Rate-Aware Flow Scheduling for Commodity Data Center Networks

Ziyang Li[1,2], Wei Bai[2], Kai Chen[2], Dongsu Han[3] Yiming Zhang[1], Dongsheng Li[1], Hongfang Yu[4]
[1]PDL Lab, National University of Defense Technology
[2]SING Group, Hong Kong University of Science and Technology
[3]Korea Advanced Institute of Science and Technology
[4]University of Electronic Science and Technology of China

***Abstract*—Flow completion times (FCTs) are critical for many cloud applications. To minimize the average FCT, recent transport designs, such as pFabric, PASE, and PIAS, approximate the Shortest Remaining Time First (SRTF) scheduling. A common, implicit assumption of these solutions is that the remaining time is *only* determined by the remaining flow size. However, this assumption does not hold in many real-world scenarios where applications generate data at diverse rates that are smaller than the network capacity.**

In this paper, we look into this issue from system perspective and find that the operating system (OS) kernel can be exploited to better estimate the remaining time of a flow. In particular, we use the rate of copying data from user space to kernel space to measure the data generation rate. We design RAX, a rate aware flow scheduling method, that calculates the remaining time of a flow more accurately, based on not only the flow size but also the data generation rate. We have implemented a RAX prototype in Linux kernel and evaluated it through testbed experiments and ns-2 simulations. Our testbed results show that RAX reduces FCT by up to 14.9%/41.8% and 7.8%/22.9% over DCTCP and PIAS for all/medium flows respectively.

I. Introduction

Many data center applications, such as web search, social networking and data mining, have stringent latency requirements. As a result, the flow completion time (FCT) is critical for application performance and user experience, and has become one of the most important optimization goals for data center transport [1, 2].

Recent proposals, such as pFabric [1], PASE [3], and PIAS [2], minimize the average FCT by approximating the ideal Shortest Remaining Time First (SRTF) scheduling. According to SRTF, we should always schedule a flow with the smallest *remaining time*. In fact, these flow scheduling proposals [1–3] schedule the flow with the smallest *remaining size* with the implicit assumption that the remaining time is determined *only* by the remaining flow size. However, this assumption does not hold in many realistic scenarios, since in practice real applications can generate data at diverse rates that may be smaller than the network capacity, due to the bottleneck sources, such as disks and CPUs. Take the disk-bounded applications (e.g., MapReduce [4]) as an example, the I/O bandwidth of hard disk drives (HDD) and solid state drives (SSD) are about 1Gbps and 6Gbps, which may be smaller than that of the underlying (e.g., 10GbE) network. Furthermore, the I/O throughput may drop significantly when multiple threads simultaneously interact with the storage [5] (as detailed in §II-A). Similarly, for computation-intensive applications (e.g., Dryad [6]), the CPU may fail to saturate the network when performing complex computation. In all these cases, the data generation rates represent the application's actual demands for the network, thus affecting the remaining times of flows (as detailed in §II-B). This issue has been neglected in most prior designs [1–3, 7][1], even the latest ones [8–10].

In this work, we argue that not only the flow size, but also the flow data generation rate should be taken into account for minimizing FCTs. Motivated by this, our goal in this paper is to design an efficient flow scheduling scheme with the following objectives:

- **Rate-aware:** The solution must be able to obtain the flow data generation rates, in order to calculate the accurate flow remaining times.
- **Minimizing average FCT:** It must leverage the flow remaining times to minimize the average FCT.
- **Being practical:** It must work with commodity switches and not require any modification of legacy TCP/IP stacks or existing applications.

To acquire the data generation rate, we have analyzed the life cycle of a flow from the system point of view and found that we can obtain rich flow information from the OS kernel. Specifically, we monitor the TCP send buffer status in the kernel and use the data copy rate from user space to kernel space to measure the data generation rate (§III-A). Then, we design RAX, a rate-aware flow scheduling solution that calculates the remaining time of a flow based on both flow size and data generation rate (§III-B).

In short, in the course of design and implementation of RAX, we make the following contributions:

- We show that the data generation rates of flows play a key role in minimizing FCTs. We analyze the complete flow life cycle, extract information such as data generation rate

Work performed when Ziyang Li was an intern student at SING Group @ HKUST.

[1]While PDQ [7] computes the maximum sending rate for each flow, it only considers the network bottlenecks instead of the application demands.

978-1-5090-5336-0/17/$31.00 ©2017 IEEE

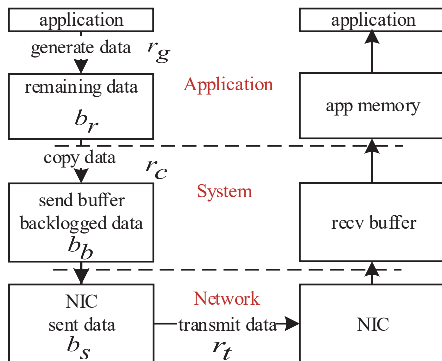


Fig. 1. The life cycle of flow

and backlogged size from the OS kernel, and use them to design RAX, the first data center flow scheduling solution considering both factors.

- We have implemented a RAX prototype in Linux kernel, which locates between the TCP/IP stack and the Network Interface Card (NIC) driver. RAX does not require any modification to user applications or legacy TCP/IP stacks, and thus supports various operating systems. RAX is a hot-pluggable kernel module, which is readily deployable on running systems.
- To evaluate RAX, we build a small testbed with 10 servers connected to a commodity Pronto-3295 Gigabit Ethernet switch. Our experiments show that RAX can accurately measure the data generation rates, and significantly outperform existing data center transports that only use flow size to estimate the remaining time. For example, RAX reduces the average FCT by up to 14.9% and 7.8% for all flows and 41.8% and 22.9% for medium flows compared to DCTCP [11] and PIAS [2], respectively. To complement the small scale 1GbE testbed experiments, we further evaluate RAX in a simulated large scale 10GbE network by using ns-2 [12]. Our simulations show that RAX reduces average FCT by 36% and 22% over DCTCP and PIAS, respectively.

II. Flow: From System Perspective

From the network's perspective, a flow refers to an application data unit travels across network [13]. Using the flow abstraction, all existing designs implicitly assume that flow data is always ready for transmission. But from the system's perspective, this is not true and the reality is much more complex. Fig. 1 shows the complete life cycle of a flow from the system's perspective. First, the data is generated from an application, then it traverses the system stacks, and eventually gets transmitted across the network.

Data generation: Applications generate data to be transmitted through the network at different rates. The data generation rate depends on various factors including application computational complexity, the CPU speed or the I/O throughput of storage devices. We denote the flow data generation rate as r_g. It depends only on the server components, such as CPU cores and hard disks, but not on the network.

Notation	Description
r_g	data generation rate from application
r_c	data copy rate to kernel
r_t	data transmission rate across network
b_r	size of remaining application data
b_b	size of data backlogged in the send buffer
b_s	size of data that has been sent into the network
C	network interface card (NIC) capacity

TABLE I
Main Variables

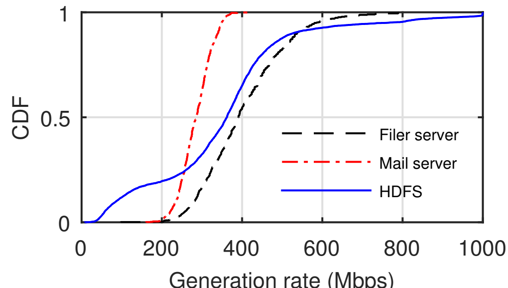


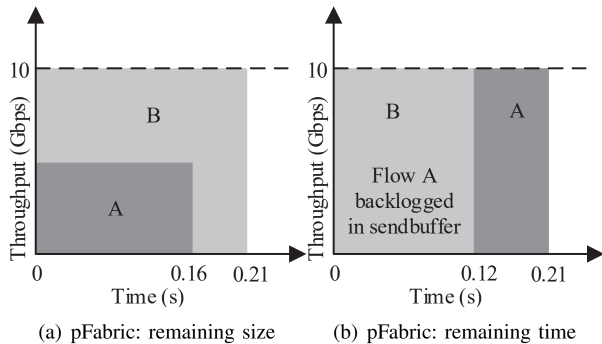
Fig. 2. [Testbed] Measured generation rate distributions.

Data copying to kernel: After the data is generated, applications call network functions such as *send()*, *sendto()* and *write()* to copy data into send buffer in the OS kernel. We use r_c to denote the data transmission rate from the application to the kernel.

Data transmission: Finally, the network interface card (NIC) transmits data from the send buffer into network. The transmission rate is denoted as r_t. Many factors determine the transmission rate (r_t), such as the link capacity, network load, and scheduling policies.

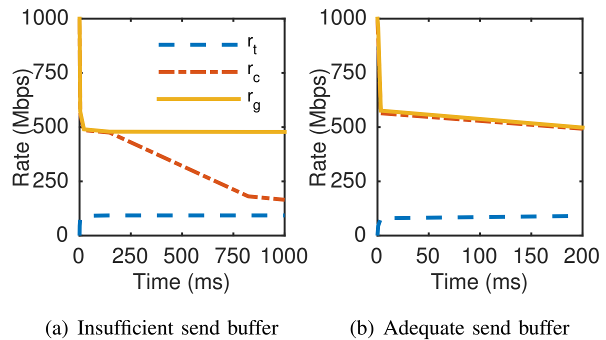
A. Does flow data generation rate vary widely?

The data generation rate (r_g) determines the real *demand* for the network, which is a crucial factor in network flow scheduling. However, we find that data generate rate varies widely across applications and changes dynamically even within an application. To study this, we measure the data generation rate (r_g) from two real world applications by modifying them. The flow data generation rate is the flow size divided by the data preparation time (excluding in-network blocked time). The first application is FileBench [14], which is a file system and storage benchmark that allows to generate a large variety of workloads. The backend storage device is hard disk, and the link capacity is 1Gbps. We choose `file server` and `mail server` as two representative workloads. The second application is HDFS [15]. We setup HDFS as the backend storage of Spark [16] and run the TeraSort benchmark. The r_g distributions are shown in Fig. 2. Applications have different demands for the network, e.g. the file server averagely generates data $1.4\times$ faster than the mail server. Also within an application, it varies widely from 100 Mbps to 100 Gbps ($1000\times$ difference). This is because some data is readily available in the memory ready to be transmitted (e.g., cached), while other data may be in the disk and may require further computation.



(a) pFabric: remaining size (b) pFabric: remaining time

Fig. 3. [Simulation] Motivating example. The average FCTs of pFabric(rem. size) and pFabric(rem. time) are 0.185s and 0.165s respectively.



(a) Insufficient send buffer (b) Adequate send buffer

Fig. 4. [Testbed] Relationships among r_g, r_c and r_t with (a) adequate and (b) insufficient send buffer. Note that r_g and r_c overlap in (a).

B. Does data generation rate impact flow completion times?

As shown above, the application data generation rate varies widely. However, previous works largely overlook this important fact, which results in sub-optimal scheduling results.

We demonstrate the impact of data generation rate on flow scheduling with a toy example. In this example, we setup two flows, Flow A and Flow B. The flow sizes are 100MB and 150MB, and the data generation rates are 5 Gbps and 10 Gbps respectively. We use ns-2 [12] to simulate these flows in a network, where a sender and a receiver connected by one switch. The link capacity is 10Gbps. We use pFabric to schedule these flows and compare it with a variant of pFabric that uses the remaining time to prioritize flows. The remaining time of a flow is the remaining size divided by the data generation rate. The original pFabric sets explicit priority for packets based on flow remaining size. Thus, it prioritizes Flow A which is shorter than Flow B. However, because Flow A cannot saturate the 10 Gbps link, the remaining bandwidth is used for Flow B as shown in Fig. 3(a). As a result, the FCTs of A and B are respectively 0.16s and 0.21s, and the average is 0.185s. In contrast, our variant that uses SRTF scheduling first schedules Flow B as shown in Fig. 3(b). The resulting FCTs for Flow A and B are respectively 0.21s and 0.12s, and the average is 0.165s. We observe that 1) data generation rate affects flow completion times; and 2) flows scheduled using the remaining time finish more quickly.

C. Can we accurately estimate the data generation rate?

The data generation rate in Fig. 2 is measured by modifying applications. This will introduce huge burden for operators in commodity data centers hosting a variety of applications, which is highly undesirable. Therefore, we take a step back and ask: *can we obtain the data generation rate (r_g) without instrumenting applications?* Before introducing our solution, we first analyze the relationships among r_g, r_c and r_t (as shown in Figure 1 and Table I).

Case 1: When $r_g = r_t$,[2] the network can fully satisfy the application demand. Once the application data is generated, it can be quickly copied to kernel space and delivered to the network without much delay. Therefore, in such scenario, $r_g = r_c = r_t$ and very little data is backlogged in the send buffer.

When $r_g > r_t$, the network becomes the bottleneck of data transfer. The excess data that cannot be immediately delivered to network will get buffered in TCP send buffer. Based on the availability of additional room in the send buffer, it results in the following two possibilities.

Case 2: If the send buffer space is not enough (e.g., already exhausted), the OS kernel will block the data copy from the application (or defer it in non-blocking mode by returning error for socket functions), thus pushing the buffer pressure back to the application. In such scenario, the following relation holds: $r_g > r_c \geq r_t$.

Case 3: If we have adequate send buffer space to buffer excess data, the data copy from the application will not get blocked. Therefore, we have $r_g = r_c > r_t$.

Based on above analysis, we find that $r_g = r_c$ when the send buffer is large enough. Hence, we configure a large TCP send buffer and directly measure r_c to estimate r_g without patching applications (details in $\S IV$).

To evaluate the effectiveness of this method, we create a simple example in which an application generates flow data at 500 Mbps and the transmission rate is limited at 100 Mbps. Fig. 4 shows the rates (r_g, r_c and r_t). r_c and r_t are measured by our *netfilter* hook. The initial value of r_g and r_c is the line rate of 1 Gbps. We give the flow a large send buffer (100 MB). When send buffer is adequate for the flow, r_c approximately equals to r_g. Fig. 4(b) demonstrates the case where send buffer is small (16 KB by default). Once the send buffer is insufficient for the flow, r_c drops from r_g to r_t, as shown in Fig. 4(a). We conclude that r_c accurately represents r_g when the send buffer is sufficiently large.

III. Rate-Aware Flow Scheduling

The key insight of RAX is to schedule flows based on their remaining *time* rather than size. In this section, we first introduce how to estimate remaining time by leveraging rich information obtained from OS kernel (e.g., estimated data generation rate). Then, we propose a new flow scheduling design that utilizes the estimated remaining time.

[2]We ignore the case that $r_g < r_t$ since the network transmission rate r_t is bounded by application data generation rate r_g.

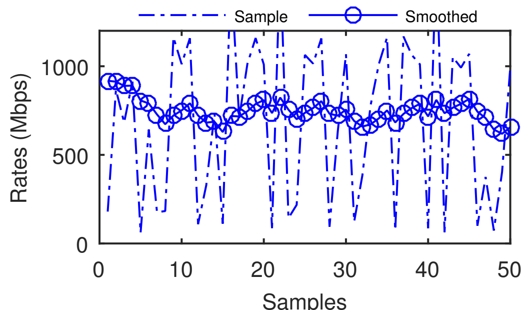


Fig. 5. [Testbed] Smoothed measured rates when the disks are in severe interference state.

A. Remaining Time Estimation

To perform Shortest Remaining Time First (SRTF) scheduling, we need to calculate the *shortest possible remaining time*: the time for a flow to finish in an *idle* network without competing with other flows.

As shown in Fig. 1, a flow is finished when all its bytes are sent into network. To simplify our analysis, we assume that the send buffer is large enough to hold any data delivered from the application. The remaining data of a flow includes the data already in the send buffer (b_b) and the data that will be generated by the application (b_r).

Therefore, a flow takes two steps to finish. The first step is to drain backlogged data in the send buffer (b_b), which can be transmitted at NIC capacity (C). So the time of the first step is b_b/C. After the first step, we need to transmit data that will be generated by the application (b_r). For the second step, the transmission is either throttled by the application or the network. So the time of the second step is $b_r/\min(r_g, C)$. Hence, the shortest possible remaining time T_r is given as follows:

$$T_r = \frac{b_r}{\min(r_g, C)} + \frac{b_b}{C} \tag{1}$$

In above formula, C is known and b_b can be read from the send buffer status. However, we have no prior knowledge of r_g and b_r. Now we show how to estimate r_g and b_r.

r_g in above formula is indeed the data generation rate *in the future*. Here, we use the latest measured data generation rate as the estimation. As shown in §II-C, when the send buffer is large enough, the copy rate r_c is a good estimation for the generation rate. To measure the copy rate (r_c) for each flow, we monitor the bytes copied (b_c) from the application within time interval (t_e). A copy rate sample τ is given as b_c/t_e. However, the application behavior may be unstable. For example, when an application reads data from the disk, its disk I/O throughput may be interfered by other I/O threads. Thus, we smooth r_c using exponentially weighted moving average (EWMA) as follows: $r_c \leftarrow \alpha\tau + (1-\alpha)r_c$ where α is the smoothing factor between 0 and 1. As shown in Fig. 5, although the raw sample rates oscillate drastically, the smoothed average rates (α is 0.25) are relatively stable.

The other unknown variable is b_r, the size of remaining data that will be generated by the application. Some previous works [1, 3, 7, 17] assume that b_r can be obtained by modifying applications. However, flow size information is not available in many cases (e.g., HTTP chunked transfer and database) [2, 18]. Moreover, modifying various applications to get flow size information will increase operation burden. Therefore, we prefer a simple approach to get or estimate b_r without touching applications.

Many works [11, 19, 20] have shown that the flow size in data centers typically follows heavy-tailed distribution: most flows are small but the majority of all bytes are from a small percent of large flows. For example, in a data mining workload [19], 80% of the flows are less than 10KB and 95% of all bytes are in the ~3.6% flows that are larger than 35MB. Under such heavy-tailed distributions, the number of bytes already sent is a good predictor for the remaining size [21]. Therefore, we can use the data sent by the application, including the data backlogged in the send buffer (b_b) and the data sent into the network (b_s), to estimate the remaining size.

In summary, the estimation of remaining time (T_{est}) can be given as follows:

$$T_{est} = \frac{b_s + b_b}{\min(r_c, C)} + \frac{b_b}{C} \tag{2}$$

where all variables can be obtained without modifying applications or OS.

B. Rate-Aware Flow Scheduling

To emulate SRTF, we need to allocate bandwidth based on the estimated remaining time (T_{est}) of each flow. Inspired by [2], we leverage class of service queues available on existing commodity switches to enforce it. Current commodity switches typically provide 4-8 queues per egress port and support strict priority queueing. To emulate SRTF, we need to classify flows with smaller estimated remaining time into higher priority queues. At the end host, RAX tracks the per-flow state, computes the estimated remaining time, and marks flow packets with the corresponding priority. The switch simply classifies packets into different queues based on the priority values carried by these packets.

According to Formula 2, the estimated remaining time keeps changing in a flow's life time. In this formula, $b_s + b_b$ keeps increasing. When r_c and r_t are stable, b_b also keeps non-decreasing. In such scenario, the flow priority keeps being demoted, thus RAX performs similar to Multi-Level Feedback Queue (MLFQ). However, RAX may perform differently when r_c changes dynamically, e.g., the flow priority may increase when r_c increases. Even during the priority demotion process, a flow may bypass first several high priorities when a *send()* function call copies a large amount of data from the application to the send buffer, resulting in a large b_b.

We need to use a series of thresholds to distinguish flow priorities. Formally, there are K priorities in total. Priority 1 is the highest while K is the lowest. Correspondingly, there are $K+1$ thresholds. A flow is assigned to priority i if its T_{est} satisfies $Q_{i-1} \leq T_{est} < Q_i$, where Q_i is the threshold of priority i. Note that $Q_0 = 0$ and $Q_K = \infty$. Compared to

previous works [1, 2], the best value for Q_i depends not only on the flow size distribution, but also on the data generation rate, which is difficult to derive. Therefore, in RAX, we employ exponential thresholds. The threshold for queue i is chosen to be

$$Q_i = Q_1 \times E^{i-1} \quad (3)$$

where E is the base that determines how much slower a flow can be in this priority queue. For the web search workload [11] in 1GbE network, a slow flow may take more than 200ms to complete, which is $154\times$ longer than the fast ones. We typically use $E = 2$ and $Q_1 = 1280\mu s$ for 1GbE network, $E = 2$ and $Q_1 = 160\mu s$ for 10GbE network. We find that such simple threshold setting works well in our testbed experiments and simulations.

Different from MLFQ, RAX may incur packet reordering when the priority increases. Packet reordering may seriously degrade the throughput. To avoid this impact, we temporarily disable the duplicate ACK mechanism at the sender side while the flow priority increases. In addition to this, another promising solution is enabling re-sequence buffer [22] at the receiver side.

IV. Implementation

Most features of RAX are implemented at the end host except for priority queueing configuration at the switch. We have implemented a RAX prototype in Linux kernel. The prototype has two main components: measurement of data copy rate and priority tagging. We now introduce them in detail while leaving the switch configuration to §V-A.

A. Measurement of Data Copy Rate

The rate measurement module measures the per-flow data copying rate to estimate data generation rate. To achieve this, we leverage *jprobes* to hook two kernel functions `tcp_sendmsg` and `tcp_sendpage` which copy data to the TCP send buffer. When hooked kernel functions are called, we identify the flow it belongs to and update corresponding statistics, such as copy data size and elapsed time since last call.

The rate we measured from system layer is the data copy rate r_c. And we intend to use r_c as an estimation of the data generation rate r_g. As previously discussed (§II-C), the send buffer size is crucial to the estimation accuracy. Linux performs auto-tuning of the TCP send buffers. By default, the minimal, initial and maximum send buffer sizes allowed for a single TCP socket to use are 4KB, 16KB and 208KB respectively, the maximum send buffer space shared by all sockets is 4MB. We can control these parameters by configuring `net.ipv4.tcp_wmem` and `net.core.wmem_max` through the `sysctl` command. We enlarge per-flow send buffer size to 2MB to make the data copy rate measurement accurate. The overall send buffer space is enlarged to 200MB accordingly to hold concurrent flows. The send buffer can easily hold more than 100 concurrent flows of realistic data center workloads [11, 19].

Having a large send buffer does not harm the network: 1) This is the recommended practice in high speed datacenter networks. In data center environment, the RTTs at 50 and 99 quantile are $326\mu s$ and $2.43ms$ respectively [23], so the BDP is at least 400KB ($326\mu s \times 10Gbps$). Thus, the default settings is not fit for high throughput and low latency datacenter networks. 2) Servers in data centers usually have tens even hundreds of GBs of RAM, so large send buffer is acceptable. 3) Similar approach has been used to buffer traffic and estimate traffic demand [24, 25].

The send buffer may still be not large enough to hold the data for an ultra long and fast flow. In this situation, r_c tends to underestimate r_g. However, the underestimation usually does matter because the ultra long flows result in low priority queues anyway.

B. Tagging Packet Priority

To enforce the desired scheduling in the network, RAX assigns priorities to flows by comparing their remaining times with the thresholds at the end hosts. RAX then tags each outgoing packet with corresponding priority.

The priority tagging module is implemented by the `NF_IP_LOCAL_OUT` hook of *netfilter* [26]. The netfilter hook intercepts all outgoing packets. When a packet comes in, the hook queries the flow table to get its flow entry with the protocol, the IPs and ports of its source and destination. Then the hook calculates priority for the corresponding flow of the packet. The priority is encoded into the DSCP field in the IP header. Finally, the netfilter hook returns `NF_ACCEPT` to admit the packet.

V. Evaluation

We evaluate RAX with both testbed experiments and large-scale ns-2 simulations. Our evaluation centers around the following questions:

- **How accurate our rate measurement is?** With enlarged send buffers, we show that the copy rate can accurately represent the data generation rate. Most of the measured rates fall within the $\pm 10\%$ range of the true value, and the coefficient of variation is no more than 0.055. We also show the impact of different send buffer capacity settings.
- **How does RAX perform in practice?** Our testbed experiments show that RAX reduces average FCT by up to 14.9% and 7.8% compared to DCTCP and PIAS respectively. The major improvement is attributed to the medium flows, RAX reduces the average FCT of medium flows by up to 41.8% and 22.9% over DCTCP and PIAS respectively. The results from multiple data generation rate distributions verify the performance improvement of RAX.
- **How robust is RAX?** We evaluate RAX with different threshold settings and show RAX is robust to misconfigured thresholds. We also prove that a single packet does not experience huge increase in delay with large send buffer.
- **How does RAX perform in large-scale clusters?** The large-scale ns-2 simulation results show that RAX greatly

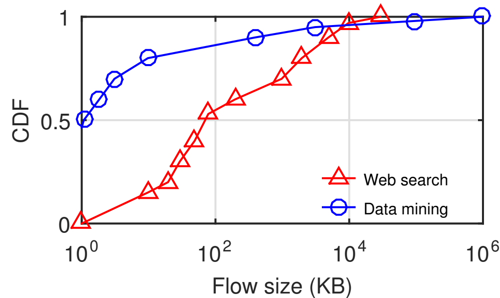


Fig. 6. Flow size distributions.

outperforms DCTCP and PIAS in average FCT by 35.6% and 21.5% respectively. RAX shows at most 26.4% performance gap in overall average FCT compared to pFabric, which is state-of-the-art near optimal information-aware scheduling scheme.

A. Testbed experiments

Testbed setup: We built a small scale prototype system for RAX with 10 servers. Each server comprises of a 4-core Intel E5-1410 2.8GHz CPU, 8GB memory, a 500GB hard disk, and a Broadcom BCM5719 NetXtreme Gigabit Ethernet NIC. The operating system running on each server is Debian 6.0 64bit operating system with Linux 3.18.11 kernel. These servers are connected to a Pronto 3295 48-port Gigabit Ethernet switch with 4MB shared memory. To measure data generation rate in the high speed network, we connected two servers to the 10GbE ports in the switch. Each server is equipped with an Intel 82599EB 10GbE NIC.

At the switch side, we set 8 priority queues and enable per-port ECN marking. The ECN marking threshold is 20KB. At end hosts, the RTOmin is 10ms, and the send buffer size per flow is 2MB if not specially mentioned in the following experiments. For RAX, the smoothing factor α for copy rate measurement is 0.25.

Workload: We modify traffic generator of [27] to generate traffic with specific data generation rate distributions. We use the web search workload [11] and the data mining workload [19] from production data centers whose flow size distributions are shown in Fig. 6. The arrival of flows conforms to a Poisson process. The flow data generation rates follow the distributions from Fig. 2, which are file server, mail server and hdfs. For testbed experiments, the traffic load is represented by the actual throughput at the receiver side.

Schemes compared: We compare RAX with DCTCP, a transport protocol that has been widely used in production data centers; and PIAS, a size-based flow scheduling scheme. We use the open source implementations of DCTCP and PIAS to conduct the following experiments.

Data generation rate measurement: We first verify the accuracy of the rate measurement. We measure the flow copy rates while controlling the data generation rate at 200 Mbps, 500 Mbps, 800 Mbps for 1Gbps link and 2000 Mbps, 5000 Mbps, 8000 Mbps for 10Gbps link. We set the send buffer space allowed for a single TCP socket to 2 MB. Since the average flow size is 1.6MB, the send buffer is capable of holding flows. We conduct 5000 flows in this experiment. These flows arrive according to a Poisson process. Fig. 7 shows the CDFs of measured copy rates using 1Gbps link (Fig. 7(a)) and 10Gbps link (Fig. 7(b)). From Fig. 7, we can see that the measured copy rates are consistent with the real flow data generation rates. More than 95% measurements fall into the generation rate $\pm 10\%$ interval. The minimal and maximum coefficient of variation are 0.017 and 0.055 respectively. With both 1Gbps and 10Gbps links, a wide range of generation rates, the measured copy rates are accurate to represent the flow data generation rates. This result confirms that we can obtain accurate generation rates when the send buffer is adequate.

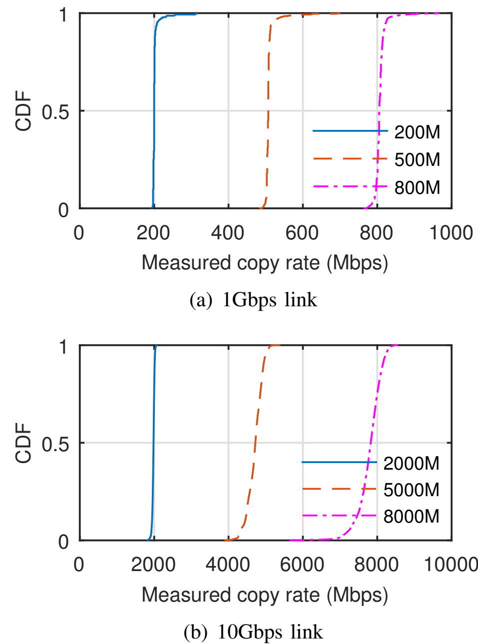
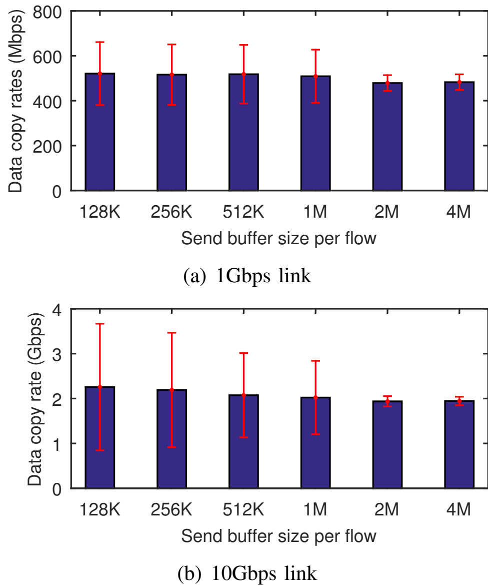


Fig. 7. [Testbed] Measured copy rates are consistent with the real generation rates.

Impact of send buffer capacity: The accuracy of data generation rate measurement relies on having large send buffers. One may be curious about the impact of send buffer capacity on rate measurement. We configure multiple send buffer sizes range from 128KB to 4MB to evaluate their impact on copy rate measurement accuracy.

For 1Gbps link, the data generation rate of all flows is 500Mbps. The average load of the sender side is 800Mps. We show the measured data copy rates in Fig. 8(a). The error bars show the measurement accuracy. When the send buffer size is only 128KB, the coefficient of variation is 0.28. After we enlarge the send buffer size from 128KB and 1MB, the coefficient of variation is still as high as 0.24. That is because the send requests are blocked due small send buffer sizes, so the rate measurement is not accurate. Once the send buffer size reaches 2MB, the coefficient of variation sharply decreases to 0.07. Note that the average flow size is 1.66MB, 2MB is large enough for send buffer to hold most flows. However,



(a) 1Gbps link

(b) 10Gbps link

Fig. 8. [Testbed] The measured data copy rates under different send buffer capacity sizes.

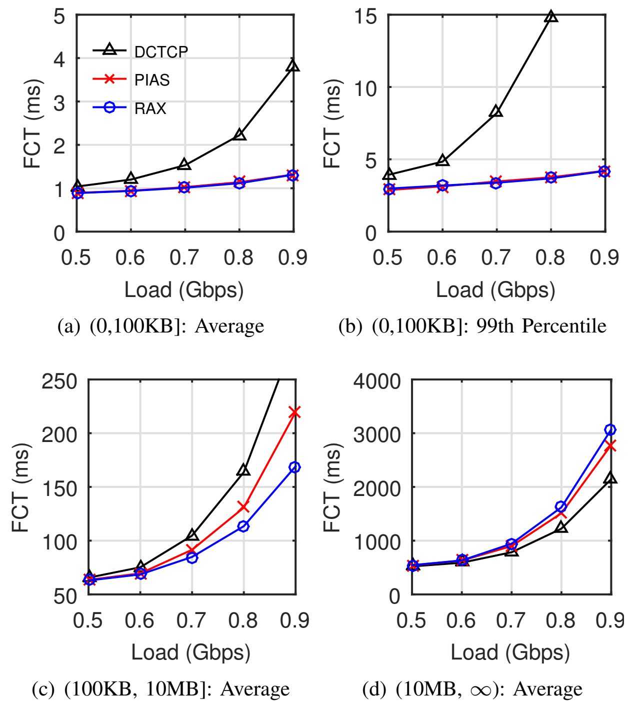
further increasing the send buffer size to 4MB does not provide significant accuracy improvement. As shown in Fig. 8(b), the experiments on 10Gbps link provide similar results.

Results with realistic workloads: Among the 10 servers, one acts as the receiver, the others are senders. The receiver fetches data from senders. The data generation rate distribution is file server. We present the results of web search workload and data mining workload in Fig. 9 and Fig. 10 respectively. The x-axis presents the average load of the receiver. For the web search workload, RAX improves the average FCT over DCTCP and PIAS by 14.9% and 7.8% respectively. RAX outperforms PIAS by medium flows, and breaks even with PIAS by short and long flows. Because PIAS always assigns short flows with the highest priority, it is hard to beat PIAS with short flows. Similarly, long flows (10 MB, ∞) usually stay in the lowest priority queue for both RAX and PIAS. The major improvement comes from the medium flows. RAX reduces the average FCT of medium flows compared to DCTCP and PIAS by up to 41.8% and 22.9% respectively.

Since the data mining workload is extremely biased, most flows are short while a few ultra long flows dominate the average FCT, the performance improvement of RAX is rather limited. For the data mining workload, RAX reduces the average FCT compared to DCTCP and PIAS by up to 3.3% and 2.5% respectively.

Results with different distributions: Besides with file server distribution, we also conduct experiments with other two data generation rate distributions in Fig. 4. The load of the receiver is 900Mbps. We present the overall average FCTs and the average FCTs of medium flows in Fig. 11. The results show that RAX outperforms DCTCP and PIAS with different distributions, especially for medium flows.

Impact on packet delays: We evaluate the impact of large send buffers on the packet delays in a testbed experiment.



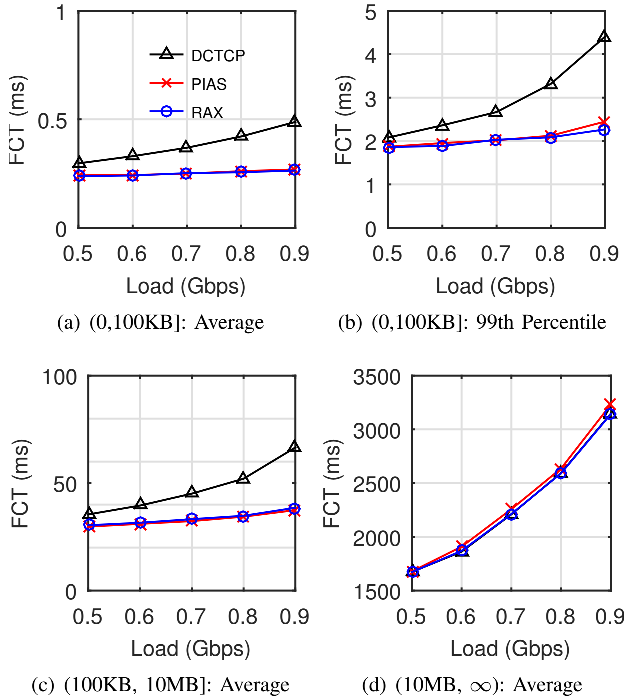
(a) (0,100KB]: Average

(b) (0,100KB]: 99th Percentile

(c) (100KB, 10MB]: Average

(d) (10MB, ∞): Average

Fig. 9. [Testbed] The FCTs of web search workload.



(a) (0,100KB]: Average

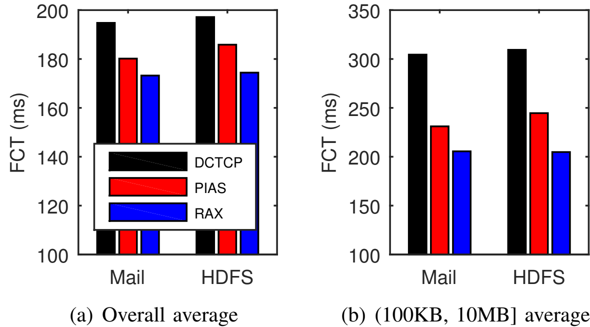
(b) (0,100KB]: 99th Percentile

(c) (100KB, 10MB]: Average

(d) (10MB, ∞): Average

Fig. 10. [Testbed] The FCTs of data mining workload.

We start flows with unlimited data generation rates from web search workload to saturate the send buffer. Then we measure the FCT from the sender to the other receiver by sending a flow with only 1KB payload. Since background flows are saturating all send buffer space, and the NIC becomes severely congested. The congestion at NIC leads to increase in flow completion times. However, it is not introduced by large send buffers. We



(a) Overall average

(b) (100KB, 10MB] average

Fig. 11. [Testbed] The average FCTs of web search workload with different data generation rate distributions.

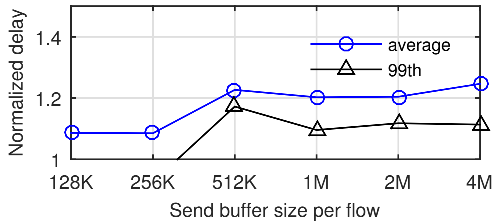


Fig. 12. [Testbed] Normalized delay when per-flow send buffer size varies.

compare multiple per-flow send buffer sizes in this experiment. To quantify the impacts, the packet delays are normalized to the default per-flow send buffer size of Linux (16KB). From Fig. 12, we can see that large send buffers do not introduce huge increase in packet delay.

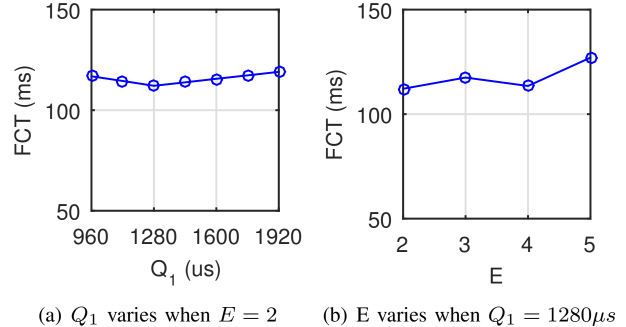
System overhead of RAX implementation: To evaluate the overhead of RAX implementation, we use `iperf` to saturate the 10 Gbps link. The goodput is around 9.4Gbps with and without RAX. The CPU overhead introduced by RAX is smaller than 1%. The memory usage is proportional to the number of concurrent flows with each flow consuming tens of bytes memory.

Wang et al. [24] demonstrated that, the total memory consumption of send buffers on each sender is always smaller than 200MB. We also observe that even when the sender is in heavy load (i.e. outgoing throughput is 900Mbps), the number of concurrent flows is no more than 60, the memory usage of all send buffers rarely goes beyond 120MB.

Parameter sensitivity: RAX has two parameters: Q_1, the threshold for the first priority and E, the base from Equation 3. To demonstrate the robustness of RAX with respect to parameter settings, we test RAX with different settings with the web search workload. We vary Q_1 from 960 to 1920 usec, while E ranges from 2 to 5. As shown in Fig. 13, we observe that the performance of RAX is relatively stable with respect to threshold settings.

B. Large-scale cluster simulations

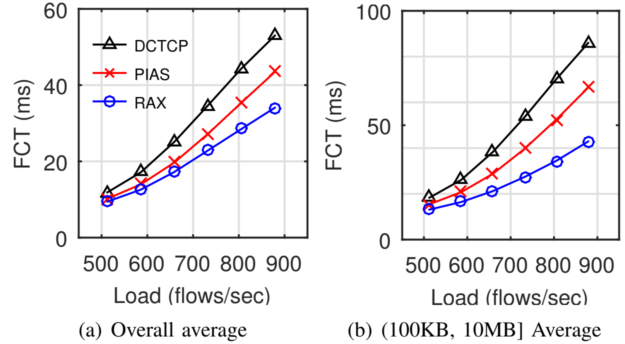
Simulation setup: We simulate a large-scale cluster by using ns-2. The topology is leaf-spine, there are 4 spine switches and 9 leaf switches in network. The link capacity between leaf and spine is 40 Gbps. Each leaf switch is connected



(a) Q_1 varies when $E = 2$

(b) E varies when $Q_1 = 1280\mu s$

Fig. 13. [Testbed] The overall average FCTs with different threshold settings.



(a) Overall average

(b) (100KB, 10MB] Average

Fig. 14. [Simulation] The average FCTs of web search workload.

with 16 end-hosts, so 144 end-hosts in total. The bandwidth between end-host and leaf switch is 10 Gbps. This is a non-oversubscribed network, and the round trip time (RTT) across the whole network is $82\mu s$. We use packet spraying for load balancing [28]. The data generation rate distribution for simulation is synthesized from distributions in Fig. 2, but with $10\times$ magnification to fit for the 10Gbps environment. For simulated experiments, the traffic load is represented by new incoming flows per seconds.

Performance in large-scale clusters: Fig. 14 shows the performance improvement of RAX over DCTCP and PIAS. As shown in Fig. 14(a), RAX reduces the average FCT by up to 35.6% and 21.5% compared to DCTCP and PIAS respectively. The simulation results are consistent with our testbed experiments. RAX achieves best performance improvement for medium flows. As shown in Fig. 14(b), RAX reduces the average FCT of medium flows by up to 50% compared to DCTCP, and up to 35.4% compared to PIAS.

The gap between RAX and the information-aware solution: We compare RAX with pFabric(rem. time) (discussed in § II-B), which is state-of-the-art near optimal information-aware scheduling scheme. We also conduct an information-aware variant of RAX (noted as RAX+) for comparison. Fig. 15 shows the overall average FCTs of RAX, RAX+ and pFabric(rem. time). RAX shows at most 26.4% performance gap in overall average FCT compared to pFabric(rem.time). RAX+ achieves comparable performance compared to pFabric(rem.time).

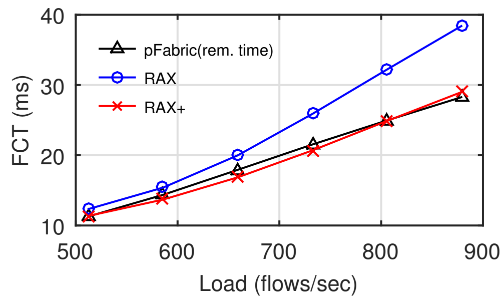


Fig. 15. [Simulation] The overall average FCTs of RAX, RAX+ and pFabric(rem. time).

VI. Related Work

Flow scheduling schemes are designed to minimize the average flow completion times. We summarize the previous flow scheduling works into two categories based on their information sources: application-informed flow scheduling [1, 3, 7, 17, 29] and network-informed flow scheduling [2, 30].

Applications-informed flow scheduling schemes directly obtain flow information from applications. These works approximate the SRTF discipline to minimize the average FCT. Three noteworthy drawbacks of application-informed schemes are as follows: (i) They need great effort on modifying applications to obtain flow size information. (ii) These designs require either unpractical support from switch hardware [1], complex arbitration [3, 7] or customized end-host transport [17, 29]. Network-informed flow scheduling schemes [2, 30] adopt the non-clairvoyant scheduling (such as LAS and MLFQ) to minimize FCTs. These works use bytes sent information to estimate the flow size, which would be efficient for heavy-tailed flow size distributions. Again, existing flow scheduling works pay close attention to the network bottlenecks, and largely overlook the application demands.

VII. Conclusion

In this paper, we investigate the importance of the flow data generation rates in minimizing FCTs. We design RAX, an efficient rate-aware flow scheduling approach, that calculates the remaining time of a flow based on both flow size and data generation rate. We have implemented a RAX prototype in Linux kernel, which is compatible with legacy transport, readily deployable with commodity hardware. Both testbed experiments and simulations show that RAX reduces the average FCT compared to DCTCP and PIAS.

Acknowledgments

This work is supported by the National Key Research and Development Program under Grant No.2014CB340303, the Hong Kong RGC ECS-26200014, GRF-16203715, GRF-613113, CRF-C703615G, the National Natural Science Foundation of China under Grant No.61222205, 61379055 and 61379053, the Program for New Century Excellent Talents in University, the Fok Ying-Tong Education Foundation under Grant No.141066, and MSIP/IITP of Republic of Korea under B0101-16-1368.

References

[1] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *ACM SIGCOMM 2013*.
[2] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *USENIX NSDI 2015*.
[3] 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 *ACM SIGCOMM 2014*.
[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *USENIX OSDI 2004*.
[5] A. Agache and C. Raiciu, "Oh flow, are thou happy? tcp sendbuffer advertising for make benefit of clouds and tenants," in *USENIX HotCloud 2015*.
[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *ACM EuroSys 2007*.
[7] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *ACM SIGCOMM 2012*.
[8] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *ACM SIGCOMM 2016*.
[9] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling mix-flows in commodity datacenters with karuna," in *ACM SIGCOMM 2016*.
[10] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, "Numfabric: Fast and flexible bandwidth allocation in datacenters," in *ACM SIGCOMM 2016*.
[11] 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 2010*.
[12] [Online]. Available: http://www.isi.edu/nsnam/ns/
[13] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling end-host network functions," in *ACM SIGCOMM 2015*.
[14] A. Wilson, "The new and improved filebench," in *USENIX FAST 2008*.
[15] [Online]. Available: http://hadoop.apache.org/
[16] [Online]. Available: http://spark.apache.org/
[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 *Co-NEXT 2015*.
[18] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM 2015*.
[19] 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*.
[20] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM 2015*.
[21] J. Nair, A. Wierman, and B. Zwart, "The fundamentals of heavy-tails: properties, emergence, and identification," in *SIGMETRICS 2013*.
[22] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh, "Juggler: A practical reordering resilient network stack for datacenters," in *EuroSys 2016*.
[23] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *ACM SIGCOMM 2015*.
[24] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan, "c-through: Part-time optics in data centers," in *ACM SIGCOMM 2010*.
[25] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *2011 Proceedings IEEE INFOCOM*, April 2011, pp. 1629–1637.
[26] [Online]. Available: http://www.netfilter.org/
[27] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ecn in multi-service multi-queue data centers," in *USENIX NSDI 2016*.
[28] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *IEEE INFOCOM 2013*.
[29] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *USENIX NSDI 2015*.
[30] A. Munir, I. Qazi, Z. Uzmi, A. Mushtaq, S. Ismail, M. Iqbal, and B. Khan, "Minimizing flow completion times in data centers," in *IEEE INFOCOM 2013*.