Packet Slicing for Highly Concurrent TCPs in Data Center Networks with COTS Switches

Jiawei Huang[§,‡], Yi Huang[§], Jianxin Wang[§], Tian He[‡]
[§]School of Information Science and Engineering, Central South University, ChangSha, China 410083
[‡]Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA 55455
Email: {jiaweihuang, huangyi90, jxwang}@csu.edu.cn, tianhe@umn.edu

***Abstract*—Modern data center nowadays leverages highly concurrent TCP connections between thousands of computer servers to achieve high system performance and service reliability. However, recent works have shown that, in the many-to-one and barrier-synchronized communication pattern, a large number of concurrent TCP connections suffer the TCP Incast problem due to packet drops in shallow-buffered Ethernet switches. This problem unavoidably leads to severe under-utilization of link capacity. In this work, we first reveal theoretically and empirically that controlling the IP packet size reduces the Incast probability much more effectively than controlling the congestion windows in the presence of severe congestion. We further present the design and implementation of Packet Slicing, a general supporting scheme that adjusts the packet size through a standard ICMP signaling method. Our method can be deployed on commodity switches with small firmware updates, while making no modification on end hosts. Another highlight of our work is Packet Slicing's broad applicability and effectiveness. We integrate Packet Slicing transparently (i.e., without modification) with three state-of-the-art TCP protocols designed for data centers on NS2 simulation and a physical testbed, respectively. The experimental results show that Packet Slicing remarkably improves network goodput across different TCP protocols by average 26x under severe congestion, while introducing little I/O performance impact on both switches and end hosts.**

I. Introduction

Modern data centers usually house a large number of computer servers networked together to handle diverse online business such as web search, large-scale cloud storage and social networking applications. The consistent trend in data center designs is always to build highly available, high performance computing and storage infrastructure using low-cost commodity components. For example, an increasing number of data centers are being built with rack-mounted servers interconnected by Top of Rack (ToR) Commercial Off-the-Shelf (COTS) Ethernet switches [1] [2]. To save cost, these COTS switches normally have small-size SRAM packet buffers.

As widely used in Internet, the mature TCP/IP have also been chosen as the *de facto* communication protocols in the data centers. Unfortunately, Transport Control Protocol (TCP), which is specifically designed for traditional networks, performs poorly in data center networks (DCNs). The reasons are twofold. First, the data center applications, such as clustered storage systems and large-scale web search, usually adopt the barrier-synchronized and many-to-one communication pattern to achieve high performance and service reliability. For example, a typical barrier-synchronized scenario occurs in clustered file systems, in which the data blocks are striped across a number of servers termed workers. Each worker has its own data fragment which is denoted as Server Request Unit (SRU) and the aggregator server initiates requests to all workers in parallel to fetch all SRUs. When the workers receive requests, they send SRUs back to the aggregator. The next round of requests cannot be initiated until all workers have finished transferring SRUs in the previous round. A stalled worker becomes the bottleneck of this clustered file system.

Second, when all workers transfer SRUs concurrently via the same output port, a shallow-buffered ToR switch would experience severe buffer overflows. If a full window of packets are lost in a TCP connection, this connection suffers Retransmission Time-Out (RTO) [3]. Once Time-Out happens, the aggregator server must wait for the stalled TCP flows to complete their retransmissions before initiating the next round of requests. Since the minimum RTO (RTO_{min}) is set to 200ms or 300ms in most operating systems, the excessive idle period of RTO (in comparison with RTT) throttles the effective throughput as low as 1-10% of the aggregator's available bandwidth capacity. Such inefficiency is termed as the TCP Incast problem in the literature.

Existing DCN TCP congestion control [4] [5] [6] have developed miscellaneous schemes to alleviate the impact of TCP Incast problem. They share a same design feature: adjusting TCP congestion windows to reduce the number of packets injected into bottleneck links. These schemes have been shown effective in handling high congestion caused by a moderate number of flows. However based on our empirical results in Section II, these protocols still cannot cope with the situation when the number of concurrent flows is *extremely* large. Under such a situation, even if all flow senders cut their congestion windows to a minimal value, the switch buffer still suffer frequent overflow [7].

In this work, we argue that existing data center protocols only focus on the adjustment of congestion windows, which is inherently unable to solve the TCP Incast problem caused by highly concurrent flows. Through theoretical analysis and empirical studies, we reveal that controlling the IP packet size reduces the Incast probability much more effectively than controlling the congestion windows in the presence of severe congestion. This is because that smaller packets allow more statistically fair sharing of shallow buffers in COTS switches, preventing Time-Out due to *a full window of packet losses* in a single TCP flow.

Specifically, to solve the TCP Incast problem, we propose Packet Slicing (PS) scheme, which improves the transfer efficiency of the very large number of parallel TCP flows by uniquely analyzing and inferring the suitable packet size for real-time network state based on the correlation among the congestion window and packet size. While the adjustment strategy of congestion window has been extensively studied in previous congestion control research, they have been exploited in isolation form the impact or assistance of varying packet size. In contrast, Packet Slicing joints the control of congestion window and packet size, allowing fellow researchers to gain deeper insight. Our contributions are as follows:

- We provide the first extensive study to exploit the root cause of TCP Incast in high concurrency. We reveal the impact of packet size on Incast probability and demonstrate experimentally and theoretically why slicing packet is more effective in avoiding Incast than cutting congestion window under severe congestion.
- We design a general supporting scheme by carefully adjusting IP packet size on widely used COTS switches. Our Packet Slicing design balances the benefit of reducing Incast probability with the cost of increasing header overhead due to smaller packet payloads by calculating the optimal packet size. The design uses standard ICMP signaling, which requires no modification of TCP protocols and can be transparently utilized by various TCP protocols.
- We evaluate our design on both NS2 simulations [8] and small-scale Linux testbed. The results demonstrates that, as with Packet Slicing, we are able to broadly improve the goodput of state-of-the-art data center protocols by average 26x. We also show that the increased number of smaller packets has almost no effect on the IO performance thanks to the Interrupt Coalescing (IC) and Large Receive Offload (LRO) [9] mechanisms widely adopted in existing COTS switches.

The remainder of this paper is structured as follows. In Section II, we describe our design rationale. The details of Packet Slicing are presented in Section III. In Section IV and V, we report the NS2 simulation and real testbed experimental results, respectively. In Section VI, we analyze the system overhead. We demonstrate existing approaches and discuss their pros and cons in Section VII. Finally, we conclude the paper in Section VIII.

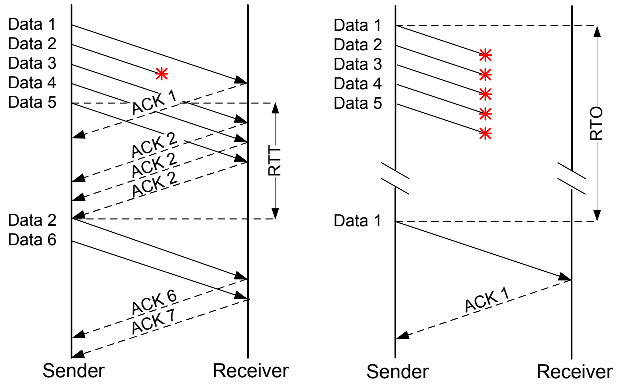
II. Motivation

In this section, we present empirical studies to demonstrate the impact of full window loss and show it is very common in the modern data centers with the COTS switches and high concurrent TCP flows. Then, we demonstrate theoretically that the Incast probability is effectively controlled by exploiting smaller packet sizes. Finally, we show the impact of different packet sizes on TCP performance.

A. Impact of full window loss

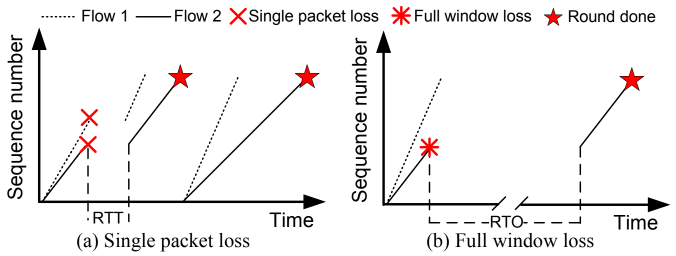
A TCP sender detects a packet loss using the two schemes described below. The first one is fast retransmit as shown in Fig.1(a). If only data packet 2 is dropped, the receiver will generate 3 duplicate ACKs indicating that packet 2 is lost. After about one RTT period (i.e., 10s or 100s microseconds), the sender retransmits the packet 2. Fig.1(b) shows the second case when all data packets sent by the sender are dropped, defined as full window loss. The sender has to wait for a certain period of time (named RTO), before recovering from the loss. Considering the 200 or 300 millisecond RTO_{min}, the recovery responding to full window loss is about three orders of magnitude slower than single packet loss.

Fig.2 shows the impact of multiple packet losses on concurrent flows in the barrier-synchronized transfer, where the next round of transfer cannot start until all senders in the previous round have finished sending. As shown in Fig.2(a), if the multiple packet losses are spread across different flows, it is very possible that the number of lost packets for each flow is lower than the congestion window size. Then, the fast retransmit scheme helps the senders to recover quickly from the packet loss. However, if the multiple packet losses occur in a single flow as shown in Fig.2(b), this unlucky flow may experience full window loss and induce Time-Out, blocking the next round of data transfer.



(a) Single packet loss (b) Full window loss

Fig. 1: Impact of packet loss on single flow



(a) Single packet loss (b) Full window loss

Fig. 2: Impact of packet loss on concurrent flows

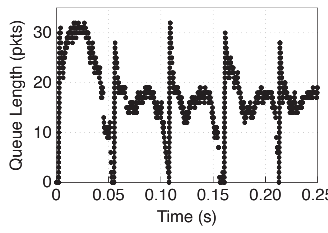
B. Full window loss in concurrent TCP flows on COTS switch

In modern data centers, the TCP's full window loss is a common phenomenon for following two reasons. First, the partition/aggregate pattern is the foundation of many large scale applications. Application requests are farmed out to worker servers, which simultaneously send response flows to the aggregator server to produce a result. This pattern inevitably brings about high flow concurrency. Taken the 44 port ToR switch as an example, the median number of concurrent flows is 36. In the multi-layer partition/aggregate pattern, the 99.99th percentile is even over 1,600 [5].

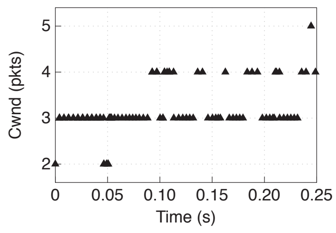
Second, like most commodity switches, the widely deployed COTS switches in data centers are shared memory switches, which use the logically common packet buffers available to all ports. However, to provide fair sharing memory, each interface has its maximum memory usage limitation, which easily leads to port queue overflow and packet loss [10]. Some kinds of switches have multi-ported memories, but are very expensive. Thus, most current COTS switches still use the shallow buffer due to the cost consideration.

The high flow concurrency and shallow buffer size together lead to insufficiency of available buffer space. For example, although a Triumph switch has 48 1Gbps ports and 4 10Gbps ports, its 4MB buffer size is shared by all ports, which means that each port only shares 78 KB buffer space in average, namely - accommodating 52 packets with 1500 Bytes packet size. Under this circumstance, even if each flow only sends 2 packets, the buffer is filled when at least 26 flows send data simultaneously and other packets are dropped, which may induce the TCP Incast problem.

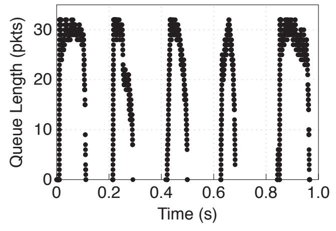
To investigate the performance of current data center TCP protocol under high concurrency and COST switches, we conducted a small experiment on a data center network testbed including 3 servers. The TCP protocol we used is well-known DCTCP [5]. We use a server with two 1Gbps NIC cards to emulate a typical shallow buffered, shared-memory ToR switch. Here we denote this server as *switch*. In this testbed, via the 1Gbps bandwidth, a worker server connects to the *switch* for sending multiple flows to an aggregator server. Each flow sends 5 rounds of 64KB SRU data. The packet size is 1.5KB. To make the output link become the network bottleneck, we limit the link speed of *switch*'s output port to the aggregator server as 100 Mbps. The buffer size is set as 32 packets. The *switch* is Explicit Congestion Notification (ECN) [11] enabled and operates in drop-tail mode. For DCTCP, we set the ECN mark threshold to 20, as recommended. The number of DCTCP flow sender (fan-in degree) is increased from 10 to 60 to test the impact of flow concurrency.



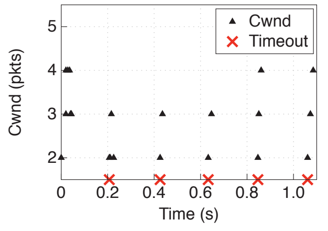
(a) Buffer queue length with 10 workers



(b) Congestion window with 10 workers



(c) Buffer queue length with 60 workers



(d) Congestion window with 60 workers

Fig. 3: DCTCP performance on COTS switch

We record the queue length of *switch*'s buffer. From Fig.3(a) and Fig.3(c), it is observed that, due to bandwidth oversubscription, queue length rapidly increases. At the same time, we pick a flow randomly from the concurrent flows and trace its congestion window. With 10 concurrent senders, Fig.3(b) shows the congestion window mainly fluctuates between 3 and 4. However, when the number of concurrent senders is 60, incurred by a high fan-in degree and thus too many ECN feedbacks, congestion windows are frequently suppressed to the minimum value 2 as shown in Fig.3(d). We observe that this flow experiences 5 full window losses, triggering TCP Time-Out event.

In short, this experimental study indicates that, even with a very small congestion window, DCTCP is still not able to cope with large number of concurrent flows in the COTS switches. Furthermore, we note that cutting congestion window reduces the number of in-flight packets for a flow. With a smaller number, the probability of all these packet are lost becomes higher under statistical multiplex. In other words, cutting congestion window has a detrimental side-effect: a larger probability of full windows losses (i.e., cause of Time-Out), although the probability of single packet loss becomes smaller (i.e., cause of retransmission). This is the fundamental drawback in current congestion control schemes in solving the Incast problem, and motivates us to explore the cutting the packet size as follows.

C. Cutting the Packet Size to avoid full window loss

When the concurrent flows inject too many flight packets into the bottleneck link, cutting congestion window still could not avoid the timeout event. However, it can be reasonably inferred that if the packet size is reduced, the buffer accommodates more packets and therefore alleviates the Incast problem.

To investigate the potential of cutting the packet size, we theoretically analyze the probability of Time-Out event with different packet size. We demonstrate that fewer Time-Out events will happen if the flow has smaller packet size. We use B to denote the buffer size of each port in bytes, which is maximum number of data bytes a port queue accommodates. Let s denote the packet size and w denote the congestion window for each flow. Given n flows, the number of all in-flight packets is thus nw. Since all n synchronized flows share the bottleneck link fairly using statistical multiplexing, the packet loss rate is calculated as $1 - B/nsw$.

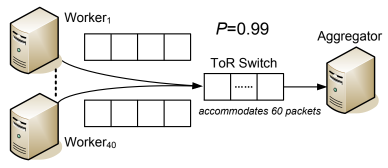
It is known that TCP Time-Out is mainly caused by a full window of packet losses, we calculate the probability of a full window of packet losses as the Time-Out probability p:

$$p = (1 - \frac{B}{nsw})^w. \tag{1}$$

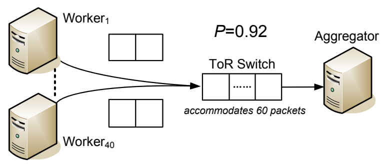
The TCP Incast probability P is defined as the probability that at least one flow experiences time out in the transfer of n flows sent by respective workers. We obtain P as

$$P = 1 - (1-p)^n = 1 - (1 - (1 - \frac{B}{nsw})^w)^n. \tag{2}$$

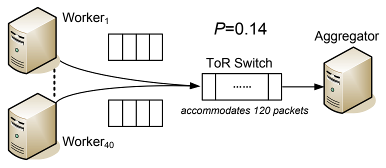
From Equ. (2), we find that the Incast probability P is determined by the congestion window w and packet size s given the fixed buffer size B and number of flows n. We deduce that, the value of P is reduced with smaller w or s.



(a) At the beginning



(b) After cutting congestion window



(c) After cutting packet size

Fig. 4: Cutting congestion window vs. packet size

Compared with cutting the congestion window size, cutting the packet size allows the same buffer to accommodate more packets. Consequently, under statistical multiplex, each flow has a higher chance to enqueue at least one packet, hence reducing the chance of a full window of packet losses. Here, we use an example to demonstrate that cutting the packet size is remarkably more efficient in reducing probability of full window losses and Incast event.

Fig.4 (a) shows that 40 concurrent flows share a bottleneck link with a buffer size of only 60 packets. Let us assume the congestion window size of each flow is 4 packets (i.e., totally 160 in-flight packets). Fig.4 (b) and (c) compare the effects of cutting congestion window with that of cutting packet size. Fig.4(a) shows that 100 packets are dropped due to overflow. Based on Equ. (1) and (2), we derive that the probability of full window losses (i.e., Time-Out probability p) of each flow is $(\frac{5}{8})^4 = 0.15$, and the Incast probability P is 0.99. As shown Fig.4 (b), if all flows halve their congestion windows, p and P become $(\frac{1}{4})^2 = 0.06$ and 0.92, respectively. In contrast, Fig.4 (c) shows that if packet size is cut into half, the buffer accommodates 120 packets and the congestion window size is still 4. Therefore, after cutting the packet size, p is $(\frac{1}{4})^4 = 0.004$, and P becomes as small as 0.14 in comparison with 0.92 in case of halving the window size.

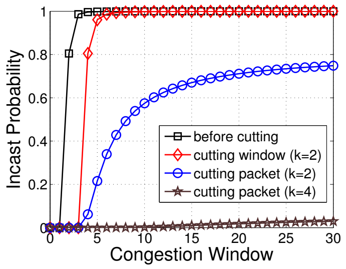
Without loss of generality, we denote k as the slicing ratio for congestion window and packet size. For example, if congestion window or packet size is halved, the corresponding slicing ratio is 2. Here, we give the comparison of Incast probability between cutting congestion window P_{cw} and cutting packet size P_{ps} as

$$P_{cw} = 1-(1-(1-\frac{B}{ns\times max(2,w/k)})^{max(2,w/k)})^n, \quad (3)$$

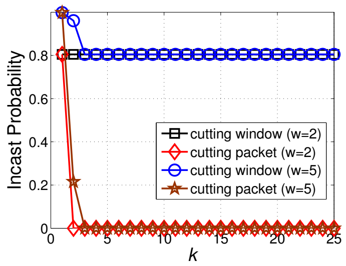
$$P_{ps} = 1-(1-(1-\frac{kB}{nsw})^w)^n \quad (4)$$

where $max(2, w/k)$ gives the minimum value of w as 2. Suppose the buffer size B and packet size s are 64KB and 1KB, respectively. Fig.5(a) plots the Incast probability of 40 concurrent flows for cutting w and s. When w is less than 5, P_{cw} is decreased after w is cut, while the decrement is limited. If w is larger than 5, cutting w becomes completely ineffective because $(1-2B/nsw)^{w/2}$ converges towards 1 with increasing w. On the other hand, if s is cut, P_{ps} is greatly decreased no matter what size of w. Moreover if $k = 4$, P_{ps} is decreased to almost 0 for all values of w.

As mentioned before, when highly concurrent flows share the bottleneck link, the congestion window size is suppressed to very small. Here, we fix w as 2 and 5, and compare P_{cw} and P_{ps} with increasing k. Fig.5(b) shows that, P_{ps} becomes almost 0 with k increasing in cutting packet, while cutting congestion window has negligible effect on P_{cw}. This confirms that, in highly flow concurrency (i.e. n=40), large number of flight packets still easily brings about Time-Out, even when w is cut into the minimum value of 2.



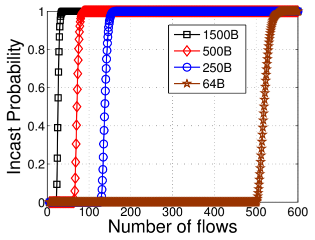
(a) Incast probability with varying w



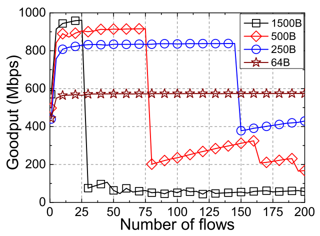
(b) Incast probability with varying k

Fig. 5: Incast probability with varying w and k

Next, we analyze the Incast probability for different packets size s with the increasing of n. From Fig.6(a), we find that it is much easier to enter Time-Out with the larger packet size. If the packet size is 1500B, the Time-Out probability reaches 1 when n is 25. While for 64B, the corresponding number of flows increases to as large as 540, indicating that the network system accommodates more than 500 concurrent TCP flows.



(a) Incast probability



(b) Network Goodput

Fig. 6: Incast probability and network goodput with varying n

Finally, we use NS2 to test the network goodput for different packet sizes. There are multiple workers respectively sending one DCTCP flow to a aggregator via a ToR switch with 64KB buffer per output port. Corresponding to the buffer size, we set the marking threshold to 20 packets for DCTCP according to [5]. Since the topology is homogeneous, the bandwidth of all links is set to 1Gbps, and the round-trip propagation delay is 100μs. RTO_{min} is set to 200ms by default.

Fig.6(b) shows that, the smaller packet size helps to avoid TCP Incast. When the packet size is 1500B, only 25 concurrent flows induce the TCP Incast throughput collapse. While for the 500B and 250B packet size, the corresponding maximum numbers of flows coexisting well without Time-Out increase to 76 and 146, respectively. On the other side, we also notice that cutting the packet size is unavoidable to degrade network efficiency due to the larger overhead of packet header. Although more than 500 flows coexist without Incast for the 64B packet size, the network goodput is reduced to 570Mbps, far from the 1Gbps link bandwidth.

D. Summary

Our observation of this particular experiment leads us to conclude that (i) slicing of packet size decreases the Time-Out probability, leading to the significantly increment in network goodput, and (ii) using smaller packet size introduces larger packet header overhead to transmit same amount of data, which suggests that adopting fixed small packer size is not optimal solution under dynamic network traffic scenarios. These conclusions motivated us to investigate a novel approach adjusting packet size. In the rest of this paper, we present our Packet Slicing method as well as a reference implementation in real testbed system.

III. Design of Packet Slicing

Since the traffic workload of a data center is highly dynamic, a fixed small packet size cannot produce optimal efficiency under all scenarios. This section presents Packet Slicing, which adjusts the packet size automatically and transparently. We note that Packet Slicing is a supporting design that is compatible with a wide range of data center TCPs.

A. Design Insight

To remind, Fig.6(b) indicates a tradeoff in our design: a smaller packet size reduces the Incast probability, meanwhile, brings about higher protocol overhead. In other words, we can provide higher network goodput of transport-layer data, if we ensure that (i) the Incast probability is decreased by cutting packets and; (ii) the overhead of extra packet headers is still under control.

The procedure involves several key challenges that are addressed in this section. First, we need to obtain the optimal value of the packet size, taking Incast probability and header overhead into consideration. Second, we need a low-cost computation method to deal with rapid changes of network dynamics. Third, we need a light-weight and compatible scheme for cutting packets with small firmware updates at switches, while making no modification on end hosts. Last, it should be compatible with existing transport layer protocols for practical deployment.

B. Model Analysis

To obtain optimal slicing ratio k, this section derives the relation between the network goodput and the slicing ratio. In our analysis, we consider the all in-flight packets in both buffer and link. Let C and RTT_{min} denote link capacity and propagation delay, respectively. We get the maximum number of packets Y that can be accommodated in switch buffer and link pipeline as

$$Y = B + C \times RTT_{\min}. \tag{5}$$

Assume the packets are evenly distributed among each flow when the number of packets reaches Y, then the congestion window w of each flow is kY/ns, given the packet size is s/k. In the next round of transfer, all flows use additive increase to enlarge their respective congestion windows by 1. Then, we obtain the packet loss rate as $1 - \frac{kY}{ns(w+1)} = \frac{ns}{kY+ns}$.

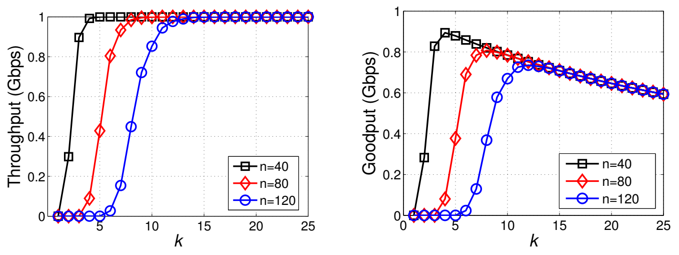
For all n flows, the Incast probability P is

$$P = 1 - (1 - (\frac{ns}{kY + ns})^{\frac{kY}{ns}+1})^n. \tag{6}$$

To study the impact of Incast probability P on network efficiency, we normalize the total network throughput T as Equ. (7). We simply suppose that T obtains its maximum value 1 when there is no buffer overflow, while it gets its minimum value 0 when P is 1.

$$T = \begin{cases} 1 - P & nsw + ns > kY \\ 1 & nsw + ns \leq kY \end{cases} \tag{7}$$

Fig.7(a) shows the network throughout with different value of slicing ratio k, where B and s are 64KB and 1.5KB, respectively. For the different number of flows from 40 to 120, higher network throughput is obtained with larger k. If network throughput is the only factor to be considered, we intuitively deduce that the packet size should be as small as possible to get the maximum network throughput.



(a) Throughput with increasing k (b) Goodput with increasing k

Fig. 7: Network throughput and goodput

If overhead is considered, however, this intuition no longer holds, as is evident from the following analysis of total network goodput G in Equ. (8). L_D and L_H are the default length of packet payload and packet header, which are 1460B and 40B, respectively.

$$G = \begin{cases} (1 - P) \times \frac{L_D}{L_D + k \times L_H} & nsw + ns > kY \\ \frac{L_D}{L_D + k \times L_H} & nsw + ns \leq kY \end{cases} \tag{8}$$

We plot the network goodput G with different values of slicing ratio k in Fig.7(b). Different from T, G is not always increasing with larger k. This suggests that we need to take packet head overhead into consideration if we manage to find the optimal packet size within each flow, then the highest network goodput is achieved. Meanwhile, the large computational complexity for calculating the optimal value of packet size in COTS switches shall be avoided when possible. In the following, we present an approximation approach to simplify the calculation process of the optimal packet size.

C. Numerical Approximation of Optimal Slicing Ratio

From Fig.7(b), we observe that the goodput G is always increasing when $nsw + ns > kY$. However, from Equ. (8), it is analytically hard to obtain the derivative of goodput G to verify such a monotonic trend. To simplify the deduction, we use the average congestion window $\hat{w}$ as a substitute for the congestion window of all flows in Equ. (8). Then we get the numerical approximation of network goodput $\hat{G}$ as

$$\hat{G} = \begin{cases} (1-(1-\frac{kY}{ns\hat{w}})^{\hat{w}})^{n} \times \frac{L_D}{L_D+k\times L_H} & ns\hat{w}+ns > kY \\ \frac{L_D}{L_D+k\times L_H} & ns\hat{w}+ns \leq kY \end{cases} \tag{9}$$

We calculate the derivative of $\hat{G}$ when $ns\hat{w}+ns > kY$ as:

$$\frac{d\hat{G}}{dk} = \frac{L_D(1-(1-\frac{kY}{ns\hat{w}})^{\hat{w}})^{n}}{L_D+k\times L_H} \times [\frac{(1-\frac{kY}{ns\hat{w}})^{\hat{w}-1}Y}{(1-(1-\frac{kY}{ns\hat{w}})^{\hat{w}})s} - \frac{1}{k} + \frac{L_D}{k^2}]. \tag{10}$$

We note that the minimum limit of packet size is 64B in current Ethernets, therefore a viable slicing ratio k ranges from 1 to 23 (note the default packet size is 1500B). To obtain the optimal slicing ratio $\hat{k}$, we analyze the following two cases.

- When $ns\hat{w} + ns > kY$, Equ.(10) shows that $\frac{d\hat{G}}{dk}$ is always larger than 0, which means that goodput $\hat{G}$ becomes larger with a larger slicing ratio k. This is because that the positive effect of cutting the packet size dominates when slicing ratio k is relatively small.
- When $ns\hat{w} + ns \leq kY$, goodput $\hat{G}$ becomes smaller with a larger k according to Equ. (9), because the overhead of extra packet headers eliminates the benefit of packet slicing, if the slicing ratio k becomes very large.

Now it becomes clear that we can obtain the approximation of optimal slicing ratio $\hat{k}$ by assigning $\hat{k}$ with a boundary value:

$$\hat{k} = \frac{ns\hat{w}}{Y}. \tag{11}$$

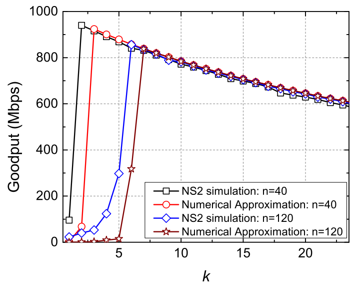
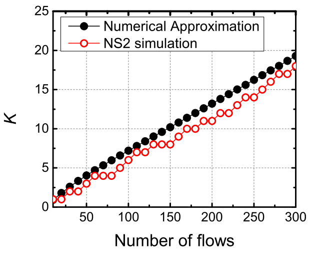
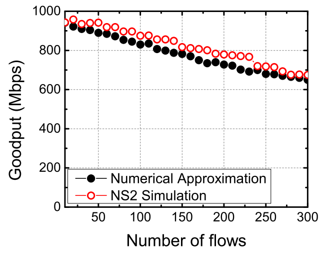


Fig. 8: Goodput comparison of NS2 simulations and numerical approximation

We evaluate the accuracy of our numerical approximation results using NS2 simulations with the same settings in Section II.B. As shown in Fig.8 and Fig.9, we maintain the numerical approximation of network goodput accurately over different number of flows. Specifically, Fig.8 shows the goodput with different k, from which we find that the numerical approximation well characterizes the evolution of network goodput. Fig.9 shows the optimal value of k achieving the highest goodput in simulation result and numerical approximation. Both the highest goodput and optimal k in numerical approximation closely follow the corresponding simulation results.



(a) Optimal k



(b) The highest goodput

Fig. 9: Optimal k achieving the highest goodput

D. Implementation

We implement the design of Packet Slicing as computing the optimal k. There are three key steps in our design. One is the notification of expected packet size: the switch should use an easy way to notify the expected packet size for all senders. We utilizes the ICMP message as the notification [12] [13]. After computing the optimal $\hat{k}$, the switch modifies the ICMP message's Next-Hop MTU field as the default packet size divided by $\hat{k}$, and sends the ICMP message to all senders. After receiving the ICMP message with the expected packet size, all senders adjust the packet size in their own flows.

The second one is estimation of concurrent number of flows n: each switch requires to maintain the number of concurrent flows. In Packet Slicing, we simply use the TCP handshake messages, SYN and FIN, to count the number of flows n. The operation overhead is negligible, because n is increased or decreased by one when switch receives a SYN or FIN message, respectively. This method is very simple, but may overestimate the number of flow, because some flows are closed without FIN message. However, if the number of flows is overestimated, the packets will be cut into smaller ones to avoid the Time-out event in a gentler manner. On the other hand, if the advanced method [14] is used, the accurate number of flows could be obtained by sampling traffic.

The last key point is utilizing the passive measurement method to estimate the congestion window w by observing packet passing through the switch [15]. In current data center TCP protocols, such as DCTCP [5], D^{2}TCP [16] and L^{2}DCT [17], the senders utilize ECN to avoid queue buildup. This facilitates the measurement by looking at the CE codepoint in the packet header to infer the congestion window. To avoid the operation cost in the passive measurement, we also propose an alternative way to estimate w for highly concurrent flows. As observed in Section II.B, the congestion window w is cut down to very the minimum value (i.e., 2) when the concurrent number of flows n becomes large. Therefore, w is approximatively estimated as the minimum value when n is very large (i.e., $> \frac{Y}{2s}$).

All these implementation issues can be addressed without modifying the transport protocols – we simply use ICMP message to adjust the packet size in the original transport protocols when the number of concurrent flows becomes large. By doing so, the transport protocols automatically avoid the Incast problem, thereby achieving higher network goodput.

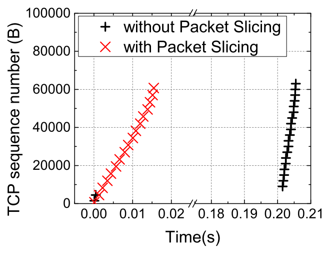
IV. Simulation Evaluation

In this section, we embed Packet Slicing with classical data center protocols in NS2 simulations. First, taking DCTCP as an example, we examine the basic performances of Packet Slicing. Then, we evaluate Packet Slicing with DCTCP, D^{2}TCP and L^{2}DCT, across various network scenarios.

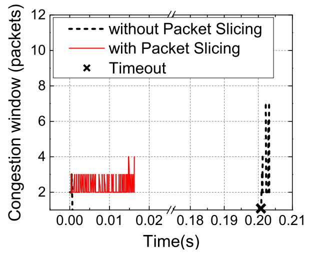
As [4] [5] [18], we use a single-rooted tree topology, where multiple servers send data (responses to queries) to one aggregator simultaneously through a bottleneck link. The default packet size and SRU size are set as 1.5KB and 64KB, respectively. According to the typical COTS switch, the buffer size is set to 64KB. Other simulation settings are as same as that in Section II.B. In the three TCP protocols, the weighted averaging factor g is 1/16 and the buffer occupancy threshold for marking CE-bits is 20 packets. For D^{2}TCP, the deadline imminence factor d is between 0.5 and 2.0. The weight of each flow decreases from initial 2.5 to final 0.125 along with data transmission in L^{2}DCT.

A. Basic Performance Results

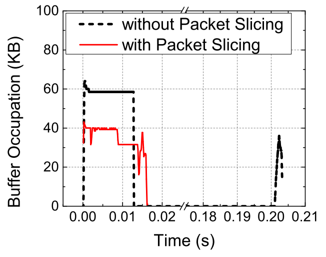
To explain how Packet Slicing avoid the performance impairments described in Section II, we set up the simulation environment with 30 DCTCP flows bursting synchronously.



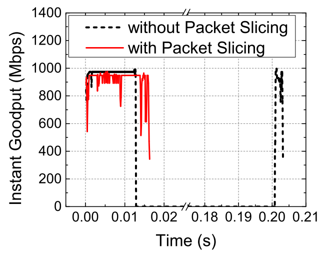
(a) TCP sequence number



(b) Congestion window



(c) Buffer occupation



(d) Instant goodput

Fig. 10: Basic performance results

1) TCP sequence number: To provide deep insights into how Packet Slicing avoid TCP Incast in detail, we randomly select one flow and trace its TCP sequence number on bytes. Form Fig.10(a), we observe that, DCTCP without Packet Slicing experiences Time-Out at 0.8ms and finally finishes its transfer after 200ms. On the contrary, with Packet Slicing, no Time-Out happens and the flow transmission finishes at 16ms.

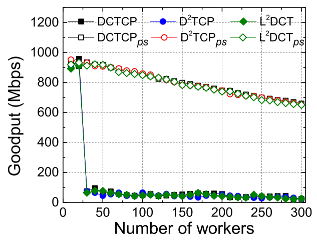
2) Congestion window and Time-Out: As shown in Fig.10(b), the selected flow with Packet Slicing always keeps the small congestion window as 2 and 3. Note that the Time-Out events include three categories as full window losses, retransmitted packets losses and last packets losses. Here, without the help of Packet Slicing, this flow experiences the full window losses and then Time-Out happens. Moreover, according to the traces of our all tests, full window losses cases are responsible for the greatest proportion, about 95%, of all Time-Out events, which is similar to the results in real systems from literature [3].

3) Buffer occupation: We compare instant buffer occupation to find out whether Packet Slicing can effectively control the queue buildup. As shown in Fig.10(c), due to the high flow concurrency of 30, DCTCP queue touches the upper bound of switch buffer 64KB quickly, which implies some packets are dropped and Time-Out may come together as well. In the case of DCTCP with Packet Slicing, the buffer occupation reaches the largest value of 43KB at the very beginning. After that, the arrived ICMP message brings the expected packet size back to the senders, then they tune their packet sizes and stabilize the buffer occupation during their remaining data transfer.

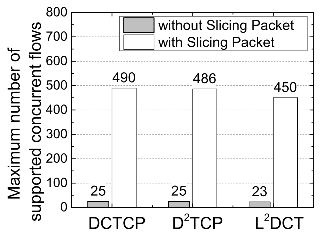
4) Instant goodput and average flow completion time: Fig.10(d) shows the instant network goodput. DCTCP with Packet Slicing achieves the almost full bottleneck link utilization throughout its transmission. Due to impact of Time-Out event, DCTCP without Packet Slicing has to wait for the long idle time, which greatly degrades the gross link utilization by up to 92% compared to Packet Slicing. We also measure the average flow completion time (AFCT) of all flows. By avoiding Time-Out event, Packet Slicing reduces AFCT by 71%.

B. Performance results under different flow concurrency

To test Packet Slicing's broad applicability and effectiveness under different flow concurrency, we evaluate the network performance of DCTCP, D^{2}TCP and L^{2}DCT with Packet Slicing enabled and disabled. For simplicity, we use symbols DCTCP$_{PS}$, D^{2}TCP$_{PS}$ and L^{2}DCT$_{PS}$ to denote DCTCP, D^{2}TCP and L^{2}DCT with Packet Slicing enabled, respectively.



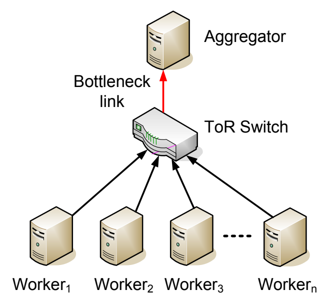
(a) Network goodput with increasing number of workers



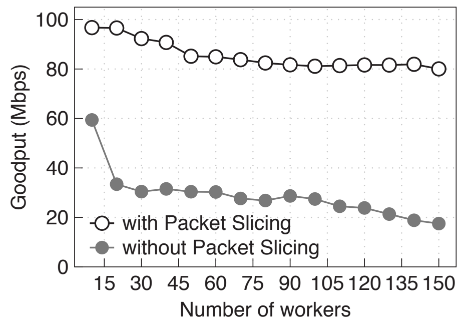
(b) Maximum number of supported concurrent flows

Fig. 11: Performance results under different flow concurrency

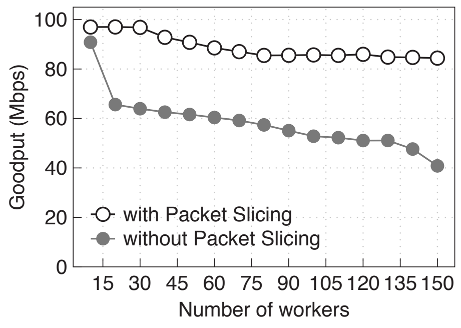
1) Total goodput: The network total goodputs with up to 300 worker servers are shown in Fig.11(a). Each worker respectively sends one flow to a aggregator server. We find that DCTCP, D^{2}TCP and L^{2}DCT experience severe degradation when the number of workers rises to 30. However, if Packet Slicing is used, all three protocols achieve very high goodput.



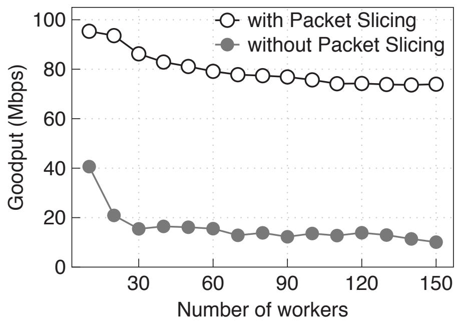
(a) Network topology



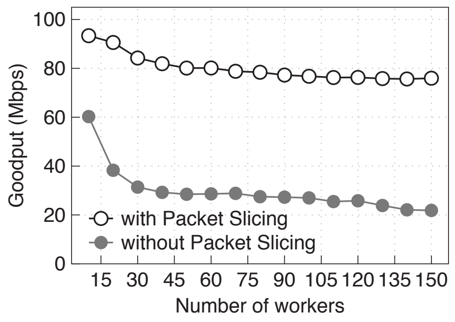
(b) SRU = 64KB, Buffer size = 32KB



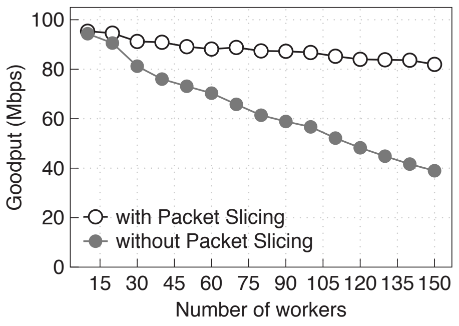
(c) SRU = 128KB, Buffer size = 32KB



(d) SRU = 32KB, Buffer size = 32KB



(e) SRU = 32KB, Buffer size = 64KB



(f) SRU = 32KB, Buffer size = 128KB

Fig. 12: Network topology and MapReduce-like application performance

For example, for the 300 workers, Packet Slicing collaborating with the three protocols obtains average 26x increase in network goodput. We also observe that the network goodput decreases with more workers in Packet Slicing, because the overhead of extra packet headers becomes higher when the packet size is adaptively reduced. This shows the tradeoff between high flow concurrency and low network efficiency.

2) Maximum number of supported concurrent flows: In fact, due to the minimum limit of packet size, Packet Slicing could not cut packets into smaller than 64B, which inevitably limits its effectiveness. Here, we vary the number of simultaneous workers to find the maximum number of supported concurrent flows, which indicates how many flows coexist well without Incast happening. As shown in Fig.11(b), with the help of Packet Slicing, the maximum number of supported concurrent flows is increased by more than 19x on average.

3) large bandwidth and randomness: We also investigated Packet Slicing's performance in scenarios including 10Gbps links and larger randomness. The test results are qualitatively similar to those presented above. We found a reduction in Incast problem for 10Gbps since the larger bandwidth could accommodate more packets, but the problem resurfaces for larger number of concurrent DCTCP flows (i.e., 45). As expected, Packet Slicing performs well at 10Gbps link and supports more than 900 concurrent DCTCP flows. Besides, we set sending time of each flow in the 10ms time interval with the uniform distribution to add the randomness in concurrent flows. This brings less synchronization and alleviates Incast problem to some extent. However, the problem still exists when the number of DCTCP flows exceeds 37. With Packet Slicing, more than 720 DCTCP flows coexist without Incast.

V. Testbed Experimentation

In this section, we conduct the testbed experiments to explore results of using Packet Slicing. To evaluate Packet Slicing's performance in the typical application of data center, we choose the MapReduce-like and web search applications.

A. Parameters and Topology

The testbed is made up of three servers that are DELL T1500 workstations with Intel Core i5 2.66 GHz dual-core CPU, 8 GB DDR3, and 500 GB hard disk. As similar as [17], we emulates the typical network topology shown in Fig.12(a). Specifically, one server with 1Gbps NIC cards acts as the ToR switch and other servers all connect to this server. The other servers are equipped with 1Gbps NIC cards and act as workers and aggregator. All servers are running CentOS 5.5 of Linux kernel 2.6.38 within our patches applied. The RTT without queuing delay is approximately 100μs between any two servers. As same as in Section II.B, we limit the link speed of switch output port to 100Mbps. We use a worker to emulate multiple workers sending data to the aggregator via the bottleneck link. Default packet size and RTO$_{min}$ is set to 1.5KB and 200ms, respectively.

B. MapReduce-like application

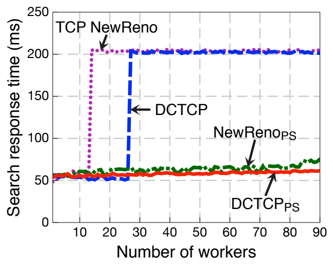
In this test of MapReduce-like application, the aggregator generates a query to each worker, and each of them immediately responds with SRU size of data. Here, we test DCTCP with Packet Slicing and compare it to the results with original DCTCP. Fig. 12(b) and Fig. 12(c) shows the experimental results with different SRU size. It is found that the larger SRU size has higher goodput. This is because, using larger SRU size, the whole transfer duration is expanded and flows can therefore utilize the spare link capacity made available by any unlucky flow waiting for a time-out event. Note that, due to 100Mbps bandwidth limitation of bottleneck link in testbed experimentation, the goodput improvement by Packet Slicing is much less than the simulation results. Nonetheless, DCTCP with Packet Slicing keep higher goodput in each SRU size.

Next, we present the network goodput with different buffer sizes in Fig. 12(d) $\sim$ Fig. 12(f). With larger buffer size, more packets are buffered and then the network goodput increases, especially for DCTCP. However, this improvement requires large buffer. On the contrary, even for small buffer size, Packet Slicing achieves much better performances.

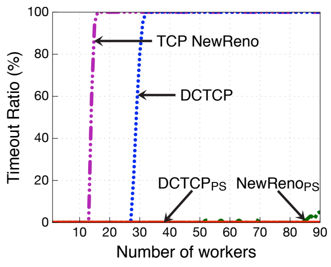
C. Web search application

In the web search application, we test the performances of DCTCP and TCP NewReno. We focus on the search response time (SRT), that is, the delay between when the query is sent by the aggregator and when the response data is completely rendered by all workers. Unlike the previous scenarios, the response data size of each worker is 500KB/n, where n is the total number of workers. We measure SRT when n increases from 1 to 90. As shown in Fig. 13(a), during the experiments, DCTCP$_{PS}$ shows generally low SRTs ranged between 53 ms and 61 ms in all cases. NewReno$_{PS}$ also achieves at most 75 ms SRT. However, both DCTCP's and NewReno's SRT increase by up to more than 200 ms because of Time-Out.

We also measure the timeout ratio, the fraction of queries that suffer at least one timeout as shown in Fig. 13(b). As expected, TCP NewReno suffers at least one timeout among the workers in most experimental rounds when the number of workers is more than 13. DCTCP starts to experience timeouts when the number of workers is 28 because DCTCP cannot prevent TCP Incast congestion when the number of workers is very large. Finally, we observe that DCTCP$_{PS}$ suffers no timeout in all cases and NewReno$_{PS}$ maintains a very small time-out ratio as well, which directly results in low SRT.



(a) Search response time

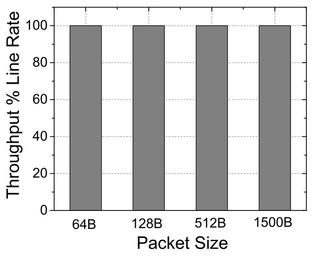


(b) Timeout ratio

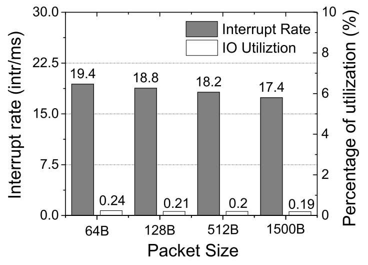
Fig. 13: Web search application performance

VI. System Overhead

From the system view, Packet Slicing cuts packets into smaller ones and increases processing overhead on switches and end hosts. In this section, we use experimental results to investigate the overhead issue of small packets. The experimental scenario involves 40 senders generate the same amount of TCP traffic to a specific receiver under a typical COTS ToR switch, Huawei S5700 switch, which has 48 ports of 1Gbps interfaces and 4MB shared buffer. All senders and receiver are the same DELL workstations used in the testbed experimentation.



(a) Switch performance



(b) Operation system performance

Fig. 14: System overhead

We firstly evaluate the effect of small packets on the forwarding performance of COTS switch. Fig.14(a) shows the throughput performance for different packet sizes in the range of 64 Bytes to 1,500 Bytes. This COTS switch exhibits full line rate traffic handling ability on all 1Gbps ports without incurring any packet loss. Even for very small packet size (i.e., 64 Byte), non blocking throughput is obtained at full line rate.

On the end hosts, we measure the interrupt rate and IO utilization. Specially, the processing overhead includes the per-packet and per-byte overhead on the operation system. The per-packet processing overhead is the major burden, which brings about the high rate of interrupts that the end hosts have to process. The per-byte processing overhead includes CRC and Checksum computations presented as the IO utilization.

From Fig.14(b), we observe that the smaller packet size (i.e., 64B) brings about 5% increasing on interrupt rate, whereas almost no effect on the IO performance. Although the result is counterintuitive, it is explained as following. Traditionally, the NIC generates an interrupt for each packet that it receives. Fortunately, most modern NICs provide Interruption Coalescing mechanism to reduce the number of interrupts generated when receiving packets. When multiple packets are received, these adapters buffer those packets locally and only interrupt the system once. Moreover, Large Receive Offload is also widely utilized in NIC to aggregate multiple incoming packets from a single flow into a larger buffer before they are passed to the network stack, thereby decreasing CPU utilization when the end system is receiving multiple small packets.

As for the per-byte processing overhead, the required resources are linear in nature and thus increase as the number of bytes increases. However, as shown in Fig.14(b), since the per-byte processing overhead only generates a tiny fraction of CPU load, not surprisingly small packets do not incur excessive overhead to IO utilization.

VII. Related Works

Literature has noticed the significant delay gap between RTO_{min} (e.g., at least 200ms) and RTT (e.g., hundreds of microsecond), which makes it critical to reduce the chance of *RTO* that directly leads to TCP throughput collapse. Various proposals target enhancing TCP protocol to address TCP throughput collapse. In order to perform accurate congestion control, DCTCP [5] leverages ECN scheme to adjust congestion window size. Thereby, DCTCP not only reduces the queueing delay, but also achieves high throughput. Compared with DCTCP, Tuning ECN [19] uses dequeue marking instead of traditional enqueue marking to accelerate the congestion feedback. As a receiver-driven congestion control scheme, ICTCP [6] adaptively adjusts the receive window on the receiver side to throttle aggregate throughput and thus alleviates Incast congestion. PAC [20] proactively controls the sending rate of ACKs on the receiver side to prevent Incast congestion. However, PAC still needs the help of switch to obtain accurate congestion state through ECN feedback. TCP-PLATO [21] introduces a *labelling* system to ensure that *labelled* packets are preferentially enqueued at switch. Therefore, the TCP sender can exploit duplicate ACKs to trigger retransmission instead of waiting for Time-Out. To maintain TCP self-clocking, CP [22] simply drops the packet payload instead of the whole packet on overloaded switch.

D^2TCP [16] considers the deadline requirements of latency-sensitive applications in DCN. D^2TCP elegantly adjusts the extent of window decreasing according to the flow urgent degree. When congestion occurs, far-deadline flows release some bandwidth to near-deadline flows. D^2TCP delivers high application throughput by meeting more deadlines. Targeting a reduction in flow completion times for short flows, L^2DCT [17] achieves the LAS scheduling discipline at the sender in a distributed way. According to the data size that has been sent, L^2DCT distinguishes long and short flows and assigns higher bandwidth to short flows, thereby reducing the mean flow completion time.

In contrast with the enhanced TCP protocols focusing on the congestion window adjustment, our solution Packet Slicing fundamentally tackles the TCP Incast problem through a different perspective: we reduce the chance of RTO by dynamically controlling the packet size. The key difference is that existing solutions tackle the congestion scenarios with a moderate number of heavy flows, while our solution addresses the congestion scenarios with a large number of highly concurrent flows, where existing solutions become ineffective.

VIII. Conclusion

This work presents Packet Slicing for highly concurrent TCP flows in data center networks. We adjust the IP packet size with low computing overhead, using only ICMP messages to mitigate the TCP Incast problem occurred in shallow buffered COTS switches. Packet Slicing is deployed only at the ToR switches, which avoids modifying thousands of servers. Packet Slicing is a supporting design compatible with a wide range of data center transport control protocols. In other words, our design can obtain significant goodput gains without modifications of transport protocols. To test Packet Slicing's broad applicability and effectiveness, we integrated Packet Slicing with three state-of-the-art data center TCP protocols and evaluated the design on the real testbed. The results indicate that with Packet Slicing, we effectively reduce the number of Time-Out events among the highly concurrent TCP flows, hence alleviating the TCP Incast problem. Consequently, we remarkably improve network goodput by 26x across different TCP protocols and increase the maximum number of concurrent TCP flows by more than 19x on average.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (61103204, 61572530), the China Hunan Provincial Science & Technology Program (2012GK4106), the International Science & Technology Cooperation Program of China (2013DFB10070).

References

[1] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. *VL2: A scalable and flexible data center network*. in Proceedings of SIGCOMM, 2009.

[2] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang , Y. Shi, C. Tian, Y. Zhang, and S. Lu. *Bcube: High performance, server-centric network architecture for data centers*. in Proceedings of SIGCOMM, 2009.

[3] A. Phanishayee, E. Krevat, V. Vasudevan, D. Andersen, G. Ganger, G. Gibson, and S. Seshan. *Measurement and analysis of TCP throughput collapse in cluster-based storage systems*. in Proceedings of USENIX FAST, 2008.

[4] V. Vasudevan, A. Phanishayee, H. Shah, l. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller. *Safe and effective fine-grained TCP retransmissions for datacenter communication*. in Proceedings of SIGCOMM, 2009.

[5] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. *Data Center TCP (DCTCP)*. in Proceedings of SIGCOMM, 2010.

[6] H. Wu, Z. Feng, C. Guo, and Y. Zhang. *ICTCP: Incast congestion control for TCP in data-center networks*. IEEE/ACM Transation on Networking. 2013, 21(2) : 345-358.

[7] J. Chen, J. Iyengar, L. Subramanian, and B. Ford. *TCP Behavior in Sub-Packet Regimes*. in Proceedings of SIGMETRICS, 2011.

[8] *NS-2 network simulator*. http://www.isi.edu/nsnam/ns/, 2000.

[9] R. Kapoor, A. Snoeren, G. Voelker, and G. Porter. *Bullet Trains: A study of NIC burst behavior at microsecond timescales*. in Proceedings of CoNEXT, 2013.

[10] P. Agarwal, B. Kwan, and L. Ashvin. *Flexible buffer allocation entities for traffic aggregate containment*. US Patent 20090207848, 2009.

[11] K. Ramakrishnan and S. Floyd. *A proposal to add explicit congestion notification (ECN) to IP*. RFC 2481, 1999.

[12] J. Mogul and S. Deering. *Path MTU Discovery*. RFC 1191. https://tools.ietf.org/html/rfc1191, 1990.

[13] J. McCann, S. Deering, and J. Mogul. *Path MTU Discovery for IP version 6*. RFC 1981. https://tools.ietf.org/html/rfc1981, 1996.

[14] J. Sanjuaas-Cuxart, P. Barlet-Ros, N. Duffield, and R. Kompella. *Cuckoo sampling: Robust collection of flow aggregates under a fixed memory budget*. in Proceedings of INFOCOM, 2012.

[15] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. *Inferring TCP connection characteristics through passive measurements*. in Proceedings of INFOCOM, 2004.

[16] B. Vamanan, J. Hasan, and T. Vijaykuma. *Deadline-aware datacenter TCP (D^2TCP)*. in Proceedings of SIGCOMM, 2012.

[17] A. Munir, I. Qazi, Z. Uzmi, A. Mushtaq, S. Ismail, M. Safdar, and B. Khan. *Minimizing flow completion times in data centers*. in Proceedings of INFOCOM, 2013.

[18] C. Wilson, H. Ballani, T. Karagiannis and A. Rowstron. *Better never than late: Meeting deadlines in datacenter networks*. in Proceedings of SIGCOMM, 2011.

[19] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang. *Tuning ECN for data center networks*. in Proceedings of CoNEXT 2012.

[20] W. Bai, K. Chen, H. Wu, W. Lan, and Y. Zhao. *PAC: Taming TCP Incast congestion using proactive ACK control*. in Proceedings of ICNP 2014.

[21] S. Shukla, S. Chan, A. Tam, A. Gupta, Y. Xu, and H. Chao. *TCP PLATO: Packet labelling to alleviate Time-Out*. IEEE Journal on Selected Areas in Communications, 2014, 32(1): 65-76.

[22] P. Cheng, F. Ren, R. Shu, and C. Lin. *Catch the whole lot in an action: Rapid precise packet loss notification in data centers*. in Proceedings of USENIX NSDI, 2014.