C2QoS: CPU-Cycle based Network QoS Strategy in vSwitch of Public Cloud

Ye Yang*†, Haiyang Jiang*, Yulei Wu‡, Yilong Lv§, Xing Li§ and Gaogang Xie¶†
*ICT, CAS, China, †UCAS, China, ‡University of Exeter, UK, §Alibaba Group, China, ¶CNIC, CAS, China
Email:*{yangye, jianghaiyang}@ict.ac.cn, ‡y.l.wu@exeter.ac.uk, §{lvyilong.lyl, lixing.lix}@alibaba-inc.com, ¶xie@cnic.cn

***Abstract*—The network Quality-of-Service (QoS) strategy in vSwitch aims to guarantee the Service-Level-Agreement (SLA) for the concurrent Virtual Machines (VMs) residing on a particular server platform. Different from a hardware switch, the traffic forwarding tasks in vSwitch are completed by processes/threads running on some dedicated CPU cores. Existing vSwitch QoS strategies, inherited from the solutions in the hardware switch, are interface-based and ignore the isolation of these IO-dedicated CPU resources among VMs. As the result, they cannot ensure VMs' SLA targets due to the resource contention. In order to resolve the issue, we propose a CPU-Cycle based QoS (C2QoS) strategy, that contains a CPU-Cycle based Token-Bucket (C2TB) mechanism and a Hierarchical Batch Scheduling (HBS) mechanism. The C2TB apportions the IO-dedicated CPU resources to each VM for ensuring bandwidth, while the HBS schedules the VMs' forwarding tasks on these CPU cores to guarantee hierarchical latency. We implement the C2QoS strategy on the Data Plane Development Kit accelerated Open vSwitch (OVS-DPDK) platform. Experimental results show that compared with existing strategies, the influence of CPU resource congestion on bandwidth is eliminated and that on latency is reduced by 80%.**

I. Introduction

The public cloud has become a trend, since many enterprises and individuals, as tenants, are deploying services on it in the form of Virtual Machines (VMs) [1]–[4]. In a public cloud, a number of VMs are deployed on a particular physical server platform and share the resources like CPU, memory and network. How to fairly share these resources among tenants and provide Service-Level-Agreement (SLA) guarantee is the foremost issue for Cloud Service Providers (CSPs).

Among these resources, network resources are different from the hardware resources like CPU and memory. On a physical server, each VM's network connectivity is implemented by a software ***vSwitch***, that is responsible for the packets classification and forwarding [5], [6]. As the results, all the VMs compete with each other for the processing capacity of the vSwitch, essentially for the CPU resources occupied by the vSwitch. To make things worse, for a CSP it is a common practice to increase the share of CPU resources with VMs and thus very limited CPU resources are left for vSwitch's forwarding tasks, e.g., the Google cloud uses no more than two physical CPU cores to perform forwarding tasks [7]. ***Therefore, the VM network resources are actually a kind of "virtual" resources, and essentially provided by the limited IO-dedicated CPU cores in the vSwitch.***

Existing network Quality-of-Service (QoS) strategies in software vSwitch are inherited from the interface-based solutions of hardware switch, and do not consider the issues of CPU resources competition among tenants. These strategies work well on the hardware switch, because the hardware circuit processing capabilities are powerful and will not change with different traffic characteristics. But in a software vSwitch, when forwarding traffic with different characteristics, the processing capabilities of these limited IO-dedicated CPU cores are variable. For example, at the same bits-per-second (BPS) rate, compared to forwarding traffic with 1518-byte packet size, forwarding traffic with 64-byte packet size consumes 10 times more CPU cycles [8]. As a result, these QoS strategies ignoring VM competing for variable processing capacities in vSwitch will cause that the SLA of tenants' networks can hardly be guaranteed in all situations.

Some recent works [8], [9] have noticed this resources competition issue, and they added a module for CPU resources isolation before the traditional QoS module. The effects of these works were limited because they are still interface-based and the variable vSwitch forwarding capacity is still ignored.

Different from existing solutions, in this paper, we propose a new CPU-Cycle based QoS strategy (C2QoS) to completely solve this issue. As the "virtual" network resources are realized by the IO-dedicated CPU cores, the VM's network SLA can be guaranteed by directly apportioning the CPU cycles to VMs. The challenges to achieve this goal include: 1) How to establish the correspondence between bandwidth and CPU usage. 2) How to assign CPU cycles to each VM to strictly guarantee its bandwidth. 3) How to provide the hierarchical SLA latency, especially for the delay-sensitive applications. To address these challenges, this paper makes the following main contributions:

- We propose a modeling methodology to build the correspondence between forwarding capacity and CPU resources in vSwitch.
- Based on the model, we propose the C2QoS strategy, containing a CPU-Cycle based Token Bucket (C2TB) mechanism for rate limiting and a Hierarchical Batch Scheduling (HBS) mechanism for ensuring latency.
- We implement the C2QoS strategy on the DPDK accelerated open vSwitch (OVS-DPDK) [10], [11] platform. The

This work was supported in part by the Key Projects of National Key R&D program of China (Grant NO. 2019YFB1804503), and Natural Science Foundation of China (Grant No. 62002344).

978-3-903176-32-4

experiments show that compared with existing strategies, the influence of CPU resource congestion on bandwidth is eliminated and that on latency is reduced by 80%.

The rest of this paper is organized as follows. Section II introduces the background and motivation. Section III presents the model between network performance and CPU usage. Section IV shows the design of C2QoS, and its implementation on OVS-DPDK is shown in Section V. Section VI carries out performance evaluation. Section VII concludes this paper.

II. Background and Motivation

A. Network QoS in vSwitch

Network QoS strategy is a well-studied topic in hardware switch and a lot of works have been proposed. The sufficient processing capacity in hardware switch is a significant advantage when dealing with the rate limiting and scheduling issues. The main reasons are argued in [12]: the overhead of processing each packet is fixed; the token buckets and queues are implemented by hardware and they can complete the corresponding functions without loss of performance; high-precision clock and hardware feedback support [13], [14].

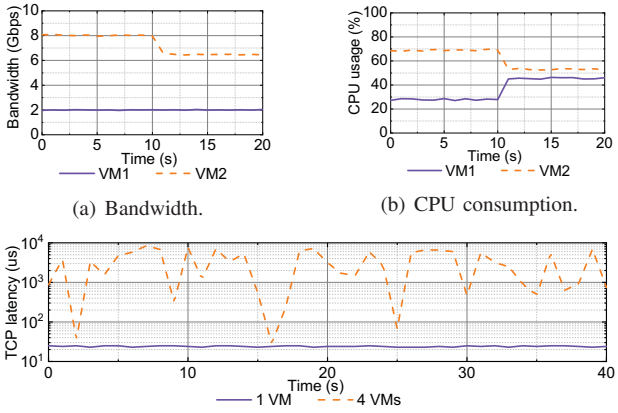
Unfortunately, none of the above advantages exists in software vSwitch. The CPU cores left for vSwitch are limited, and meanwhile their processing capacity is variable when forwarding traffic with different characteristics. For example, in Google's experiments, forwarding a flow with 64-byte packets at a speed of 512 Mbps will consume more CPU cycles than forwarding a flow with 1518-byte packets at a speed of 2.4 Gbps [8]. On the other hand, as a software based process, the vSwitch has particular bottleneck and resource competition points, which are completely different from hardware switch. These differences make that the QoS strategies inherited from hardware switch cannot work well in the vSwitch.

B. Bandwidth issue

The existing rate limiting in vSwitch is realized by token bucket methods which are based on the interface rate. As the competition for IO-dedicated CPU resources is ignored, one tenant may "legally" squeeze the CPU resources and harm the bandwidth of others. We use the three-color-marker (TCM) algorithm [15]–[17] in OVS-DPDK to demonstrate this issue.

It should be noted that all the experiments in this paper use the same configurations: Intel Xeon CPU E5-4603 v2 2.20GHz (32 logical cores on 4 NUMA nodes), 64GB DDR3 memory at 1333MHz, one Intel 82599ES 10-Gigabit Dual Port Network Interface Controllers (NICs) and Ubuntu 16.04.1 (kernel 4.8.0) as operation system. The cloud platform is built on QEMU 2.10, DPDK 17.11.2 and OVS 2.9.2. Each VM is assigned with 2GB memory and 1 logical core.

In the Fig. 1(a)-(b), the VM1 and VM2 share one CPU core in OVS-DPDK for forwarding, and their bandwidths are limited to 2 Gbps and 8 Gbps respectively. Within the first 10 seconds, their bandwidths are well guaranteed. Starting from the 10th second, VM1 sends small packets (reduces the packet size to 64-byte). In order to achieve the same throughput (2 Gbps) as before, VM1's forwarding tasks in vSwitch consume nearly 20% more CPU cycles as shown in Fig. 1(b). That leads to up to 20% bandwidth drop to VM2 due to reduced available IO-dedicated CPU resources.



(a) Bandwidth. (b) CPU consumption.

(c) Average TCP latency.

Fig. 1. Bandwidth and latency issues in the existing QoS strategies. (a)-(b) use pkt-gen from netmap [18] inside VMs as traffic generator, and (c) uses qperf [19] to measure TCP latency.

C. Latency issue

In addition to the bandwidth, the VM latency also cannot be guaranteed in vSwitch. The existing traffic scheduling mechanisms in vSwitch only work at the *egress* packet processing stage [20]–[22]. The main task in this stage is to send packets out on interfaces. These mechanisms guarantee the latency in the *egress* stage but ignore the latency in the other stages, such as the *ingress* (copying packets from interfaces) and *classification* stages. For the hardware switch, these mechanisms work well because the powerful processing capabilities make the resource contentions mainly occur in the *egress* stage, especially when traffic gathering happens on a particular port. But in vSwitch, during the *ingress* stage, concurrent VMs compete for CPU resources to execute the expensive packet copying. Due to the absence of scheduling in the *ingress* stage, VMs indiscriminately queue up for packet copying tasks to be completed, which causes mutual influence and high latency.

We also use experiments to demonstrate this issue. Fig. 1(c) shows the comparison of average TCP latency when running 1 VM and 4 VMs on one host. In both cases, one CPU core is used to forward traffic in the vSwitch. It can be seen that with 4 VMs, each tenant suffers hundreds of times higher latency indiscriminately due to waiting for the CPU core to sequentially process other VMs' *ingress* operations.

D. Motivation

The reason why VM's bandwidth and latency cannot be guaranteed is that the existing QoS strategies ignore the IO-dedicated CPU resources competition inside the vSwitch. The lack of management and apportionment of CPU resources brings a series of flaws including bandwidth isolation and high latency, which may be exploited by greedy tenants and

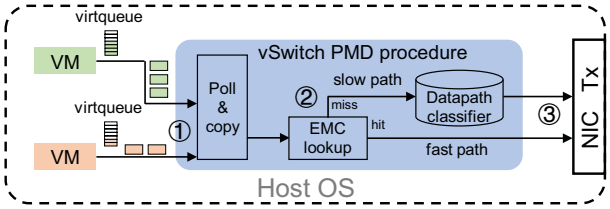


Fig. 2. Datapath in OVS. The process of sending packets from VM to NIC can be divided into three stages: ①*ingress*, ②*classification* and ③*egress*.

attackers. Some previous works have mentioned this issue, and some solutions have been proposed, e.g., Addanki et al. [9] considered separately apportioning IO-dedicated CPU resources and bandwidth on the software router, and Kumar et al. [8] proposed a method by using a CPU-based weighted fair queue to isolate CPU competition among VMs. But all of these works have limited effects because they only add a CPU isolation module before or after the existing interface-based QoS mechanisms, but fail to consider the variable vSwitch forwarding capacity and the different resource competition points in the software vSwitch process.

Essentially, the network forwarding capacity of vSwitches is not a kind of hardware resources, but a kind of "virtual" resources that are provided by IO-dedicated CPU resources in the vSwitch. Starting from this point, the motivation of this paper is to adopt the CPU resources apportionment, that reflects the network forwarding capacity more directly, into the VM network QoS solution. In order to do that, we first propose a modeling methodology to build the relationship between CPU resources and network forwarding capacity in vSwitch. Based on the vSwitch network performance model, we design and implement a new VM network QoS strategy.

III. Bandwidth-CPU Model

To guide the design of QoS strategy, we first need to model the correspondence between forwarding capacity and CPU usage in vSwitch. Our modeling is based on OVS-DPDK platform, which is the state-of-the-art implementation and has been widely adopted by the industry.

A. OVS Packet forwarding procedure

In the OVS-DPDK, several Polling Mode Driver (PMD) threads are launched and bound to IO-dedicated CPU cores. As shown in Fig. 2, the PMD procedure in the vSwitch consists of three stages delivering packets from the VM to the external network. The first stage is *ingress*, the PMD thread copies a batch of packets from the VM memory to the vSwitch buffer. Next, in *classification* stage, the PMD thread looks up their destination port based on the five-tuple. If the five-tuple is found in the Exact Match Cache (EMC), it goes to the next stage. But if it is missed, the PMD thread will use more CPU cycles to search in the more comprehensive classifiers and then go to the next stage. Finally, it is in the *egress* stage that the PMD thread writes the packet descriptors to the NIC queue, and then the NIC can send packets out. According to these three stages, we divide the CPU cycles consumed by the VM forwarding tasks into three parts:

$$C = C_{ingress} + C_{classification} + C_{egress} \quad (1)$$

Then we experiment to study how these three parts are affected by factors including traffic characteristics and other deployment issues.

B. Impact of network traffic characteristics

The network traffic characteristics that can be changed by the tenant behavior inside VM include: sending rate (packets-per-second, i.e. PPS), packet size and the number of flows. We launch one VM on the OVS-DPDK platform and assign one CPU core as the IO-dedicated CPU resources. During each experiment, we vary one characteristic and record the results while keep the other two with a certain value.

Sending rate (PPS). In Fig. 3(a), we find the CPU cycles consumed in the three stages are proportional to the PPS (1500 packet size and single flow during the experiment).

Packet size. As shown in Fig. 3(b), when sending rate (PPS) remains constant, increasing the packet size will only increase $C_{ingress}$ and have nothing to do with $C_{classification}$ and C_{egress}. The increase of $C_{ingress}$ is due to the fact that only the stage *ingress* contains packet copying, so the larger packet requires more time to copy (10^5 PPS and single flow during the experiment).

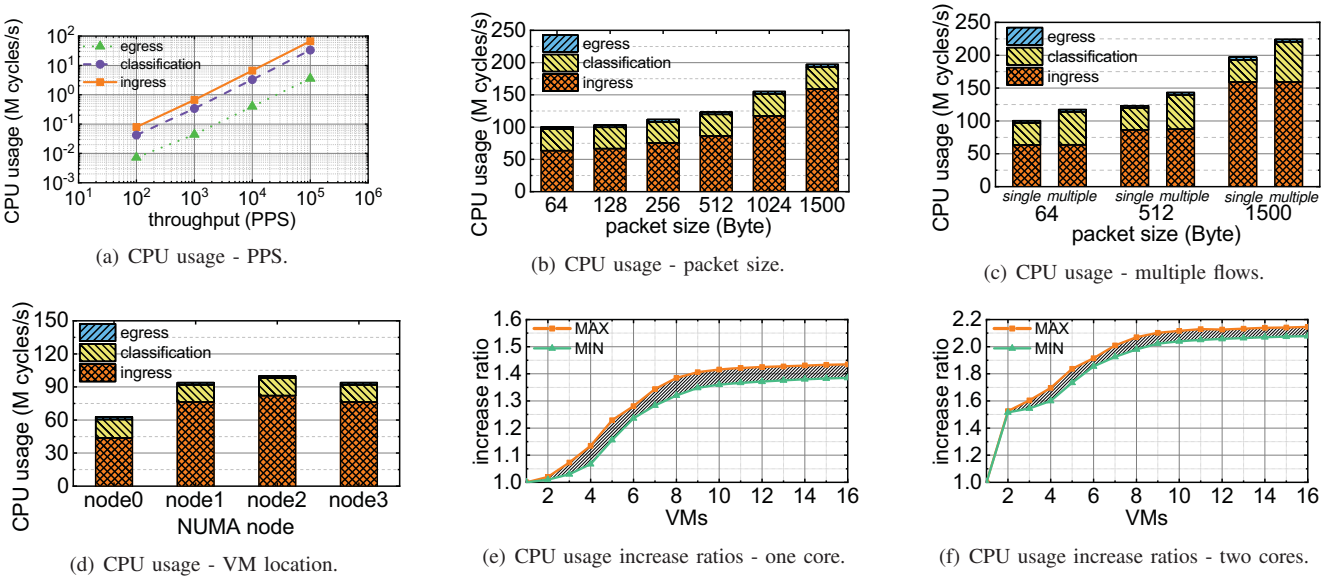
Number of flows. In Fig. 3(c), comparing with only sending one flow, sending concurrent flows will cause the packet classification to frequently miss in EMC lookup, and will enter the longer search path. Thus $C_{classification}$ is increased (1500 packet size and 10^5 PPS during the experiment).

The existing QoS strategies adopted by CSPs only consider one of the above three characteristics (i.e., PPS), but the other two characteristics can easily undermine the SLA performance via affecting CPU consumption (see Figs. 3(b)-(c)). From the experiments, a certain bandwidth-CPU relationship for single VM in the vSwitch can be established if the traffic characteristics are included in the SLA. So in this way, we can allocate particular CPU resources for each VM according to the bandwidth-CPU relationship, and further isolate the IO-dedicated CPU consumption among VMs, which will resolve the SLA issue. For example, as the iMIX traffic represents the average packet size and number of flows, CSPs can calculate the CPU resources C_{single}, required for purchased bandwidth of single VM under iMIX traffic, and allocate to the VM.

C. Impact of deployment issues

In multi-tenant scenario, we still need to consider the influence of deployment issues, which include: VM memory location, the number of VMs on the same server and the number of CPU cores used for forwarding.

VM memory location. In Fig. 3(d), when forwarding at the same rate, the VMs on NUMA [23], [24] node 1, 2 and 3 need 40% more CPU cycles than the VM on node 0 to complete the forwarding task. That is mainly due to the increase in $C_{ingress}$ by memory access across nodes (the IO-dedicated CPU core located on NUMA node 0).



(a) CPU usage - PPS.

(b) CPU usage - packet size.

(c) CPU usage - multiple flows.

(d) CPU usage - VM location.

(e) CPU usage increase ratios - one core.

(f) CPU usage increase ratios - two cores.

Fig. 3. The relationship between CPU consumption and bandwidth. (a)-(c) show the impact of traffic characteristics on CPU consumption, and (d)-(f) depict the impact of different deployment situations. The "M cycles/s" in Figs means million cycles/s. Pkt-gen from netmap is used as traffic generator in all cases.

Number of VMs. As shown in Fig. 3(e), with the number of VMs grows, the competition on memory bus and cache will increase the CPU consumption of all VMs' forwarding tasks. The maximum and minimum curves show the CPU consumption increase ratio when increasing the number of VMs on the same NUMA node and on different nodes. It is obvious that the former situation will lead to higher competition.

Number of CPU cores. Comparing Fig. 3(e) and Fig. 3(f), it can be found that using two logical cores for forwarding will consume about 1.47 times more CPU cycles than one logical core for forwarding in any cases. It is mainly due to competition for locks in the code, e.g. the synchronization among multiple PMD threads.

As these deployment factors are mainly independent with each other, the influence can be expressed as $\prod R_i * C_{single}$, where R_i represents the growth rate of CPU consumption under the influence of each factor. For example, if 4 VMs are deployed in NUMA node 1 and 2 CPU cores are assigned for forwarding. To calculate the CPU cycles required for each VM in practical, the coefficients R_i to be multiplied under the above three deployment configurations are 1.5, 1.14 and 1.47, respectively, according to Figs.3(d)-(f). Only when all these factors are considered, the CPU cycles allocated to each VM can really ensure its purchased bandwidth.

IV. C2QoS DESIGN

Based on the bandwidth-CPU model developed in Section III, we are able to design C2QoS strategy. For bandwidth guarantee, we propose C2TB mechanism using CPU cycles to achieve isolation enhanced rate limiting. To ensure hierarchical latency for tenants, we propose HBS mechanism to schedule all VMs' forwarding tasks on the IO-dedicated CPU cores. In this section, we will illustrate in detail the designs.

A. CPU-cycle based token bucket mechanism

To guarantee VM bandwidth through the CPU resources apportionment, C2TB needs two steps: allocating particular VM the necessary IO-dedicated CPU resources; using the allocated CPU resources to strictly limit the forwarding rate.

Firstly, we construct a new kind of token bucket for each VM. Different from the traditional token buckets that use the bits or number of packets as tokens, the tokens in C2TB are the remaining available IO-dedicated CPU cycles of each VM. The token generation rate of each VM is the IO-dedicated CPU cycles/s we allocated to it. Using the modeling methodology in Section III, we can set the token generation rate to a fit value that can strictly ensure tenant's purchased bandwidth. An example of this kind of bandwidth allocation under C2TB is shown in Fig. 4. Assuming the CPU resources required to achieve 1 Gbps and 4 Gbps bandwidth under iMIX traffic characteristics are 0.2G cycles/s and 0.8G cycles/s according to the measurement-driven model. So the token generation rates of the 4 VMs are set to 0.8G cycles/s, 0.8G cycles/s, 0.2G cycles/s and 0.2G cycles/s, respectively.

After the token generation rates have been configured, the next step is to use the available CPU cycles of each VM to limit its forwarding rate. The traditional token bucket algorithms drop the packets exceeding available tokens during batch processing. But in C2TB, since it is unknown how many CPU cycles will be consumed, it is impossible to decide how many packets should be dropped in one batch. For efficiency, we adopt the following policy: we allow the number of tokens to be negative, and whether the tokens are greater than 0 determines whether this batch I/O processing task can be executed. For each VM, only if its tokens are greater than 0, it can be forwarded a batch of packets and the CPU cycles consumed is subtracted in its token bucket after the batch

C2TB	VM1	VM2	VM3	VM4
weight	4	4	1	1
purchased bandwidth	4 Gbps	4 Gbps	1 Gbps	1 Gbps
generation rate (cycles/s)	0.8G	0.8G	0.2G	0.2G

Fig. 4. Token assignment method in C2TB. Allocating necessary CPU cycles/s to each VM that can achieve the purchased bandwidth.

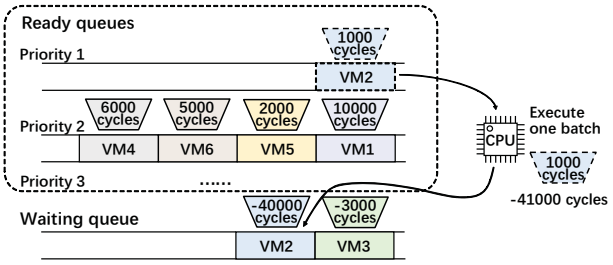


Fig. 5. HBS design. The IO-dedicated CPU cores always pick up the VM with highest priority in the virtual "ready queues" to process batch I/O task. So the traditional undifferentiated execution in polling running mode is replaced by the "smart" hierarchical batch I/O scheduling.

processing completed. As the CPU cycles used for each VM's packet forwarding tasks are fairly assigned in C2TB, the VM bandwidth can be guaranteed with good isolation.

B. Hierarchical batch scheduling mechanism

As the existing scheduling mechanisms only work at the *egress* stage and cannot avoid the high latency of CPU resources contention in other stages, we turn to think about scheduling the entire batch I/O processing procedure (including *ingress*, *classification* and *egress*) for VMs. In the field of CPU task scheduling, we find the task scheduling model in vSwitch is much closer to the works in [25], [26], which schedule tasks on CPU cores to ensure that light load tasks will not be blocked too long by heavy load tasks. That inspires us to propose HBS to schedule batch I/O forwarding tasks on the IO-dedicated CPU cores for hierarchical latency guarantee.

The HBS design is shown in Fig. 5. All VMs are placed in virtual queues and there are two kinds of queues: waiting queue and ready queue. As the C2TB allows CPU to skip VMs with negative tokens, we put these VMs that should be skipped into the waiting queue. The VMs with tokens greater than 0 are queued in the corresponding ready queues according to their priorities. The CPU cores will only poll and dequeue the VMs in the ready queues and do batch I/O forwarding, according to the priority. So in the ready queues, the VMs in high-priority queues have absolute execution privileges than the VMs in low-priority queues. For example, in Fig. 5, although the number of tokens in VM2 is the smallest among the VMs in the ready queues, VM2 will be dequeued and forwarded one batch of packets first because it has the highest priority. After the batch processing, VM2 will be placed into the waiting queue because it has consumed 41000 tokens and its available tokens are negative. To ensure fairness that VMs in the same queue have similar latency, each virtual queue in the HBS follows first-in-first-out (FIFO) policy.

With hierarchical execution privileges, the worst latency of VMs in each queue can be guaranteed and calculated. We assume a case that the numbers of VMs in all 8 ready queues are $\{N_1, N_2, N_3, \cdots, N_8\}$, respectively. The time used for one batch processing is c. So the worst-case latency of VMs in these queues are $\{N_1 * c, (N_1 * k_1 + N_2) * c, (N_1 * k_1 + N_2 * k_2 + N_3) * c, \cdots, (N_1 * k_1 + N_2 * k_2 + \cdots + N_7 * k_7 + N_8) * c\}(k_i \geq 1)$. Compared with original sequential execution that each VM equally suffers the worst $\sum N_i * c$ latency, HBS can provide hierarchical worst latency guarantee for VMs with different requirements. Meanwhile, as C2TB guarantees available CPU cycles/s for VMs, HBS will not cause VM starvation. That helps CSPs formulate more flexible SLA policies based on the tenants' latency sensitivities.

V. Implementation

According to the design in Section IV, we implement the C2QoS strategy in the OVS-DPDK platform. We modified the PMD thread's main loop function and used C2TB to replace original port ingress policy, which is implemented by TCM in DPDK [15]. For efficiency, we use the *rdtsc* instruction [27] to calculate CPU consumption in the packet forwarding procedure. In each PMD thread, the sequentially execution running mode is replaced by HBS that finds VM with highest priority in the ready queues to execute batch I/O processing. The task of maintaining queues in HBS is undertaken by another manager thread, which is woken up every 50us to update the tokens of each VM. All of these modifications require no more than 300 lines of code and are easy to realize. The effectiveness and overhead of C2QoS will be evaluated in detail in Section VI.

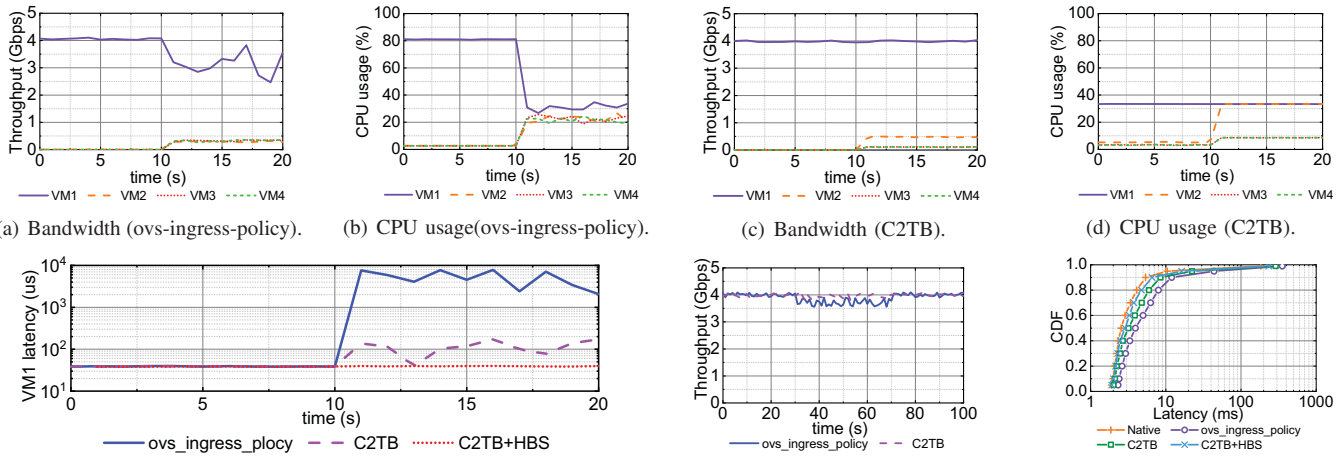
VI. Evaluation

In this section, we evaluate the VM network QoS guarantee under C2QoS and the OVS-DPDK existing "ovs-ingress-policy" QoS strategy. The configurations here are the same as that in Section II-B.

A. TCP bandwidth and latency

In this experiment, we use iperf [28] and qperf [19] tools to evaluate VMs' TCP bandwidth and latency. We launch 4 VMs with 4 Gbps, 4 Gbps, 1 Gbps, and 1 Gbps purchased bandwidth, respectively. One dedicated CPU core is used in OVS-DPDK for forwarding. In our benchmark setting, VM1 acts as a well-behaved tenant to send 1500-byte packets all the time, while the other 3 VMs act as "noisy" neighbors and send 64-byte packets from the 10th second.

In Figs. 6(a)-(b), with the ovs-ingress-policy, VM1 works well and keeps 4 Gbps bandwidth within the first 10 seconds. But in the last 10 seconds, due to the other 3 VMs send small packets and "legally" compete for CPU resources, VM1 bandwidth is affected and reduced by up to 30%. As Fig. 6(b)



(a) Bandwidth (ovs-ingress-policy). (b) CPU usage(ovs-ingress-policy). (c) Bandwidth (C2TB). (d) CPU usage (C2TB).

(e) VM1 TCP latency. (f) Ftp bandwidth. (g) Nginx response time.

Fig. 6. Experimental results. (a)-(e) show the results of TCP bandwidth and latency. We launched 4 VMs with 4 Gbps, 4 Gbps, 1 Gbps, and 1 Gbps purchased bandwidth respectively. (f)-(g) show the results of application performance. 2 VMs with 4 Gbps are deployed as the Ftp server, and 2 VMs with 1 Gbps are deployed as the Nginx server. We perform a pressure test on Nginx VMs during the 30th-70th seconds.

shows, 4 VMs compete to get similar CPU resources and that harms high weight VM1.

In C2TB, as shown in Figs.6(c)-(d), the CPU resources are strictly allocated and isolated (VM1&2: 36%, VM3&4: 9%), so VM1 bandwidth is stable and unaffected by other VMs' behaviors in the last 10 seconds. But for the other 3 VMs, they can only achieve a very low bandwidth. This is the goal of C2TB, that only guarantees the bandwidth under specific conditions, such as iMIX traffic characteristics. So sending extreme traffic (small packets) will only reduce the three "noisy" neighbors' own bandwidth without affecting VM1.

Fig. 6(e) shows the VM1 latency tested by qperf. We can see that compared with ovs-ingress-policy, C2TB can reduce part of the additional latency of VM1 by skipping ports with negative tokens. But the latency under C2TB is still unstable. With HBS, we set VM1 to be placed in Priority 1 queue and that ensures VM1's forwarding tasks are always executed first. The results show the latency of VM1 under C2TB+HBS remains low and close to native performance.

B. Application results

To make it more practical, we consider some common applications on the public cloud. In this experiment, 2 VMs with 4 Gbps are deployed as Ftp servers and 2 VMs with 1 Gbps are deployed as Nginx servers. The Ftp servers send traffic all the time, while Nginx servers bear pressure test during 30th-70th seconds by using wrk [29]. The bandwidth of Ftp server and response latency of Nginx server are evaluated.

The Ftp bandwidth is shown in Fig. 6(f), the Nginx servers' traffic during 30th-70th seconds cause a bandwidth drop of about 11% on the Ftp servers under ovs-ingress-policy, while C2TB strictly guarantees the bandwidth of Ftp servers all the time. For the latency in Nginx pressure test, we obtain the request response time distribution in Fig. 6(g). Under ovs-ingress-policy, the response time of Nginx requests is doubled compared to native performance. When only using C2TB, 50% additional latency is reduced by skipping ports with negative tokens. But with C2TB+HBS, the additional latency is reduced by more than 80% and these Nginx servers achieve almost the native performance. Therefore, the C2QoS can ensure the network performance of both latency-sensitive and bandwidth-sensitive services while sharing the same physical resources.

C. Overhead

As we added a new module to vSwitch, the overhead needs to be measured and it mainly reflects on two aspects: the performance decrease and the additional CPU overhead. For the first concern, in the single-VM and multi-VM experiments, the OVS-DPDK using C2QoS strategy has no performance drop compared with the original version. This is because the *rdtsc* instruction we used to measure CPU cycles is very light and has very little effect on forwarding performance. For the additional CPU overhead, C2QoS only consumes 0.018% of resources on IO-dedicated CPU cores and it will not go up with the increase in the number of VMs. On the manager core, 2.08% of resources are used for tokens counting and queues managing when deploying 28 VMs. So the additional CPU overhead in C2QoS is also acceptable for cloud platforms.

VII. Conclusion

This paper focused on the VM network QoS and addressed the key issue of competition on IO-dedicated CPU resources among VMs. To solve this issue, we proposed C2QoS, consisting of C2TB and HBS mechanisms. In C2TB, according to a measurement-driven bandwidth-CPU model, we limited VM's bandwidth by directly assigning CPU cycles to a particular VM. The HBS mechanism scheduled the VMs' entire batch I/O forwarding tasks on the IO-dedicated CPU cores and that provided hierarchical latencies for VMs according to priorities. The implementation on the OVS-DPDK platform showed that C2QoS eliminated the influence of CPU resource congestion on bandwidth and reduced effects on latency by 80%.

REFERENCES

[1] "alibabacloud." https://www.alibabacloud.com/.
[2] "googlecloud." https://cloud.google.com/.
[3] "azure." https://azure.microsoft.com/.
[4] "aws." https://aws.amazon.com/.
[5] D. Firestone, "Vfp: A virtual switch platform for host sdn in the public cloud," in *Conf. on Networked Systems Design and Implementation*, 2017.
[6] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, *et al.*, "The design and implementation of open vswitch.," in *Conf. on Networked Systems Design and Implementation*, 2015.
[7] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, *et al.*, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *Conf. on Networked Systems Design and Implementation*, 2018.
[8] P. Kumar, N. Dukkipati, N. Lewis, *et al.*, "Picnic: predictable virtualized nic," in *Conf. of the ACM Special Interest Group on Data Communication*, 2019.
[9] V. Addanki, L. Linguaglossa, J. Roberts, and D. Rossi, "Controlling software router resource sharing by fair packet dropping," in *IFIP Networking Conference and Workshops*, 2018.
[10] "Data plane development kit." https://www.dpdk.org.
[11] "Open vswitch." http://www.openvswitch.org/.
[12] K. To, D. Firestone, G. Varghese, and J. Padhye, "Measurement based fair queuing for allocating bandwidth to virtual machines," in *work. on Hot topics in Middleboxes and Network Function Virtualization*, 2016.
[13] F. Checconi, L. Rizzo, and P. Valente, "Qfq: Efficient packet scheduling with tight guarantees," *IEEE/ACM Transactions on Networking*, vol. 21, no. 3, 2012.
[14] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, *et al.*, "Programmable packet scheduling at line rate," in *Conf. of the ACM Special Interest Group on Data Communication*, 2016.
[15] "Dpdk traffic metering." http://doc.dpdk.org/guides/prog_guide/traffic_metering_and_policing.html.
[16] "Rfc2697(srtcm)." https://www.rfc-editor.org/rfc/rfc2697.html.
[17] "Rfc2698(trtcm)." https://www.rfc-editor.org/rfc/rfc2698.html.
[18] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *USENIX Annual Technical Conference*, 2012.
[19] "qperf." https://linux.die.net/man/1/qperf.
[20] "qdisc." https://lwn.net/Articles/564978/.
[21] A. Saeed, N. Dukkipati, V. Valancius, *et al.*, "Carousel: Scalable traffic shaping at end hosts," in *Conf. of the ACM Special Interest Group on Data Communication*, 2017.
[22] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, *et al.*, "Eiffel: efficient and flexible software packet scheduling," in *Conf. on Networked Systems Design and Implementation*, 2019.
[23] "What is numa." https://www.kernel.org/doc/html/latest/vm/numa.html.
[24] "Numa locality." https://www.kernel.org/doc/html/latest/admin-guide/mm/numaperf.html.
[25] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained nfs for flexible per-flow customization," in *Conf. on emerging Networking EXperiments and Technologies*, 2016.
[26] J. Mace, P. Bodik, M. Musuvathi, *et al.*, "2dfq: Two-dimensional fair queuing for multi-tenant cloud services," in *Conf. of the ACM Special Interest Group on Data Communication*, 2016.
[27] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2019.
[28] "iperf." https://iperf.fr/.
[29] "wrk." https://github.com/wg/wrk.