CORAL: A Multi-Core Lock-Free Rate Limiting Framework

Zhe Fu[1, 2], Zhi Liu[1, 2], Jiaqi Gao[1, 2], Wenzhe Zhou[4], Wei Xu[4], and Jun Li[2, 3]
[1] Department of Automation, Tsinghua University, China
[2] Research Institute of Information Technology, Tsinghua University, China
[3] Tsinghua National Lab for Information Science and Technology, China
[4] Huawei Technologies
{fu-z13, zhi-liu12, gaojq12}@mails.tsinghua.edu.cn, {wenzhe.zhou, wei.xu1}@huawei.com, junl@tsinghua.edu.cn

***Abstract*—Rate limiting is a fundamental function for managing quality of network service. Unfortunately, the poor scalability of today's rate limiting approaches restricts the performance on multi-core platform. In this work, we present CORAL, a lock-free framework that effectively implements high performance rate limiting on multi-core platform. The key idea is that CORAL uses virtual class queues to isolate simultaneous access to the same queue by different CPU cores and two additional parameters to synchronize the QoS constraints among multi cores. Experimental results show that this lock-free design obtains around 50% higher limiting rate compared to existing locking method, and shows great scalability and stable performance over different number of cores and packet sizes.**

Keywords—quality of service; rate limiting; multi-core; lock-free

I. Introduction

QoS (Quality of Service) functions, which includes guarantees of latency and minimum bandwidth, rate limiting, bandwidth shaping and sharing for different flows, are playing an important role in quantitatively measuring the quality of service and providing differentiated services for network flows. As a fundamental part of QoS, rate limiting is used to control the rate of traffic at the input and output side. A router could smooth out the traffic by limiting the rates of different flows, and a firewall could perform access control by limiting the rate of specific incoming flows.

Nowadays, rate limiting is performed by either hardware or software. Modern NICs support multiple hardware transmit queues. When transmitting a packet, a request will be sent to the NIC by the OS to notify the arrival of the packet, and an arbiter is used by the NIC to compute the fetching order of packets from different ring buffers. The NIC first looks up the physical address of the packet, and then initiates a DMA (Direct Memory Access) transfer of the packet contents to its internal packet buffer. Finally, a scheduler decides whether or when the packet will be transmitted.

Hardware based rate limiting ensures low CPU overhead and high accuracy. At the same time, storing masses of per-class packet queues and ring buffers for each queue on the NIC may result in poor scalability. For example, if 4,000 queues are used in the NIC and each queue stores 15KB packet data (about 1,000 packets), it would consume about 60 MB of SRAM of the NIC, which is too expensive for commodity NIC [1]. Besides, the widespread use of containers produces larger scale of network flows. Current NIC hardware only supports 8-128 rate limiters [2], which carries significant limitations for fine-grained rate limiting and other QoS applications in these scenarios.

Most operating systems have already supported software based rate limit function. As the foundation of modern network, Linux offers a very rich set of tools for managing and manipulating the transmission of packets. For example, TC (Traffic Control) is a user-space utility program used to configure Linux kernel packet scheduler. It uses QDisc (Queuing Discipline), which can be configured with traffic classes, to enforce flexible and scalable traffic control policies. But software based rate limiting implementation usually encounters the problem of high CPU overhead due to lock contention and frequent interruption. Previous experiments [3] show that software based rate limiting implementation consumes about 5 times more kernel CPU utilization of that by hardware based methods.

With the rapid development of SDN (Software Defined Network) and NFV (Network Function Virtualization), more and more network functions are virtualized and implemented on general-purpose processor platform. While allowing flexible deployment and live migration, the poor performance of these implementations has become a bottleneck for supporting high bandwidth network traffic processing. Recently, the development of data plane technology such as DPDK (Data Plane Development Kit) [4] and fd.io [5] bring new possibilities into the implementation of high performance QoS functions. However, it is still a challenge to effectively map queues on multi cores while reducing overhead as much as possible.

In this paper, we present CORAL, a scalable multi-Core lOck-free RAte Limiting framework. Specifically, we introduce virtual QoS class queue to isolate simultaneous access to the same queue by different CPU cores, and use two additional parameters demand rate and supply rate attached to each virtual class queue to synchronize the QoS constraints among multi cores. Experimental results show that compared to existing

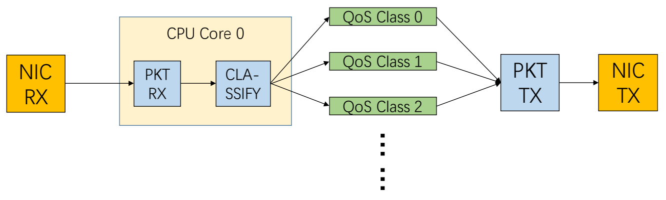

Figure. 1. Single-core rate limiting

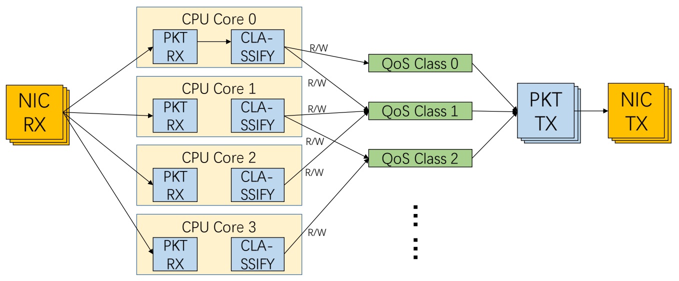

Figure. 2. Multi-core rate limiting with lock

multi-core rate limiting implementations, around 50% higher limiting rate can be achieved with CORAL. In addition, CORAL shows great scalability as the number of CPU cores increases.

This paper is organized as follows. Section II gives the overview of related work. In Section III, we analyze the shortcomings of state-of-art multi-core rate limiting implementations, and propose the design of CORAL. Section IV discusses the experimental results of CORAL, and compares the results with the existing method. Finally, Section VI concludes the paper.

II. Related Work

In [6, 7], generic token bucket management methods for QoS requirements are introduced. The tokens normally represent a single packet or a unit of predetermined bytes, and are added into a bucket at a fixed rate. The bucket is checked to see whether it contains sufficient tokens when a packet arrives. If the bucket contains enough tokens that packet needs, the packet is passed and several tokens (usually equivalent to the length of the packet) are removed. Otherwise, no tokens are removed from the bucket, but the packet is dropped or marked as non-conformant for further processing. Leaky bucket is another algorithm used to limit the rate of network traffic. Unlike token bucket algorithms, leaky rate algorithm deliver packets at a constant rate, and it lack the power of handling bursty traffic. Hierarchical token bucket (HTB) [8] algorithm allows for complex and granular control over traffic. It classifies traffic in a multilevel hierarchy, based upon a variety of parameters such as IP addresses, priority or interface. Nevertheless, these generic algorithms are not optimized for multi-core scenarios, thus don't scale well when the number of CPU cores is increasing.

Recently, quite a number of researches propose more advanced rate limiting solutions for specific situations. Gatekeeper [9], and EyeQ [10] both limit the rate between each VM (Virtual Machine) pair to guarantee bandwidth for multi-tenant datacenter network. QCN [11] and DCTCP [12] use rate limiters to reduce congestions in data centers where bursty correlated traffic coupled with small buffers often result in poor application performance [13]. All these methods take advantage of rate limiting, but none of them focus on solving the performance bottleneck of rate limiter itself. In addition, with growing number of VMs and flows in data centers and virtualized network environment, the number of limiters is expected to increase, bringing more challenges to high performance rate limiting.

In [14] a hardware priority queue architecture for link scheduling in high-speed switches is presented, guaranteeing QoS requirements in high speed networks. ServerSwitch [15] proposes a programmable NIC for configurable congestion control management. SENIC [1] offloads rate limiting to NetFPGA and leave the rest task to software, aiming at reducing CPU load while supporting thousands of rate limiters. However, all of these work rely heavily on specific hardware, and will lose scalability on general-purpose processor platform.

III. Design

Receive-Side Scaling (RSS) is a network feature of NIC which enables efficient distribution of input packets. With the support of RSS, network receive processes are distributed across several hardware-based receive queues, which allows multiple processing cores process network traffic simultaneously and relieve bottlenecks in receive interrupt processing caused by overloading a single core. RSS is the fundamental technology of high performance packets processing on multi-core platform. In this section, we will first analyze the disadvantages of current multi-core rate limiting methods, and then present a new scalable lock-free rate limiting framework to overcome these shortcomings.

A. *State-of-art Methods*

Figure 1 shows the situation where a single core takes responsibility of packets receiving and rate limiting. After received from the NIC, the packets are classified by the CPU core and sent to several queues of various QoS classes for fine-grained traffic control. When it extends to multi-core scenario, every CPU core receives packets from the NIC and uses its own classifier to send packets to different QoS classes. However, since traffic distribution by RSS is determined by the NIC driver, rather than the classifier of each CPU core, packets to different CPU cores may be classified as the same QoS class and sent to the same queue again. This will lead to simultaneous reading and writing operation to one queue from different CPU cores as shown in Fig 2. Rate limiting would fail or become abnormal without additional synchronization protections. As a result, locking operation to the queue is necessary in order to accurately limit the rate of each class. For example, before CPU core 0 wants to perform a write operation to QoS class queue 0 in Fig. 2, queue 0 must be in unlocked status, and must keep locked until CPU core 0 finished write operation. During the locking time, operations to this queue from other CPU cores have to wait until the lock becomes available.

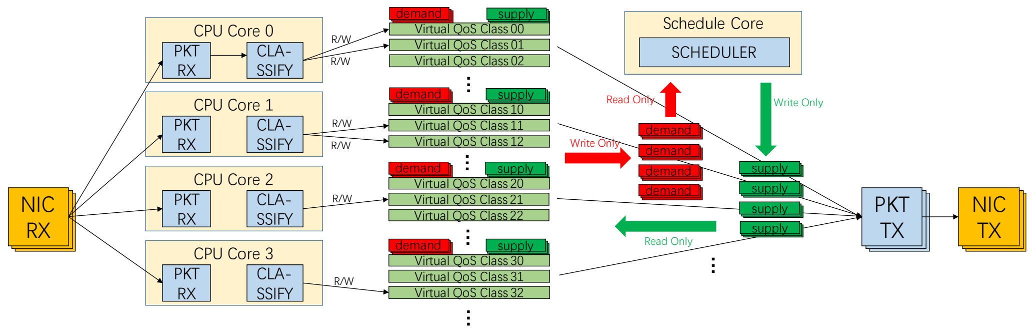

Figure. 3. CORAL design

Obviously, lock contention and frequent interruption often lead to high CPU load, which degrades performance of packet processing on multi-core platform. In next subsection, a scalable lock-free rate limiting framework will be presented.

B. *CORAL Design*

The root cause for frequent lock contention and interruption is that multi cores share the queues of the same QoS class. Due to the fact that the NIC RSS driver distributes packets for multi-core processing while the user-defined classifier classifies packets to different QoS classes, it cannot be expected that packets are sent to the appropriate QoS class queues directly from the NIC.

In the design of CORAL, we introduce the concept of *virtual QoS class queue*. Virtual class queue itself is not a complete Qdisc, but acts as a portion of a complete Qdisc. Figure 3 depicts the relationship between virtual class queues and traditional class queues for classful rate limiting. Virtual QoS class queue 00, 10, 20 and 30 are four sub class queues that together make up the class queue 0. On the other hand, virtual class queue 00, 01 and 02 are the sub class queues which are mapped on core 0 and can only be accessed by CPU core 0.

Mathematically, the virtual QoS class queue is defined as follows:

Definition 1: For a rate limiting implementation on m cores platform with n QoS classes, there are $m * n$ virtual QoS class queues, where $vcq(i,j)$ denotes a virtual class queue mapped on CPU core i and used for QoS class j, $0 \le i < m$ and $0 \le j < n$.

The introduction of virtual class queue eliminates the lock contention caused by multi-core's access to shared resources. In order to ensure the classful rate limiting effects, another two parameters are attached to each virtual class queue: *demand rate* and *supply rate*.

Demand rate is a parameter that calculates the demand rate of input packets in this virtual class queue. At a period T of token update, w packets arrive in virtual class queue $vcq(i,j)$, so the demand rate $dr(i,j)$ for this virtual class queue is:

$$dr(i,j) = \frac{w}{T} \tag{1}$$

Supply rate is one or a set of parameters which determines the actual packets transmission rate of this virtual class queue. Under given supply rate parameters, the virtual class queue is expected to dequeue at a specific average rate.

The design of virtual class queues together with demand and supply rate parameters enables a lock-free rate limiting implementation for classful QoS on multi-core platforms. As shown in Fig. 3, each CPU core only needs write permission to the demand rate parameters and read permission to the supply rate parameters of its own virtual class queues. A scheduler (which may be assigned in an independent CPU core) takes charge of all demand rate collection and supply rate update. The scheduler only needs one permission to the parameter as well: read permission to all demand rate and write permission to all supply rate parameters. More specifically,

RULE 1: For virtual class queue $vcq(i,j)$, demand rate $dr(i,j)$ can only be written by CPU core i and read by the scheduler, and supply rate $sr(i,j)$ can only be written by the scheduler and read by CPU core i.

The scheduler periodically fetches the demand rate of each virtual class queue and recalculates the supply rate parameters correspondingly. Various scheduling algorithms can be applied for different purposes. For instance, for a rate limiting scheme on m cores platform and n QoS classes, there are $m * n$ virtual QoS class queues. If the demand rate of each virtual class queue is $dr(i,j)$, $0 \le i < m$, $0 \le j < n$, a practical algorithm to obtain supply rate parameter $sr(i,j)$ is:

$$sr(i,j) = \frac{dr(i,j)}{\sum_{j=0}^{n} dr(i,j)} * cr(j) \tag{2}$$

Where $cr(j)$ denotes the committed rate for QoS class j.

Algorithm 1 shows the simplified pseudo-code of packets processing procedure for each virtual class queue. Every period T, $dr(i,j)$ is calculated by equation (1). The updating of $sr(i,j)$ is performed by the scheduler, as described in Algorithm 2.

Algorithm 1 – Packets Processing

```
Input:   $sr(i,j)$, packet pkt with length len, time t
Output:  packet processing action

$t_{diff} \leftarrow t - t_{last}$
$num_{tokens} \leftarrow num_{tokens} + t_{diff} * sr(i,j)$
if $num_{tokens} < len$ then
    droppacket(pkt)
else
    sendpacket(pkt)
    $num_{tokens} \leftarrow num_{tokens} - len$
endif
$t_{last} \leftarrow t$
```

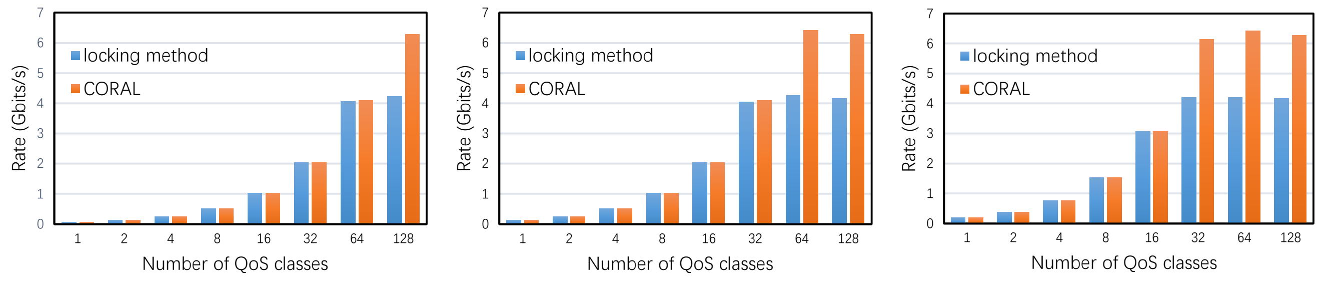

Figure. 4. Evaluation results for different committed rate (a) $cr = 64\text{MBits/s}$ (b) $cr = 128\text{MBits/s}$ (c) $cr = 192\text{MBits/s}$

Algorithm 2 – Scheduling parameters updating

```
Input:   number of CPU cores $m$, number of QoS classes
         $n,\ dr(i,j),\ cr(j),\ 0 \le i < m,\ 0 \le j < n$
Output:  $sr(i,j)$

foreach $j \in [0,\dots,n)$ do
    $sum \leftarrow 0$
    for $i = 1 \rightarrow m$ do
        $sum \leftarrow sum + dr(i,j)$
    end
    foreach $i \in [0,\dots,m)$ do
        $sr(i,j) \leftarrow dr(i,j) * cr(j) / sum$
    end
end
```

It must be noted that diverse fairness algorithms can be employed by the scheduler, including FIFO (First In, First Out), WFQ (Weighted Fair Queuing), max-min, etc.

IV. Evaluation

In this section, we conducted a series of experiments to evaluate the performance and scalability of CORAL. Experiments are conducted on two HP Z228 SFF workstations with Intel(R) Core(TM) i7-4790 CPU platform (8 logic cores), Intel(R) 82599ES 10 Gigabit Ethernet Controller, and DPDK 16.04 installed for low-level packet processing. Pktgen [16] is used to send traffic at wire rate and perform statistical analysis, and schedule algorithm described in Section III.B is employed to update the demand and supply rate of each virtual class queue.

A. Committed Rate

Committed rate cr is the rate at which the tokens are added to the buckets. It is usually measured in bytes of IP packets per second. In our implementation, each QoS class could have its own committed rate. For QoS class j, $cr(j)$ denotes its committed rate. To make the evaluation results more indicative, all QoS classed are assigned the same committed rate as shown in Fig.4. Packets of size 64 bytes are generated from pktgen with random source and destination IP, ensuring that all QoS classes in the rate limiter are evenly covered. The number of QoS classes ranges from 1 to 128, and four CPU cores are used for rate limiting in each experiment.

In Fig. 4(a), both of the original locking method and our CORAL framework obtain a linear increasing output rate when the number of QoS classes is below 128. It is also proved that expected limiting rates are achieved by both methods by calculating using equation (3):

$$output\ rate = \sum_{j=0}^{n} cr(j) = n * cr \tag{3}$$

Where there are m cores platform with n QoS classes, $0 \le i < m$ and $0 \le j < n$.

When there are 128 QoS classes, the expected output rate is $128 * 64\text{Mbits/s} = 8.192\text{Gbits/s}$. Though both methods fail to achieve that value, carol still gets nearly 50% more throughput (6.27Gbits/s compared to 4.181Gbits/s). Figure 4(b) and 4(c) shows similar experimental results.

B. Maximum Supported Limiting Rate

In this subsection, several experiments are conducted to evaluate the maximum supported limiting rate of the locking method and our lock-free framework. Figure 5 depicts the results of the rate limiter with four CPU cores and 16 QoS classes in total. As the number of QoS classes grows, the output rate of the locking method decreases due to frequent lock contention. Since the source and destination IP of input packets are randomized, the more QoS classes the rate limiter needs to deal with, the more likely different cores access one QoS class queue at the same time.

In contrast, the performance of CORAL remains stable due to the introduction of virtual class queue isolating the simultaneously access to the same queue by different CPU cores. Experimental results indicate that under the circumstance of four CPU cores with 16 QoS classes, CORAL achieves the maximum limiting rate of 6.373Gbits/s, 48% more than the locking rate limiting method at the same condition (4.297Gbits/s).

C. Packet Size

TABLE I. Experiment Results of Different Packet Size

Packet Size	**64 Byte**	**320 Byte**	**576 Byte**	**832 Byte**
Rate	512 Mbits/s	512 Mbits/s	512 Mbits/s	512 Mbits/s
Packet Size	**1088 Byte**	**1344 Byte**	**1500 Byte**	**Mixed**
Rate	512 Mbits/s	512 Mbits/s	511 Mbits/s	512 Mbits/s

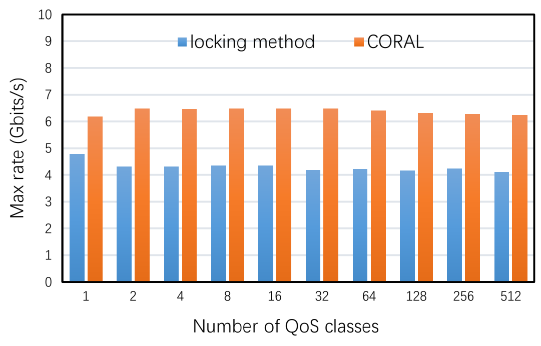

Figure. 5. Maximum supported limiting rate for different QoS classes number

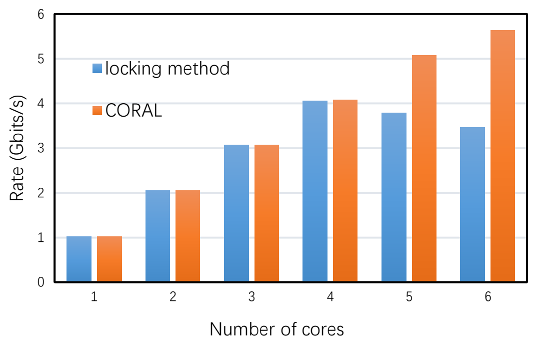

Figure. 6. Experimental results of different number of CPU cores used for rate limiting

This subsection shows the experimental results of different input packet size. Four CPU cores are used to limit the rate, and 8 QoS classes are set with 64Mbits/s committed rate for each. Pktgen is configured to generate packets varying from small packet size such as 64bytes, 320bytes to large packets of 1500bytes. Moreover, a mixed packet set filled with these small and large packets is generated as well. As Table I. illustrates, the output rate of CORAL, which is exactly the sum of each class's expected commit rate, stays almost constant regardless of the packet size.

D. Number of CPU Cores

In order to evaluate the scalability of CORAL, several experiments are conducted using different number of CPU cores, ranging from 1 to 6. We assign 16 QoS classes to each CPU core and 64Mbits/s committed rate for each QoS class. Random packets with size of 64 bytes are sent from the pktgen to the limiter. Figure 6 states that as the number of cores increases, the output rate of CORAL keeps improving. On the contrary, the comparative method achieves the highest output rate when 4 CPU cores are used. In the case of 6 cores, CORAL achieves 5.634Gbits/s output rate while the locking method only reaches at 3.463Gbits/s output rate: CORAL obtains more than 60% performance improvement. These evaluations prove that CORAL scales well with more CPU cores in use.

V. Conclusion and Future Work

In this paper, we present CORAL, a framework that effectively implements high performance rate limiting on multi-core platform. Virtual class queues are introduced to isolate simultaneous access to the same queue by different CPU cores, and two additional parameters demand rate and supply rate are attached to each virtual class queue to synchronize the QoS constraints among multi cores. In comparison with the existing multi-core rate limiting approach, CORAL nearly achieves 50% to 60% higher maximum supported limiting rate. The experimental results also show that CORAL has great scalability over different number of CPU cores as well as stable performance among packets of various sizes. Our future work will focus on further optimizing the performance of CORAL to support bandwidth of 40Gbits/s and beyond.

References

[1] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. "SENIC: scalable NIC for end-host rate limiting." In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pp. 475-488. 2014.

[2] Intel 82599 10GbE Controller. http://www.intel.com/content/dam/doc/datasheet/82599-10-gbecontroller-datasheet.pdf.

[3] S. Radhakrishnan, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. "NicPic: Scalable and Accurate End-Host Rate Limiting." In 5th USENIX Workshop on Hot Topics in Cloud Computing. 2013.

[4] Data Plane Development Kit, http://dpdk.org.

[5] The Fast Data Project (FD.io), https://fd.io.

[6] M. A. Franklin., P. Crowley, H. Hadimioglu, and P. Z. Onufryk, Network Processor Design: issues and practices. Vol. 2. Morgan Kaufmann, 2003.

[7] G. Varghese. Network algorithmics. Chapman & Hall/CRC, 2010.

[8] Hierarchical token bucket, http://luxik.cdi.cz/~devik/qos/htb.

[9] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. "Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks." In WIOV. 2011.

[10] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. "EyeQ: practical network performance isolation at the edge." In 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pp. 297-311. 2013.

[11] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman. "Data center transport mechanisms: Congestion control theory and IEEE standardization." In 46th Annual Allerton Conference on Communication, Control, and Computing, pp. 1270-1277. IEEE, 2008.

[12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data center tcp (dctcp)." In ACM SIGCOMM computer communication review, vol. 40, no. 4, pp. 63-74. ACM, 2010.

[13] Z. Liu, X. Wang, W. Pan, B. Yang, X. Hu, and J. Li. "Towards efficient load distribution in big data cloud." In 2015 International Conference on Computing, Networking and Communications (ICNC), pp. 117-122. IEEE, 2015.

[14] S. Moon, J. Rexford, and K. G. Shin. "Scalable hardware priority queue architectures for high-speed packet switches." IEEE Transactions on Computers 49, no. 11 (2000): 1215-1227.

[15] G. Lu, C. Guo, Y. Li, X. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. "ServerSwitch: A Programmable and High Performance Platform for Data Center Networks." In 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11), pp. 15-28. 2011.

[16] Pktgen, https://github.com/pktgen/Pktgen-DPDK.