Tailor: Trimming Coflow Completion Times in Datacenter Networks

Jingjie Jiang*, Shiyao Ma*, Bo Li*, and Baochun Li†
*Department of Computer Science and Engineering, Hong Kong University of Science and Technology
†Department of Electrical and Computer Engineering, University of Toronto

***Abstract*—Tasks in a data-parallel job communicate with each other through a number of concurrent flows, which is described as a *coflow*. These flows are correlated in the sense that the performance of a coflow is dictated by the flow that takes the longest time to complete. Minimizing coflow completion times, however, turns out to be a challenge, given the correlation across flows and how they are routed collectively through a datacenter network. In this paper, we propose *Tailor*, a simple yet effective mechanism with the objective of trimming the coflow completion times in a datacenter network. To achieve our objective, *Tailor* takes advantage of OpenFlow in a software-defined datacenter network. By monitoring and rerouting live flows to links with lighter loads, *Tailor* guarantees that the coflow completion time is minimized dynamically and converges to its lower bound. Our experimental results in both Mininet and large-scale simulations have shown that *Tailor* is much more effective than flow-level schemes when it comes to reducing coflow completion times. It also outperforms existing scheduling-only coflow mechanisms and achieves similar performance with the state-of-the-art hybrid mechanism, yet with much lower complexity.**

I. Introduction

Data-parallel jobs in private datacenters, such as web search queries and MapReduce, require transferring intermediate results across shared links in datacenter networks. For example, in MapReduce jobs [1], a large amount of data is shuffled between the *map* and *reduce* stages. A coflow, consisting of many concurrent flows between different senders and receivers, succeeds only after all its constituent flows have finished. In other words, the completion time of a coflow is equivalent to the completion time of the bottleneck flow that lags behind the most. It is natural to believe that the performance of these coflows hinges upon their completion times. This is especially the case for real-time applications, where longer latencies lead to significant financial loss [2].

Existing works on reducing flow completion times [3], [4], [5], [6] have largely focused on minimizing the average flow completion time, by improving the performance of each individual flow in isolation. This is insufficient, and sometime even unnecessary for reducing the completion time of a coflow. To illustrate this observation, consider the scenario where a small flow and a large flow belonging to the same coflow get congested at a switch. A per-flow fairness strategy would give them equal bandwidth to send data [4]; a per-flow priority strategy [5], [6] would let the flow with smaller remaining size to use up all the bandwidth first; the dynamic routing in Hedera [3] will do nothing since there is no conflict between two large flows. In any case, the larger flow is slowed down, prolonging the completion time of the coflow.

Based on the observations above, researchers have recently turned to enabling coflow-aware scheduling through per-flow rate limiting (*e.g.*, [7], [8], [9]), per-coflow priority queues [10] and SDN-based routing [11]. Since link congestions are commonly seen through the datacenter network, existing coflow schemes with static routing [7], [8], [9], [10] are restricted by the default path selection and may incur severe performance loss [11]. On the other hand, existing routing schemes either neglect coflow semantics [3], or too complicated to be efficient in practice [11].

In this paper, we propose to coordinate flows in a coflow through dynamic routing. The key challenge is how to find the performance bottleneck of a coflow, and how to speed up such bottleneck to trim the coflow's completion time. This problem is further complicated by the fact that the bottleneck changes from time to time when the scheduling and routing strategies come into action: after accelerating the slowest flow, the second slowest flow may become the new bottleneck. More importantly, since the senders and receivers of flows spread across a datacenter, these flows would go through multiple switches in the network. It is therefore difficult for edge switches to grasp the dynamic status of flows. In addition, switches must perform consolidated routing policies to coordinate the flows in a coflow.

To tackle these challenges, we design *Tailor*, a coflow-aware routing scheme to dynamically trim coflow completion times in software-defined datacenter networks. Software-defined networking (SDN) fits into such a scenario naturally due to its global visibility and centralized control over the network. The logically centralized controller builds up a real-time view of the network, and therefore is able to locate the current bottleneck flow of a coflow. Leveraging such information, the controller makes globally optimal routing decisions. Since the bottleneck of a coflow is likely to be an elephant flow with sizes in the MB to GB range [12], the delay between the controller and switches while updating forwarding rules has little influence. All the flows follow default routing using ECMP (Equal Cost Multipath) initially, and only the flows on a bottlenecked link would be influenced by the dynamic routing procedure.

The highlight of *Tailor* revolves around its effectiveness

The research was supported in part by grants from RGC under the contracts 615613 and 16211715, a grant from NSFC and Guangdong joint project under the contract U1301253, and the NSERC Strategic Networks grant titled "Smart Applications on Virtual Infrastructure."

978-1-5090-2279-3/16/$31.00 ©2016 IEEE

in trimming coflow completion times without involving complicated per-flow rate control. To characterize the essence of such coflow awareness, we first formulate the problem of minimizing coflow completion times as a maximum concurrent flow problem, and solve this problem by means of dynamic routing. Second, by leveraging the global view of the completion status of all live flows, we take advantage of the controller to locate the bottleneck of each coflow, which simplifies the modifications to both the switches and end-hosts. Compared to complex switches needed in explicit rate control schemes (*e.g.*, [5]), switches in *Tailor* are only responsible for forwarding and monitoring. Finally, we design a simple yet effective routing protocol that converges to the optimal routing assignment. Our experimental results in both Mininet and large-scale simulations have clearly shown that *Tailor* enjoys the advantages of coflow-aware routing while keeps the system simple and efficient at the same time.

II. The Spotlight Shifts from Flows to Coflows

A. Background

Two characteristics are critical in current datacenter networks. First, typical datacenters, using topologies such as Fat tree [13] or DCell [14], strive to provide full bisection bandwidth and avoid congestion at core switches. Congestion at access and aggregate switches, however, is still common due to imbalanced network load and improper virtual machine (VM) placement. Second, there usually exist multiple equal cost paths from sources to destinations. A load balancing scheme could distribute flows evenly in general, but it is still possible that flows belonging to the same coflow happen to be placed into the same set of paths. A coflow-aware scheme is needed to better handle such inefficiency.

Data-parallel applications, running on top of MapReduce-like frameworks (*e.g.*, Hadoop [15], Pregel [16] and Dyrad [17]) and in-memory computation engines (*e.g.*, Spark [18]), usually involve multiple computation stages in a single job. The intermediate results have to be transferred from machines that run tasks in the upstream stage to the machines that run tasks in the downstream stage [7]. Such transfers have significant impact on the user-perceived performance.

From a network perspective, a coflow is defined as a group of *concurrent* flows that transfer intermediate results among different stages of a specific job. There usually exists a barrier between different stages of a job [7], which means the next stage of computation starts only after it has received all the data from the previous stage. Data pipeline [19] only avoids data transmission within a single stage, while the final computation result still cannot be obtained until receiving all the input data. To sum up, the completion time of the slowest flow in a coflow, instead of the average completion time of all flows, determines application performance and should be brought into the spotlight.

B. Motivating Example

We reiterate the significance of embracing both coflow-awareness and dynamic routing through the following example. In Fig. 1: the three flows of a coflow all go through the

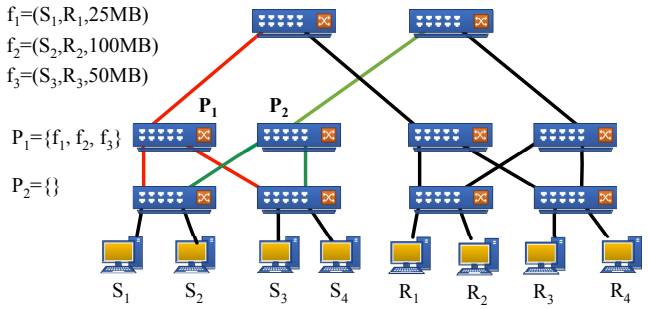

Fig. 1. A motivating example: all the links in the network are of 1 Gbps. The completion time of a coflow can be reduced if its bottleneck flow can be dynamically rerouted, *e.g.*, routing f_1 to P_2. The completion time of this coflow could be further reduced, if the currently slowest flow f_2, instead of f_1, could be rerouted from P_1 to P_2.

egress port P_1 based on the default routing, whereas P_2 is currently idle. It is clear to see that no matter what scheduling strategy is used ([6], [5], [4], [8]), the completion time of the slowest flow would be $\frac{100+50+25\ \text{MB}}{1\ \text{Gbps}} = 1400$ ms. With dynamic routing, we can utilize the idle bandwidth on P_2 to further speed up the coflow. Suppose flows on a link equally share the remaining bandwidth. If we first reroute f_1 and then f_3 to P_2, the completion times of the three flows will be

$$200 = \frac{25\ \text{MB}}{1\ \text{Gbps}}, \quad 900 = 200 + \frac{(100 - 25 * 0.5)\ \text{MB}}{1\ \text{Gbps}}$$
$$500 = 200 + \frac{(50 - 25 * 0.5)\ \text{MB}}{1\ \text{Gbps}} \tag{1}$$

Alternatively, if we reroute f_2, the completion times will be

$$400 = \frac{25\ \text{MB}}{1\ \text{Gbps}}, \quad 800 = \frac{100\ \text{MB}}{1\ \text{Gbps}}$$
$$600 = 400 + \frac{(50 - 25)\ \text{MB}}{1\ \text{Gbps}} \tag{2}$$

It is clear to see that the average flow completion time is minimized in the first solution, whereas the coflow completion time is minimized in the second solution. Through this example, we have two key observations: 1) under static routing, coflow performance is usually suboptimal; 2) minimizing the average flow completion time is neither sufficient nor necessary for optimizing coflow performance.

C. Problem Formulation

Unlike the computation and storage resources which can be statically allocated to different VMs, the network resources are shared by multiple active coflows, and henceforth coflows may influence the performance of each other. Consequently, inter-coflow scheduling must be carefully tackled, such that each coflow can get reasonable amount of bandwidth. We adopt a priority-based inter-coflow scheduling mechanism since it is proved to effectively reduce the average coflow completion times [10] [8]. As our main focus is to coordinate flows within the same coflow (*i.e.*, intra-coflow scheduling), we do not restrict our algorithms to any specific priority assignments (*e.g.*, FIFO [10], smallest effective bottleneck first [8]). Instead, our intra-coflow scheduling can be built upon any priority-based inter-coflow scheduling mechanism.

Formally, we define a coflow C_k consisting of n flows as $C_k = \{f_i, i = 1, 2, ..., n\}$. Denote the completion time of a flow f_i as t_i. The average and longest completion times of all the flows within C_k, denoted as t_a and t_c are shown in Eq. (3).

$$t_a = \frac{1}{n} \sum_{f_i \in C_k} t_i, \qquad t_c = \max_{f_i \in C_k} t_i \tag{3}$$

Our objective is to minimize the completion time of a coflow, namely t_c, by scheduling and coordinating the flows within this coflow. Given a physical network $G = (V, E)$, where each link $l = (u, v) \in E$ has a capacity constraint. Each flow from a sender to a receiver can then be viewed as a commodity in the flow network. $f_i \in C_k$ is identified through a 3-tuple: (s_i, r_i, d_i), where s_i is the source (*i.e.*, the sender) of this flow, r_i is the sink (*i.e.*, the receiver) of this flow and d_i is the total amount of data to be transferred. The problem to minimize the completion time of a coflow can be formulated as shown in Eq. (4) - Eq. (6).

$$\min \max_{1 \le i \le n} \frac{d_i}{\sum_{w \in V} f_i(s_i, w)} \tag{4}$$

$$\text{s.t.} \quad \sum_{f_i \in C_k} f_i(u, v) \le B_l^k, \forall\ l = (u, v), \tag{5}$$

$$\sum_{w \in V} f_i(u, w) = \sum_{w \in V} f_i(w, u), \forall u \ne s_i, r_i. \tag{6}$$

B_l^k in Eq. (5) indicates the *remaining* bandwidth that has not been used by coflows with higher priorities on the link l. This inequality essentially corresponds to the capacity constraints of a flow network. $f_i(u, v) \ge 0$ is the bandwidth f_i gets on link l, and the denominator in Eq. (4) thus represents the total bandwidth allocated to f_i. It is obvious that the fraction in Eq. (4) is a flow's completion time, which is the reciprocal of the objective function in the maximum concurrent flow problem. Therefore, minimizing the maximum flow completion time in a coflow is equivalent to a NP-hard *maximum concurrent flow problem*.

III. Challenges and Design Insights

From our problem formulation, we can see that the completion time of a flow can be trimmed by controlling how much bandwidth it could get and which path it should go through. We next analyze the challenges to design such coflow-aware scheduling and routing strategies.

A. Challenges

The lower bound of both the average and longest completion time of flows in a coflow C_k is given by

$$t_c \ge t_a \ge \frac{\sum_{f_i \in C_k} d_i}{\max(\sum_{l=(s_i,v)} B_l^k, \sum_{l=(u,d_i)} B_l^k)} \tag{7}$$

This lower bound can be achieved when the aggregated remaining bandwidth of access links can be arbitrarily shared by all the flows in a coflow. If we adopt weighted fair sharing, and assign the size of a flow as its weight, all the flows of a coflow can complete at the same time as shown below (B_k is the denominator in Eq. (7)):

$$\forall\ f_i, f_j \in C_k, \frac{d_i}{B_i} = \frac{d_j}{B_j} = \frac{\sum_{f_i \in C_k} d_i}{B_k} \tag{8}$$

where B_i is the average bandwidth f_i gets throughout its lifetime. In this case, both the average and longest completion time of flows in C_k reach the lower bound.

Nevertheless, the placement of senders and receivers restricts the bandwidth sharing within a coflow. Even if the placement of VMs is balanced, the dynamic workloads of concurrent coflows make it non-trivial to achieve complete load balancing in practice. Given the restriction of VM placement and the distribution of network load, finding the optimal concurrent flow in a network is NP-hard since flows are not infinitely splittable [20]. In the following, we discuss the design of scheduling strategies under the static routing scenario. For networks where dynamic routing are allowed, we analyze the potential benefit of such support.

B. Design Insights

1) Optimality with Static Routing: Multipath routing mechanisms have been adopted in datacenters for better load balancing and higher bandwidth utilization. Under such mechanisms, however, a flow still takes one static path based on the information of its packet header. Throughout its life time, a flow is forbidden to take other paths. In this case, even if a link is underutilized, flows that do not go through this link cannot benefit from such idle bandwidth. Therefore, we can only schedule flows on a single link, rather than across the entire network. Under such circumstances, existing coflow scheduling schemes (*e.g.*, Orchestra [7], Varys [8] and Aalo [9]) assign the weight of a flow proportional to its size such that all the flows in a coflow finish at the same time. This strategy is proved to be optimal for a single coflow under static routing [7].

2) Optimality with Dynamic Routing: To fully utilize the potential advantage of dynamic routing, we need to have up-to-date information of the idle bandwidth on each link. Furthermore, one needs to carefully examine the change of completion times after a bottleneck flow is rerouted. These two requirements lead us to leverage the global visibility and central control of SDN to conduct network-wide monitoring, computing and routing. The controller in a datacenter network periodically collects statistic information of all ports on each switch and decides which flow should be re-routed to which path. Furthermore, if flows can take multiple paths concurrently, the network load would be more balanced, and hence coflows can probably finish faster. Nevertheless, the arriving packets of a flow would be out of order severely, incurring frequent false detection of network congestion. Therefore, we only focus on dynamic routing for indivisible flows, and discuss the possible extensions for splittable flows in Sec. VI. Although Rapier [11] also relies on SDN for better performance, their controller has to decide the path and rate for every single flow in the network, which not only incur too much communication traffic for deploying forwarding rules, but also increases the time consumed by the decision procedure.

IV. Design

As we have analyzed previously, dynamic routing can reduce the completion time of a flow by rerouting it to a path with lighter load. Nevertheless, existing routing protocols, such as Hedera [3], restrict themselves to flow-level performance and treat all the flows in a coflow equally. This is insufficient for trimming the completion time of a coflow. Instead of routing f_1 to P_2, we reroute the currently slowest flow f_3 to P_2. The longest completion time of these flows is reduced to 800 ms, which is 100 ms shorter than the completion time of the same coflow in Fig. 1.

To effectively trim the completion time of a coflow, we need to know the load of each link across the network and spot the slowest flow of a coflow throughout the datacenter. Therefore, a global view of the network state is necessary since each single switch merely possesses local information and makes routing decisions accordingly. Therefore, we leverage SDN to tackle this problem. A central controller in a datacenter network builds up a logical view of the network through the periodical sampling provided by network monitoring tools such as sFlow-RT [21]. Based on the global information, the controller makes globally optimal routing decisions. Minimizing the completion time of a coflow can be formulated as a NP-hard maximum concurrent flow problem. With the presence of a central controller, it is possible to implement a constant-factor approximation algorithm to guide the routing of flows [20]. Nevertheless, such an algorithm involves multiple computation steps, while only guarantees $(3.23 + o(1))$-approximation. In contrast, the protocols of *Tailor* use simple yet effective heuristic algorithms, avoiding unnecessary complexity of the controller.

A. Components of Tailor

When a sender initiates a new flow, it reports the meta information of this flow, including the flow name *(srcip, srcport, dstip, dstport, protono)* and the estimated flow size to *Tailor*. Flow sizes could be *predicted* by mining application logs [22] and could be dynamically *adjusted* to approximate the real values. If switches already know how to route this flow based on the default forwarding rules that the controller has pushed to them, this flow will transmit without incurring control overheads. Otherwise, the controller will receive a packet-in message, and performs the following steps sequentially: 1) assigns an ECMP route to the flow by hashing on its meta information, 2) updates the path attribute in the flow information table; 3) installs new rules to the corresponding switches along the selected path.

To enable dynamic routing, we adopt a real-time monitoring system to collect the statistic information of switches through the sFlow protocol [21]. The information regarding the states of flows in a coflow, such as the volume of data a flow has sent and its current rate, is stored in a flow database. The information about the states of switches, such as the total amount of data that each port has forwarded, is stored in a switch database. *Tailor* periodically queries the sFlow collector from time to time leveraging the RESTful HTTP API to poll these two databases. *Tailor* can thus dynamically locate the bottleneck flow of a given coflow. It then revokes the dynamic routing algorithms as described in Algorithm 1 and Algorithm 2. Such routing decisions are then passed to the network controller, which communicates with switches in the data plane to install new forwarding rules through the OpenFlow protocol [23].

B. Algorithms of Tailor

In the following, we present the routing protocols in *Tailor*. We first work through a rerouting procedure and present our first dynamic routing protocol called *Simple Tailor*.

Simple Tailor periodically queries the flow database to estimate the remaining completion time of each flow within a coflow. Once located the slowest flow f_0, it first searches for all the alternative paths for this flow. Denote the set of paths for the sender-receiver pair of f_0 as $P_{s_0 \to r_0}$ and the set of flows within a coflow C_k going through a link l as F_l^k. The most intuitive way to speed up the slowest flow of a coflow is rerouting it to a path whose bottleneck link has more bandwidth than f_0 currently gets. We present the detail of such a routing strategy in Algorithm 1. The algorithm first records all the candidate paths in line 3-5, then selects the path with the most available bandwidth to reroute the bottleneck flow.

Algorithm 1 Simple Tailor

1: **procedure** CANDIDATE(f_0, p_0)
   ▷ find all candidate paths with larger bandwidth
2: **for** $p_c \in P_{s_0 \to r_0}$ **do**
3: $B_c = \min_{l \in p_c}(B_l^k - \sum_{f_i \in F_l^k,} B_i(t))$;
4: $B_0 = B_0(t)$;
5: **if** $B_c > B_0$ **then** $\mathcal{P} = \mathcal{P} \cup \{p_c\}$
return a set of capabale paths $\mathcal{P}$
6: **procedure** MAIN
   ▷ p_i is the path f_i currently takes.
7: Sort $f_i \in C_k$ in the decreasing order of t_i.
8: **Denote** $p_c = \arg\max_{p \in \text{CANDIDATE}(f_0, p_0)} B_p$
9: Reroute f_0 to p_c ▷ reroute the bottleneck flow f_0

We verify Simple Tailor's performance through the preliminary results shown in Fig. 2.

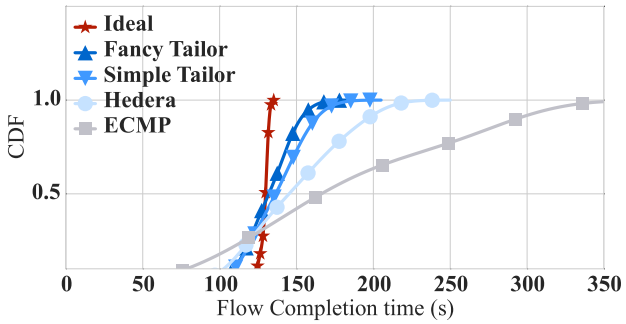

Fig. 2. The distributed functions of flow completion times within a coflow to compare Tailor with existing schemes. The coflow completion time under Simple Tailor is 205 seconds, which is 45 seconds faster than Hedera. Fancy Tailor further expedites the coflow by 30 seconds.

It is clear to see that *Simple Tailor* significantly outperforms Hedera and ECMP in terms of the coflow's completion time.

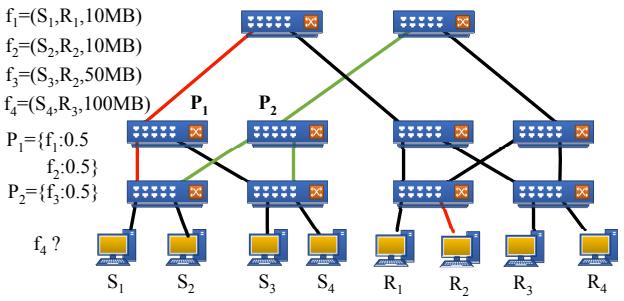

Fig. 3. An example to illustrate the limitation of *Simple Tailor*: an incoming flow f_4 would choose P_2 since it has more available bandwidth, but it would add too heavy burden on P_2, aggravating the imbalance of network load.

Although this strategy is effective and straightforward, it neglects a fact that the flows belonging to the same coflow on a link would equally share the available bandwidth without explicit rate control. To better understand the limitation of this strategy, consider the situation shown in Fig. 3: a newly-initiated flow f_4 has to go through P_1 or P_2, and it is easy to identify that f_4 would become the bottleneck given its large volume. Based on *Simple Tailor*, f_4 would be routed to P_2 since there is no idle bandwidth on P_1. Nevertheless, this routing decision would exacerbate the already imbalanced network load: by allocating the underutilized bandwidth share on P_2 to f_4, its rate is 0.5 Gbps before f_3 finishes. As a result, the coflow completion time would be as long as 1200 ms.

To avoid the limitation of *Simple Tailor*, we propose a new routing protocol called *Fancy Tailor* which takes the path load into consideration, as shown in Algorithm 2.

Algorithm 2 Fancy Tailor

1: **procedure** FEASIBLE(f_0, p_0)
   ▷ find all paths whose flows finish earlier than p_0.
2: **for** $p_c \in P_{s_0 \to r_0}$ **do**
3: $t_o = \max_{l \in p_0, f_i \in F_l^k} \frac{d_i(t)}{B_i(t)}$, $t_c = \max_{l \in p_c, f_i \in F_l^k} \frac{d_i(t)}{B_i(t)}$
4: **if** $t_c < t_o$ **then** $\mathcal{P} = \mathcal{P} \cup \{p_c\}$
   return $\mathcal{P}$
5: **procedure** MAIN
6: **Sort** $f_i \in C_k$ in the decreasing order of t_i.
7: **Denote** the most congested link of p_0 as l_0.
8: **if** Opt **then** $F_0^k = F_{l_0}^k$
9: **else** $F_0^k = \{f_0\}$ ▷ only reroute the bottleneck flow
10: **for** $\forall f_j \in F_0^k$ **do**
11: **Denote** $p_c = \arg\min_{p_c \in \text{FEASIBLE}(f_j, p_j)} t_c$
12: $\overline{F}_l^k = F_l^k \cup \{f_j\}$
13: $t_o = \max_{l \in p_j, f_i \in F_l^k} \frac{d_i(t)}{B_i(t)}$, $t_c = \max_{l \in p_c} \frac{\sum_{f_i \in \overline{F}_l^k} d_i(t)}{B_l^k}$
14: **if** $t_c < t_o$ **then** Reroute f_j to p_c;

We capture the load of a path by introducing a concept called *path completion time*, which is defined as the maximum completion time of flows belonging to a coflow on the bottleneck link of the path. Following such a definition, a path p_c is called feasible if it is one of the several equal cost paths for f_j, and its completion time is smaller than that of p_j, the path f_j currently takes. By default, f_j is the bottleneck flow, which indicates we only dynamically route the bottleneck flow. Once acquiring all the feasible paths, we select a new path, p_c, with the smallest completion time. Tailor then estimates the completion time of p_c after a possible rerouting in line 13 of Algorithm 2. Only if the completion time of p_c is smaller than p_j's original completion time would the rerouting be conducted. Line 13 indicates that C_k can preemptively acquire the bandwidth that have been allocated to coflows with lower priorities and use up B_l^k. Following *Fancy Tailor*, f_4 would be routed to the egress port P_1 since its completion time is shorter then that on port P_2 before rerouting; the completion time of this coflow could reduce to 960 ms consequently.

Since the condition in line 14 of Algorithm 2 guarantees the completion time of the newly selected path is smaller than the original path, it is impossible to reroute a flow on p_c back to p_j. In this way, the dynamic routing algorithm avoids oscillation. It is possible that a flow on a path other than p_j and p_c becomes the new bottleneck after rerouting, but the completion time of the new bottleneck flow is guaranteed to be smaller than that of the original bottleneck flow before rerouting. Therefore, the completion time of a coflow decreases monotonically.

However, it is possible that rather than re-routing the bottleneck flow itself, it would be more beneficial to route some flows on the congested link of the path that the bottleneck flow currently takes. We further design an optional feature (line 8) which allows reroute every flow on the congested link that the bottleneck flow runs through. This procedure guarantees the completion time of a coflow keeps decreasing until the lower bound is reached. It is worth noticing that to discover all the candidate paths of a given flow, *Tailor* needs to examine the state of each link along a candidate path. This requires querying all the switches related to a flow. In a k-ary fat-tree topology, the number of switches involved is no more than $\frac{k^2}{4} + k$ [13]. Therefore, we only reroute the bottleneck flow in practice to ensure the computation overhead can be easily offset by the performance gain. In Fig. 2, it is clear to see that Fancy Tailor further reduces the coflow's completion time by 30 seconds in comparison with Simple Tailor.

Along with the benefits of rerouting comes the out-of-order delivery problem. Since a flow no longer follows a single path throughout its lifetime, a packet sent later could follow a faster path and arrive the destination earlier than its precedents. Fortunately, out-of-order packets only occur in the short transient period when the flow path changes, which should be kept as infrequent as possible. In addition, adopting reorder buffers at receivers and increasing TCP receive window can mitigate the negative effects of slight out-of-order deliveries.

C. Properties of Tailor

1) Simple funcationalities in switches: In *Tailor*, the control functionality of switches is minimized. Each switch in the network establishes a secure channel to the central controller which is in charge of any control decision. Switches are only responsible for forwarding and monitoring. Per-flow state management and computation are unnecessary in switches. With respect to forwarding, once receiving a packet from

a flow, a switch searches its flow table for corresponding rules. If a match is found, it forwards this packet to the assigned port. Otherwise, a packet-in event is triggered, and the packet is sent to the controller for further processing. The controller computes a suitable path for this flow, and then configures corresponding switches along the path. With respect to monitoring, every switch maintains the statistics of its ports and flows passing through each port, such as total byte counts and durations of flows. The available bandwidth of each port can be computed through comparing the difference of total byte counts sampled in consecutive periods by sFlow agents. Similarly, the actual rate of a flow is computed. A flow entry in the databases is deleted when its remaining size equals to zero, or it is inactive for a certain period of time.

2) Minimal involvement of endhosts: In private datacenters, the modification of end-hosts' protocol stack is possible. Nevertheless, in cloud datacenters where many tenants run their own applications, a transparent implementation is preferred. Therefore, instead of integrating the extra flow information into packets, we rely on a dedicated database server to collect the information of all flows in a coflow. sFlow agents periodically sample the switch counters and update the remaining size of each flow to the server. In this way, endhosts only need to send an initial message to the server, while the following process remains unchanged. If we can arbitrarily modify the network stack of end-hosts, the coflow ID and the remaining flow size can be piggybacked in packet headers. Switches can extract such information and report to the real-time monitor directly. In addition, rather than merely relying on TCP's AIMD congestion control, the assistance from endhosts can help to improve the agility of congestion control. The controller can notify a sender the proper rate it should send data at either in the ACKs sent from the receiver, like EyeQ [24], or through a separate rate control protocol, like SRQ [10]. These techniques can be integrated into our mechanism to further improve the network performance. *Tailor* requires no change of application codes except posting the flow digest to the controller before transmission.

V. Evaluation

We evaluated *Tailor* through a set of experiments both on our Mininet-based emulator and flow-level simulator. Through analyzing extensive experiment results, we demonstrate that *Tailor* outperforms existing flow-level protocols and scheduling-only coflow schemes in terms of coflow completion times. *Tailor* achieves similar performance with Rapier [11] with much simpler algorithms.

Schemes Compared: we compare *Tailor* with the following schemes in terms of coflow performance.

- **Baseline**: all the flows are statically routed using ECMP;
- **Routing-only (Hedera)**: dynamically reroute large flows using the global first fit routing algorithm [3];
- **Scheduling-only (Varys)**: statically route flows using ECMP but dynamically decide flow rates based on the smallest-effective-bottleneck-first strategy [8];
- **Joint-optimization (Rapier)**: combine routing and flow scheduling according to the approximate solution to Problem (4) [11].

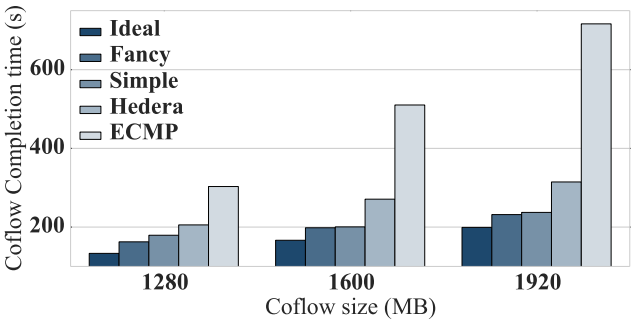

Fig. 4. Coflow performance for shuffle traffic under different routing protocols: the amount of data one host sends to each receiver ranges in [20MB, 30MB]. *Tailor* outperforms Hedera by 24.69% on average, and surpasses ECMP significantly.

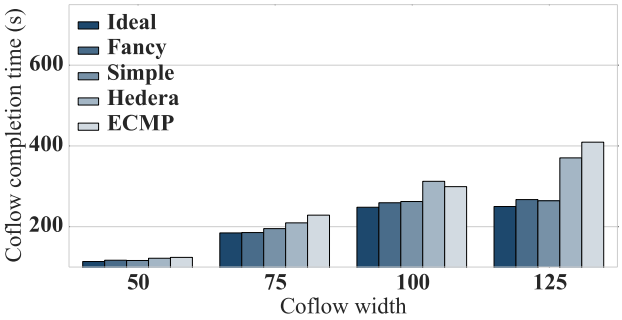

Fig. 5. Coflow performance for random traffic under different routing protocols: 80% of flows with sizes ranging from 10KB to 1MB; 10% of flows with sizes ranging from 10MB to 100MB. The advantage of Tailor is less significant compared with shuffle communication, since there are fewer large flows in the network.

Communication Patterns: We generate synthetic traffic based on realistic traffic distributions found in datacenter networks [10], [8], and use the same traffic for each group of experiments. We emulate two types of communication patterns among end-hosts for comprehensive evaluations:

- **Random communication**: an end-host initiates a flow to any other host in the network with equal probability. Following the heavy-tailed distribution observed in datacenter networks [25], we generate the synthetic traffic such that 80% of the flows in a coflow are mice flows whose sizes are smaller than a threshold, and 10% of the flows are elephant flows with sizes larger than the given threshold.
- **Data shuffle (many-to-many)**: a host sends to another host under the same switch with probability P_e, and to its same pod with probability P_p, and to the rest of the network with probability $1 - P_e - P_p$. Each sender sends a fixed amount of data to all its receivers. Under this communication pattern, all the flows of a coflow are of the same size.

A. Implementation and Emulations

We implement our coflow-aware scheduler as a module in the controller (Pox [26]) connected to Mininet [27]. We conduct emulation on a server equipped with a four-core eight-thread 2.8 GHz Intel Xeon E5-1410 CPU and 8GB of RAM.

Tailor runs in a virtual machine with 6 virtual cores and 6GB of RAM on that server. OpenvSwitches are deployed and configured to connect to the sFlow monitor for information sampling. The controller conducts polling on network states with the help of sFlow every 10 seconds. We use a $k = 4$ Fat-tree [13] topology in our emulation, and set the bandwidth of each interface to 10 Mb/s owing to the hardware restriction of our machine, *esp.*, the computing power of the CPU. In accordance with the link capacities, we set the sizes of flows in the network ranging between 10KB and 100MB. Each coflow consists of 64 flows.

We implement ECMP by hashing on flow's 5-tuple, and extend an implementation of Hedera on Mininet [28] by enabling Hedera to accept the traffic we generate as input and generate the completion time of each flow as output. We further use the performance under an ideal network where the ingress/egress bandwidth of every switch in the Fat-tree network is infinite as a benchmark. We do not evaluate the performance of Varys and Rapier for the emulation experiments.

The overall emulation results are shown in Fig. 5 and Fig. 4. In general, both Simple Tailor and Fancy Tailor outperform Hedera and ECMP significantly in all scenarios. Compared to Hedera, the performance gain of Fancy Tailor in terms of coflow completion time reduction is 14.61% on average in Fig. 5. By further comparing the Simple and Fancy Tailors, we find the coflow completion times of Fancy Tailor further drop 10.04% compared to the Simple Tailor on average, and are within several percent of the ideal performance.

Since all the flows in a shuffle coflow have are relatively large, the network is more congested than in random communication pattern. Under such circumstance, it is more likely for multiple large flows to conflict. The advantage of *Tailor* is more significant as shown in Fig. 4: the coflow completion times of *Tailors* are 24.69% less than Hedera. Since ECMP treat all the flows equally without considering their sizes, it performs very poorly under shuffle communication: the coflow completion times of ECMP are $2-3\times$ of Tailor.

B. Large-scale simulations

Simulation Methodologies: Since packet-level network simulators, such as ns-3 [29], are inefficient [8], we write our own event-driven simulator like in [8], [3], [11]. For the simulations, we set k=16 such that there are 1024 end-hosts connected by the fat-tree network. The bandwidth of each network interface in the network is 1 Gbps. There are 64 coflows in the network, whose priorities are determined by the arrival times in Tailor. For many-to-many communication, each coflow consists of 64 flows with equal sizes. For random communication, each coflow has the same number of constituent flows, but the size of each flow follows a long-tail distribution. We adopt per-coflow queue at each switch to ensure strict priorities among different coflows. We only use the Fancy Tailor for simulations.

Impact of Coflow Width: we vary the number of flows in a coflow (*i.e.*, its width) and evaluates the performance of different schemes under the random communication pattern. When the coflow width is small, the network is lightly loaded due to the fixed number of coflows. The superiority of coflow-aware scheme is indistinct. As the network traffic increases, the difference between coflow-aware and flow-level schemes becomes significant.

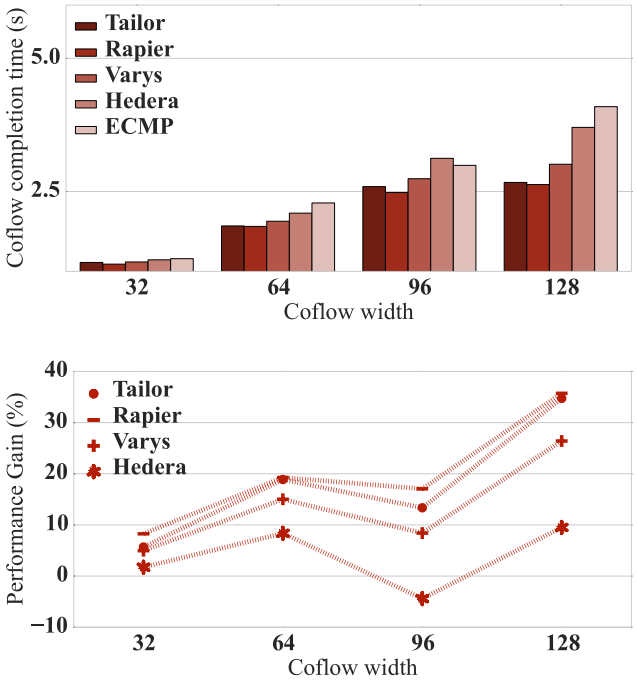

Fig. 6. Coflow performance for random traffic under different coflow schemes: the flow sizes range in [10KB, 100MB]. *Tailor* outperforms Varys in all cases, and the advantage increases with the coflow width. Tailor only achieves similar performance gain with Rapier, but it is much simpler compared to Rapier's per-flow routing and rate limiting.

As shown in Fig. 6, compared to the baseline, Tailor, Varys and Rapier reduces coflow completion times by 34.75%, 26.40% and 35.72% respectively when the coflow width is 128. In addition, it is worth noticing that the completion times of coflows under Hedera are sometimes even longer than the static ECMP routing. The reason for such phenomena is that Hedera equally treat all the flows with rates larger than a given threshold by rerouting them to feasible paths. Although this strategy succeeds in improving the network throughput, which equivalently minimizes the average completion time of flows, it fails to spot the bottleneck flow of a coflow. Henceforth, it is possible in Hedera that the slowest flow gets starved for a long time before it acquires sufficient bandwidth.

Impact of CoFlow Size: we further vary the total amount of traffic of a coflow (*i.e.*, its size) by adjusting the size of each flow in a shuffle coflow.

Apparently, the advantages of coflow-aware routing schemes (*i.e.*, Tailor and Rapier) are much more significant under many-to-many communication, which is consistent with the results we get through emulations. However, the coflow-aware rate limiting (*i.e.*, Varys) does not improve coflow performance over flow-level schemes very much. The reason is that when network bandwidth is insufficient, the key to coflow performance is the selection of routing paths, rather than the rate limiting of flows along a fixed path.

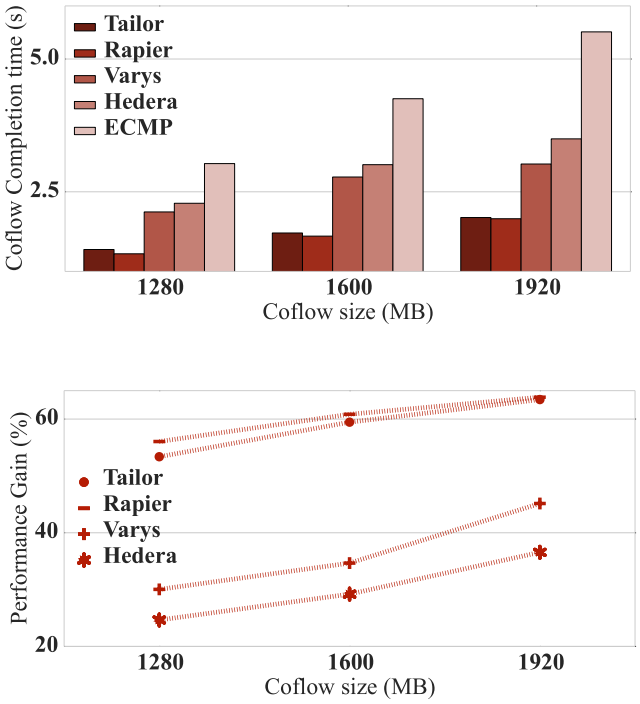

Fig. 7. Coflow performance for shuffle traffic under different coflow mechanisms: the amount of data one host sends to each receiver ranges in [20MB, 30MB]. The advantage of Tailor and Rapier over Varys is much more obvious due to the more congested network under many-to-many communication, which demonstrates the effectiveness of dynamic routing.

Under all circumstances, Tailor outperforms Varys significantly due to its capability of selecting better paths for the bottleneck flows. The performance of Tailor and Rapier is similar. However, Tailor achieves such performance gain with much simpler routing algorithms. As analyzed in Sec. IV-C, Tailor does not require sophisticated rate control mechanism at end-hosts such that we do not need to modify the server's kernel as Rapier does. Furthermore, Rapier needs a third-party LP solver to find the approximate solution to the coflow scheduling problem, which cannot guarantee a good approximate ratio. In contrast, Tailor asymptotically reaches the best path selections through dynamic routing and ensures strict priority through per-coflow queue.

C. Overhead

The control overheads stem from two modules in *Tailor*: the dynamic routing module in the controller and the monitoring module of sFlow. These two modules are decoupled, and their communication is conducted via standard HTTP APIs. Both modules bring about computation overheads and communication overheads. With respect to routing, the controller periodically spots the slowest flow and reroutes it if applicable. The selection of the slowest flow is essentially a traversal of all the ongoing flows, which can be done in $O(n)$ time, where n is the coflow width. However, finding a new proper path for that specific flow requires traverse all the possible paths, and then select the most suitable one. To reduce such overhead, *Tailor* periodically caches all the possible routes of a certain *(srcip, dstip)* pair. Also, updating each forwarding rule would generate 72B traffic [23]. Since there are only decades of coflows simultaneously running in the network [8], such overhead can be effectively offset by performance gain.

With respect to the real-time monitoring, sFlow is a quite mature protocol with the support of merchant silicons from many vendors. The computation overhead caused by packet sampling at switches is ignorable in practice. For the communication overhead, we examine the traffic generated between switches and the sFlow central collector. In a Fat-tree topology, there are $\frac{5}{4}k^2$ switches and $\frac{5}{4}k^3$ interfaces. Therefore, the sampling traffic in the whole network during a polling interval can be computed as 1MB. Compared to the large traffic volume in datacenter networks, the communication overhead is quite small. To further reduce the overhead, the sampling rate and polling interval could be dynamically tuned.

VI. Extensions of Tailor

Integrating with task schedulers: Currently, we try to improve coflow performance given the senders and receivers of flows. In other words, we assume a task distributor (scheduler), (*e.g.*, Sparrow [30]) has already select machines for each computation task of a specific job. Nevertheless, it is the job completion time, not the coflow completion time, that directly influences user-perceived performance. We should integrate task distribution and coflow scheduling to jointly improve job performance. We leave such a comprehensive scheme for future work.

Routing for splittable flows: In order to minimize the number of out-of-order packets, *Tailor* only reroutes a flow as a whole. To support routing for splittable flows, both the congestion control algorithms at endhosts and the routing algorithm in the controller need to change. With respect to endhosts, receivers can adopt reorder buffers, which are responsible for reordering scrambled packets before delivering the packets to upper layers in the network stack. Alternatively, receivers can increase the receive window in TCP segment and avoid the false detection of congestion caused by slightly out-of-order packets. As for the controller, it needs to select several candidate paths and determines the portion of data a flow sends to each path. We can extend *Tailor* to support splittable flows by searching for k candidate paths instead of merely looking for one. The amount of data routed to each path should be proportional to the available bandwidth on that path.

VII. Related Work

Flow scheduling: The literature of network resource allocation has largely focused on scheduling of each individual flow (*e.g.*, [6] [31] [5]), aiming at minimizing the average flow completion times. Since they have not taken the coflow semantics into consideration, bandwidth might be wasted to flows that can already finish ahead of the other flows within the same coflow. This potentially slows down other flows and thus slows downs the coflow as a whole. Instead of merely prioritizing latency-sensitive flows, Hedera [3], MPTCP [32] and Conga [33] improve the overall performance by distributing network load more uniformly leveraging dynamic routing, flow dividing and switch upgrading respectively. Again, these

methods restrict themselves to flow-level performance, and thus cannot improve coflow-level efficiency as *Tailor* does.

Coflow scheduling: Chowdhury *et al.* in [7] propose a global control architecture, Orchestra, to conduct weighted bandwidth sharing in both intra- and inter- coflow level. Baraat [10] aims to reduce the average coflow completion times by using FIFO-LM inter-coflow scheduling. Varys [8] further improves coflow scheduling by combining priority-based inter-coflow scheduling (*i.e.*, smallest effective bottleneck first) and weighted sharing among flows within a coflow. Aalo [9] adopts the generalized *least serviced first* heuristic to avoid collecting coflow information in advance. These methods, however, have not utilized the benefit of multipath topologies in typical datacenter networks. Restricted to a static path throughout a flow's lifetime, the scheduling is merely effective for unipath networks.

Identifying such inefficiencies, Zhao *et al.* propose Rapier [11], a centralized scheduler to conduct both scheduling and routing. Despite the similar architecture with *Tailor*, their per-flow scheduling and routing rely on solving an ILP optimization problem. By controlling every single flow within the network, the controller is likely to crash by millions of control messages, and switches are overwhelmed by tremendous forwarding rules. In contrast, *Tailor* only deals with the bottleneck flow of a given coflow, the majority of fast and small flows follow their default routes. Exempting per-flow rate limiting, *Tailor* has successfully minimized the modification of endhosts, making the routing scheme transparent to datacenter tenants.

VIII. Conclusion

In this paper, we have proposed and studied the problem of minimizing the completion times of coflows in software-defined datacenters. Through an in-depth analysis of coflow-aware scheduling and routing, we find that the the global visibility and centralized network control available in software-defined networking are important for better coflow performance. Based on this observation, we have designed and implemented *Tailor*, a coflow-aware routing mechanism, which dynamically identifies and reroutes the bottleneck flow of a given coflow. The rerouting process guarantees that the completion time of a coflow decreases monotonically and converges to the lower bound. The experiment results both on our Mininet-based prototype and simulations have demonstrated that *Tailor* reduces the coflow completion times significantly achieves much better coflow performance with little overhead.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. USENIX OSDI*, 2004.

[2] "Latency is everywhere and it costs you sales," http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. USENIX NSDI*, 2010.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proc. ACM SIGCOMM*, vol. 41, no. 4, 2011, pp. 63–74.

[5] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proc. ACM SIGCOMM*, vol. 42, no. 4, 2012, pp. 127–138.

[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," in *Proc. ACM SIGCOMM*, 2013.

[7] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," in *Proc. ACM SIGCOMM*, vol. 41, no. 4, 2011, pp. 98–109.

[8] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient Coflow Scheduling with Varys," in *Proc. ACM SIGCOMM*, 2014.

[9] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM SIGCOMM*, 2015, pp. 393–406.

[10] F. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. ACM SIGCOMM*, 2014.

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

[12] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proc. the VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, 2012.

[13] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, vol. 38, no. 4, 2008, pp. 63–74.

[14] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *Proc. ACM SIGCOMM*, vol. 38, no. 4, 2008, pp. 75–86.

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD*, 2010.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[18] 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 *Proc. USENIX NSDI*, 2012.

[19] "Google Cloud Platform: App engine pipeline API," https://github.com/GoogleCloudPlatform/appengine-pipelines.

[20] T. Leighton and S. Rao, "Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms," *Journal of the ACM (JACM)*, vol. 46, no. 6, pp. 787–832, 1999.

[21] "sFlow," http://www.inmon.com/products/sFlow-RT.php.

[22] Y. Peng, K. Chen, G. Wang, W. Bai, Z. Ma, and L. Gu, "Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing," in *Proc. IEEE INFOCOM*, 2014, pp. 19–27.

[23] 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 Communnication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[24] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical network performance isolation at the edge," in *Proc. USENIX NSDI*, 2013.

[25] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM conference on Internet measurement (IMC)*, 2010.

[26] "Pox," https://github.com/noxrepo/pox.

[27] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proc. ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets)*, 2010.

[28] "Hedera on Mininet," http://www.inmon.com/products/sFlow-RT.php.

[29] "The Network Simulator NS-3." http://www.nsnam.org/.

[30] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 69–84.

[31] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *Proc. ACM SIGCOMM*, vol. 41, no. 4, 2011, pp. 50–61.

[32] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in *Proc. ACM SIGCOMM*, vol. 41, no. 4, 2011, pp. 266–277.

[33] 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 *Proc. ACM SIGCOMM*, 2014, pp. 503–514.