}$'s score is the smallest among all tenants owning a queue on L (line 16). If the algorithm finds an available queue, either opportunistic or unoccupied, for each link of T's TR, we say T's TR has an opportunistic queue (line 17) and enqueue T. ALl dequeued tenants are served in shared queues during the next interval.

Queue allocation state is updated after handling T (line 9). Once queue allocations for all tenants are finished, QShare computes weight for each queue (line 10). For a queue $\mathcal{Q}_i$ on link L, its normalized weight is the ratio of reserved bandwidth in $\mathcal{Q}_i$ to the total amount of reserved bandwidth on link L.

D. Policy Enforcer

To enforce queue allocation decisions inside network, QShare needs to perform (i) packet tagging and (ii) network configuration. Packet tagging is to ensure that packets are served in correct queues. We use dscp tagging to achieve this. To avoid ambiguity, the D-tenants (tenants with dedicated queues) whose TRs share at least one common link cannot use the same dscp value. D-tenants whose TRs are non-overlapping can reuse the same dscp value. Since only 64 dscp values are available, finding the smallest possible number of dscp values in a legal assignment can be reduced to the k-coloring of a graph, which is NP-hard [9]. To address the dscp usage concerns, we analyze the efficiency of a greedy assignment in large scale datacenters based on production datacenter settings (see details in §VII-C1).

Network configuration involves configuring the queues on each link with proper weights and dscp values, which requires WFQ configuration on both ports of the link. For edge links connecting servers and switches, software WFQ is required on hypervisors. To achieve automation, QShare designs a network action container to perform configuration in a batch: operations on different switches are parallelized via multi-threading so that the marginal configuration latency is negligible [19].

Finally, QShare can run ElasticSwitch-like rate allocation mechanisms [24] for tenants without dedicated queues to improve worst-case performance. However, QShare imposes smaller overhead than ElasticSwitch [21] since it only performs rate allocations for tenants without dedicated queues.

VI. Implementation

The prototype of QShare contains both user-space and kernel-space programs, as shown in Figure 3. The user-space

[2]Payment for computation resources is not considered as QShare focuses on bandwidth management.

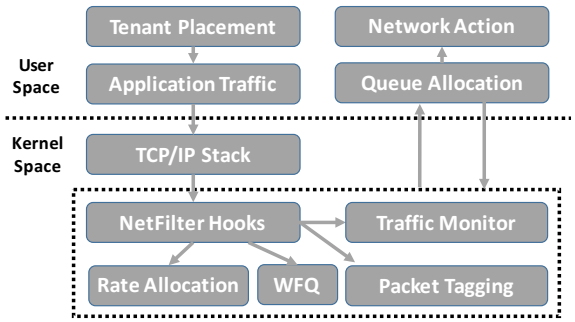


Fig. 3: The software implementation of QShare.

programs, executed globally, are responsible for managing the whole datacenter whereas the kernel-space program, running on each hypervisor, manages the local hypervisor. Two spaces interact with each other such that queue allocation decisions are made based on the distributed measurement reported by all hypervisors, and meanwhile the allocation decisions are pushed back to the kernel modules for enforcement on hypervisors. The implementation has $\sim$2000 lines of code (Python in user space and C in kernel space).

The user-space programs include tenant placement, queue allocation and the network action container. The kernel-space module, built on NetFilter [2], includes tenant traffic monitor, rate allocation (for tenants in shared queues), software WFQ (for tenants with dedicated queues) and packet dscp tagging. On each hypervisor, a user-space deamon (not plotted) based on Netlink [1] interacts with the kernel module.

Note that implementing a hypervisor that can support all kinds of VM management is out of this paper's scope. Our prototype builds a simple hypervisor that can support QShare-related operations, such as identifying the VMs of each tenant.

VII. Evaluation

Our evaluation centers around the following questions:

(i) How does traffic dynamic affect QShare's performance? With correct predictions on demand trend (not the exact demand), QShare achieves *perfect* work-conserving bandwidth guarantees: all bandwidth guarantees are satisfied and meanwhile the bottleneck link is fully utilized (§VII-A1). Even when demand trends are *completely unpredictable*, QShare drives the bottleneck link to over 91% utilization (§VII-A2).

(ii) How well can QShare benefit applications? Given the above desirable properties, QShare benefits tenant applications by reducing their flow completion times (FCTs) by up to 50% compared with the state-of-the-art solutions [13, 21] (§VII-B).

(iii) How well can QShare manage large scale datacenters? Based on observations from production datacenters, we show that QShare can assign dedicated queues to $\sim$90% of all tenants in any control interval in fully reserved datacenters. Thus, QShare offers tenants at least 3$\times$ throughput gain over their guaranteed bandwidth (§VII-C).

Testbed Setup. We build a physical testbed containing 10 servers and each server provisions 10 VM slots, for a total of 100 VMs. We evenly distribute the servers into two racks inter-connected by two Pronto-3297 48-port Gigabit switches. Thus, the topology is 5:1 oversubscribed and the core link is the bottleneck. Each port supports up to 8 WFQ queues. We embed multiple tenants in the testbed, with random sizes from 2 to 20 VMs. We develop a client/server program to generate traffic. The clients initiate long-lived TCP connections to randomly selected servers and request flow transfers. All VMs run both the client and server programs.

A. Work-Conserving Bandwidth Guarantees

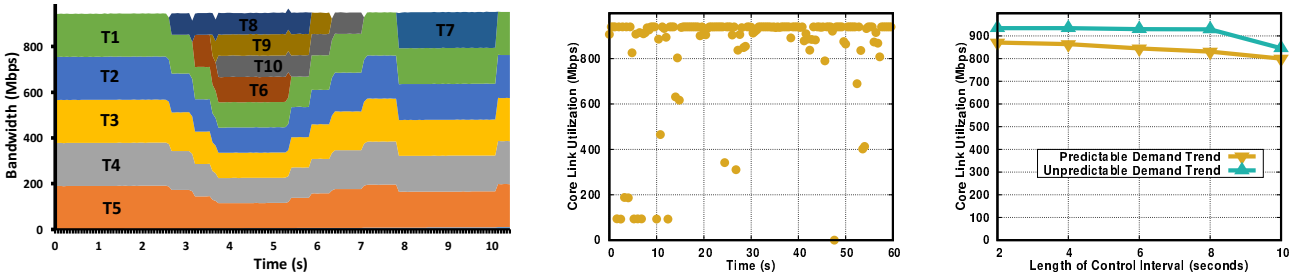
In this section, we consider how traffic dynamics may affect QShare's performance for enabling work-conserving bandwidth guarantees. We consider the following two scenarios. The first case is that a tenant's demand trend is predictable: *i.e.,* once a tenant has high traffic demand, this trend continues for few seconds. Trend predictability is not over-optimistic since hot spots in production datacenters can last over tens of seconds [16]. The second case is that the demand trend is *completely unpredictable*: *i.e.,* a tenant's future demands are independent on its current or previous demands. In both cases, QShare does not impose any constraint or assumption on VM communication patterns, *i.e.,* one client can request flow transfers from arbitrary servers at any time.

To quantify the worst-case performance degradation caused by traffic unpredictability, we first disable the ElasticSwitch-like rate allocations for the tenants without dedicated queues, and allocate them at most their guaranteed bandwidth.

1) Predictable Demand Trend: In this experiment, we consider 10 tenants competing on the core link. Each tenant is guaranteed 94 Mbps bandwidth on the core link. To generate traffic, we randomly pick 5 tenants (referred to as T1 to T5) as high-demanded tenants whose clients request sufficient flow transfers during our measurement period. The remaining tenants (referred to as T6 to T10) have insufficient demands during the measurement period. Low-demanded tenants may initiate their flow transfers at any time during the measurement period. We first fix the length of control interval as 4 seconds and consider other lengths in §VII-A2.

Figure 4(a) plots the runtime core link bandwidth obtained by each tenant in a 10-second measurement period. During this period, QShare's tenant-queue binding algorithm assigns each of the tenants in T1 to T7 a dedicated queue on the core link; T8, T9 and T10 are served in a shared queue. When low-demanded tenants are inactive at the early stage, T1 through T5 fairly share the entire core link capacity. Later on, low-demanded tenants T6, T8, T9 and T10 become active. As T8, T9 and T10 are in the share queue, they all obtain their guaranteed bandwidth. T1 to T6, each exclusively occupying a queue, equally share the remaining capacity. At about 8 second, T7 becomes active and fairly shares the core link with T1 to T5. It is clear that all tenants receive at least their guaranteed bandwidth regardless of their communication patterns and other tenants' demands. Meanwhile, the core link is always fully utilized. Thus, QShare achieves perfect work-conserving bandwidth guarantees.

2) Unpredictable Demand Trend: We now consider the case when tenant demand trend is unpredictable. Since traffic



(a) Per-tenant runtime bandwidth utilization given predictable demand trends.

(b) Aggregate core link utilization given unpredictable traffic trend.

(c) Average core link utilization with varying control intervals.

Fig. 4: Figure 4(a) plots runtime bandwidth utilization of each tenant given correct demand trend prediction. QShare achieves perfect work-conserving bandwidth guarantees in this case. Figure 4(b) plots the aggregate core link utilization given completely unpredictable demands. Only few under-utilized cases are observed, yielding over 91% average utilization. Figure 4(c) shows the average core link utilization given different lengths of the control interval.

predictability is only relevant to the tenant-queue binding module, we mainly focus on the performance of work conservation when handling unpredictable demands. To generate unpredictable traffic demands, each client requests flow transmissions from randomly selected servers. Flow sizes are sampled from the empirical datacenter workloads [4]. When the current flow finishes, a client randomly switches between being active (*i.e.,* requesting a new ow transmission) or dormant (*i.e.,* sleeping for a random period of time between 0 to 1 second before requesting a new flow transfer).

Figure 4(b) illustrates the runtime core link utilization over a one-minute measurement period. We measure the aggregated link utilization from all tenants at the granularity of 0.1 second. As illustrated in Figure 4(b), in spite of unpredictable demands, under-utilized cases are rare, rendering over 91% average link utilization (plotted Figure 4(c)). This is because that QShare does not rely on good TM estimation to achieve work conservation. Instead, for any D-tenant (tenant with a dedicated queue), its VMs can burst traffic using arbitrary communication patterns, allowing them to effectively grab possible spare bandwidth. As long as one VM pair from all D-tenants is high-demanded, it can drive the core link to full utilization. Mathematically, the probability that all VM pairs from D-tenants have insufficient demands is low. In particular, assuming each VM pair independently determines to be either active or dormant with equal probability during a small time interval, the probability that the core link observes insufficient demands in the small interval[3] is $(\frac{1}{2})^{\mathcal{N}}$, where $\mathcal{N}$ is the number of VM pairs from all D-tenants. Thus, demand unpredictability has minor effects on work conservation.

We further plot the average core link utilization for different lengths of the control interval in Figure 4(c). For predictable demand trend, QShare achieves perfect work conservation as long as the length of control interval is comparable with how long the trend lasts. For unpredictable trend, the utilization drops slightly as the length of control interval increases.

The takeaway of this evaluation is that to achieve good work conservation, (i) QShare does not require perfect demand trend prediction and (ii) it is sufficient to perform tenant-queue allocation at coarse time granularity (*e.g.,* seconds). *Thus, QShare's dynamic queue-tenant binding module does not need to react quickly enough to capture traffic bursts, which significantly reduces the stress for large scale deployment.*

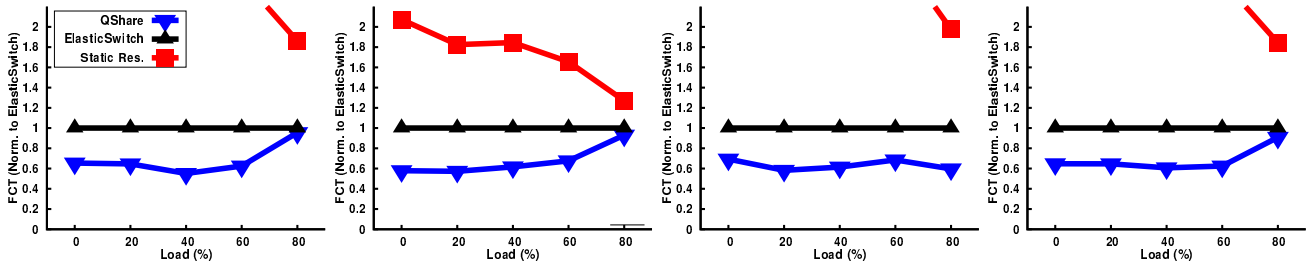
Fairness. The benefits of enabling ElasticSwitch-like rate allocations for tenants without dedicated queues are two-fold. First, it improves the link utilization for those under-utilized cases shown in Figure 4(b). Second, it improves the fairness for sharing spare bandwidth as both tenants in the shared queue and tenants with dedicated queues can utilize such bandwidth.

B. Tenant Application Benefits

Given the desirable property in §VII-A, QShare can benefit tenant applications by significantly reducing their flow completion times (FCTs). In this section, we demonstrate QShare's edges over ElasticSwitch [21], Trinity [13] and static reservation for improving FCTs. Among all embedded tenants, we consider one tenant $\mathbf{T}$ with 10 VMs evenly distributed in two racks. Tenant $\mathbf{T}$ has 94 Mbps guaranteed bandwidth on the core link. We consider the shuffle phase of MapReduce jobs where a client requests flow transfers from all servers (recall that a VM runs both the client and server program). The flow sizes are sampled from empirically observed traffic patterns in two deployed datacenter traces [10] and [4]. Each client requests a new flow once the previous one is finished.

In the experiment, we create different datacenter fabric loads by varying the guaranteed bandwidth of background tenants (*i.e.,* the tenants competing with $\mathbf{T}$ on the core link). The load is computed as the ratio of total guaranteed bandwidth from background tenants to the core link capacity. The results for using the enterprise datacenter workload [4] are plotted in Figure 5 (results for using the data-mining workload [10] are similar and we omit them for brevity). Because of the efficient resource utilization, QShare greatly reduces FCTs compared with both ElasticSwitch [21] and the static bandwidth reservation. Such improvement is even more significant for smaller fabric loads. In spite of its improvement over static reservation, ElasticSwitch [21] has a non-trivial performance degradation from QShare (up to $2\times$ long FCTs) even if it adopts very aggressive RA to probe available bandwidth (scarifying bandwidth guarantees [21]).

[3]Given a small interval (*e.g.,* sub-millisecond), a small flow transmission may be considered as sufficient demand.



(a) Overall average FCTs (b) Small flows (≤100KB) (c) Medium flows (d) Large flows (≥10MB)

Fig. 5: FCT results for varying fabric loads (some results for static reservation are out of the plot scope). QShare achieves much smaller FCTs than both static reservation and ElasticSwitch [21] (Trinity [13] has roughly the same FCTs as ElasticSwitch [21]).

We are aware that ElasticSwitch's performance depends on parameter settings and system tuning. Our self-implemented ElasticSwitch prototype uses the default parameter setting in its paper. We do not further plot the results of Trinity [13] since ElasticSwitch with aggressive RA has roughly the same performance with Trinity in terms of bandwidth utilization, whereas Trinity has reordering and starvation issues.

C. QShare in Large Scale

In this section, we first study the extent of switch queue scarcity in large scale datacenters. Further, we show QShare's benefit for providing tenants more bandwidth than their guarantees and improving link utilization efficiency in large scale datacenters. We consider a three-layer multi-rooted tree topology with 1024 servers and 100 VMs per server, for a total of 100 thousand VMs. The network interface of each server is 10 Gbps and the switch port capacity is 40 Gbps. The network topology is constructed based on the k=16 fattree [3] topology. By disabling certain links and switches, we can create a topology with different oversubscription ratios.

1) The Extent of Switch Queue Scarcity: Based on measurements in the production datacenters [6, 24], the number of VMs requested by each tenant follows an exponential distribution with mean 49. To better represent various bandwidth requirements from tenants, each VM randomly samples its required bandwidth from $\{10, 50, 100, 200, 300\}$ Mbps. In the experiment, we keep embedding tenants until either network resources or computation resources are fully reserved, *i.e.,* the datacenter operates at 100% load. To do a stress test for queue scarcity, we assign more weight to c_q in Algorithm 1. We test three different over-subscription ratios $1:1$, $4:1$ and $16:1$.

The tenant placement results are tabulated in Table I. Overall, the extent of queue scarcity is moderate, counterintuitive to the common assumption [20]. For instance, only ∼4% switch ports are overloaded in the $1:1$ over-subscribed topology. Two thirds of the tenants are assigned dedicated queues throughout their lifetime due to the lack of queue contention, *i.e.,* on any link of their TRs, the number of competing tenants is less than 8. ∼90% of all tenants can have dedicated queues, either permanently or opportunistically, in any control interval, indicating that only a small fraction of tenants need to run rate allocations on hypervisors. After placement, we analyze the dscp concern in §V-D. dscp 0 is reserved for tenants in shared queues. For each tenant with dedicated queues, we greedily assign it the next non-conflicting dscp value. It turns out that 64 dscp values are sufficient for fully reserved datacenter.

O. R.	$\mathbf{R}_{\mathbf{N}_L<9}$	$\mathbf{R}_{\mathbf{N}_L\in[9,12]}$	$\mathbf{R}_{\mathbf{N}_L>12}$	$\mathbf{R}_{\mathbf{N}_D}$	$\mathbf{R}_{\mathbf{N}_I}$
1 : 1	96.7	3.26	0	66.7	90.4
4 : 1	95.1	4.88	0	67.2	90.1
16 : 1	92.1	7.89	0	66.7	90.6

TABLE I: Tenant placement results in a large scale datacenter. $\mathbf{R}_{\mathbf{N}_L<9}$ is the percentage of ports serving less than 9 tenants. $\mathbf{R}_{\mathbf{N}_L\in[9,12]}$ and $\mathbf{R}_{\mathbf{N}_L>12}$ have similar definitions. $\mathbf{R}_{\mathbf{N}_D}$ is the percentage of tenants permanently assigned a dedicated queue and $\mathbf{R}_{\mathbf{N}_I}$ is the percentage of tenants assigned a dedicated queue, either permanently or opportunistically, in any control interval.

The takeaway for the evaluation is that in reality the problem of queue scarcity is moderate. By performing dynamic tenant-queue binding, QShare can effectively address such scarcity.

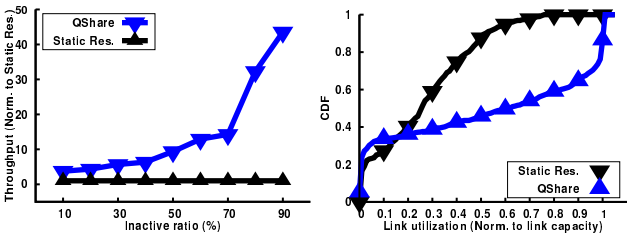
2) QShare's Performance in Large Scale: In this section, we evaluate QShare's performance based on large scale simulations. Due to the scalability of accurately simulating detailed packet-level commutations involving billions of VM pairs, our simulator does not further study the performance of ElasticSwitch [21] and Trinity [13] since both of them require GP that depends on accurately modeling packet-level communications. Instead, our simulator focuses on modeling tenant-level throughput, assuming tenant applications can use the network with arbitrary communication patterns. The experiment is performed on a 16:1 over-subscribed and fully reserved datacenter, since it has the highest level of queue scarcity compared with other settings. We define the inactive ratio r_{in} as the percentage of low-demanded tenants.

Throughput Gain. The throughput gain for a tenant is defined as the ratio of its actual achieved throughput to its guaranteed bandwidth. For simplicity, we assume the throughput gain for tenants in shared queues is 1 (no gain). For a tenant $\mathbf{T}$ with dedicated queues, its bandwidth gain on different links of its TR may be different since the actual demands on each link vary. We quantify the throughput gain of $\mathbf{T}$ as the smallest bandwidth gain obtained on any link of its TR. Thus, our experiment shows the worst-case throughput gain for $\mathbf{T}$ when the link with the smallest bandwidth gain is the bottleneck.

Figure 6(a) illustrates the average throughput gain given varying inactive ratios. Overall, QShare produces significant throughput gains (*e.g.,* over 3× for all inactive ratios) over bandwidth guarantees. The throughput gain increases dramatically (up to ∼50) as the inactive ratio increases, demonstrating that QShare can effectively utilize spare bandwidth.

	SecondNet [12], Oktopus [6], TIVC [25]	CloudMirror [18]	ElasticSwitch [21], Trinity [13]	EyeQ [15], GateKeeper [22]	Silo [14]	QJump [11]	QShare
BG	**Yes**	**Yes**	Tradeoff [21]	**Yes**	**Yes**	**Yes**	**Yes**
WC	No	No	Tradeoff [21]	**Yes**	No	**Yes**	**Yes**
Multi-tenant isolation & placement	**Yes**	**Yes**	No	**Yes**	**Yes**	No	**Yes**
Others	**None**	Application driven	TM estimation; Starvation & reordering [13]	Non-congested network core	**None**	**None**	**None**

TABLE II: Property comparison with closely related works. "BG" and "WC" mean bandwidth guarantee and work conservation.



(a) Throughput gain (b) Link utilization

Fig. 6: QShare's performance in large scale.

Utilization Efficiency. A direct benefit of work conservation is that network links are more effectively utilized. Specifically, consider that tenant $\mathbf{T}$'s throughput gain allows it to receive an extra 100 Mbps bandwidth besides its guaranteed bandwidth. This extra bandwidth will distribute among the links of $\mathbf{T}$'s TR, driving these links to higher utilization. Without loss of generality, we consider a communication pattern that spreads $\mathbf{T}$'s throughput gain across $\mathbf{T}$'s links proportionally to $\mathbf{T}$'s guaranteed bandwidth on these links. As the throughput gain is obtained as the minimal bandwidth gain among all links, this distribution will not drive any link to over 100% utilization.

Figure 6(b) plots the CDFs of normalized link utilization (to the link capability) in the datacenter given r_{in}=0.5. The results show that QShare achieves better efficiency in link utilization than static reservation. For instance, with QShare, half of the links' utilization is over $\sim$60% compared with $\sim$25% in static reservation; $\sim$14% links are fully utilized with QShare compared with 0 percentage in static reservation.

VIII. Related Work

Table II summarizes the properties of closely related work. SecondNet [12], Oktopus [6], and TIVC [25] provide static, non work-conserving bandwidth guarantees. EyeQ [15] and GateKeeper [22] achieve work-conserving bandwidth guarantees only if the network core is congestion-free, which may be not true for many datacenters [7, 16]. ElasticSwitch [21] and relies on challenging TM prediction and has a tradeoff between providing accurate bandwidth guarantees and being sufficiently work-conserving. Trinity [13] improves ElasticSwitch's work-conservation in static context via in-network priority queuing. However, it inherits the challenge of TM estimation and further raises starvation and packet reordering issues. Silo [14] and QJump [11] provide both bandwidth and in-network latency guarantee, but Silo is not work-conserving and QJump lacks the tenant placement and isolation.

IX. Conclusion

This paper presented QShare, the first comprehensive in-network solution enabling work-conserving bandwidth guarantees in multi-tenant datacenters. At its core, QShare's tenant placement module provides accurate bandwidth guarantees, and its tenant-queue binding module dynamically assigns high-demanded tenants dedicated switch queues to achieve work conservation. Our evaluation results show that QShare improves state-of-the-art solutions in two aspects: (i) it does not rely on challenging traffic matrix prediction to achieve good performance and (ii) it eliminates the tradeoff of providing good bandwidth guarantees and being work conserving without raising starvation or packet reordering issues.

X. Acknowledgements

We want to thank the anonymous reviewers for insightful comments. This research was partially supported by China 973 Program No.2014CB340300, HK GRF-16203715, ECS-26200014, CRF-C703615G and NSF CNS-1717313.

References

[1] Netlink. http://man7.org/linux/man-pages/man7/netlink.7.html.
[2] The Netfilter Project. http://www.netfilter.org.
[3] M. Al-Fares et al. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, 2008.
[4] M. Alizadeh et al. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*, 2014.
[5] S. Angel et al. End-to-end performance isolation through virtual datacenters. In *USENIX OSDI*, 2014.
[6] H. Ballani et al. Towards predictable datacenter networks. In *SIGCOMM*, 2011.
[7] T. Benson et al. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.
[8] N. G. Duffield et al. A flexible model for resource management in virtual private networks. In *ACM SIGCOMM*, 1999.
[9] M. R. Garey et al. Some simplified NP-complete problems. In *ACM STOC*, 1974.
[10] A. Greenberg et al. VL2: a scalable and flexible data center network. In *ACM SIGCOMM*, 2009.
[11] M. P. Grosvenor et al. Queues Don't Matter When You Can JUMP Them! In *USENIX NSDI*, 2015.
[12] C. Guo et al. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *ACM CoNEXT*, 2010.
[13] S. Hu et al. Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud. In *IEEE INFOCOM*, 2016.
[14] K. Jang et al. Silo: Predictable message latency in the cloud. In *SIGCOMM*, 2015.
[15] V. Jeyakumar et al. EyeQ: Practical network performance isolation at the edge. In *USENIX NSDI*, 2013.
[16] S. Kandula et al. The nature of data center traffic: Measurements & analysis. In *ACM IMC*, 2009.
[17] G. Kumar et al. Virtualizing traffic shapers for practical resource allocation. In *USENIX HotCloud*, 2013.
[18] J. Lee et al. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM*, 2014.
[19] Z. Liu et al. Enabling work-conserving bandwidth guarantees for multi-tenant datacenters via dynamic tenant-eue binding, 2017.
[20] L. Popa et al. FairCloud: Sharing the network in cloud computing. In *ACM SIGCOMM*, 2012.
[21] L. Popa et al. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM*, 2013.
[22] H. Rodrigues et al. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *USENIX WIOV*, 2011.
[23] A. Roy et al. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.
[24] A. Shieh et al. Sharing the data center network. In *USENIX NSDI*, 2011.
[25] D. Xie et al. The only constant is change: Incorporating time-varying network reservations in data centers. In *ACM SIGCOMM*, 2012.
[26] Q. Xu et al. Optimization framework for multi-tenant data centers, 2017. US Patent 9813301 B2.