Minimizing Datacenter Flow Completion Times with Server-based Flow Scheduling

Jie Zhang¹, Dafang Zhang^{1,*}, Kun Huang², and Zheng Qin¹

¹College of Computer Science and Electronics Engineering, Hunan University

²Institute of Computing Technology, Chinese Academy of Sciences

{jiezhang, dfzhang, zqin}@hnu.edu.cn, huangkun09@ict.ac.cn

Abstract—Minimizing flow completion times (FCT) is a critical issue in datacenter networks. Existing approaches either cannot minimize FCT (i.e., DCTCP) or are costly to deploy with hardware modification on switches (i.e., pFabric). This paper presents a server-based flow scheduling (SFS) scheme for enabling easy and rapid deployment in servers while almost retaining the same minimal FCT as state-of-the-art pFabric. SFS combines traffic control and flow scheduling to keep in-switch queue very short, hence switches do not need to schedule flows and such function can be moved to servers. With SFS, each server keeps its highest-priority flow active and pauses the other low-priority ones. Then flows are completed one by one for minimizing FCT. We propose two novel techniques to achieve SFS. First, the bidirectional flow scheduling technique is used for each server to locally schedule sent or received flows. Second, the most recently seen flow coordination technique is used to coordinate senders and receivers to address the priority disagreement problem. We show that SFS scales to thousands of senders in Incast scenario and achieves zero-queue in the entire networks. Experimental results on NS2 show that SFS outperforms DCTCP and pFabric in Incast scenario. On real workloads in FatTree, SFS achieves 4X faster than DCTCP and closely approaches pFabric in tail FCT of short flows.

I. INTRODUCTION

Minimizing flow completion times (FCT) is a critical issue in datacenters. Many cloud services following partition/aggregation pattern usually generate a large amount of short flows, and the overall response time is determined by each individual FCT. A flow missing deadline will not contribute to the final result, affecting user-perceived experience and causing revenue loss. Especially, in current deployed datacenter networks (DCN), the latency-sensitive short flows suffer from large latency. FCT can be as high as tens of milliseconds while in theory they could finish in 10-20 microseconds [1]. The reason is that short flows often get queued up behind large flows, which results in long FCT.

Recently, several approaches have been proposed to reduce FCT. There are two categories of solutions: rate control [2–4] and flow scheduling [1, 5–8]. Rate control schemes vary rate based on information of congestion and deadline to keep queue short, hence flows experience low latency during transmission and can catch up with their deadlines. Although these schemes generally improve latency and are also easy to deploy in switches with no modification, they cannot achieve the optimal performance. A recent study [5] points out that minimizing FCT requires preemptive flow scheduling, so that rate control is a poor and ineffective technique for flow

scheduling and minimizing FCT [1]. Having recognized the above limitation, Alizadeh et al. [1] proposed a switch-based flow scheduling scheme called pFabric. pFabric achieves near theoretically optimal FCT even at the 99th percentile for short flows. It is the best known prior approach minimizing FCT. Unfortunately, pFabric is a clean-state approach that requires non-trivial hardware modification. Deploying it in datacenters built with large amounts of commodity switches may be either infeasible or costly, hence its performance gain cannot be achieved easily in today's datacenters. An alternative way is to deploy pFabric on current class of service (CoS) queues [1]. However, our experimental results show that such deployment can potentially leads to unpredictable FCT for insufficient priority queues.

Inspired by the fact that it is easy to modify software on servers, in this paper, we propose a server-based flow scheduling (SFS) scheme. Our goal is not to beat pFabric, but instead to offer a tradeoff between performance and deployment, while minimizing FCT. SFS is a combination of traffic control and flow scheduling. It controls the total amount of flighting packets to make the entire network zero-queue, hence flow scheduling can be moved from switches to servers. We argue that switches can guarantee low latency of small flows by scheduling small flows not queued up behind large flows in switches. However, if the network is kept zero-queue, we can achieve the same performance for small flows. Thus, the function of flow scheduling can be moved from switches to servers, where various scheduling policy can be easily implemented in software. In addition, recent studies [9–12] have shown the possible opportunity to achieve such a zero-queue network using full bisection bandwidth and packet spraying routing.

With SFS, flows are bidirectionally scheduled by senders and receivers to minimize FCT. Each sender or receiver keeps its highest-priority flow active while pausing the other low-priority ones. Then flows are completed one by one according to their priorities for minimizing FCT. At the receiver-side, unlike existing approaches [1, 5–8, 13–15], we propose a novel backward scheduling technique, which is to control ACK packets to indirectly schedule incoming data packets. The ACK packets of low-priority flows are captured to prevent their senders from generating new data packets. The ACK packets of the highest-priority flow are released to let the highest-priority flow monopolize all bandwidth. By doing this,

1
2
3 receivers also control the total amount of fighting packets and
4 make the entire network zero-queue, which in turn becomes
5 the footstone of SFS. At the sender-side, we use the forward
6 scheduling technique [1, 5–8, 13–15], which is to directly
7 schedule sent flows. The data packets of the highest-priority
8 flow are sent out while the data packets of the other low-
9 priority flows are captured and stored.

10 The bidirectional flow scheduling technique is still insuffi-
11 cient. SFS has a specific problem called priority disagreement,
12 which interrupts transmission and increases FCT. A flow can
13 be the highest-priority one at its receiver, while a low-priority
14 one at its sender. In this case, the receiver is idling as its
15 highest-priority flow is interrupted by the sender while other
16 low-priority ones have been paused by itself. To address
17 this problem, coordination between senders and receivers is
18 necessary. We borrow idea from temporal locality and design a
19 technique called most recently seen (MRS) flow coordination,
20 which ensures that servers will avoid idling and seize all
21 opportunities to transmit packets. When priority disagreement
22 occurs, the sender notifies its receiver. Then the receiver
23 temporarily activates a low-priority flow to saturate network by
24 releasing its ACK packets. What's more, the receiver always
25 selects the MRS flow to activate. The MRS flow usually has
26 the highest probability to build a data transmission with its
27 sender. Therefore, activating it can quickly saturate network
28 and efficiently eliminate the idle time caused by priority
29 disagreement.

30
31 Using SFS, we provide realistic benefits to today's DCN.
32 SFS requires no modifications on switches, no additional de-
33 vices for arbitration, no global information from the network,
34 no complex congestion control and rate estimate. It scales to
35 thousands of senders in Incast [16] scenario. It also achieves
36 zero-queue in the entire network. We conduct experiments
37 in NS2 to evaluate its performance. In Incast scenario, SFS
38 outperforms DCTCP and pFabric in both total request com-
39 pletion time and average individual FCT. And it tolerates up
40 to 2000 senders without any packet loss. In FatTree topology
41 with realistic workload, SFS's 99th percentile FCT of short
42 flows can be 4X faster than DCTCP, and approaches that of
43 pFabric. When SFS works in FatTree at extremely high load,
44 the max queue length is less than 60KB, and the average queue
45 length is less than 2.5KB.

46
47 The remainder of this paper is organized as follows. In
48 Section II, we overview the related work and describe our
49 motivation. In Section III, we describe SFS's design in detail.
50 In Section IV, we evaluate SFS's performance. Finally, we
51 conclude the paper in Section V.

52 II. RELATED WORK AND MOTIVATION

53
54 In this section, we briefly review the related work for min-
55 imizing FCT and describe our motivation. Also, we introduce
56 two TCP mechanisms utilized by our SFS.

57 A. Prior Schemes for Minimizing FCT

58
59 There are two categories of schemes for minimizing FCT:
60 rate control and flow scheduling. The rate control schemes

61
62 vary data transmission rate of the sender to reduce FCT of
63 latency-sensitive short flows. For instance, DCTCP [2] and
64 HULL [3] use an ECN-based congestion control algorithm
65 with appropriately throttling long elephant flows to keep queue
short for reducing the FCT of short flows. D2TCP [4] is an
extension to DCTCP, which reduces the fraction of missed
deadlines in a differentiated-deadline way, where far-deadline
flows back off more, and near-deadline flows back off less.
These schemes improve latency and are also easy to deploy
with no modification on switches.

However, the rate control schemes above suffer from the
limitations of inefficiency for minimizing FCT. This is because
they use an unpreemptive way for rate control by adjusting
TCP congestion window. To overcome the limitations, several
efforts have recently been done with flow scheduling [1, 5–8].
For instance, pFabric [1] is a switch-based flow scheduling
scheme, which achieves near theoretically optimal FCT by
using the shortest-remaining-flow-first policy. But this scheme
is not easy to implement due to non-trivial modifications to
switch hardware. Unlike pFabric, our SFS is a server-based
flow scheduling scheme, which not only achieves comparable
performance with pFabric but also is easier to deploy on
servers.

Several variants of pFabric have been proposed to en-
hance practical implementations and optimize the perfor-
mance. PASE [6] combines rate control [2], flow scheduling
[1] and arbitration [5] to achieve the minimal FCT for ex-
tremely high network load. This scheme requires additional
dedicated controllers co-located with TOR switches to arbitrate
flows, without modification of switch hardware. PIAS [7] uses
CoS (Class of Service) queues of switches to roughly schedule
flows with no prior knowledge of flow sizes required by
pFabric [1] and no additional controllers required by PASE [6].
This scheme may induce high FCT for same-priority flows due
to limited number of available CoS queues. On the contrary,
SFS is a software-based solution with large amounts of queues
in servers for fine-grained flow scheduling. Fastpass [8] uses
a centralized arbiter to determine the sending time and the
route path for every packet, achieving zero-queue and low
latency. Such centralized scheduling may have the limitation
of scalability.

Closely related work to ours is pHost [17], which shares
similar idea as SFS. There are two main differences. First,
pHost cannot efficiently handle the issue of flow priority
disagreement. When priority disagreement occurs, the receiver
in pHost uses linear searching on its flow list to find a proper
flow for saturating the idle network, leading to bandwidth
waste. On the contrary, our SFS proposes the MRS coordi-
nation mechanism to improve bandwidth utilization. This
mechanism uses heartbeat packets notified by sender-side
servers to activate the MRS flow with temperature-based sort
searching. Second, pHost may induce too much packet loss
for high network load and massive short flows. This scheme
uses an aggressive sending policy like pFabric to improve the
whole performance of short flows in the case of low or medium
network load, which in turn may lead to additional packet loss

when network load becomes high. Unlike pHost, SFS uses a conservative policy for sending packets to avoid packet loss in the case of high network load such as Incast and web search scenarios.

In addition, recent studies focus task-aware coflow scheduling on a group of flows. Baraat [13] is a decentralized approach used for light-tailed coflow distribution, while Varys [14] is a centralized solution that outperforms Baraat by exploiting global knowledge. Aalo [15] is an improvement of Varys, which schedules coflows without any prior knowledge and by avoiding coordination for tiny coflows. This work is orthogonal to our SFS. We focus SFS on a single flow rather than a group of flows. SFS can be applied to coflow scheduling for reducing the completion time of coflows, which is left as the future work.

B. Motivation

This paper is motivated by pFabric, which is the state-of-the-art approach for minimizing FCT. In pFabric, a switch dequeues packets based on the priorities carried, rather than based on FIFO (First In First Out) implemented by commodity switches. However, deploying pFabric is costly or even infeasible due to non-trivial hardware modification in modern data centers with large amounts of switches. Therefore, it is critical to make easy implementations while achieving approximately same performance as pFabric.

There is an alternate way proposed in [1] to mimic pFabric by using CoS queues. This scheme works as follows. Flows that are smaller than a particular size are assigned to the highest priority queue, while flows that are larger than the above size but less than the second threshold are assigned to the second highest priority queue. However, this scheme may potentially induce unpredictable or even high FCT. This is because there are limited priority queues (typically 4 to 8 in commodity switches), so that flows of different sizes can be assigned to the same priority queue and contend for the shared bandwidth. We conducted an experiment on NS2 [18] to validate the drawbacks above of this scheme. In this experiment, we setup the Incast scenario, where a receiver requests a file of 100MB that is stripped across N senders each transmitting data of $100\text{MB}/N$ simultaneously. We add an appropriate background flow to compete with these flows, and assign 10Gbps to the bandwidth of each link. Figure 1 shows that the variant of pFabric can degrade the performance with limited CoS queues of 4 and 8 in switches. From Figure 1(a), we see that compared to the optimal pFabric, the variant increases the request completion time by up to 16ms. Figure 1(b) shows that unlike pFabric with one-by-one scheduling the variant does not schedule the flows so that all the flows are finished at the same time.

Other flow scheduling schemes also have various limitations and defects. They either cannot be easily implementable or suffer from poor performance in certain cases. For instance, D3 [19] and PDQ [5] introduce non-trivial modifications to switch hardware. PASE [6] requires a complex control plane for arbitration. Fastpass [8] suffers from the scalability

(a) Total request completion time

(b) Individual FCT

Fig. 1: An appropriate background flow in Incast scenario can degrade the performance of the practical variant of pFabric (4-8 CoS queues).

problem due to its centralized control architecture. PIAS [7] and the variant of pFabric discussed above are sensitive to the thresholds and the number of queues.

Our work is inspired by the fact that it is easy to implement a flow scheduling scheme in software on servers. In this paper, we propose a server-based flow scheduling scheme called SFS to achieve both two goals of easy implementations and minimal FCT. SFS uses two techniques of bidirectional flow scheduling and most recently seen flow coordination to minimize the FCT. Especially, SFS integrates traffic control into flow scheduling by utilizing two TCP mechanisms, which are introduced in the next subsection.

C. TCP Mechanisms Utilized by SFS

TCP handshake [20] is used to establish TCP connection between servers. The process works as follows: The sender sends a synchronous (SYN) packet to the receiver; the receiver replies with a synchronize-acknowledgement packet, which is the first ACK packet; the sender replies with an ACK packet and starts transmitting data packets. If the sender does not receive the first ACK packet, the TCP connection cannot be established and data packet will not be sent. SFS utilizes this TCP mechanism to block the first ACK packet for efficiently restricting the number of active TCP flows in Incast [16].

TCP self-clocking [21] is another mechanism that slides TCP congestion window and triggers continuous transmission.

The arrival of an ACK tells the sender that the network can accept another packet, and the sender sends a data packet in response. Through this mechanism, TCP can maintain continuous and stable transmission and quickly fill up the pipeline. Without sufficient ACK packets, however, self-clocking stops and causes transmission stop and timeout. Unlike CP [22] targeting maintaining self-clocking, SFS stalls self-clocking to efficiently pause low-priority flows. PAC [23] also stalls self-clocking for addressing the issue of Incast [16], which is not the primary target of SFS.

III. SERVER-BASED FLOW SCHEDULING

SFS is a combination of traffic control and flow scheduling. It controls the total amount of fighting packets to make the entire network zero-queue, hence flow scheduling can be moved from switches to servers. The reason that switches need to schedule flows is that with switch-based flow scheduling, small flows will not get queued up behind large flows and their FCT can be minimized. However, if we can keep network zero-queue, short flows can also experience very low latency during transmission and switches do not need to schedule flows. Such function can be moved to servers, where various scheduling policy can be easily implemented through software. In fact, for SFS, a zero-queue network is the sufficient and necessary conditions. On one hand, a zero-queue network is the reason that we can deploy flow scheduling on servers. On the other hand, a zero-queue network is the result of server-based flow scheduling, which controls the total amount of fighting packets to make in-switch queue very short.

With SFS, flows are bidirectionally scheduled by senders and receivers to minimize FCT. A sender always sends its highest-priority flow, and a receiver receives its highest-priority flow most of time. A flow can only be transmitted under permissions from both sides. All flows are completed one by one according to their priorities to minimize FCT. The architecture of SFS is shown in Figure 2. It contains a sender-side forward scheduler and a receiver-side backward scheduler. Packets generated from TCP/IP stack are classified and pushed into the scheduling modules inside the schedulers. These scheduling modules then prioritize packets and schedule flows by blocking the packets of low-priority flows and releasing the packets of the highest-priority flow. The MRS flow coordination technique contains a set of mechanisms implemented inside the schedulers. These mechanisms monitor the state of the scheduling modules and adjust scheduling, coordinating senders and receivers to address the priority disagreement problem.

In the following subsections, we first set some basic assumptions and describe the two scheduling modules. And then we discuss the priority disagreement problem and address it with the MRS flow coordination. Finally we explain SFS's benefits.

A. Basic Assumptions

Priority: We propose a prioritization scheme similar to pFabric [1]. At the sender-side, flows get their priorities from applications, and these priorities are carried by SYN packets

Fig. 2: The architecture of SFS. The receiver-side backward scheduler controls ACK packets to indirectly schedule received flows. The sender-side forward scheduler controls SYN packets and data packets to directly schedule sent flows. The MRS flow coordination technique monitors and adjusts flow scheduling to coordinate senders and receivers to address the priority disagreement problem.

and data packets. Different from pFabric, the receivers in SFS would copy the priorities from SYN packets and data packets to the corresponding ACK packets, which could be used for receiver-side flow scheduling. Like pFabric, we use remaining flow size as flow's priority and the scheduling policy is shortest-remaining-flow-first. A short flow has a small remaining flow size hence its priority is high, while a long flow has a big remaining flow size hence its priority is low. During transmission, the remaining flow size of a flow continuously decreases. So with time passing, a long flow can become a short flow with high priority.

Topology: We focus on FatTree [9] topology, as it is by far the most common topology in practice. The oversubscription ratio is 1:1. Such setting is reasonable because recent researches trend to provide full bisection bandwidth [10, 11].

Load-balancing: We use packet spraying [12], which has been used in [1] to balance load among multiple paths. Packet spraying evenly distributes packets with the same destination among all downstream paths. As a result, it achieves almost perfect load balance and excellent bandwidth utilization of multiple links in a DCN. Note that the non-oversubscribed network, the packet spraying routing and the window-based flow scheduling technique are three essential building blocks for a zero-queue network, in which congestion may happen only at the edge layer.

TCP: We use half-duplex TCP like [1]. The size of TCP congestion window is fixed to be equal to bandwidth delay products (BDP), which ensures that flow can achieve line rate. To mitigate the impact of packet reordering, we disable TCP congestion avoidance and fast recovery. The TCP timeout value is fixed to 500us, which is approximately 5 times inter-pod RTT (round trip times) in our experiment setup.

B. Bidirectional Flow Scheduling

Flow scheduling in SFS is bidirectional. At the receiver-side, flows are backward scheduled. Their ACK packets are released or paused to let their senders start or stop sending data packets. At the sender side, flows are forward scheduled. Their SYN packets and data packets are directly scheduled to be sent or paused.

Fig. 3: Example of flow scheduling at the receiver. The dotted line indicates that the blue flow is paused by the window-based flow scheduling module of server B for low priority.

1) Receiver-side Window-based Flow Scheduling:

This technique is the core of SFS. It directly determines which flow is running, which flow should be paused, hence the whole scheduling is driven by this technique. This technique also controls the total amount of fighting packets to make network zero-queue, which in turn becomes the footstone of SFS.

A server uses a window-based flow scheduling module to schedule all received flows. When a server acts as a receiver, it receives data packets from many senders and feeds back ACK packets to them. This module activates the highest-priority flow and pauses the other low-priority flows by starting or stopping feedback of their ACK packets. It's well known that TCP needs the first ACK packet to establish a connection [20], and it also needs persistent ACK packets to slide congestion window and trigger continuous transmission [21]. This module gives the green light to the ACK packets of the highest-priority flow while pausing the ACK packets of the other low-priority flows. As a result, the highest-priority flow monopolizes all bandwidth, and the other low-priority flows wait for their ACK packets to start transmission. Figure 3 illustrates an example. Both server A and server D try to send data packets to server B. As the priority of the green flow is higher, the window-based flow scheduling module of server B activates the green flow while pausing the blue flow. When the green flow completes, the window-based flow scheduling module of server B releases the ACK packets of the blue flow to activate it.

This module contains two components: a list of flows and a flow window. As Figure 4 shows, ACK packets from TCP/IP stack are pushed into the list, and then the NIC polls these ACK packets from the entries of this list. To restrict the number of active flows, the NIC can only poll the entries inside the flow window, hence the flows outside the flow window are paused because of TCP self-clocking [21] stop.

List of Flows: The receiver extracts priorities from ACK packets to create such list, in which flows are sorted in decreasing order of priority. The entry of this list contains information such as flow id, flow's temperature (described in

Fig. 4: Diagram of the window-based flow scheduling module in the receiver-side backward scheduler. ACK packets from TCP/IP stack are pushed into the flow list, in which flows are sorted in decreasing order of priority. The flow window restricts the number of active flows. The NIC can only polls ACK packets from the flows inside the window. The flows outside the window are paused because of self-clocking stop.

subsection C), current priority and a pointer¹ to the actual ACK packet. If a new flow comes, the receiver registers the new flow and inserts it into the list. If a flow completes, the receiver removes the flow from the list. As the priority of a flow continuously changes during transmission, the receiver also continuously updates the entries in this list. Luckily, such updating mostly happens at the head of the list, which is the entry of the highest-priority flow, so the overhead will not be excessive. In addition, the size of the list can also be set to a fixed value (e.g., 100 flows). With an appropriate TCP timeout value, the flows that fail to register because of low priority can retry later.

Flow Window: Inspired by the TCP congestion window, we design a flow window for the receiver to restrict the number of active flows. The flows inside the window are active, while the flows outside the window are paused. Like TCP slides congestion window to maintain continuous transmission, a receiver slides the flow window to complete flows one by one. The size of this flow window is a bit tricky. If the flows in the list are large, the flow window should be small enough to let the highest-priority flow monopolize all bandwidth. However, if the flows are small (e.g., just several packets), the flow window should be big enough to activate sufficient flows to saturate network. We use the total number of fighting packets to dynamically adjust the window size. As mentioned before, the congestion window of all TCP flows is fixed to be equal to BDP, so we can infer that the number of fighting packets caused by flow i ($FightingPackets_i$) is determined by $Min(BDP, Remainingflowsizesize_{flow\ i})$. We then can calculate the total number of fighting packets caused by all flows inside the flow window ($FightingPackets_{in\ window}$). We adjust the window size to keep $FightingPackets_{in\ window} \in [0.5BDP, 1.5BDP]$, hence the flow window is always maintained in a proper size. For example, in Figure 4, there are three flows and currently the size of the flow window is one. Assume that the BDP is equal to 8 packets of 1500B. If the receiver detects that the remaining size of the red flow is less than four packets, the flow window will increase to two to activate the green flow. When the red flow completes and is

¹As TCP uses cumulative acknowledgment, the receiver only needs to keep one ACK packet for every flow.

Fig. 5: Example of flow scheduling at the sender. The dotted line indicates that the blue flow is paused by the priority-based flow scheduling module of server D for low priority.

removed from the list, the number of fighting packets now becomes 16 (two times BDP caused by the green flow and the blue flow). So in order to keep the number of fighting packets in range of $[4, 12]$, the flow window will be decreased to one to let the green flow monopolize all bandwidth. Such procedure repeats and all flows in the flow list complete one by one.

2) Sender-side Priority-based Flow Scheduling:

A server uses a priority-based flow scheduling module to schedule all sent flows. Recent research [14] shows that the sender side can also be the bottleneck. For example, in Figure 5, server D simultaneously sends two flows to server B and server C. Although these two flows are the highest-priority flow at their receivers, none of them can reach line rate as they two share the same sender. With the help of a priority-based flow scheduling module, the green flow with medium priority then can monopolize all bandwidth, and the blue flow with low priority will be paused and wait for its turn. As Figure 6 shows, the priority-based flow scheduling module adopts a design similar to the window-based flow scheduling module. But there are two differences. First, the sender does not need a flow window to restrict the number of active flows. The receiver needs the flow window to control the number of fighting packets and avoid congestion if multiple senders simultaneously connect to it. While a sender cannot jam its own uplink not matter how many receivers it connects to. Second, every entry of the sender's flow list contains a FIFO queue to store pointers to the actual data packets. The receiver does not need such queue in its entry. As mentioned before in the footnote, TCP uses cumulative acknowledgment, hence for every entry, the receiver only needs to keep one pointer, which is the one pointing to the ACK packet with the highest sequence number. Other ACK packets can be directly dropped to save network bandwidth. The priority-based flow scheduling module works in the following way. SYN packets and data packets generated from TCP/IP stack are pushed into the flow list and stored in respective queues. The NIC then polls available packets from the queue of the highest-priority flow and outputs them. For example, in Figure 6, we can

Fig. 6: Diagram of the priority-based flow scheduling module in the sender-side forward scheduler. SYN packets and data packets from TCP/IP stack are pushed into the flow list, in which flows are sorted in decreasing order of priority. Each entry in this list contains a FIFO queue to store pointers to the actual packets. The NIC always polls available packets from the highest-priority flow. For example, the NIC first tries to poll the red flow. If the queue of the red flow is empty, the NIC then tries the green flow.

see that currently the NIC is polling the packets of the red flow. Normally, the ACK packets of the red flow will come back in time and make the TCP/IP stack generate new data packets for it. As long as the data packets of the red flow are persistently pushed into its queue, the packets of the other low-priority flows stay in their queues. However, sometimes the ACK packets may be delayed (e.g., encounter congestion or be paused by the receiver). In such case, the NIC will deplete all data packets of the red flow and then turn to the queue of the green flow. Once the red flow receives its ACK packets and generates new data packets, the NIC will turn back to its queue to poll packets.

One may notice that the NIC actually polls packets from these two modules: the window-based flow scheduling module in the receiver-side backward scheduler and the priority-based flow scheduling module in the sender-side forward scheduler. So which one should be polled first? We know that prioritizing ACK packets over data packets can accelerate TCP self-clocking and improve throughput. What's more, such setting also avoids issues about two-way TCP [24]. So the NIC first polls ACK packets from the window-based flow scheduling module. If the polling does not return a valid pointer, the NIC then turns to the priority-based flow scheduling module for SYN packets or data packets.

C. MRS Flow Coordination

The bidirectional flow scheduling technique is still insufficient. SFS has a specific problem called priority disagreement, which interrupts transmission and increases FCT. To address this problem, SFS coordinates senders and receivers to maintain continuous transmission.

1) Priority Disagreement Problem:

Flow scheduling at both sides using the same policy causes priority disagreement. A flow can be the highest-priority one at its receiver, while a low-priority one at its sender. In this case, the receiver is idling as its highest-priority flow is interrupted by the sender while the other low-priority ones have been paused by itself. Figure 7 illustrates an example. In Figure 7(a), we can see that the green flow is the highest-priority flow on server B, and the blue flow is paused for low priority. In Figure 7(b), server D starts to transmit a red flow, which has a higher priority than the green one. Now server B becomes

(a) The green flow with medium priority is the highest-priority flow at its receiver. The blue flow is paused by the window-based flow scheduling module of server B for low priority.

(b) Server D starts to transmit a new flow with high priority. The green flow becomes a low priority one at its sender and it can never start till the red flow completes. Server B is idling as it cannot receive any packets from neither the green flow nor the blue flow.

Fig. 7: Example of the priority disagreement

idling as it cannot receive any packet from neither the green flow nor the blue flow. In fact, the paused blue flow should seize the moment to transmit packets and reduce its FCT. The fundamental reason of priority disagreement is that SFS is a decentralized server-based scheme, which locally schedules flows without global information. It should be noted that priority disagreement only impacts the receiver side. As mentioned before, the sender-side priority-based flow scheduling module always outputs the available highest-priority flow. If the highest-priority flow of a sender is paused by the receiver, the TCP/IP stack stops producing data packets for this flow, hence the NIC automatically turns to the other flows for packets.

2) Solution:

Coordination between senders and receivers can address the priority disagreement problem and maintain continuous transmission. We borrow idea from temporal locality and design a technique called MRS flow coordination, which has the following three functions. First, when priority disagreement occurs, the sender notifies its receiver. For example, in Figure 7, when the red flow emerges, server D notifies server B that “you cannot receive the packets of the green flow from me”. Second, the receiver activates the MRS flow to saturate network. Concepts about temporal locality inspires us that among all paused flows, the MRS flow usually has the highest probability to build a data transmission with its sender. Activating it can quickly saturate network. Third, the receiver evolves the scheduling policy back to the shortest-remaining-flow-first for minimizing FCT. Activating the MRS flow actually adopts a scheduling policy called MRS-flow-first. Such policy is useful for temporarily saturating network but cannot minimize FCT. So if possible, the receiver should evolve the scheduling policy back to the shortest-remaining-flow-first for minimizing FCT. The MRS flow coordination technique contains a set of mechanisms implemented inside the sender-side forward scheduler and receiver-side backward scheduler. These mechanisms monitor the state of the flow scheduling modules and adjust their scheduling to realize the

above three functions.

Sender-side Notification: This mechanism is designed for a sender to inform its receiver that priority disagreement has occurred. To achieve such function, the sender should monitor the changes of currently outputting flow. For example, in Figure 7(b), we can see that the outputting flow of server D has changed from a medium-priority flow to a high-priority flow. Visually we can say that every time the outputting flow moves towards the left side of the sender’s flow list, priority disagreement occurs and the original outputting flow is interrupted. To notify the receiver, the sender sends one more data packet of the original outputting flow, and this packet will be marked as priority disagreement. For example, in Figure 7(b), when the data packets of the red flow emerges, server D will send one more data packet of the green flow to notify server B. It should be noted that only data packets can trigger sender-side notification, while SYN packets cannot do this. For example, when the SYN packet of the red flow emerges, server D just creates an entry for it in the list and outputs this SYN packet, and then continues outputting the data packets of the green flow. This is because currently server D does not know whether the red flow can complete the TCP handshake [20] and establish a connection. Maybe it can, or maybe it will be paused by its receiver. On the contrary, when TCP/IP stack starts generating data packets of the red flow, server D can confirm that the red flow has permission from its receiver, hence it is time to notify server B that “I cannot output the green flow for you now”. A problem with this mechanism is that an empty queue can make it invalid. For example, in Figure 7(a), on server D, assume that the queue of the green flow has poured out all data packets because its ACK packets do not come back in time. At this moment, the data packets of the red flow emerges (Figure 7(b)) and the sender is supposed to notify the receiver of the green flow. Unfortunately, the queue of the green flow is empty hence the sender cannot do this. To avoid such problem, the sender keeps one data packet of each flow staying in its queue unless it is the last data packet of this flow. Accordingly, we modify the basic assumption we

set before and now the TCP congestion window is set to be equal to BDP plus one.

Flow's Temperature: Like storage system uses temperature to recognize the most recently used (MRU) data [25], the temperature of flows helps a receiver² to recognize the MRS flow. A hot flow means that the receiver recently gets packets from this flow, which also indicates that this flow has enough priority at its sender side and can potentially build a data transmission. On the contrary, a cold flow means that the receiver has not got packets from it for a while, which also indicates that this flow does not have enough priority to be sent out, hence it has little chance to build a data transmission. The hottest flow is the MRS flow, which has the highest probability to build a data transmission. We set the upper bound of temperature to 7 degree and the lower bound to 0 degree. Every time a receiver gets a packet, it finds the corresponding flow in its list and refreshes its temperature to 7 degree. With time passing, the temperature of all flows decreases and eventually a flow can become frozen (0 degree). We use a timer to realize such function. Every 100us a receiver decreases the temperature of all flows in its flow list by one. For an activated flow (i.e., the flows inside the flow window), its temperature will be stabilized at 7 degree as its data packet continuously comes. For a paused flow (i.e., the flows outside the flow window), there are two cases. As we have fixed the TCP timeout value to 500us, every 500us a TCP timeout event occurs and a data packet is generated. If this flow has enough priority at its sender side, this data packet will be sent out. When it reaches its receiver, its temperature will be refreshed to 7 degree before decreasing to 0 degree (A flow of 7 degree needs at least 600us to be frozen). On the contrary, if this flow does not have enough priority at its sender side, it eventually becomes frozen and the receiver knows that it cannot build a data transmission. It is important to noted that receiving a packet marked as priority disagreement will instantly make the corresponding flow frozen.

Heartbeat Packet: We consider the data packet generated by TCP timeout event as heartbeat packet because actually it is used to notify a receiver that a paused flow is “alive” at its sender side. Such function is very similar to hadoop’s heartbeat packet which is used to tell master the state of worker. The difference is that the heartbeat packet generated by TCP timeout event carries payload and may consume considerable network bandwidth. To save bandwidth, the sender cuts down the payload of heartbeat packet. Similar design has been used in [1] to generate little packet to probe the state of network. Note that the heartbeat packet is still scheduled by sender and it will not trigger sender-side notification. For example, in Figure 6, we can see that there are three flows and currently the NIC is polling the packets of the red flow. Assume that the red flow does not have enough priority at its receiver side hence its ACK packets do not come back. After depleting all packets in the queue of red flow, the NIC starts to poll

the green flow. After 500us, the red flow experiences a TCP timeout event and generates a heartbeat packet. Assume that the red flow is still the highest-priority one at its sender side, the NIC polls this heartbeat packet and then continues polling the green flow. In this way, this heartbeat packet is sent out and reaches its receiver. It then refreshes the temperature of the red flow to 7 degree and the receiver knows that this paused flow is “alive” at its sender side and can be used to build a data transmission. To further save bandwidth, the sender only keeps one heartbeat packet for each flow in its flow list. For example, assume now the sender is stably transmitting the red flow and the packets of the other two flows are staying in their queues. After 1000us, the green flow experiences two TCP timeout events and generates two heartbeat packets with the same sequence number. The sender keeps the first heartbeat packet and drops the second one as its sequence number is not bigger than that of the first one. When the red flow completes or is paused by its receiver, the sender turns to the green flow, outputting the heartbeat packet and data packets.

Candidate-window Open: This mechanism is used to activate the hottest flow (the MRS flow) to saturate network in case of priority disagreement. As mentioned before, the NIC can only poll ACK packets from the flows inside the flow window. We modify the original setting and now the NIC can poll ACK packets from two places: the flows inside the flow window and the single flow inside the candidate-window. Normally, the candidate-window is close as the flows inside the flow window are enough to saturate network. However, when priority disagreement occurs, the flows inside the flow window may become frozen hence the number of fighting packets is not enough to saturate network. In this case, it is necessary to activate a paused flow outside the flow window. The opening of the candidate-window is triggered by the polling attempt of the NIC. Every time the NIC tries to poll ACK packet from a flow, the receiver checks whether the flow is frozen. If it finds a frozen flow inside the flow window and currently the candidate-window is closed, it will open the candidate-window to activate the hottest flow. Sometimes the activated flow inside the candidate-window may also encounter priority disagreement and become frozen. In that case, the polling attempt from this frozen flow will re-trigger candidate-window open mechanism to activate another paused flow. Figure 8(a)-(d) show an example of candidate-window open. And Figure 8(e)(f) show an example of re-triggering candidate-window open. Note that the network will be idling for about one RTT during the activation. For example, in Figure 8(c), the receiver has activated flow E and is waiting for the responded data packets. The network is idling until the data packets of flow E continuously arrivals as shown in Figure 8(d).

Candidate-window Move: This mechanism is used to move the candidate-window towards the left side of the flow list. The candidate-window open mechanism actually adopts a scheduling policy called MRS-flow-first. Such policy can quickly build a data transmission to saturate network, but cannot minimize FCT. So if possible, the receiver should evolve the

²Senders do not need to recognize MRS flow, so only receivers log flow’s temperature.

Fig. 8: Examples of flow's temperature, candidate-window open, candidate-window move and candidate-window close. The flows in the list are sorted in decreasing order of priority. The entry shows flow id and current temperature. The hottest flow (the MRS flow) is painted in red.

scheduling policy back to the shortest-remaining-flow-first. We know that the flow list are sorted in decreasing order of priority and flows at the left side have higher priority. So moving the candidate-window towards the left side means evolving the MRS-flow-first policy back to the shortest-remaining-flow-first policy. The candidate-window move mechanism works in the following way. When the candidate-window is open, if the receiver gets a packet belonging to a paused flow, it checks the position of this flow. If the position of this flow is at the left side of the candidate-window, the receiver moves the candidate-window to this flow. In this way, the original MRS flow is paused and a new MRS flow with higher priority is activated. Figure 8(g) shows an example of candidate-window move. We can see that a heartbeat packet of flow C refreshes its temperature to 7 degree. Although flow C and flow D have the same temperature, the priority of flow C is higher, so the candidate-window moves to flow C to activate it.

Candidate-window Close: This mechanism is used to stop the activated flow inside the candidate-window. It is triggered by receiving a packet belonging to the flows inside the flow window, which may indicate that priority disagreement has ended. Figure 8(h) shows an example of candidate-window close. We can see that receiving packets from flow A will instantly close candidate-window to stop flow C. A problem with this mechanism is that sometime there might be multiple flows inside the flow window and only one of them is frozen. In this case, the receiver needs to check whether the *FlightingPackets_{in window}* is enough to saturate network, but this time the frozen one will not contribute to the *FlightingPackets_{in window}*. If the *FlightingPackets_{in window}* is not less than 0.5BDP, the candidate-window will be close.

D. Benefits Discussion

Software Implementation: The possibility of a software-only deployment is the key advantage for SFS. Since it can

be implemented in software without modifications to network switches, it provides an attractive alternative to more idealized schemes like pFabric [1] when the network has full bisection bandwidth. Currently we implement SFS in NS2 [18]. A possible testbed implementation is to intercept traffic from/to the NIC driver and pass it to the schedulers. Such function is supported by plenty of works such as Netgraph (FreeBSD), Netfilter (Linux), and Ndis Miniport drivers (Windows), hence the implementation should not be difficult. Besides, the architecture of SFS is similar to ICTCP [26], which is an implemented scheme on testbed. So we think the testbed implementation of SFS is feasible.

Generic Framework of Flow Scheduling: Essentially, SFS is not only a FCT minimization approach but also a generic framework for many flow scheduling schemes. For example, we can use deadline as a flow's priority to implement the earliest-deadline-first policy in SFS. Likely, we can also use the number of sent bytes or the starting time of jobs as the priority to implement other scheduling policy in SFS, such as MLFQs used by PIAS [7], FIFO-LM used by Baraat [13] and D-CLAS used by Aalo [15].

Incast: Incast [16] is a crucial issues that severely decreases TCP throughput in DCN. In Incast scenario, multiple senders concurrently transmit packets to a single receiver through a switch. The aggregated packets then overwhelm the switch's buffer. So TCP timeout occurs and some flows experience long idle time, leading to low throughput. SFS scales to hundreds or even thousands of senders in Incast scenario. As flow scheduling is performed before TCP establishes connection, synchronous senders are serialized and they cannot pump data packets into the bottleneck simultaneously. Most of time only one flow is active and the other flows are waiting for their ACK packets to start transmission. Flows are completed one by one and no packet loss occurs. In fact, the number of senders is only restricted by SYN packets. When a mass of senders

simultaneously send their SYN packets to the receiver, these SYN packets form an Incast scenario. Lucky, SYN packet is very small (typically 40 bytes). A 225KB buffer can hold more than 5000 SYN packets.

Zero-queue: SFS achieves zero-queue in FatTree topology [9] with full bisection bandwidth, hence the short flows experience very low latency on their paths. Most of the time, a receiver only activates one flow, which turns the many-to-one traffic pattern into a sequence of one-to-one traffic pattern. So no congestion occurs on the edge layer. For the same reason, the total number of active flows is not more than the total number of servers. As the number of fighting packets caused by every flow is equal to BDP, the total rate of active TCP flows is not more than the total bandwidth provided. In addition, the traffic of all flows is evenly distributed among multiple paths by packet spraying, so no congestion occurs on core layer and aggregation layer too.

IV. EVALUATION

In this section we conduct experiments to evaluate SFS's performance using packet-level simulations in NS2 [18], where pFabric is evaluated [1]. Our evaluation consists of two parts. First, we evaluate SFS's performance in Incast scenario with many-to-one traffic. And then we evaluate its performance in a large FatTree topology with many-to-many traffic generated from realistic workloads.

We compare two schemes with SFS in the experiments: DCTCP and pFabric. We set the parameters of three schemes to achieve the optimal performance for fair evaluation. DCTCP is set as follows: The switch's ECN threshold for DCTCP is set to 15 packets. Like experiments setup in [1], we use a fixed small TCP timeout value (1000us) to make DCTCP robust to Incast. pFabric is set as follows: The customized TCP for pFabric starts to send data packet right after sending the SYN packet, so the flows less than one BDP can be finished within one RTT³. The TCP timeout value is fixed to 500us. SFS is set as follows: The TCP congestion window is fixed to BDP plus one (9 packets) and the timeout value is fixed to 500us to generate heartbeat packets.

A. Performance Evaluation in Incast Scenario

In this subsection, we evaluate SFS's performance in Incast scenario, which is a typical many-to-one traffic pattern in DCN. We show that SFS switches between flows seamlessly to complete them one by one. As a consequence, it outperforms DCTCP and pFabric in request completion time and individual FCT. We also show that SFS's window-based flow scheduling technique efficiently control the number of fighting packets hence SFS tolerates up to 2000 senders.

Seamless Flow Switching: We simultaneously generate 5 large flows of size 20MB to a single destination host. The

³On page 7 of pFabric paper, the author describes why pFabric can finish small flows within one RTT while PDQ [5] needs at least two RTT to finish a flow. Like PDQ, SFS also needs the first RTT for flow initiation (SYN/SYN-ACK exchange), which is one of the reasons that SFS cannot catch up with pFabric.

Fig. 9: (a) Per-flow throughput of Incast scenario with 5 senders. (b) Queue length of the bottleneck link during flow switching.

bandwidth of all links is 10Gbps. Figure 9(a) shows the individual throughput of the five flows. We can see that the flows are indeed completed one by one without gap, and the whole transmission completes after 80.02ms. This is only 20us larger than the best possible completion time of 80ms for a 100MB transfer at 10Gbps. So SFS does achieve seamless flow switching in Incast scenario. We inspect the flow switching from another perspective. We repeat this experiment with DCTCP and pFabric and this time we measure the queue length of the bottleneck link. Figure 9(b) shows the result. The curve of DCTCP oscillates around its ECN threshold (22.5KB). This is because DCTCP controls data transmission rate but does not schedule flows. The curve of pFabric stays close to max queue length (36KB), and the dramatical declines to the nadir indicates flow switching. The curve of SFS is very close to zero. This is because SFS performs flow scheduling before TCP connection establishment. Multiple senders are serialized and they cannot pump data packets to the receivers simultaneously.

Request Completion Time: We setup an Incast scenario following [1]. A receiver node requests a 100MB file that is striped across N sender nodes. The senders respond with 100MB/N of data simultaneously. The request completes when all the individual flows have finished. We add a large background flow to compete with workflows [13], which should degrade DCTCP's performance. We vary the number of senders

(a) Total request

(b) Average individual FCT

Fig. 10: Total request and individual flow completion times in Incast scenario.

to evaluate SFS's performance. The total request completion times are shown in Figure 10(a). As expected, without flow scheduling, DCTCP's workflows have to compete with the background flow, which results in a rather long request completion time. As we increase the number of senders, workflows grab more bandwidth hence the total request completion time decreases and gets close to the ideal request completion time of 80ms. pFabric shows an opposite trend. Its total request completion time increases as the number of senders increases, which is due to the overhead of serially scheduling flows [1]. The more senders, the more overhead. SFS achieves the best performance, the total request completion time is only 80.6ms when there are 200 senders. In Figure 10(b), we can see that SFS also achieves the best average individual FCT. pFabric's individual FCT is slightly more. DCTCP is the worst, as it does not schedule flows at all.

Scalability: We test SFS in Incast scenario with 2000 senders. The total request completion time increases to 85.9ms. We measure the queue length of the bottleneck link. The max queue length is 87KB, which is caused by data packets of the background flow plus a mass of SYN packets at the beginning of the Incast. The average queue length is only 0.058KB and no packet loss occurs. So SFS does scale to thousands of senders in Incast scenario.

B. Performance Evaluation in FatTree Topology

In this subsection we evaluate SFS's performance in a FatTree topology with many-to-many traffic generated from realistic workloads. We show that SFS efficiently schedules flows according to their priorities, hence it achieves comparable performance to that of pFabric. We show that SFS achieves zero-queue in the entire network, which supports our hypothesis that in such network, flow scheduling can be moved from switches to servers. We also show that the MRS flow coordination technique efficiently activates flow to saturate network and the heartbeat packets only consume tiny bandwidth. Finally, we discuss issues about oversubscribed network and the overhead of flow switching.

Topology: We use a 8-pod FatTree [9] topology with full-bisection bandwidth. Such topology has 128 servers and 16 core switches. Each switch has 8 ports. The bandwidth of all link is 1Gbps. The propagation delay of all link is 1us. For data packet, the inter-pod RTT is 86us and the intra-rack RTT is 28.6us. For SYN packet, the inter-pod RTT is 16us and the intra-rack RTT is 4.6us.

Workload: We obtained the web search workload and the data mining workload used in [1] from the author. The distribution of these workloads follow traffic patterns that have been observed in production data centers. In the web search workload, over 95% of all bytes are from the 30% of the flows that are 1-20MB. In the data mining workload, 95% of all bytes are from 3.6% flows that are larger than 35MB, while more than 80% of flows are less than 10KB. We use these workloads to determine the size of each flow generated between random pairs of servers. Flows arrive following a poisson process. We vary the flow arrival rate to obtain a desired level of load to thoroughly evaluate SFS's performance. The size of data packet is 1.5KB.

FCT: Figure 11 and Figure 12 show the FCT performance for different schemes in these workloads as we vary the load from 0.1 to 0.9. Note that each flow's completion time has been normalized to the best possible value (flow size/bandwidth). The results are in line with our expectations. SFS's performance is better than DCTCP and approaches that of pFabric.

On web search workload as shown in Figure 11, compared with DCTCP, SFS reduces up to 63% average FCT for all flows. The improvement in 99th percentile FCT for short flows is more salient. SFS can be 3X-4X faster than DCTCP in case of high load. As expected, SFS cannot outperform pFabric, which achieves 1.3x average FCT for all flows and 1.6x 99th percentile FCT for short flows smaller than SFS. The reason is two-fold. First, pFabric can complete flow smaller than one BDP (8 packets in our experiment setup) within one RTT [1], while SFS requires at least two RTT to complete such flow. As mentioned before, the customized TCP for pFabric starts to send data packet right after sending the SYN packet, so flows less than one BDP can be finished within one RTT. On the contrary, SFS always needs the first RTT for flow initiation (SYN/SYN-ACK exchange), then starts transmission at the second RTT. Second, SFS suffers from priority disagreement

Fig. 11: Web search workload in non-oversubscribed FatTree: Normalized FCT statistics across different flow sizes.

Fig. 12: Data mining workload in non-oversubscribed FatTree: Normalized FCT statistics across different flow sizes.

and the MRS-flow-first policy is not the optimal scheduling policy for minimizing FCT. pFabric always uses the scheduling policy of shortest-remaining-flow-first to minimize FCT. SFS also uses such policy most of the time. However, in case of priority disagreement, SFS temporarily uses the MRS-flow-first policy, and then evolves the scheduling policy back to the shortest-remaining-flow-first. The MRS-flow-first policy is efficient to saturate network but cannot minimize FCT. So it is reasonable that SFS cannot achieve the best performance as pFabric does. However, as we have stated, due to the non-trivial hardware modification, pFabric's excellent performance cannot be achieved easily in datacenters with a mass of commodity switches. On the contrary, SFS is readily deployable and its performance gain is more realistic. For the long flows, as expected, SFS is worse than DCTCP and pFabric. DCTCP never schedules flows and pauses the long flows, so it achieves the best performance in average FCT of long flows. pFabric schedules flows and frequently pauses the long flows, so it has larger FCT of long flows than DCTCP. SFS also pauses the long flows frequently, and the little idle time in MRS flow coordination further increases the FCT of the long flows. So SFS has larger average FCT, which closely approaches pFabric. The results sound a bit unsatisfied, but SFS generally achieves significant reductions in average FCT (see Figure 11(a)) as

well as maintains stable tail FCT for short flows across all loads (see Figure 11(b)).

On data mining workload as shown in Figure 12, all the schemes perform better because the congestion is less. Compared with DCTCP, SFS reduces up to 43% average FCT for all flows and up to 59% tail FCT for short flows. Also, pFabric achieves 1.4x average FCT for all flows and 1.7x tail FCT for short flows smaller than SFS. Comparing Figure 11(a) and Figure 12(a), we see that the gap between pFabric and SFS in Figure 12 is larger. This is because pFabric requires just one RTT to finish 80% flows with no more than seven packets in data mining workload, while SFS requires at least two RTTs to finish them. Note that due to the size of packet, the first RTT for SYN/SYN-ACK exchange is much smaller than the second RTT for data packet/ACK exchange, so the normalized FCT of SFS for small flows can still be less than two. For the long flows, pFabric and SFS perform better than DCTCP, which is surprisingly converse on web search workload (see Figure 11(c)). Compared with DCTCP, SFS reduces up to 46% FCT for long flows and closely approaches pFabric. The reason is that the distribution of flow size can significantly impact the performance of flow scheduling schemes. In web search workload, the flows that less than 10MB account for 65% traffic, which can significantly degrade the performance

(a) Max queue length of all links

(b) Average queue length of all links

Fig. 13: The queue length of all downstream links with SFS under 0.9 load.

of flow scheduling schemes on long flows. On the contrary, in data mining workload, the flows that less than 10MB only account for 2% traffic, which is hard to impact the performance of flow scheduling schemes on long flows.

Queue Length: To test whether SFS achieves zero-queue in FatTree. We repeat the above experiment with web search workload for SFS at extremely high load (0.9 load). We measure the queue length of all downstream links, which are the links from core layer to aggregation layer, the links from aggregation layer to edge layer and the links from edge layer to servers. In our experimental topology, there are in total 384 links and each layer has 128 links. Figure 13 shows the results. From Figure 13(a), we can see that at the core layer and the aggregation layer, the max queue length never exceeds 5KB, and congestion only happens at the last hop. This is because sometimes candidate-window moves several times in very short time and multiple paused flows are successively activated. Their fighting packets accumulate and form a small Incast at the last hop and temporarily increase the queue length. The performance in Figure 13(b) is more salient. The average queue length never exceeds 2.5KB. And the links in core layer and aggregation layer have a average queue length of only 0.5KB (less than one data packet). So SFS dose maintain zero-queue in the entire network for a long time even at extremely high load. Besides, we can also observe the power of packet spraying, which evenly distributes traffic

Fig. 14: Respond time of priority disagreement.

among multiple paths. We see that the blue curve and green curve is much flatter than the purple curve, and the max queue length of the core layer links are almost the same, and the links to the same rack have similar average queue length.

Efficiency of MRS Flow Coordination: When receiving a priority disagreement notification, we want to know how much time it takes to get responded data packet from the activated flow inside the candidate-window. We hope this time should be small as network is idling in this period. We measure the respond time of all priority disagreement events in the web search experiment under 0.9 load. Figure 14 shows the experiment result that the respond time follows a long tail distribution. We see that the curve start from 28.6us, which is the intra-rack RTT. About 73% priority disagreement events get responded data packet within one inter-pod RTT (86us), and 99% priority disagreement events get responded data packet within 370us. So the MRS flow coordination technique does efficiently activate flow to saturating network in case of priority disagreement. However, we also observe several events that have respond time that more than 20000us (no shown in Figure 14). We carefully examine these events and find that the reason is that all flows in the receiver's flow list are frozen hence activating any of them will not get response.

Ratio of Heartbeat Packet to Data Packet: This ratio should be small so that the heartbeat packets will not consume too much bandwidth. We count all bytes of the heartbeat packet and the data packet sent from every sender-side forward scheduler in the web search experiment under 0.9 load. The max ratio is 6‰ and the average ratio is only 3‰. So the heartbeat packets only consume tiny bandwidth.

Impact of Oversubscription: SFS builds on non-oversubscribed network and packet spraying to achieve zero-queue. In this case, congestion happens only at the last hop and can be easily addressed with flow scheduling at the same time. However, in case of oversubscribed network, SFS cannot keep zero-queue and its performance significantly degrades. To accommodate this case, we combine SFS with DCTCP. We repeat the web search experiment above in a FatTree topology with three times servers (that is the oversubscription of 3:1). Figure 15 shows the results. We note that x-axis varies from 3% to 33% in server load, which can induce 8% to

Fig. 15: Web search workload in 3:1 oversubscribed FatTree: Normalized FCT statistics across different flow sizes.

96% switch load at the edge layer. From Figure 15(b), we see that SFS cannot achieve low latency for short flows as expected, since congestion occurs at the edge layer or beyond. But the combination scheme of SFS and DCTCP still can enable low latency for short flows. Compared with DCTCP, the combination scheme reduces up to 33% average FCT for all flows and up to 36% tail FCT for short flows. For the long flows in Figure 15(c), we see that pFabric, SFS and the combination scheme almost have the same performance, which is different from Figure 11(c). This is because the built up queue reduces the idle time of network hence SFS achieves higher throughput for long flows.

Discussion on Overhead of Flow Switching: The nature of SFS is an end-to-end arbitration. Typically, an arbitration-based scheme suffers from high flow switching overhead, which refers to the overhead of pausing and unpausing flows. SFS has such limitation due to the following two aspects. First, when flows travel multiple hops, SFS cannot guarantee seamless flow switching. The first packet of the unpaused flow might encounter queue at any hop along its path hence cannot catch up with the last packet of the paused flow. Second, The MRS flow coordination induces the idle time (see Figure 14) that wastes network bandwidth. As indicated in prior work [1], when network load is extremely high, just a small amount of wasted bandwidth can cause a disproportionate slowdown for the long flows. To reduce the idle time in flow switching, a possible solution is to maintain a small amount of queues in network. The results in oversubscribed FatTree as seen in Figure 15 show that the queues are a contradiction that is a cure for stable throughput for long flows but the cause of large latency. Therefore, designing good flow scheduling must take into account the tradeoffs, which is left as our future work.

V. CONCLUSION

This paper presents SFS, a server-based flow scheduling scheme to minimize FCT. SFS combines ideas of traffic control and flow scheduling to achieve a zero-queue network, which in turn becomes the reason that we could move flow scheduling from switches to servers. Compared to switch-based flow scheduling scheme, SFS is easier to implemen-

t, while achieving comparable performance. We conducted experiments on NS2 to evaluate its performance. In Incast scenario, SFS outperforms DCTCP and pFabric. On real workloads in FatTree, SFS achieves 4X faster than DCTCP and closely approaches pFabric in the tail FCT of short flows. Therefore, SFS offers a tradeoff between performance and deployment, and can be an option for datacenters to improve performance.

ACKNOWLEDGMENT

This work was supported by the National Basic Research Program of China (973) under Grant 2012CB315805, the National Science Foundation of China under Grants 61173167, 61472130, 61472131 and 61272546 and the Prospective Research Project on Future Networks of Jiangsu Future Networks Innovation Institute under Grant BY2013095-1-05.

We thank Mohammad Alizadeh for providing simulation codes of DCTCP and pFabric. We also thank Hong Xu for providing the instructions about our experiments.

REFERENCES

- [1] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," in *Proc. of SIGCOMM*, 2013.
- [2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. of SIGCOMM*, 2010.
- [3] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proc. of NSDI*, 2012.
- [4] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," in *Proc. of SIGCOMM*, 2012.
- [5] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proc. of SIGCOMM*, 2012.

- [6] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: Synthesizing existing transport strategies for data center networks," in *Proc. of SIGCOMM*, 2014.
- [7] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proc. of NSDI*, 2015.
- [8] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized "zero-queue" datacenter network," in *Proc. of SIGCOMM*, 2014.
- [9] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. of SIGCOMM*, 2008.
- [10] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," in *Proc. of SIGCOMM*, 2009.
- [11] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A scalable fault-tolerant layer 2 data center network fabric," in *Proc. of SIGCOMM*, 2009.
- [12] A. Dixit, P. Prakash, Y. Hu, and R. Kompella, "On the impact of packet spraying in data center networks," in *Proc. of INFOCOM*, 2013.
- [13] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. of SIGCOMM*, 2014.
- [14] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proc. of SIGCOMM*, 2014.
- [15] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. of SIGCOMM*, 2015.
- [16] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *Proc. of SIGCOMM*, 2009.
- [17] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "pHost: Distributed near-optimal datacenter transport over commodity network fabric," in *Proc. of CoNEXT*, 2015.
- [18] "The network simulator ns-2," <http://www.isi.edu/nsnam/ns/>.
- [19] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron, "Better never than late: Meeting deadlines in datacenter networks," in *Proc. of SIGCOMM*, 2011.
- [20] J. Postel, "Transmission Control Protocol," Internet Engineering Task Force, RFC 0793, Sep. 1981.
- [21] V. Jacobson, "Congestion avoidance and control," in *Proc. of SIGCOMM*, 1988.
- [22] P. Cheng, F. Ren, R. Shu, and C. Lin, "Catch the whole lot in an action: Rapid precise packet loss notification in data center," in *Proc. of NSDI*, 2014.
- [23] W. Bai, K. Chen, H. Wu, W. Lan, and Y. Zhao, "PAC: Taming TCP Incast congestion using proactive ack control," in *Proc. of ICNP*, 2014.
- [24] M. Heusse, S. A. Merritt, T. X. Brown, and A. Duda, "Two-way TCP connections: Old problem, new insight," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 2, pp. 5–15, Apr. 2011.
- [25] J. W. Hsieh, T. W. Kuo, and L. P. Chang, "Efficient identification of hot data for flash memory storage systems," *Trans. Storage*, vol. 2, no. 1, pp. 22–40, Feb. 2006.
- [26] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data center networks," in *Proc. of CoNEXT*, 2010.