Skipping Congestion-links for Coflow Scheduling

Shuo Wang*, Jiao Zhang*†, Tao Huang*†, Tian Pan*, Jiang Liu*, Yunjie Liu*†, Jin Li‡ and Feng Li‡
{shuowang, jiaozhang, htao, pan, liujiang, yunjieliu}@bupt.edu.cn
{mark.lijin, frank.lifeng}@huawei.com
*State Key Laboratory of Networking and Switching Technology, BUPT, China
†Beijing Advanced Innovation Center for Future Internet Technology, Beijing, China
‡Network Technology Lab, Huawei, Nanjing, China

***Abstract*—Data transfer duration accounts for a great proportion of job completion time in big-data systems. To reduce the time spent on data transfer, some traffic scheduling mechanisms at coflow-level are proposed recently. Most of them abstract datacenter networks as an ideal non-blocking big-switch, and the bottleneck is located at egress or ingress ports of end-hosts instead of in networks. Thus, they mainly focus on how to allocate port capacities of end-hosts to jobs without considering in-network congestion. However, link congestion frequently occurs in datacenter networks due to network oversubscription and load imbalance. When link congestion occurs, bottleneck locations will move from the ports of end-hosts to network links.**

In this paper, we design and implement SkipL, a congestion-aware coflow scheduler which could detect congestion and schedules coflows at end-hosts to effectively reduce coflow completion time. In addition, to be easily deployed in cloud environments, SkipL does not require to control flow routes. SkipL prototype system is implemented in Linux. The results of experiments conducted in a real small testbed and simulations conducted in the flow-level simulator show that SkipL reduces the average Coflow Completion Time(CCT) compared to the per-flow fair sharing scheduling method and Varys.

***Index Terms*—Datacenter; Scheduling; Coflow Scheduling;**

I. Introduction

In the past few years, big data processing applications have changed traditional business boundaries and more and more customers are planning to use big data platforms. To allow customers to distribute and process their data on the cloud, large providers such as Google, Microsoft, Amazon, provide big-data frameworks (e.g., MapReduce [1], Dryad [2], and Spark [3]) in their datacenters. Big data analytic jobs generate massive data traffic among thousands of machines in datacenter networks. It is stated that the time spent on forwarding these massive data accounts for more than 50% of the jobs' completion time [4].

However, optimizing data transfer time of a job is not as easy as reducing the completion time of an individual flow. Each big-data job contains many tasks and each task generates a batch of synchronous flows. Until all the synchronous flows of a task are completed, the job can't finish data transfer and begin the next process. For example, in MapReduce, reducers will establish many flows to obtain data from different mappers [1]. The reducing process could not start until receiving all the data. A set of synchronous flows is termed as *coflow* [5].

Since existing load balancing mechanisms (e.g., [6]–[10]) mainly focus on balancing the load at flow-level instead of coflow-level. Recently, many coflow scheduling mechanisms have been proposed to reduce the data transfer time of a job. Current coflow scheduling mechanisms can be classified into two categories. The first type of work localizes the data of the job to reduce the total quantity of data that has to be sent across racks [11], [12]. Although data localization is a good way to reduce flow transfer time and bandwidth utilization, cross-rack data transfer is inevitable. Because most of the jobs need to run concurrently on thousands of machines, and those machines can hardly be placed into a single rack.

The second class of work tries to optimize data transfer phase by scheduling flows at coflow-level and minimizes Coflow Completion Time (CCT). However, few existing coflow scheduling mechanisms could timely handle congestion in networks (e.g., [13]–[18]). On the one hand, most of them abstract datacenter networks as a non-blocking switch and only consider scheduling coflows under this ideal network [13]–[15]. However, link congestions are high likely to occur in Clos topologies due to poor load balancing mechanisms(e.g., Equal-Cost Multi-Path Routing (ECMP)) [6]. Furthermore, datacenter networks are usually oversubscribed, and a large fraction of bandwidth is consumed by background data transfers [19]. On the other hand, although some coflow schedulers consider network topologies and link bandwidth, they could only reactively reschedule coflows when a new coflow arrive or complete [16]–[18]. However, bandwidth may change rapidly after a schedule due to the small interarrival time between two background flows [20]. In this situation, these schemes are generally unable to reschedule coflows based on changed bandwidth.

In this paper, we propose SkipL, a congestion-aware coflow scheduler that detects in-network congestion and schedules coflows based on the in-network congestion information. Firstly, we investigate that coflows contain a large number of concurrent flows that are usually balanced to different paths. Thus, the global congestion information of different paths can be collected by monitoring the rates of these flows. Accordingly, we design a simple in-network congestion measurement algorithm to detect congestion and estimate the available bandwidth of links at end-hosts. Based on the congestion and available bandwidth information, we propose a scheduling algorithm. It computes coflows' estimated completion time according to the measured congestion information. Based on the estimated coflow completion time, our algorithm schedules

978-1-5386-2704-4/17/$31.00

coflows according to Shortest Processing-Time-First (SPTF) [13] principle.

We have implemented SkipL and built a small-scale testbed with 8 hosts and 4 switches to evaluate its performance. Furthermore, we also implement a flow-level simulator to perform large-scale simulations. The testbed experiments show that SkipL could accurately detect congestion and reduce the CCT by 20% compared to Varys. The simulation results show that under topologies with 512 hosts in big-switch topology, SkipL reduces the average CCT up to 38% compared to the per-flow fair sharing mechanism and has 20% smaller CCT than Varys when coflow width is 32.

The main contributions of SkipL are:

1) We analyze and show that in-network congestion could greatly increase the completion time of coflows.
2) We propose a simple coflow scheduling mechanism to detect and handle in-network congestion at end hosts, SkipL, which doesn't need to control routing of flows and estimate the available bandwidth of links at end-hosts.
3) A prototype system and a large-scale flow-level simulator are implemented, and the performance of SkipL is extensively evaluated.

The remainder of the paper is organized as follows. Section II shows the background and summarizes the recent work on scheduling coflows. In Section III, the motivation of our work is shown through an example. In Section IV, the details of SkipL is described. In Sections V and VI, the performance of SkipL is evaluated. Finally, the paper is concluded in Section VII.

II. Background and Related Work

A. The Coflow Abstraction

Coflow is defined as a collection of flows that have the same performance metric, independent input and output [13], such as shuffle in MapReduce. The key properties for a coflow are defined as follows:

- *Coflow length:* the bytes of the largest flow of a coflow.
- *Coflow width:* the number of flows of a coflow;
- *Coflow size:* the sum of all bytes of flows of a coflow;

B. Related Work

How to schedule coflows and allocate bandwidth to each flow is an NP-hard problem. Most of the previous coflow scheduling mechanisms use the heuristic algorithm to schedule coflows. Their approaches are mainly based on two design principles: First In First Out (FIFO) and Shortest Processing-Time-First (SPTF) [13]. These two design principles and their related work are shown in Table I.

TABLE I: Summary of related work

Related work	Coflow Order	Handle Congestion	Prior Knowledge
Orchestra [4]	FIFO	No	Flow size
Baraat [15]	FIFO with multiplexing	No	NA
Varys [13]	SPTF	No	Flow size
Aalo [14]	FIFO with priority queue	No	NA
Rapier [16]	SPTF with optimization	No	Flow size Topology
SkipL	SPTF	Yes	Flow size Topology

FIFO: The FIFO based heuristic algorithms allocate bandwidth according to the arrival time of coflows and give more bandwidth to earlier arrived coflows such as Orchestra [4]. To avoid head-of-line blocking, Baraat [15] allows a coflow with higher priority to grab back bandwidth assigned to a lower priority coflow. To schedule coflow without requiring prior coflow information, Aalo [14] places coflows into strict priority queues and allocate bandwidth to coflows according to the priority.

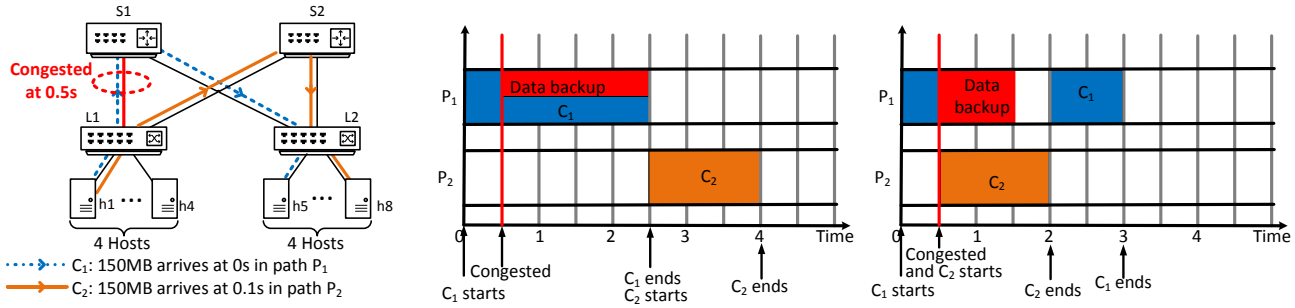
SPTF: The heart of SPTF based heuristic algorithms is to schedule coflows who have the smallest metric first. If two coflows have the same metric, it will schedule coflows according to their arrival time. The metric can be coflow length, width, size or CCT in different mechanisms. Based on this principle, Varys [13] abstracts the network as a non-blocking big-switch, estimates the minimal CCT of each coflow, and schedules coflows who has the smallest estimated CCT first. Instead of using the big-switch model, RAPIER [16] considers the real network topologies and assumes bottleneck also exists in the network. It jointly considers the scheduling and routing of all coflows and further reduces the CCT by solving an optimization problem with link bandwidth constraints. To reducing scheduling overhead, OPTAS [21] designs a distributed protocol to schedule tiny coflows without the centralized scheduler.

Besides, some previous coflow scheduling mechanisms ([17], [18]) try to model the coflow scheduling problem. CORA [17] formulate a completion-time optimal resource allocation problem and implement a framework to achieve max-min fairness among job utilities. Similar with CORA, [18] formulates the coflow scheduling problem, and tries to optimize coflow completion times with utility max-min fairness.

In sum, most of the coflow scheduling mechanisms introduced above only reschedule coflows when a coflow arrives or completes. This might cause that they are unable to reschedule coflows when network bandwidth changes or congestion occurs. In contrast, SkipL reactively detects the congestion and reschedule coflows when congestion occurs, a coflow arrives or completes.

III. Motivation

Before presenting our design, we use a simple example to show our key observations in Figure 1, where two coflows C_1 and C_2 arrive at 0s and 0.1s respectively, and each coflow has a 150MB flow. In Figure 1a, we assume that the link bandwidth is 100MBps, C_1 is routed through the path $P_1(h1 \rightarrow L1 \rightarrow S1 \rightarrow L2 \rightarrow h5)$, and C_2 is routed through the path $P_2(h1 \rightarrow L1 \rightarrow S2 \rightarrow L2 \rightarrow h8)$. Then, a 100MB data backup flow F_b collides C_1 at the red link ($L1 \rightarrow S1$) at 0.5s causing the



(a) Motivation topology (b) Congestion-agnostic scheduler (c) Congestion-aware scheduler

Fig. 1: Impact of the network congestion (a). A data backup flow collides the coflow C_1 at 0.5s causing congestion on path P_1 (b). Schedule Coflows according their arrive time without handling congestion (c). Reschedule coflows when detecting congestion.

congestion. Ideally, we assume C_1 and F_b share the bandwidth of P_1 equally when they collide. Note that C_1 and C_2 may contain other flows in realistic environments. For simplicity, we omit these flows and only use the two flows to represent the two coflows. Because most of the flows of a coflow may be small or not be congested, and the CCT isn't directly affected by their completion times.

The first type of coflow scheduler is unaware of the link bandwidth usage and link congestion, and it just reschedules coflows when a coflow arrives or finishes (e.g. Varys [13]). As shown in Figure 1b, because C_1 arrives first and has the same size as C_2, C_1 will be served first. The congestion-agnostic coflow scheduler gives all the capacity of $h1$ to C_1. Note that the port of $h1$ is the bottleneck of C_1 and C_2. Thus, C_2 has to wait until the completion of C_1. At 0.5s, when the data backup flow starts on P_1, the bottleneck will move from the host $h1$ to the network causing the rate of C_1 reduces to 50MBps. Since the congestion-agnostic coflow scheduler is unaware of the congestion, C_1 will still send data on the congested path and has a large CCT. As a result, C_2 has to wait a long time before starting, and its CCT also increases. *The average CCT is* $\frac{2.5+4}{2} = 3.25s$ *by using the congestion-agnostic scheduler.*

The optimal coflow scheduler is able to *detect link congestion and timely reschedule coflows when congestion seriously affects the CCT of a coflow.* As shown in Figure 1c, similar with the congestion-agnostic scheduler, the congestion-aware scheduler will serve C_1 first. But when the data backup flow arrives causing the rate reduction of C_1, the congestion-aware scheduler could detect the rate variation and reschedule all the coflows. Then, C_1 is paused, and the port capacity of $h1$ is reallocated to C_2. As a result, when C_2 finishes, the congestion also disappears. C_1 can use all the bandwidth in P_1 to continue its transfer. *The average CCT is* $\frac{2+3}{2} = 2.5s$ *saving* 23% *average CCT compared to the congestion-agnostic scheduler.*

More importantly, we think this example is not a corner case. In multi-tenant environments such as public clouds [22], [23], big-data applications usually share the network with other applications such as cloud storage [24], database, online video. These applications could generate a large number of background flows competing for the bandwidth with coflows. Although data center networks provide multiple paths to increase bandwidth, popular load balancing mechanisms based on flow hashing, e.g., ECMP [6], can easily cause the background flows collide with coflows due to harsh collisions or asymmetric topologies. Furthermore, by analyzing realistic Facebook workloads [25], we find that 50% of coflows have more than 10 flows and 10% of coflows have more than 1000 flows, Thus, the large width of coflows also increases the possibility of collision. On the other hand, the background flows generated by these applications are usually more than tens of megabytes. Once the congestion occurring, some coflows will be congested for a very long time decreasing the whole performance.

The conclusion is that *in-network congestion could negatively reduce the performance of the coflow schedulers, and the coflow schedulers should timely detect the congestion and reschedule coflows.*

IV. Design

SkipL is a centralized coflow scheduling mechanism to optimize the average CCT of coflows when large background flows cause link congestion in networks. In this section, we present the architecture of SkipL. Next, we show the scheduling framework of SkipL. After that, we discuss how to detect the congestion and measure the available bandwidth of links.

A. Problem Statement

When large background flows arrive and cause severe congestion, SkipL must timely detect the congestion and measure the available bandwidth. It must decide which coflows to pause and then preempts existing coflows to minimize the average CCT. In addition, the measurement algorithm must run in real-time with high accuracy and low complexity.

When preempting coflows, Varys doesn't control any switches either the route of any flow. The reason is that in the most of multi-tenant datacenters, users only have authorities to control their own machines or virtual machines. The network

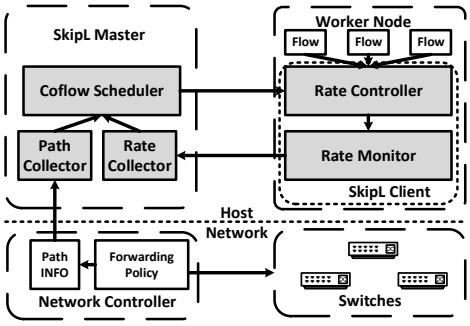


Fig. 2: Architecture of the SkipL.

and the routing of flows are controlled by datacenter administrators. Furthermore, coflows generally contain hundreds of flows, and coflow schedulers must take great care to avoid flow collisions during rerouting the flows. Some schemes like [26], [27] can avoid the collision during the flow migration, but they take a long time to finish. SkipL is able to minimize the average CCT without rerouting any flows.

B. Architecture

SkipL has a centralized SkipL master and serval SkipL clients running on the worker nodes of big-data systems shown in Figure 2. The SkipL master cooperates with the big-data system. As prior works [13], [15]–[18], the information about a coflow can be readily derived from big-date applications. Thus, SkipL master leverages the collected information to decide when to start the flows of coflows and at what rate to serve them. Then the scheduling decisions are enforced through distributed SkipL clients.

SkipL clients running on each work node receive scheduling decisions from the master node, and they mainly have two responsibilities. First, They need to make sure that each flow sends data at their desired rates assigned by scheduling decisions. For example, when the desired rate of a flow is zero, the SkipL client should pause the data transfer of the flow until the desired rate is non-zero. Second, They need to periodically report the remaining byte size and the throughput of each flow to SkipL master.

C. Scheduling Algorithm

SkipL uses algorithm shown in Algorithm 1 to reschedule coflows when a coflow arrives, finishes or experiences congestion. Inspired by [13], [16], SkipL schedules coflows following the SPTF principle. More specifically, it first estimates each coflow's CCT (shown in IV-D) based on available bandwidth information (Line $3 \sim 4$). Then all the coflows are sorted according to their estimated CCT (Line 6). Next, coflow with the smallest estimated CCT is served first (Line 8), and this coflow is temporarily removed from the sorted coflow set (Line 10). These processes are repeated again and again until there are no coflows in the coflow set. Finally, for work-conservation purpose, the remaining bandwidth is distributed to coflows (Line 11) according to their arrival time.

There are two important algorithms in SkipL for handling congestion. The Algorithm 2 is used to detect congestion and get congestion level at end-hosts. When congestion is detected by the Algorithm 2, SkipL will call Algorithm 1 to handle congestion. The Algorithm 3 is used to estimate CCT. In our algorithm, coflows that have smaller estimated CCT will be allocated with more bandwidth since a small estimated CCT indicates a coflow is less affected by the congestion.

Algorithm 1 SkipL

1: **function** RESCHEDULECOFLOWS(Coflows {C})
2: $\{C\}$ ▷ all the coflows
3: **for** $C \in \{C\}$ **do**
4: ESTIMATECCT(C)
5: **end for**
6: $\{C^*\} \leftarrow$ sort $\{C\}$ according T_C
7: **for** $C \in \{C^*\}$ **do**
8: ALLOCATEBANDWIDTH(C)
9: Remove C from $\{C^*\}$
10: **end for**
11: CONSERVATION($\{C\}$)
12: **end function**
13: **function** ALLOCATEBANDWIDTH(Coflow C)
14: **B** $\leftarrow$ Available port capacities of each host
15: **for** subflow $f \in C$ **do**
16: $\tau \leftarrow$ ESTIMATEDCCT(c)
17: $f.r = \frac{f.remainingSize}{\tau}$
18: **end for**
19: **for** subflow $f \in C$ **do**
20: $B_f^{source} -= f.r$
21: $B_f^{destination} -= f.r$
22: $A_l += f.r, l \in path_f^c$
23: **end for**
24: **end function**
25: **function** CONSERVATION(Coflows C)
26: **B** $\leftarrow$ Remaining port capacities of each host
27: **for** subflow $f \in C$ **do**
28: $minRate = Min(B_f^{source}, B_f^{destination})$
29: $minRate = Min(minRate, B_l), l \in path_f^c$
30: $f.r += minRate$
31: $B_f^{source} -= minRate$
32: $B_f^{destination} -= minRate$
33: **end for**
34: **end function**

D. Estimating Congestion

Detecting congestion is one core part of SkipL. In order to timely reschedule coflows and handle congestions, we design and implement a host-based congestion measurement algorithm. When designing this algorithm, we need to answer the following sub-problems.

How to detect congestion? Transport protocols generally detect congestion by monitoring the in-network delays. However, in-network delays are very bursty in microseconds and very challenging to measure [28]. Therefore, instead of using

Algorithm 2 EstimateAvailableBandwidth

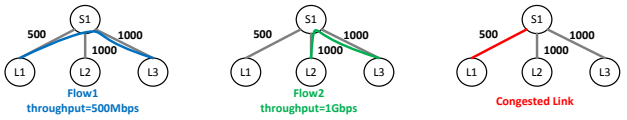
```
1: function EstimateAvailableBandwidth(Link l)
2:     r_l ← according to Equation 1
3:     r̃_l ←according to Equation 2
4:     l.lastTime                ▷ maintain recent congestion time
5:     if r_l − r̃_l > T then
6:         l.lastTime = TimeNow()
7:         l.B = r̃_l
8:         return
9:     end if
10:    if TimeNow() − l.lastTime > silenceTime then
11:        l.lastTime = −1
12:        l.B = Link Capacity
13:        return
14:    end if
15:    if r_l − r̃_l ≤ T then
16:        if l.lastTime ≥ −1 and r_l < l.B then
17:            return
18:        end if
19:        l.lastTime = −1
20:        l.B = Link Capacity
21:        return
22:    end if
23: end function
```

delays, SkipL uses flow rate to detect congestion. Intuitively, if big congestion occurs in a link, it will reduce all flows' throughput in that link. Assume the scheduler decides a flow to send data at a desired rate r, but the actual throughput of the flow is $\widetilde{r}$. SkipL master compares each flow's throughput $\widetilde{r}$ and desired rate r, and if $r > \widetilde{r}$, SkipL master can determine that the flow is congested.

For example, in the above motivation example (Figure 1), if there is no congestion in the path $P1$, C_1's desired rate is 100MBps. It can also get 100MBps bandwidth in P_1, thus we can assume that there is no congestion in link P_1. If congestion occurs in P_1 at 0.5s, although C_1's desired rate is 100MBps, it can only send data at 50MBps due to equally sharing bandwidth with the data backup flow. Thus, we can see that the throughput of flows is directly related to whether congestion occurs in networks.

However, the throughput of flows is not always stable in practice. We find that the sending rate of a flow needs a short time to converge to its equal sharing bandwidth due to the TCP window evolution. Especially, when we update the flow rates, the throughput of flows will have a large variation. To obtain the converged throughput, SkipL compares the throughput of time t and $t-1$. If $\widetilde{r}(t) - \widetilde{r}(t-1) < \delta$, we assume the throughput is stable and we update the throughput $\widetilde{r} = \widetilde{r}(t)$.

How to quantify congestion? A flow's throughput may only reflect whether congestions occur in links. It is challenging to directly use the throughput to guide the scheduling of coflows. Therefore, SkipL uses the available bandwidth of links to quantify the level of congestions, and use available bandwidth to guide the scheduling of coflows. Note that, we define the available bandwidth of a link as link capacity subtracts the rates of background flows, and it stands for the maximum unused bandwidth that a coflow can get in the link.



(a) 1st update (b) 2nd update (c) Final result

Fig. 3: Throughput of flows at two successive updates, and measuring the available bandwidth based on the two updates.

However, how to correctly estimate the available bandwidth is a big challenge in datacenter networks. Previous work PAPIER [16] estimates available bandwidth using Software Defined Network (SDN) technologies [29]. Generally, the controller in SDN firstly gets the throughput of all flows by periodically querying the flow entries from switches, and then calculates the available bandwidth. However, this mechanism may induce large traffic to the SDN control plan and increase the overhead of the controller [30]. Therefore, SkipL avoids directly querying available bandwidth information from the centralized controller. SkipL only uses the centralized control to get the paths of flows.

Estimating available bandwidth: Before proposing our estimation algorithm, we use Figure 3 to show our main idea and considerations. A flow path usually contains multiple links in datacenter networks, and the throughput of a flow is only determined by the bottleneck link in the path. Therefore, if a flow is congested, we can hardly determine which link is the bottleneck link in the path. For example, in Figure 3a, at the first update, the path of $Flow1$ contains two links, and the throughput of $Flow1$ is 500Mbps. We don't know whether the first link or the second link in the path is the bottleneck. To solve this problem, SkipL will update flow rates and collects throughput information several times, and then estimate the available bandwidth by analyzing all the collected information. For instance, since SkipL can not figure out which link of $Flow1$ is the bottleneck, it temporarily assumes both links are congested, and each link has 500Mbps maximum available bandwidth. Then, the SkipL will pause $Flow1$ and start $Flow2$ since $Flow2$ only has one potential congested link. In the next update shown in Figure 3b, if the throughput of $Flow2$ is 1Gbps, SkipL could know that there is no congestion on links of $Flow2$, and link $L1 \rightarrow S1$ is the real bottleneck as shown in Figure 3c.

Formally, we assume for a coflow j denoted as $coflow_j$, it has a set of flows $\{f^j_{sd}\}$ from source s to destination d. The path of flow j is $path^j_{sd}$. The remaining bytes size, throughput, desired rate for a flow f^j_{sd} is denoted as $data^j_{sd}$, $\widetilde{r}^j_{sd}$ and r^j_{sd} respectively. Note that, r^j_{sd} is allocated by SkipL through Algorithm 1, and it represents the maximal rate that f^j_{sd} can reach. $\widetilde{r}^j_{sd}$ is the current flow throughput measured by SkipL at end-hosts. Then we define the designed rate r_l and the throughput $\widetilde{r}_l$ for a link l as follows:

$$r_l = \sum_{k:f_{sd}^j \in l} r_{sd}^k \quad (1)$$

$$\widetilde{r}_l = \sum_{k:f_{sd}^j \in l} \widetilde{r}_{sd}^k \quad (2)$$

The estimate algorithm is shown in Algorithm 2. This algorithm estimates the available bandwidth B_l of link l according to r_l and $\widetilde{r}_l$. Initially, B_l is the capacity of the link. The available bandwidth B_l is updated in the following cases:

- Case 1 (Line 5): If $r_l - \widetilde{r}_l > T$, it is high likely that congestion occurs in the link, and the maximum available bandwidth is the link throughput.
- Case 2 (Line 10): If $r_l - \widetilde{r}_l \leq T$ for $slienceTime$, we assume the congestion disappears.
- Case 3 (Line 16): If congestion had occurred in the link and the new updated throughput is small than the latest estimated available bandwidth, we assume the congestion does not disappear. In contrast, if new updated throughput is large than the latest estimated available bandwidth, we can assume the congestion disappears.

E. Handle congestion

Algorithm 3 EstimateCCT

```
1: function ESTIMATECCT(Coflow coflow_j)
2:     {B_l} ← link available bandwidth in current time
3:     {A_l} ← link allocated bandwidth in current time
4:     for f_sd^j ∈ coflow_j do
5:         τ = data_sd^j / min{B_l − A_l}, l ∈ path_sd^j
6:     end for
7:     RETURN(τ)
8: end function
```

To effectively handle congestion, SkipL estimates the CCT of each coflows to determine the arrangements of them. The calculation method of estimated CCT is similar with MADD [13] algorithm in Varys, but we add link constraints. In Algorithm 3, if some flows of a coflow are forwarded in a congestion link with less available bandwidth, the estimated CCT of the coflow will be increased. Thus, all the coflows that use the congestion link are punished and their allocated bandwidth are decreased.

F. Implementation

The SkipL prototype implements the master-client architecture shown in Figure 2 in about 3700 lines of C++ code. The coflow schedulers run in the master with each flow rate information reported by the clients. The clients communicate with the master and enforce the scheduling decisions of the master. For balancing flows, we leverage SDN and implement a simple ECMP application in ONOS controller [31].

SkipL clients have two main functions: reporting flow rates to SkipL master and limiting flow rates according to scheduling decisions. The two functions are implemented in the user space. More specifically, the client writes 64KB data each time via $write()$ system call and records the execution time t. Then, the send thread of client sleeps $rates/64KB - t$ for rate limiting. We also set the send socket buffer size of TCP to 128KB to reduce the buffered data at hosts. SkipL clients resynchronize every Δ milliseconds. Each client will report the remaining flow size of each flow and flow rates during the period to SkipL master.

After receiving all updated information, SkipL master detects link congestions based on flow rates and flow paths information. Since ONOS controller doesn't support load balancing, we implement a simple ECMP application that hashes flows based on TCP destination port and provides REST API for flow path querying. Thus, SkipL master can query each flow's path from the controller via $libcurl$ [32].

V. Experimental Evaluation

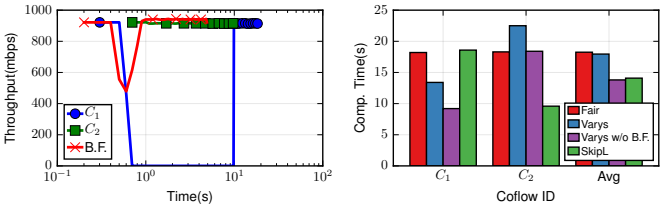
A. Evaluation Settings

Scheme compared: We evaluate the following schemes:

- **Fair:** In this mechanism, coflows are scheduled at flow-level. The bandwidth of a link is shared equally by all the flows sent on it.
- **Varys:** We implement Varys and compare our proposed mechanism with it. Varys only reschedules all coflows when a coflow arrives or finishes, and schedules the coflow with minimal CCT first.
- **SkipL:** SkipL is implemented according to the design described above. Besides scheduling coflows when a coflow arrives or finishes, SkipL reschedules all coflows when a coflow experiences severe congestion in links.
- **without Background Flow (w/o B.F.):** In this setting, we evaluate SkipL or Vayrs using the same workloads but don't generate any background flows in the network. Through comparison previous schemes with this setting, we can inspect how congestion affects the performance of coflow schedulers and better understand the performance improvement brought by SkipL.

Metric: We choose average coflow completion time among all coflows as our main performance metric. The average CCT is the division results of the summation of all CCTs and the total number of coflows. To clearly show the improvements of different mechanisms, the average CCT is normalized to Fair mechanism.

Testbed: We built a small testbed that consists of 8 hosts, 2 leaf switches, 2 spine switches as shown in Figure 1a. Each host is a Dell OPTIPLEX with a 2-core Intel I3-3220 3.3GHz CPU, 4G memory, and a Broadcom BCM5719 NIC. We use the OpenvSwitch 2.5 software switch that supports OpenFlow 1.3 as our switch [33]. OpenvSwitch software runs in DELL R720 with two 6-core Intel E5-2609 1.9GHz CPUs, 64G memory, and 8 Broadcom BCM5719 NICs. The default OS is Ubuntu 16.04 64 bit version with Linux 4.4.0-21 kernel. The base round-trip-time in our testbed is around 700μs, and we set the coordination interval Δ to 100ms as Varys does.



(a) Throughput of SkipL (b) Completion time

Fig. 4: [Testbed] Function verification.

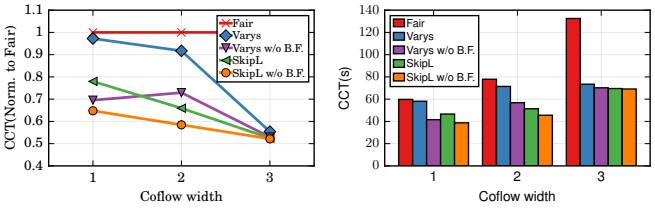


Fig. 5: [Testbed] Impacts of coflow width.

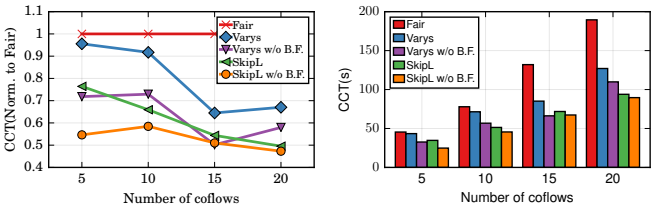


Fig. 6: [Testbed] Impacts of coflow number.

B. Experimental Results

Firstly, we show SkipL can timely detect in-network congestion and correctly reschedule coflows. Then the performance of SkipL is compared with Fair and Varys in our testbed with different workloads. At last, we show that the measurement algorithm of SkipL can monitor flow rate variations at microseconds level.

Function verification: In this experiment, we use the leaf-spine topology shown in Figure 1a and inject two coflows and a background flows into the network to show SkipL can timely detect in-network congestions. The first coflow C_1 sends 1GB data from $h1$ to $h5$ at 100ms, the second coflow C_2 sends 1GB data from $h1$ to $h8$ at 200ms, and the background flow $B.F.$ sends 500MB data from $h2$ to $h6$ at 200ms. By analyzing the experimental results, we find that the throughput of C_1 reduces to 500Mbps at 600ms. Figure 4a depicts the throughput of coflows, showing that SkipL detects the congestion at 600ms and reallocates bandwidth to C_2 to handle congestion. Figure 4b shows the completion times of coflows. We can see that the completion time of C_1 increases about 4s in Varys compared to Varys w/o B.F. due to the background flow. From this experiment, we can see that SkipL can save $\frac{17.9-14.1}{17.9} = 21.2\%$ of the average CCT compared to Varys.

Impact of coflow width: To evaluate how the performance of SkipL is influenced by the coflow width. We send 10 coflows with the same width and injects 4 cross-rack background flows arriving at 8s interval. The data size of all the flows is 1GB. We let each coflow only generate one cross-rack flow. This is because that our topology only has two paths, and the number of cross-rack flows usually less than the number of paths in realistic environments.

As shown in Figure 5, the average CCT is increased with the coflow width, and SkipL always has the best performance compared to Fair and Varys. We also observe a trend that the performance difference of schemes and schemes without background flows becomes small with the coflow width. The reason is that when the coflow width is relatively small, the bottleneck is in cross-rack links. When the coflow width becomes large, more flows compete for the capacity of hosts within the same rack, and the bottleneck is not in the cross-rack links. As a result, the cross-rack background flows have less impact on the performance.

Impact of coflow number: We fix the coflow width to 2 and evaluate how coflow number influences the performance. There are also 4 background flows with 1GB data arriving at 8s interval, and each coflow only has one cross-rack flow.

From Figure 6, we can see that the average CCT is increased with the coflow number, and SkipL improves the performance by up to 30% compared to Varys. Secondly, the performance difference between Varys and Varys w/o B.F. is stable. Note that since Varys schedules coflows when coflows arrive or finish, Varys has more chances to reschedule coflows when coflow number increases. However, the stable performance difference between Varys and Varys w/o B.F. indicates that increases the number of scheduling times cannot help Varys improve the performance when congestion information is unknown. Contrarily, we can see the performance difference between SkipL and SkipL w/o B.F. decreases with coflow number. This is because that when the coflow number is large, it is possible that there exist coflows whose paths are different with the paths of congested coflows. Thus, SkipL can find these uncongested coflows and gives them more bandwidth to avoid blocking by congestion links.

Accuracy of measurement: To show SkipL can accurately measure the rates of background flows by analyzing flow rates. In this experiment, we generate two flows starting at 0s and two background flows starting at 1s and 5s respectively. All the flows share the same bottleneck link. We monitor the packet counter of the queue at each host and use the port rate of host as the ground truth rates of background flows. All the rates are updated at 100ms interval.

Figure 7 depicts the measured and the ground truth throughput of background flows. Note that the throughput of background flows is the sum of all the rate of the background flows. Since all the flows share the same bottleneck link, after the first background flow starts at 1s, the throughput of $f0$ and $f1$ decreases to about 314Mbps, and after the first background flow starts at 5s, the throughput of $f0$ and $f1$ decreases to about 237Mbps. We can see SkipL accurately measured the throughput variation. Furthermore, the curve of measured

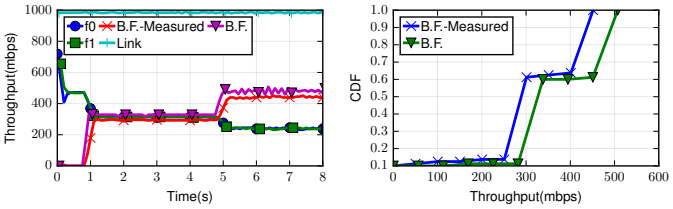


Fig. 7: [Testbed] Accuracy of measurement.

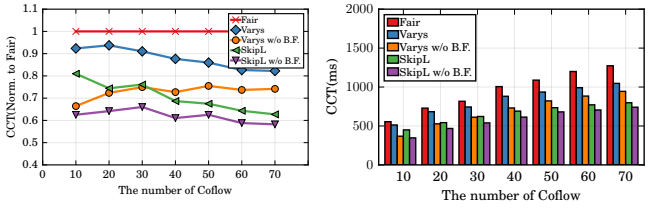


Fig. 9: The impact of coflow number in big-switch topology.

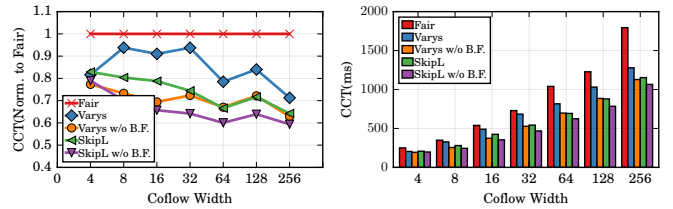


Fig. 8: The impact of coflow width in big-switch topology.

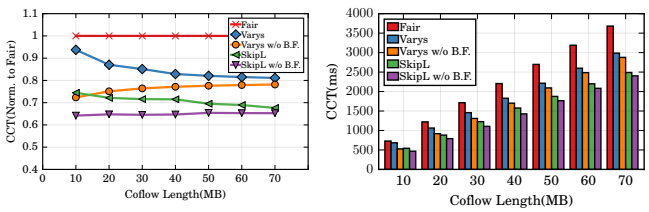


Fig. 10: The impact of coflow length in big-switch topology.

throughput of background flows has a stable deviation from the ground truth. This deviation is the system error. The reason is that we assume the link bandwidth is 1000Mbps and use the link bandwidth subtracting the throughput of all the flows as the measured background throughput. Actually, the link bandwidth is up to 950Mbps. Therefore, SkipL has a stable measurement deviation.

VI. Simulations

We implemented a flow-level simulator to evaluate the performance of SkipL. Similar with [13], our simulator only accounts for flow arrivals and departures. We use two topologies in our simulation. Firstly, we choose the big-switch topology used by Varys and show how host-side congestion influences the performance. Then we evaluate all the mechanisms in spine-leaf topology to show how in-network congestion impacts the performance.

A. Evaluation Results of Big Switch Topology

For simplicity, we assume that there are different links between any two pair of hosts to simulate the action of a big-switch. If congestion occurs in one link, it only affects the two end-hosts of the link.

Impact of coflow width: To evaluate the performance of SkipL under different coflow width in large topologies, we randomly generate 20 coflows between 512 hosts, and each flow's size is randomly selected from 10MB to 20MB. To induce congestion in links, 200ms congestion is generated at the beginning in all the links that are used by more than two different coflows.

Figure 8 shows the average CCT of different mechanisms. The x-axis represents varying coflow width from 4 to 256. The y-axis is the average CCT normalized to the result of Fair.

We can see that SkipL exhibits smaller average CCT than Fair and Varys. When the coflow width is small, the performance gap between SkipL and Varys is small. However, as the coflow width increasing, the performance gap between SkipL and Varys w/o B.F. increases at first and then decreases. This is because that when a coflow width is small, the congestion affects only a small group of coflows. Thus, it greatly increases the CCT of some coflows, and adjusting coflow order is effective to avoid congestion. However, when a coflow width is too large, the congestion affects almost all of the coflows, thus handling congestion has fewer impacts on the CCT. When the coflow width is 32, the performance gap between SkipL and Varys reaches 20%. When the coflow width is 256, SkipL performs the best and only has about 65% of CCT of Fair. From this, we can see that SkipL successfully handles most of the congestion.

Impact of coflow number: To evaluate the impact of coflow number, we fix the coflow width to 32 and randomly generate coflows between 512 hosts by varying coflow number from 10 to 70. Flow size is randomly selected from 10MB to 20MB, and congestion duration is 200ms.

Figure 9 depicts the normalized average CCT of overall coflows under different coflow number. We can see that CCT in SkipL could be reduced by up to $\sim$ 38% compared to FAIR. The performance of SkipL and Varys is stable when coflow number increases. This is because that we only generate background flows in the links used by more than two coflows. Thus, the number of congested links is mainly determined by the coflow width. When coflow number increases, the percentage of congestion link becomes small. Furthermore, the performance gap between Varys and Varys w/o B.F. or SkipL and SkipL w/o B.F. becomes small as the increasing of coflow number. This indicates that when coflow number is large, the performance of schedulers is less influenced by the congestion.

Impact of coflow length: In this scenario, the coflow length is changed from 10MB to 70MB, 20 coflows with 32 width are randomly generated between 512 hosts, and the congestion duration is fixed to 200ms.

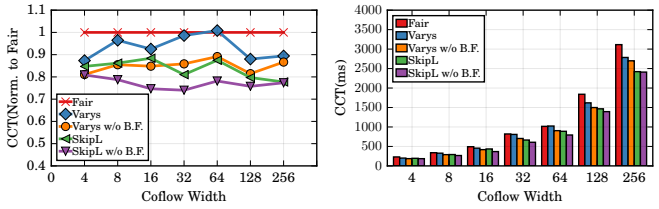


Fig. 11: The impact of coflow width in leaf-spine topology.

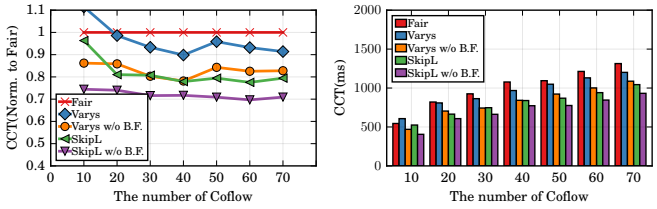


Fig. 12: The impact of coflow number in leaf-spine topology.

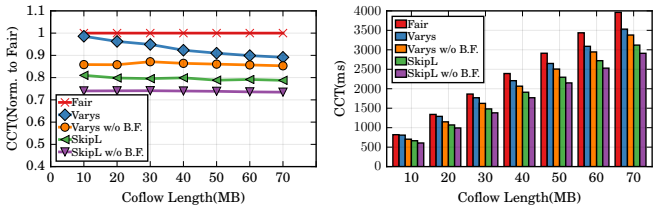


Fig. 13: The impact of coflow length in leaf-spine topology.

Figure 10 draws the normalized average CCT of overall coflows. The result shows that the performance of SkipL and other methods are very similar with Figure 9. SkipL performs the best and reduces 30% of CCT compared to Fair. The performance gap between Varys and SkipL is relatively large when coflow length is 10MB. The performance gap between SkipL and Varys remains stable when coflow length is large than 50MB. This because that when coflow length is 10MB, the optimal CCT is 80ms which smaller than the congestion duration, and congestion has great impacts on coflow's CCT. However, when coflow length is increased to 50MB, the optimal CCT is 200ms. It is equal to congestion duration, and congestion has fewer impacts on CCT.

B. Evaluation Results of Spine-Leaf Topology

In the following simulation, we use the leaf-spine topology as our default topology. The network has 512 hosts, 16 leaf switches, 16 spine switches. All hosts and switches are connected by 1Gbps links, and each leaf switch connects 32 hosts. Therefore, the oversubscription of the network is 2:1. Besides, we use ECMP [8], [34] as our default routing algorithm which is deployed by most of datacenter networks.

We repeat the simulations in the previous subsection, and only replace the topology from the big-switch to the leaf-spine topology. Differently with the big-switch topology, in the leaf-spine topology, we have to choose positions for coflows. Therefore, we split flows into two types. The first type is leaf-only flows. They achieve great data localities, and only transfer data to the hosts connected to the same leaf. This type of flows accounts about 75% of the total number of flows of a coflow. The second type is cross-leaf flows. They achieve poor data localities, and only transfer data to the hosts connected to the other leafs. For each coflow, we randomly choose two leafs, equally place leaf-only flows in two leafs, and let cross-leaf flows transfer data from one leaf to another.

In addition, in order to induce congestion in the network, 200ms congestion is generated at the beginning in all the leaf to spine links that are used by more than two different coflows. If the congestion occurs in a link, it will take up 80% bandwidth of the link, and all flows in the link share the residual 20% bandwidth. The rates of these flows are proportional to their desired rates.

Impact of coflow width: The simulation results are shown in Figure 11. We can see that all the mechanisms have very similar performance when the coflow width is 4. This is because that the coflow width is too narrow that there only one cross-leaf flow for a coflow. These small number of cross-leaf flows may never experience any congestion at all. As the increasing of the coflow width, SkipL has better performance than Varys. Compared to the Fair mechanism, SkipL can reduce the average CCT by up to 75% (when coflow width is 32). Besides, compared to Figure 8, we can find that all schemes have worse performance in the leaf-spine topology. Especially, when the coflow width is 256, the average CCT of Fair is about 1700ms in big-switch topology, but increases to about 3200ms in leaf-spine topology. The average CCT in leaf-spine topology increases almost twice than big-switch topology. This is mainly caused by routing mechanism. In leaf-spine topology, ECMP routing algorithm will randomly hash flows to multi-paths, and randomly hashing may cause many flows collide at one link, thus reducing the performance. Finally, the performance gap between SkipL and SkipL w/o B.F. is less than the performance gap between Varys and Varys w/o B.F., which indicates SkipL can better handle congestions.

Impact of coflow number: Figure 12 shows the simulation results of varying the number of coflows in the network. Firstly, as shown in the Figure, when the number of coflow is 10, the performance of SkipL is very close to Fair, and Varys even has a bad performance than Fair. Also, there is a 20% performance gap between SkipL and SkipL w/o B.F.. In fact, when there are only 10 coflows, the congestion may affect most of the coflows, and it is hard to find a coflow that is not affected by congestion. Thus, reducing average CCT by rearranging coflows is not effective and very challenging for SkipL. Secondly, when there are more coflows, SkipL has a 10% better performance than Varys. Compared to Figure 9, the performance gap between SkipL and Varys in the big-switch topology is large than it in the spine-leaf topology by 5%. This shows that how topologies influence the performance. In the big-switch topology, the performance of SkipL isn't influenced by the flow collisions caused by ECMP. Thus, it can have a better performance than in the leaf-spine topology. Finally, the

performance gap between SkipL and SkipL w/o B.F. keeps stable when the number of coflow increases. Therefore, SkipL has a stable performance when there are a large number of coflows.

Impact of coflow length: In this simulation, we vary the coflow length and the simulation results are shown in Figure 13. We can see that Varys has very poor performance when coflow length is 10MB. In fact, the optimal completion time of 10 MB flow is about 80ms in our simulation, and the default congestion duration is 200ms. If a congestion occurs, it will increase the CCT of a coflow by about 250% times. Therefore, congestion will have more influence on small flows. On the other hand, SkipL improves performance by 25% compared to Varys when coflow length is 10MB. Therefore, it is necessary to handle congestion when congestion duration is long, and handling congestion can greatly improve performance in this situation. Compared to Figure 10, the performance of all schemes reduces about 10%. The performance gap is mainly caused by topology as mentioned before. But the performance gap between SkipL and Varys keeps stable, this indicates SkipL has a stable performance when coflow length is large.

VII. Conclusion

Existing coflow scheduling mechanisms are mainly based on FIFO and SPTF, and few of them could timely reschedule coflows when large background flows cause the in-network congestion. However, we investigate that handling congestion that greatly affects flow's throughput can further reduce the average CCT of coflows. Based on this investigation, we proposed a congestion-aware coflow scheduling mechanism, which only detects congestion and estimates the available bandwidth of links at end-hosts. A prototype system is implemented in Linux, and a flow-level simulator is implemented to extensively evaluate the performance of SkipL through a series of simulation. The results of experiments and simulations show that SkipL could largely reduce the average CCT compared with per-flow fair sharing mechanism and Varys, and can timely handle congestion when it occurs in networks.

Acknowledgements

This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61502049, Young talent development program of the CCF and CAST, National High-Tech Research and Development Plan of China (863 Plan) under Grant No. 2015AA016101, Beijing New-star Plan of Science, Technology under Grant No. Z151100000315078, and Huawei Innovation Research Program (HIRP).

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *ACM SIGOPS*, 2007.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-memory Cluster Computing," in *USENIX NSDI*, 2012.

[4] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," in *ACM SIGCOMM*, 2011.

[5] M. Chowdhury and I. Stoica, "Coflow: a Networking Abstraction for Cluster Applications," in *ACM HotNets*, 2012.

[6] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters," in *ACM SIGCOMM*, 2014.

[7] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-Packet Load-Balanced, Low-Latency Routing for Clos-Based Data Center Networks," in *ACM CoNEXT*, 2013.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *USENIX NSDI*, 2010.

[9] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, "Scalable, Optimal Flow Routing in Datacenters Via Local Link Balancing," in *ACM CoNEXT*, 2013.

[10] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A Centralized Zero-Queue Datacenter Network," in *ACM SIGCOMM*.

[11] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters," in *USENIX ATC*, 2014.

[12] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can," in *ACM SIGCOMM*, 2015.

[13] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM*, 2014.

[14] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM*. ACM, 2015.

[15] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized Task-aware Scheduling for Data Center Networks," in *ACM SIGCOMM*, 2014.

[16] Y. Zhao, K. Chen, W. Bai, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *IEEE INFOCOM*, 2015.

[17] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Need for Speed: CORA Scheduler for Optimizing Completion-Times in the Cloud," in *IEEE INFOCOM*, 2015.

[18] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing Coflow Completion Times with Utility Max-Min Fairness," in *IEEE INFOCOM*, 2016.

[19] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *ACM SIGCOMM*, 2013.

[20] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *ACM SIGCOMM*, 2011.

[21] Z. Li, Y. Zhange, D. Li, K. Chen, and Y. Peng, "OPTAS: Decentralized Flow Monitoring and Scheduling for Tiny Tasks," in *IEEE INFOCOM*, 2016.

[22] "Amazon EC2," http://aws.amazon.com/ec2.

[23] "Microsoft Azure," http://azure.microsoft.com.

[24] "Dropbox," http://dropbox.com.

[25] "Coflow-Benchmark," https://github.com/coflow/coflow-benchmark.

[26] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating data center networks with zero loss," in *ACM SIGCOMM*, 2013.

[27] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic Scheduling of Network Updates," in *ACM SIGCOMM*, 2014.

[28] C. Lee, C. Park, K. Jang, S. Moon, and D. Han, "Accurate Latency-based Congestion Feedback for Datacenters," in *USENIX ATC*, 2015.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, 2008.

[30] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch." in *USENIX NSDI*, 2013.

[31] "ONOS Controller," http://onosproject.org/.

[32] "Libcurl," https://curl.haxx.se/libcurl/.

[33] "OpenvSwitch," http://openvswitch.org.

[34] 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 *ACM SIGCOMM*, 2009.