Creek: Inter Many-to-Many Coflows Scheduling for Datacenter Networks

Hengky Susanto[1], Ahmed M. Abdelmoniem[2] Hao Jin[3] Brahim Bensaou[4]
Huawei Tech.[1] , HKUST[2,4], Texas A&M University[3]
hengky_susanto@huawei.com[1] amas@cse.ust.hk[2] haojin@tamu.edu[3] brahim@cse.ust.hk[4]

***Abstract*— In datacenter networks, many data transfers usually constitute semantically a *coflow* group. Typically, a coflow is considered completed when all transfers in a coflow are completed, and hence the data and information are useful to applications. That is why, applications' performance are optimized whenever the completion time at the level of a coflow rather than the individual flows is minimized. The current popular coflow scheduling algorithms are centralized based approach, but they incur high overhead cost. The decentralized approach in the Many-to-Many scenario also incurs high communication overhead cost caused due to the communication among the local controllers. Therefore, in this paper, we present a coflow scheduling mechanism that aims to minimize the coflow completion time for coflow with Many-to-Many communication pattern. And by product the communication overhead costs are minimized. Using a testbed implementation in our mini datacenter and large-scale network simulation, we demonstrate that our scheduling scheme improve the coflow completion time on average by up to 1.8× compared to the baseline in both cases. These are achieved while preserving compatibility with existing commodity switches and network protocols.**

I. Introduction

Network traffic in modern datacenters is often a result of the communication requirements at the application level. Recently, the term *coflow* provided a meaningful semantic that translates application requirements to matrices which can be understood at network level (e.g., the data plane layer). In networking context, a coflow consists of a set of concurrently active flows set to complete a specific data transfer started by the application. Typically, the completion of data transfer of all flows within the same coflow signifies the completion of the communication stage for the application. Applications strive to achieve faster completion of their communication tasks which greatly depends on minimizing *coflow's completion time* (CCT). However, this dependency may lead to inter coflow bottleneck, which can severely degrade the performance at the application level.

To address the dependency problems, many recent proposals puts this problem into the form of CCT minimization. The popular approaches are usually designed in centralized manner [4,5,6,7,8,9] where a single centralized scheduler is responsible for scheduling the coflows of the entire network. However, a high overhead cost is required for maintaining such a centralized system. Alternatively, there are various decentralized state of the art solutions. For instance, Barat [3] requires switch modifications where the task of scheduling coflows is performed at switches. However, lacks access to coflow level information because switches only have access to information at flow level, which leads to less optimal outcome. In addition, because this decentralized solution requires elaborate software modifications in the switches, it is harder to deploy. Stream [27] does not require switch modification but requires local controllers of the same coflow to exchange information, which may result in communication overhead cost. Moreover, decentralized schemes also commonly suffer from sub-optimal outcome because of the lack of a complete picture of coflow states and the inability to achieve global coordination between the local controllers. In this paper we present *Creek*: a decentralized inter coflow scheduler for coflows with Many-to-Many communication patterns without requiring hard modification and imposes minimal communication overhead cost. Creek is designed to resolve the challenges in decentralized scheduling systems, while at the same time possessing key advantages of centralized system. Creek is capable of acquiring a more complete picture of coflow states and accomplishes an approximate global coordination, achieves near optimal performance, without the overhead cost of centralized solutions.

The key to the solution depends on understanding the communication pattern which provides insights to achieve the objective of minimizing CCTs effectively. One-to-Many is a pattern where a single node receives data transfer from many senders and forms a single coflow [19,22,23,24]. Many-to-Many is a pattern where many receivers receives data transfer from many senders [18,20]. In other words, that is a single Many-to-Many coflow consists of multiple Many-to-One coflows, which is the focus of this paper.

In this paper, we present, ***Creek***, our inter coflow scheduler for coflow with Many-to-Many communication pattern which acquires the necessary information on coflows at receiver's end. The scheduling policy is enforced and communicated by leveraging existing network components (e.g., functionalities that are commonly available in commodity switches) and the mechanics of existing transport protocol (e.g., TCP/IP). For inter coflows scheduling decision, Creek employs Conditional Shortest Job First (C-SJF), where coflow is scheduled based on the condition of coflow state in SJF fashion. To reduce the communication overhead required for the receivers of a coflow to communicate with each other, Creek resorts the information management to a third party, which can be a designated node that store coflow information.

In our performance analysis, we evaluate our solution through actual testbed experiments and large scale simulation experiments. In the testbed experiments, we implement Creek and deploy the prototype in a mini datacenter testbed. This also shows that the solution is production deployments

friendly. Moreover, the experiments demonstrate that Creek outperforms the baseline by 1.8×. In the large scale simulation, we evaluate Creek performance by replaying an actual production trace of coflow traffic workload from a 3000 servers (150 racks) in Facebook production datacenter [4].

Specifically, the evaluation is performed by using widely accepted traces from Facebook along with two benchmarks: TPC-DS [5] query and Facebook's Tao structure [28]. In our evaluation, Creek superseeds both Baraat and the traditional per-flow fair sharing scheme by 1.85× on average, and achieves comparable performance with the centralized scheme. As for mice coflow CCT, Creek is up to 28× better than per flow fair sharing and up to 18× better than Baraat. Here, Creek also achieves similar outcome with centralized system. At last, finding in [4] shows that priority based scheme follows diminishing return behavior, and in this paper we provide an insight to this behavior through theoretical and (testbed and simulation) experiment results.

Our contributions can be summarized as follows:

1. Propose a coflow scheduling scheme for coflows with Many-to-Many communication patterns, which minimizes the communication overhead between receivers .
2. Deploy our solution in our mini datacenter. Evaluate solution in large-scale setting.

This paper is organized as follows. We present previous related work in section II and the system model in section III. Then, we describe Creek in section IV. Simulation results are presented in section V, then concluding remarks in section VI.

II. Related Work

One of the early works on this theme is Orchestra [6], where the semantic among flows is accounted in the design of the flow transfers optimization in datacenter. Sincronia [2], Varys [4], Aalo [5], and NC-DRF [21] by prioritizing smallest-total-size-first based approach in their scheduling mechanisms which improved the performance in [6]. RAPIER [7] extends [4] by incorporating routing algorithms into the scheduling scheme.

Likewise, CORA [8] also extends the problem in [4] by integrating the resources allocation solution into the flow scheduling scheme. In later development, the authors of [9] extended the problem in [4] and have taken into account the importance level of different coflows. Then they reformulated the problem into a weighted CCTs minimization problem. The aforementioned schemes falls into the centralized scheduling category that typically provide near optimal scheduling. However, these approaches are critized for incurring very high cost of centralized system management and are generally hard to realize because they require significant switch modifications and/or a complex control plane.

On the other hand, as an alternative, there is the decentralized approach. In this approach, Baraat [3] dominates as the state-of-the-art decentralized coflow scheduling system. Baraat relies on various heuristics which is based on a multiplexed First-In First-Out (FIFO) principles.

In Baraat, whenever large coflows are observed in the network, Baraat improves the CCT by processing mice flows in the background. Otherwise, mice flows are processed according to the trivial FIFO scheduling. Even though, Baraat proves to be effective, it has a few drawbacks. First, its

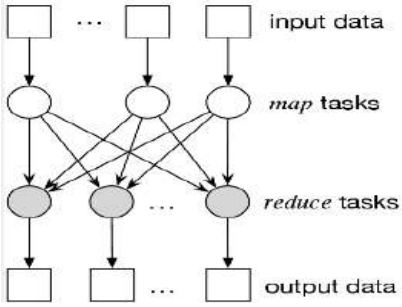

Fig. 1. Data Shuffle between mappers and reduces in Hadoop [18].

Flow size	The length of a flow
Coflow size	The sums of all flows in a coflow in bytes.
Coflow width	The number of parallel flows in a coflow.
Coflow length	Largest or longest flow in coflow in bytes.

Table 1. Terminology

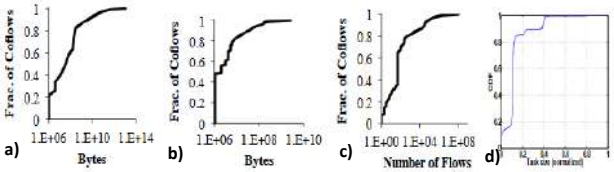

Fig. 2. CDF plot: a) coflow size , b) length, and c) width from Facebook datacenter [4], and d) coflow size in Bing, Microsoft datacenter [3].

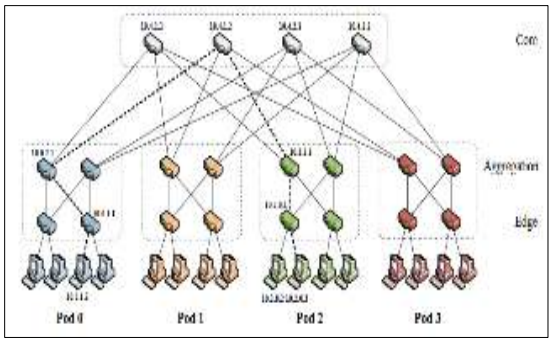

Fig. 3. FatTree network topology [30].

scheduling decisions are made locally at switches which limits the scheduler access to only flow level information. So, the scheduler has an incomplete information about coflow states and results in sub-optimal performance. Second, Baraat also requires modifications to switches' source code which makes it not deployment friendly.

Stream [27] is another recently proposed decentralized scheduler which opportunistically choose the receiver in Many-to-One and Many-to-Many communication patterns. However, since Stream requires its receivers of a same coflow to communicate with each other for coordination, Stream has high overhead communication cost. In our work, we adopt different approach where we solve the general coflow scheduling problem in decentralized manner for Many-to-Many patterns, without requiring hardware modification with minimal communication overhead.

III. System Model

In this section, we discuss the coflow abstraction, describe coflows in production, and the network model used in the study.

Coflow Abstraction. A coflow is generally characterized by the number of its concurrent flows (width), its total transferred bytes (size), and its longest flow in bytes (length). These characteristics determine the state of coflow during its lifetime. For example, a coflow state is known by tracking the number of completed flows of the coflow, the number of bytes transferred/received of the coflow, etc.

Coflows in Production. In this work [4], the authors observe that coflow sizes follow a heavy trailed distribution. Even though, in Facebook datacenter, large coflows of at least 10 Gb make 8% and ones of at least 1 Gb make 15% of all coflows, they are responsible for 98% (99.6%) of the traffic, respectively. This means that most coflows are small in the size and contribute the least bytes to the network as shown in Figures 2a, 2b, and 2c. This confirms to the findings in [3, 6] from Microsoft's datacenter, as shown in Figure 3d, which shows that coflow sizes abide to the heavy tailed distribution. Another work [14] which studied data-mining application and found it also has a very heavy tailed distribution where 95% of all data bytes come from flows larger than 35MB which make for only 3.6% of all flows. This again confirms that data mining application generates more small sized flows, but the traffic in the network comes from few large sized flows.

Network Model. In this work, the Tree-based topology [3, 7, 8, 10, 25] like FatTree [30] (Figure 3) is considered. We conduct the experiments in the testbed and NS-3 simulator using FatTree topology. From the experiments, we find that the processing and queuing times are significant in the aggregation and core switches which confirms with the findings in [10,11 25]. Moreover, we find that the bottleneck has shifted from ToR switches and becomes more evenly distributed among different layers. This is due to the high speed NICs matching the speeds of core switch ports.

IV. SCHEDULING SCHEME

A. Problem Formulation

The problem for the offline case of coflow scheduling can be formulated as follows: we have n number of coflows in a system numbered by c = 1, 2, …, n. Then, the problem is formulated as an optimization problem as follows.

$$minimize \sum_{c=1}^{n} t_c \,, \tag{1}$$

$$\sum_{f \in l} x_f \leq \mathcal{B}_l, \quad \forall l \in L, \tag{1.a}$$

$$w_f \leq \mathcal{T}_w, \ \forall f \in c, \ \ over\ t_c, w_f \geq 0 \tag{1.b}$$

The variable t_c refers to the completion time of a coflow c and can be evaluated via the expression. $t_c = \max(t_f |\ \forall f \in c)$, where t_f refers to the completion time of a flow f. Put it another way, t_c is denotes the completion time of the *slowest* flow in a coflow. First, constraint (1.a) ensures that the aggregate flow of link l does not exceed link capacity $\mathcal{B}_l$. Second, constraints (1.b) ensures that the starvation and packet out-of-order problems are eliminated. It is also vital to observe that the above formulation for CCTs minimization is an NP-Hard problem [3,4] and is reducible to the well-known *Open Shop Problems* [12].

B. Decentralized Coflow Scheduling Mechanism

Prior works focused on the many-to-one scenario with the assumption that coflow size is unknown *a priori*. In this paper, however, we address the more difficult many-to-many scenarios.

Generally, Creek uses the C-SJF scheduler to reduce the CCTs by simply giving priority to smaller coflows over larger ones. In C-SJF, *first* the coflow size is compared to a threshold $\mathcal{T}$ at the receiver's end. Then, if the coflow size exceeds $\mathcal{T}$, then the coflow priority is demoted. The problem is coflow size is unknown a priori, hence prior size measurement is not be possible. We resolve this by initially assigning every coflow to the highest priority and the priority is dynamically demoted based on the number of bytes received for the coflow. The receiver then updates the workers with the new priority values by piggybacking the priority on the ACK packets.

Creek also takes into account the coflow condition when deciding the priority (e.g.. the number of completed flows). It also ensures compatibility with the commodity switches, by performing the scheduling at the receiver's end as the information on coflow and its flows are readily available at the receiver side.

Creek enforces SJF by leveraging the multi queues commonly available in the commodity switches, to realize a multi-level feedback queue (MLFQ). As have been pointed out in [5], MLFQ may result into the starvation of some flows and Weighted Fair Queuing (WFQ) may provide a better solution. In Creek, MLFQ is adopted because priority queues provides better in-network prioritization and potentially achieves lower CCT. Moreover, WFQ may introduce the out-of-order problem for TCP flows. Having said that, later we propose an algorithm that ensures starvation free operation for Creek.

Coflow Priority Decision. Consider K priority queues in the commodity switches [1] and given coflow c, priority P_f^k denotes k^{th} priority queue assigned to flow $f \in c$, such that $1 \leq k \leq K$. Then, the priority assignment is as follows: $P_f^1 > P_f^2 > \cdots > P_f^k > \cdots > P_f^K$, where P_f^1 is the highest priority and P_f^K is the lowest priority. Every P_f^k mirrors to a threshold τ_k. Not that, most of existing commodity switches only support a maximum of 8 priorities queue [1]. Let P_f denote the priority assigned to f, such that $P_f = P_f^k$. Initially, all f is assigned to P_f^1, such that $\forall f \in c,\ P_f = P_f^1$. Therefore, given flow size $x_f \geq 0$, the priority P_f is decided as follows.

Coflow management. In coflows that create many-to-many communication patterns, the coflow typically may consist of many sub-cofows. In such case, there would be many receivers in a single coflow. Hence, sub-coflows of the same coflow is considered as a single entity and the completion of the coflow relies on the completion of all of its sub-coflows. Some of the many scheduling challenges with this pattern in decentralized settings are keeping track of the relationship among sub-coflows of the same coflow, deciding the appropriate priority values when coflow information is sparse, and a sub-coflow may not know about some of the other sub-coflows.

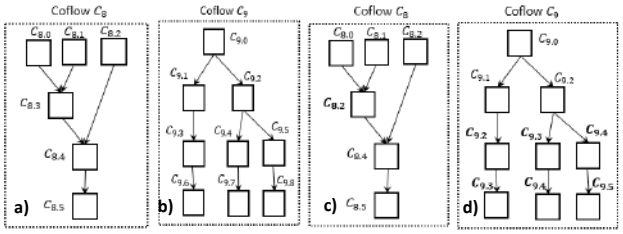

Fig 4. Coflow dependency 4(a and b), where each layer represents the webserver, cache follower, cache leader, and database in Cloudera' TPC-DS (fig. 4c), and Facebook's Tao Architecture (Fig 4.d).

To address these challenges, Creek utilizes shared-storage to allow sub-coflows of the same coflow easily exchange necessary status information with each other. In other words, the receivers of the same coflow will share and access the same data storage.

A task manager allocates a small amount of space at a designated storage space in a server to every new coflow. Thus, all receivers of this coflow use this storage to provide information, such as updates and queries on the total bytes have been sent. Hence, the number of communication within a coflow can be reduced from $O(n^2)$ down to $O(n-1)$, where is n denotes the number of receivers of a coflow.

However, one of the practical challenges is how to synchronize receivers of a coflow, such that information can be updated appropriately. This problem is known as *race condition* in operating system. There are multiple receivers sharing a common buffer but there is only one receiver that can update the information. This problem is solved using locking mechanism such as Mutex, a mutual exclusion based scheme. Thus, only a single receiver with Mutex is allowed to update and modify the information in the shared storage space. We also utilize Mutex semaphore based locking mechanism to resolve the race condition between receivers of a coflow in our testbed implementation.

Starvation Mitigation. To resolve starvation, when the wait exceeds a waiting threshold, worker of the starving flow retransmits packets that have not been acknowledged with higher priority assignment. The duplicate packets will be dropped at the receiver by TCP [29] if there is any. By doing so, the solution also avoids packet TCP out of order problem. The process is repeated until the flow escapes the starvation. Then, upon receiving a packet from the starving flow, the receiver compares the priority of the recent sent packet with the priority currently assigned to the starving flow. If it does not match, then the receiver increases that coflow priority and notifies the worker of the starving flow with new priority through the ACK packet. As pointed in [25], ECN can help in mitigating starvation, but it may accidently mark packets from mice coflows, because ECN is not designed to be aware of coflows

Setting threshold. The value of threshold is important in determining the system performance. If the threshold is too small or too large, packets of short flows may prematurely experience queuing delay behind elephant flows. Although threshold is commonly used in system design [3,4,10,25,28], there is very little study on how threshold should be set, such that system achieves optimality. We observe doing this does not guarantee convexity, and therefore it is possible that this is a non-convex problem (an NP-Hard problem).

Authors of [25] attempt to formulate threshold setting into convex optimization problem, but it uses too many constraints in the formulation, which is not realistic. From our experiments, we derive two observations: (i) Thresholds should be able to quickly direct traffic into appropriate queue. (ii) To mitigate starvation, the wait of the lowest priority should not exceed TCP retransmission timeout (RTO) [29]. Using these two rules of thumbs, our threshold leads to very minimal starvation in our testbed experiments. At this point, however, the threshold is decided by using exhaustive search, which may imply a higher overhead cost for larger systems. We will further investigate setting of threshold using machine learning techniques proposed in [26] in our future work.

Data structure. One challenge in implementing Creek is keeping track of the amount of bytesent generated by a large number of coflows. In practice, multiple coflows arrive and complete the task. Thus, information on coflows must be added or removed to the data structure when coflows start and complete respectively. For this reason, the data structure must be adaptive to the dynamics of start-complete cycle while at the same time keeping the computation cost low (e.g. lookup operation).

In our testbed implementation, we use two dynamic arrays available in C++ library (e.g. vector) to track coflows and sub-coflows' bytesent. We assume that coflow ID is unique globally is unique within a coflow. Then, Creek utilizes these IDs as coflow index and the information is inserted such that the IDs are sorted in increasing order, which is linear using the existing technique. Since the structure is dynamic caused by the start-complete cycle, straightforward hashing is not suitable for lookup operation. To resolve this, Creek utilizes binary based search algorithm [32], which is $O(\log n)$ and n denotes the array size. This is possible because the array is sorted.

Without careful coordination between inserting, deleting, updating, and information retrieval operation, the system may end up in a race condition where different threads are competing to modify the same information or data structure. This can result in inaccurate information update. For example, two threads are performing simultaneous updates at the same location in memory causing the new information to only reflect one of the updates instead of both updates.

We mitigate this issue by utilizing *strict priority queue non-preemptive scheduling* (SPQ-NS). Here, an operation (e.g. delete, update, insert, or read) cannot be interrupted when it is being performed even when there is an operation in higher queue waiting to be performed. Operations in each queue of SPQ-NS is performed in first-in-first-out order and the coordination between operation is done using Mutex. Here, information update and insert operations are assigned to the highest priority, information retrieval is assigned to a lower priority, and delete operation is assigned to the lowest priority. Moreover, deletion is only performed once every interval time (e.g. every 1 second). By doing this we prevent race condition.

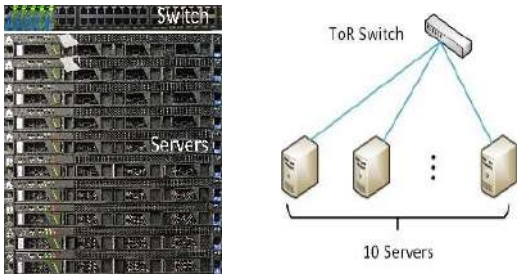

Fig. 5. Testbed Topology with 10 servers and ToR Switch.

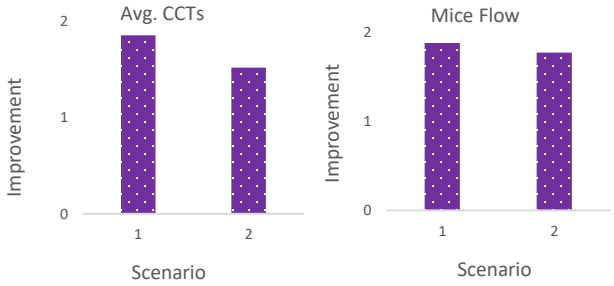

Fig. 6. Testbed experiment. *Scenario 1:* There are 30 coflows with each coflow has 3 receivers and each receiver is serving 5 flows. *Scenario 2*: There are 20 coflows with each coflow has 2 receivers and each receiver is serving 2 to 3 flows.

V. Evaluation

The performance of the proposed scheduling scheme is evaluated via number of experiments in 10Gbps testbed as well as large-scale network simulation using NS3 and Facebook traces from [4,5]. The main metrics used for evaluation are the average CCT and performance improvement factor, which is described as follows:

$$Improvement = \frac{Compared\ CCT}{Creek's\ CCT} \quad (2)$$

If the improvement is greater (smaller) than one, Creek is faster (slower).

The main findings are summarized below:

1. In the testbed experiment, Creek significantly reduces the average coflows CCTs relative to TCP by up to 1.8× and the average mice flows FCTs by up to 1.833×
2. In the simulation experiments, Creek outperforms decentralized approaches such as *Baraat*, *Per-Flow-Fair-Sharing* (FS), and *Stream* by up to 1.82×, while achieving comparable outcomes with the centralized scheme *Aalo*.

A. Testbed Experiment

Prototype: Creek prototype is built on top of the existing TCP implementation and synthesized as a loadable kernel module in Linux. Then, we implement *client and server* model to emulate multiple workers and receivers by utilizing socket programming at the application level. In this model, packets are transmitted from clients acting as workers to server acting as receivers. Our prototype randomly generates 216 and 432 TCP flows with different sizes according to heavy tailed distribution; then, these flows are randomly clustered into 20 and 30 coflows respectively with each coflow has 2 and 3 receivers. In this experiment, the TCP kernel module is modified so that the coflow ID can be inserted into the IP option field in TCP packet header [29]. Moreover, we used local memory to store coflow information, such as total bytes sent.

Testbed: Figure 5 shows the testbed used in evaluation. It consists of 12 datacenter-scale servers are connected together via a ToR 48-port 1 Gigabit Ethernet switch (Pica8 P-3297) and a control-plane 4-port 10 Gigabit Ethernet switch. The ToR switch supports strict priority queuing with at most 8 classes of services queue [1]. Each server is a HUAWEI RH1288 V2 with 24-core Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz, 64G memory, a 2T hard disk, and Broadcom BCM5719 NetXtreme Gigabit Ethernet NIC. Each server runs Ubuntu 14.04.2 LTS with Linux 4.0 kernel. In the ToR switch, strict priority queues are enforced and packets are classified based on the DSCP field [1,29].

Experiment: To evaluate Creek, we create two experimental scenarios. In the experiments, 10 machines are running client application sending data to a 11th machine running server application. In the first one, the experiment is conducted by starting 432 TCP flows which are classified into 30 coflows. In the second one, 216 TCP flows are initiated to make up for 20 coflows. In both scenarios, to reflect a more realistic environment, the 12th server is used to generate background traffic using iperf, which is a popular Linux traffic generator, at the speed of 500 Mbps (which is the equivalent of 50% of the link capacity). This is a common traffic pattern seen in many datacenters [11]. In both scenarios, we compare the CCTs of our scheduling scheme to the CCTs of using regular TCP [29]. This set of experiment is conducted using 8 priority queues. Later in the section, we conduct another experiment to measure the performance of using different number of priority queues. One of challenges performing testbed experiments is to generate sufficient traffic load that mimics bursty traffic without causing Denial of Service (DoS) [29]. In our testbed, traffic with 435 connections or larger causes Denial of Service.

The testbed results, as shown in Figure 6(a), demonstrate that Creek, when compared to TCP, it can improve the average performance by 1.8× and 1.533× in the first and second scenario respectively. Specifically, the average CCTs of 30 coflows with TCP is 14.9 and 9.73 milliseconds in the first and second scenario respectively; on the other hand, the average CCTs in our scheduling scheme is 8.1 and 6.47 milliseconds respectively. Similarly, Figure 6(b) depicts that using our coflow scheduling also improve the average performance of mice flows by 1.8× and 1.7× with 20 and 30 coflows respectively. This experiment shows that the proposed scheme performs better than TCP, especially in networks with higher traffic load.

B. Large Scale Simulation Experiments

To evaluate our proposed scheduling scheme in large scale network, we develop a flow-level simulator where it accounts for the coflow arrival and departure events at the flow level. It updates the rate and remaining volume of each flow when event occurs. We model a data center with 3465 hosts and 720 switches of 10 Gigabit (10G) link speed in Fat-tree topology [30] of size k=24.

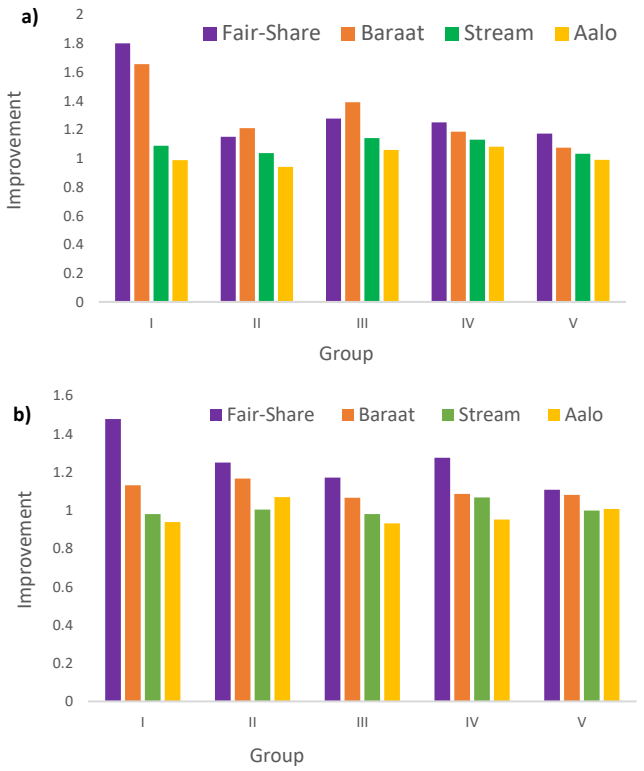

Fig. 7, Large scale experiments using (a) TPC-DS and (b) FB-Tao benchmark.

	I	II	III	IV	V
Size B	6MB-1GB	1GB-10GB	10GB-100GB	100GB-1TB	>1TB

Table 2. Five categories of coflow with different size in many-to-many pattern (size B).

In simulation experiments, Creek's performance is compared to baseline Per-Flow-Fair-Sharing, Baraat [3], Stream [27], and Aalo [5]. *Per-Flow-Fair-Sharing* (PFS) mechanism is a scheduling scheme that divides the resource capacity equally among flows traversing the same link, which is also the baseline in our analysis. *Baraat* is a First in First out with limited multiplexing scheme. *Stream*, is also a decentralized scheduling scheme, which opportunistically leverages coflow communication pattern.

Realistic traffic pattern and load. Creek is evaluated using real traffic pattern and traffic load by replaying 526 coflows from actual production traffic traces from 3000 servers in Facebook production datacenter [4,5], which capture a one-hour Hive/MapReduce trace. In our simulation, Equal-cost multi-path routing (ECMP) [29] routing protocol, which is used in datacenters to route and load balance network traffic, is also used in the simulations. Moreover, TCP is the dominant transport protocol in datacenter, hence we implement rate limiters that acts like TCP for all the schemes, except for Baraat whose rate limiter follow its design [3].

Traffic Pattern. To run the simulations, Cloudera's Industrial benchmark is used. Specifically, the TPC-DS query-42 (TPC-DS) [4], and Facebook Tao (FB-Tao) [28,31] traces are used to create many-to-many scenario (because Facebook trace only consists of coflow with many-to-one). We use these benchmarks and insights from [3,4,19,23,24,31] to synthesize the original trace to generate realistic trace of many-to-many pattern. The coflow sizes for the many-to-many pattern is shown in table 2.

Scenario 1: TPC-DS benchmark. Figure 7(a) shows that Creek is at least 1.82× better than Baraat and FS. And it shows similar performance as the centralized scheme Aalo. Creek also outperforms Baraat, FS, and Aalo in Group I by almost 1.8×, 1.6×, and 1.2×, respectively. All in all, compared to Baraat and FS, Creek is at least 1.83× better and Creek's and Aalo's performance are comparable.

Scenario 2: FB-Tao benchmark. Figure 7(b) shows that, on average, Creek superceeds Baraat, FS, and Stream by 1.6×, 1.2×, and 1.1× respectively. And, for small coflows, Creek is only within 1% to Aalo. In conclusion, Creek is better than both Baraat and FS, by at least 1.2× across all groups. This is because Creek can achieve similar performance of Stream but without its communication overhead. Creek also has comparable performance to Aloa across the various groups.

Creek's ability to quickly differentiate coflow according to its states with information at sub-coflow level allows for its superiority over Baraat and FS. This allows Creek to quickly divert coflows and allocate appropriate resources earlier, which avoids delay. In contrast, Baraat and FS severely suffer longer delay. Moreover, by resorting the information management to a third party, Creek achieves slightly better performance compared to Stream, but with significantly lesser communication overheads (i.e., $O(n-1)$ as in IV).

On average, Creek's overall performance is comparable to a centralized scheme Aalo. This is because Aalo only realizes a coflow is a mice coflow when it is completed; this means mice coflow is processed together with larger coflow in Aalo. Creek on the other hand is a sub-coflow based system, and therefore mice coflow can be quickly recognized as soon as a sub-coflow is competed. This enables Creek to prioritize mice coflow before its completion and quickly separate it from larger coflows, which results in lower CCTs. This approach takes advantage of the fact that sub-coflows of a mice coflow is typically small. For large coflows consisting of many mice sub-coflows, one of the parents of mice sub-coflows can recognize and separate it.

Finally, Aalo is performs better than Creek (by ~0.1×) because it is a centralized scheme with global information (i.e., Aalo can be more precise in distinguishing coflows with similar characters, which benefits these two categories). However, Creek compensate for this by achieving superior performance in all categories compared to the decentralized schemes.

VI. Conclusion

Creek is a decentralized coflow scheduler that aims to minimize CCT for Many-to-Many communication patterns and the communication overhead between receivers . The results from both testbed and large scale network simulation experiments show that Creek is a simple but effective coordination between receivers can improve applications' performance in datacenters. Creek outperforms decentralized schemes like Baraat, FS, and Stream, and performs comparably well to centralized schedulers like Aalo.

Reference

[1] http://www.pica8.com/documents/pica8-datasheet-picos.pdf

[2] S. Argawal, et al, , Sincronia: Near-Optimal Netwok Design for Coflows", ACM SIGCOM, 2018.

[3] F. Dogar, et al, "Decentralized Task-Aware Schduling for Data Center Networks", ACM SIGCOMM, 2014.

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

[5] M. Chowdhury and I. Stoica, "Efficient Coflow Schduling Without Prior Knowldege", ACM SIGCOMM, 2015.

[6] M. Chowdhury, et al,"Managing Data Transfer in Computer Clusters with Orchestra", ACM SIGCOMM, 2011.

[7] Y. Zho, et al, "RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks", IEEE INFOCOM 2015.

[8] Z. Huang, et al "Need for Speed: CORA Scheduler for Optimizing Completion Time in the Cloud", INFOCOM 2015.

[9] Z. Qiu, et al, " Minimizing the Total Weighted Completion Time of Coflows in Datacenter Networks", ACM SPAA, 2015.

[10] M. Alizadeh, et al, "pFabric:Minimal Near-Optimal Datacenter Transport", ACM SIGCOMM, 2013.

[11] M. Alizadeh, et al,"Data Center TCP (DCTCP)", SIGCOMM, 2010.

[12] S. Gawiejnowicz, "Time-Dependent Scheduling", Springer 2008.

[13] M. Alizadeh, et al., "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters:, ACM SIGGCOMM, 2014.

[14] A. Greenberg et al., "VL2: a Scalable and Flexible Data Center Network", SIGCOMM 2009.

[15] M. Chowdhury and I. Stoica, "Coflow: A Networking Abstraction for Cluster Applications", USENIX HotNets, 2012.

[16] A. Munir, et al, "Friends, not Foes – Syntehsizing Exiting Transport Strategies for Data Center Networks, ACM SIGCOMM, 2014.

[17] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild", ACM IMC, 2010.

[18] J. Dean and S. Ghemawat, "MapReduce: Simplifed Data Processing on Large Clusters", USENIX OSDI, 2004.

[19] M. Isard, et al, " Distributed Data-Parallel Programs from sequential Building Block", EuroSys, 2007.

[20] M. Zhaharia, et al., "Resilent Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Custer Computing", USENIX NSDI, 2008.

[21] L. Wang and W. Wang, "Fair Coflow Scheduling without Prior Knowledge", IEEE ICDCS, 2018.

[22] R. Chaiken, et al."SCOPE: Easy and Efficient Parallel Processing of Massive Dataset", VLDB, 2008.

[23] G. Malewicz, et al.,"Pregel: A System for Large-Scale Graph Processing", ACM SIGMOD, 2008.

[24] Y. Low, et al., "Distrubted GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". PVLDB 2012.

[25] W. Bai, et al, "Information-Agnostic Flow Scheduling for Comodity Data Centers", USENIX NSDI, 2015.

[26] P. Poupart, et al., "Online Flow Size prediction fo Improved Network Routing", IEEE ICNP, 2016.

[27] H. Susanto, J. Hao, K. Chen, "Stream: Decentralized Inter Coflow Scheduling for Datacenter Networks", IEEE ICNP, 2016.

[28] N. Bronson, et al, "TAO: Facebook's Distributed Data Store for the Social Graph", USENIX ATC, 2013.

[29] J. Kurose and K. Ross, "Computer Networking, a Top Down Approach 6th addition", Pearson, 2013.

[30] M. Al-Fares, A. Laukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", ACM SIGCOMM, 2008.

[31] A. Roy, et al, "Inside the Social Network's (Datacenter) Network," in ACM SIGCOMM 2015.

[32] C., L., R., S., " Introduction to Algorithm", MIT Press, 2001.