Stream: Decentralized Opportunistic Inter-Coflow Scheduling for Datacenter Networks

Hengky Susanto, Hao Jin, and Kai Chen
SING Group, Hong Kong University of Science and Technology
{hsusanto, hjinae, kaichen}@cse.ust.hk

***Abstract*— Coflow scheduling can improve application-level communication performance for data-parallel clusters. However, most prior coflow scheduling schemes are based on the centralized approach, which achieve good performance but suffers from high control overhead and scalability issue. On the other hand, state of the art decentralized solution requires switch modification, which makes it hard to implement. In this paper, we present Stream, the decentralized and readily-implementable solution for coflow scheduling. The key idea of Stream is to opportunistically take advantage of many-to-one and many-to-many coflow patterns to coordinate coflows without resorting to the centralized controller, and then emulate shortest coflow first scheduling to minimize the average coflow completion time (CCT). We implement Stream with existing commodity switches and show its performance using both testbed experiments and large-scale simulations. Our evaluation results show that Stream's performance is comparable to the centralized solution, and outperforms the state of the art decentralized scheme by 1.77x on average.**

I. INTRODUCTION

Network traffic in today's data-parallel clusters is often shaped by requirements at the application-level, and coflow provides an abstraction that bridges application-level semantic and the network [4, 15]. At the network level, coflow refers to a set of parallel flows associated with a specific task given by the application, and all flows in a coflow must be completed for the completion of a communication stage. In other words, minimizing coflow completion time (CCT) may result in a shorter completion time of the corresponding task and improve performance at application-level.

A number of proposals formulate coflow scheduling into CCT minimization problem. Most of the prior schemes are based on a centralized approach [4-9], where a single controller makes the coflow scheduling decision for the entire system. The centralized approach achieves good performance but suffer from a high control overhead (e.g., synchronization, fault tolerance, scalability, etc.). Alternatively, the state of the arts decentralized solution [3] requires customized modification in switches and this makes implementation and deployment difficult. In this paper we present *Stream*, a decentralized and readily-implementable solution for coflow scheduling, which opportunistically takes advantage of many-to-one and many-to-many coflow communication patterns without relying on a central controller.

Many-to-one is a communication pattern of a single receiver communicating with multiple senders to complete a single coflow [19-24, 35]. We observe that receiver is a natural position for coordinated coflow scheduling, since the overall coflow information can be available there. To minimize average CCT, Stream emulates conditional Shortest Job First (C-SJF) by prioritizing smaller coflows over larger ones. Stream assigns a priority to each coflow by considering its total number of bytes received at the receiver and other conditions such as the number of completed flows of the same coflow; the priority is gradually decreased as the total number of bytes received increases. Then, SJF is enforced by utilizing priority queuing, a built-in function available in today's commodity switches, to ensure smaller coflows are prioritized over larger coflows.

We extend the above scheme to the many-to-many pattern, where multiple receivers communicate with multiple senders [18, 20, 23] and a coflow consists of multiple sub-coflows. Each sub-coflow is many-to-one. For this, Stream also utilizes the receiver to schedule coflows. However, in this scenario receivers of different sub-coflows may not directly exchange information with one another, therefore the receivers of a coflow may not have a full picture of the coflow (e.g., the total bytes received in a coflow). The lack of shared information may lead to poor outcomes. To resolve this challenge, Stream complements C-SJF with three additional schemes. (i) Weighted-Priority: the priority decision is weighted such that sub-coflows of the same coflow that arrive later will be deprioritized faster. (ii) Information-Relay: Stream also takes advantage of senders that are serving multiple receivers of the same coflow by relaying information (i.e. bytes received) between these receivers. (iii) Child-to-Parent: in multi-stage scenario where the completion of a parent sub-coflow is dependent on the completion of child sub-coflows of the same coflow, the receiver of child sub-coflow, upon its completion, will communicate its size (bytes received) to the receiver of the parent sub-coflow. This allows the receiver of the parent sub-coflows to make better scheduling decision. By this design, Stream is effective in prioritizing smaller coflows over larger coflows in many-to-many scenarios.

We have implemented a Stream prototype and deployed it in a small-scale testbed. Our implementation verifies that Stream can be readily deployed in today's commodity datacenters without requiring modification to switch hardware. Our testbed experiment results show up to 1.3× faster CCT on average and 1.87× faster with mice coflow compared to TCP fair sharing.

We further evaluate Stream through a large-scale trace-driven simulation with a production trace of coflow traffic from Facebook datacenter [4]. In the many-to-one scenario, Stream shows 1.4× and 1.77× faster CCT on average compared to the state of the art decentralized solution [3] and

978-1-5090-3281-5/16/$31.00 ©2016 IEEE

per-flow fair sharing scheme respectively (and 2.7× and 5.1× faster respectively with mice coflows). At the same time, Stream achieves comparable outcomes to the centralized scheduler, Aalo [5]. For further evaluation, we include two other case studies: multi-wave coflow (flows of the same coflow arriving at different times) and bursty traffic (coflows arriving within the same interval). Stream improves the average CCT by up to 2.8× in the multi-wave study and at least 1.9× faster in bursty traffic study compared to the decentralized and per-flow fair sharing schemes. In many-to-many scenario, the evaluation is performed by utilizing two benchmarks: TPC-DS [5] query and Facebook Tao structure [28]. Stream outperforms both Baraat and per-flow fair sharing by up to 1.85× faster on average. Compared to centralized scheme, Stream achieves comparable performance in both case studies.

This paper is organized as follows. We begin by presenting background information in section II. Stream is described in detail in section III. Simulation results are presented in section IV, followed by previous related work in section V and concluding remarks in section VI.

II. BACKGROUND

This section provides a general overview of coflow structure, a description of coflows in production, and network model.

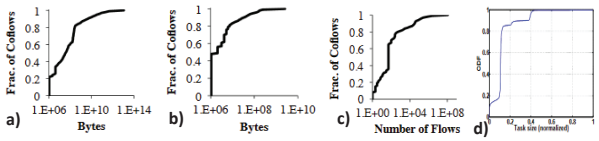

Fig. 1. CDF plot: a) coflow size , b) length, and c) width from Facebook datacenter [4], and d) coflow size in Bing, Microsoft datacenter [3].

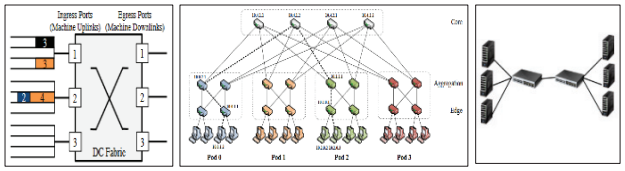

Fig. 2. Network topologies: Big-Switch topology [4,5] (*left*), FatTree topology [30] (*middle*), and a simple network topology (*right*).

Coflow structure. Cluster computing applications today generally follow many-to-one model. For example, mapper and reducers in Map-Reduce [18] are respectively the receiver and senders in a coflow. Spark [20] is another framework that utilizes many-to-one pattern for enabling data reuse in applications. Others include Dryad [19], DryadLINQ [21], SCOPE [22], Pregel [23], GraphLab [24], Tachyon [35], etc. To confirm this finding, we have analyzed production trace of coflow traffic from Facebook datacenter [4] and we observe that many-to-one pattern is also found in the production trace. Since this pattern is common among applications, we design our proposed coflow scheduling scheme to take advantage of the receiver in many-to-one model to coordinate the flow transmission of coflow without resorting on a central controller.

Coflows in production. The authors of [4] discover that coflow size follows the heavy trailed distribution. Only 8% (15%) of coflows has the size of at least 10 GB (1 GB) in Facebook datacenter, yet they are responsible for 98% (99.6%) of the traffic. In other words, the majority of coflows are relatively small coflow size (Figure 2a, 2b, and 2c). Findings of authors of [3] from Bing search application in Microsoft's datacenter (Figure 4d), and a further investigation in [6] also concur that coflow size follows the heavy tailed distribution. A similar trend is also observed in [14] where the data-mining distribution has a very heavy tail with 95% of all data bytes belonging to 3.6% of flows larger than 35MB. This means the system is predominantly populated by shorter flows, but the traffic is mostly taken up by minority flows.

Network model. The two popular network topologies (Figure 2) often used in scheduling scheme design for datacenter are: (i) Big-Switch-based topology [4,5,10,13], a non-blocking datacenter fabric where processing and queue delay are negligible. This model only focuses on bottleneck in ingress and egress ports (machine NICs) which allows simpler computation. (ii) Tree-based topology [2, 3, 7, 8, 10, 25] like FatTree [30]. To choose between these two topologies, we conduct few experiments in our testbed and NS-3 simulator with network topology illustrated in Figure 2 (right). We discover that processing and queuing delay in switches in non-edge network does matter. Our finding confirms the results of [2, 10, 11 25]. The bottleneck shifts from edge and becomes more distributed because today's NIC speed catches up to switches processing speed [31]. For this reason, we adopt the tree based topology and incorporate it into our design.

III. STREAM DESIGN

Stream is a decentralized solution that opportunistically takes advantage of many-to-one and many-to-many patterns to coordinate coflows. In the design, we consider the following coflow characteristics: the number of parallel flows (width), total bytes (size), and the longest flow in bytes (length). These characteristics determine the state of a coflow, for example, the number of flows in a coflow that have been completed, the amount of bytes received per coflow, etc. As prior works [3-9, 37], Stream assumes that information on coflow ID can be derived from upper layer applications.

A. Problem Formulation

Consider the following offline scheduling problem with n coflows in a system indexed by c = 1, 2, …, n. Then, the objective of scheduling problem is as follows.

$$minimize \sum_{c=1}^{n} t_c , \quad (1)$$

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

$$w_f \leq \mathcal{T}_w, \ \forall f \in c, \quad (1.b)$$

$$p_i < p_{i+1}, \ \ \forall p_i, p_{i+1} \in f, \quad (1.c)$$

$Over\ t_c, w_f \geq 0$. Notation t_c denotes the completion time of coflow c and it is described as the following expression: $t_c =$

$\max(t_f | \ \forall f \in c)$, where t_f denotes the completion time of flow f. In other words, t_c is determined by the completion time of the *longest* flow's completion time in a coflow. Constraint (1.a) assures aggregate flow traversing link l does not exceed link capacity $\mathcal{B}_l$. Constraints (1.b) and (1.c) assure starvation and packet out-of-order respectively are mitigated. It is also important to note CCTs minimization is an NP-Hard problem [3, 4] and reducible to *Open Shop Problems* [12].

B. Many-to-one Pattern

We begin by addressing coflow scheduling problem in many-to-one scenario on the premise that coflow size is unknown *a priori*.

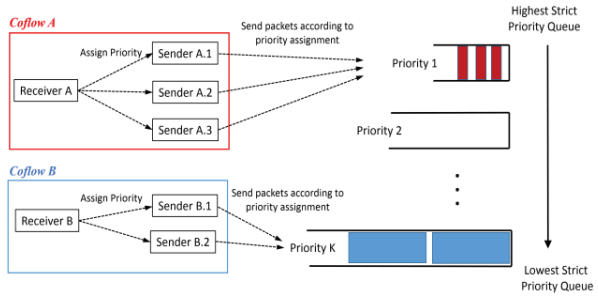

Figure 3. Stream overview in many-to-one scenario.

Generally, Stream utilizes C-SJF to minimize the average CCT by prioritizing smaller coflows over larger ones. Figure 3 summarizes Stream's C-SJF: the receiver determines the priority of a coflow and communicates it to each sender. Next, the senders transmit data with the priority determined by the receiver. The priority is then enforced at switches by utilizing strict priority queuing, a built-in function available in today's commodity switches.

C-SJF is accomplished by first comparing the coflow size to a demotion threshold $\mathcal{T}$ at the receiver's end: if the coflow size exceeds $\mathcal{T}$, then the coflow will be deprioritized, which results in deprioritization to all its flows. However, since coflow size is unknown a priori, a straightforward measurement may not be possible. To address this issue, our solution is inspired by [5, 25]. Initially every coflow is assigned to the highest priority and the priority is later adjusted as the information on the amount of bytes received becomes available at the receiver. Then, the receiver notifies its senders with new priority updates by embedding the updates in the ACK packet. Secondly, the scheme takes coflow condition into consideration in deciding the priority (e.g. number of completed flows). Thirdly, to ensure compatibility with the existing commodity switches, Stream performs the scheduling at the receiver's end because information on coflow and its flows are accessible there. Lastly, SJF is enforced by utilizing multiple queues, which is commonly available in the existing commodity switches, to implement strict priority queuing (SPQ).

Although it has been pointed out in [5] that SPQ may introduce the risk of starvation and Weighted Fair Queuing (WFQ) may provide a better solution, SPQ is preferable for two reasons: first of all, priority queuing provides better in-network prioritization and potentially achieves lower CCT. Secondly, WFQ may cause TCP packet out of order problem. We will address the starvation concern later in this paper.

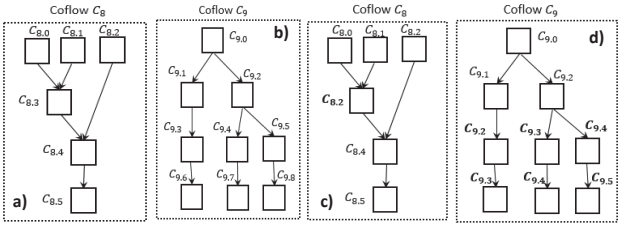

Fig 4. (a) Coflow dependency in Claudera' TPC-DS [4], and (b) Facebook's Tao Architecture [28,32], where each layer represents the webserver, cache follower, cache leader, and database. (c) Coflow sub-ID of TPC-DS and (d) Tao Architecture generated in Weighted-Priority Approach.

Coflow priority decision. Here, we present Stream's priority decision mechanism. 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 arrangement is defined 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 is associated to threshold τ_k. Currently, existing commodity switch typically supports 8 priority queues [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.

$$P_f = K - \left\lceil K.\min\left(1, \frac{\tau_k + \alpha\, \mathcal{H}_c}{\sum_{f \in c} x_f}\right)\right\rceil, \quad \text{for } \exists x_f > 0, \qquad (2)$$

$$\mathcal{H}_c = \tau_k \left(\frac{n_c^{fnsh}}{n_c} + \frac{n_c}{\sum_{f \in c} x_f}\right), \qquad (3)$$

where n_c^{fnsh} and n_c in (3) denote the number of flows in coflow c that have completed and the total number of flows in c. The ratio $\frac{\tau_k}{\sum_{f \in c} x_f}$ in (2) enforces SJF emulation. Observe that P_f decreases as $\sum_{f \in c} x_f$ grows, which results in $\frac{\tau_k}{\sum_{f \in c} x_f} < 1$. This equality implies that coflow with $\sum_{f \in c} x_f > \tau_k$, for $k > 1$, will be deprioritized. The ceiling function in (2) assures that P_f is an integer. The rationale behind the ratio $\frac{n_c^{fnsh}}{n_c}$ in (3) is to prioritize coflow that is suspected to be near completion. Ratio $\frac{n_c}{\sum_{f \in c} x_f}$ is also utilized to influence smaller coflows to be given higher priority. Since information may not be *a priori* known in every framework, n_c is adjusted as new information becomes available. To summarize the discussion, $\mathcal{H}_c$ can be interpreted as a function that captures coflow conditions. This function can be further developed as part of our future work. At last, to assure packet arriving out of order is avoided, $P_f = max(P_f, P_f')$, where P_f' is the previous decided priority.

C. Many-to-many Pattern

A coflow with many-to-many pattern may consist of multiple sub-coflows and there may exist dependency between sub-coflows. As illustrated in Figure 4, coflow with this pattern can be modelled with Directed Acyclic Graph

(DAG). Similar observations are made in [5], that first sub-coflows of a same coflows must be treated as a single entity. Second, a parent sub-coflow only completes when the child sub-coflows it depends on are completed. Some of the challenges with this pattern in decentralized environment include keeping track of the relationship among sub-coflows from the same entity, deciding an appropriate priority when coflow information is sparse, and sub-coflows within the same entity may not be aware of the existence of other sub-coflows. To address these challenges, Stream utilizes Weighted-Priority, Information-Relay, and Child-to-Parent approaches. With these approaches, Stream opportunistically gathers information on bytes received. Then Stream utilizes C-SJF to coordinate coflow where each receiver of the same coflow manages its own sub-coflow.

Algorithm 1: Sub Coflow ID Assignment

```
1. InternalID[ ] // set of IDs proposed by parent
2. |Parents|     // number of parents
3. Procedure Set_SubCoflowID (InternalID[ ] )
4.     If  D = D' , ∀D, D' ∈ InternalID[ ], then
5.         SubCoFlowID = InternalID[ ] + |Parents| – 1.
6.     Else SubCoFlowID = max(InternalID[ ])
7. End procedure
```

Weighted-Priority (WP). Here, we propose a scheme to weigh the priority decision such that sub-coflows of the same coflow that arrive later will be deprioritized faster. Stream utilizes coflow's internal ID that is used to identify its sub-coflows to weigh coflow priority. Internal ID determined using algorithm 1 can be utilized as an indicator of the number of sub-coflows that is locally discovered by a sub-coflow. For example, if the internal ID=4, it means there are at least 3 others sub-coflows in the entity. It can also be utilized to describe dependency between sub-coflows. For example, parents sub-coflow has a lower ID number than its children. Stream extends eq. (2) of C-SJF scheme and leverages internal IDs to weight the priority of each sub-coflow by the following equations.

$$P_f = K - \left\lceil K.\min\left(1, \frac{\tau_k + \alpha\, \mathcal{H}_c}{W \sum_{f \in c}\, x_f}\right)\right\rceil \qquad (4)$$

Here, weight $W = \alpha.log(m+1)$ when $m > 1$. Otherwise, $W = 1$. The log function is to limit W's influence on priority decision. Variable m denotes number of sub-coflows that is discovered so far. Weight W in eq. (4) is employed to allow a faster deprioritization of sub-coflows that are members of a large coflow. The internal ID is generated by parent sub-coflows when they are invoking new sub-coflows (children) using algorithm 1. The ID of the first batch of sub-coflows in an entity is provided by "master" (or "manager") whose task is to invoke the first batch of sub-coflows [18, 20, 21, 23, 24]. When there are two or more parents assign different ID to the same child, the largest ID is selected by the child. If there are two or more parents assign a child with the same ID, then child's ID = ID + n_parents-1, where n_parent denotes the number of child's parents. For example, sub-coflow $C_{8,4}$ in Figure 4c and 4d

Information-Relay (IR). In applications like Map-Reduce [18], multiple receivers of the same coflow may share common senders. In other words, a sender may serve multiple receivers of the same entity at the same time. Stream takes advantage of these senders to relay information (i.e. bytes received) between receivers of the same coflow. The sender first observes coflow ID and sub-coflow (internal) ID, for example, the coflow ID of coflow $C_{8.1}$ (in Figure 3.a) is C_8 and the internal ID is 1. Then, by comparing the coflow ID, the sender knows that it is serving multiple receivers of the same coflow. On this basis Stream leverages senders to relay information (such as bytes received) between receivers by piggybacking in data sent to its receivers. Then, the receiver sums up the information on bytes received gathered from its peers to determine the priority. Let S denotes the total amount of bytes received by receiver's peers and β denotes a weight factor, the priority is determined by extending eq. (4), which is described as in the following equation, eq. (5).

$$P_f = K - \left\lceil K.\min\left(1, \frac{\tau_k + \alpha\, \mathcal{H}_c}{W\,(\beta\, S + \sum_{f \in c}\, x_f)}\right)\right\rceil \qquad (5)$$

Child-to-Parent (CP). We observe that the receiver of parent sub-coflow is a natural position for gathering information (bytes received) of its child sub-coflows because it has access to the receivers of child sub-coflows. CP is carried out in two stages. In the first stage, when a child sub-coflow completes, the receiver of the child sub-coflow sends a tuple, <Responses to query, sub-coflow size (bytes received)>, to the receiver of the parent sub-coflow. In the second stage, upon receiving a tuple, the parent sub-coflow sums up the sub-coflow size of its child sub-coflows and determine the priority utilizing eq. (5). This approach enables Stream to capture large coflows that are made up of many mice sub-coflows.

In addition, we also observe that threshold-based approaches [5, 27] process large coflows and mice coflows together until one of them exceeds the threshold for mice coflow. Most likely that a mice coflow is made up of a few mice sub-coflows. Thus, to detect large coflows earlier, the threshold for highest priority is configured to detect mice sub-coflows and, the larger coflows will be detected by parent coflows using the approach described in the previous paragraph.

By combining WP, IR, and BU with C-SJF, Stream obtains the approximation of the number of sub-coflows, as well as of the current coflow sizes. This allows Stream to quickly direct coflows to the right queues and allocate appropriate resources.

D. Practical Consideration

Multi-wave. Flows from the same coflow may arrive at different times due to failures or stragglers [33]. Stream is capable of handling events with multiple waves of arrival flows as long as the flows use the appropriate coflow and sub-coflow ID. The receiver keeps track of the amount of data received regardless of the number of waves.

Starvation mitigation. To resolve starvation issue, when the waiting exceeds pre-defined threshold, the sender 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. 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 received packet with the priority currently assigned to the starving flow. If they do not match, then the receiver increases that coflow priority and notifies the sender of the starving flow with new priority through the ACK packet.

Setting threshold. Although threshold is commonly used in system design [3,4,10,25,27], there is very little study on how threshold should be decided, such that system achieves optimality. Authors of [25] attempt to formulate threshold setting into convex optimization problem, but it uses too many constraints in the formulation, which may not be realistic. We attempt to compute the threshold for each priority queue by utilizing eq. (6) from queuing theory [26]. We observe that doing this does not guarantee convexity, and therefore it is possible that this is a non-convex problem (an NP-Hard problem). At this point, the thresholds are decided using exponentially-spaced threshold used in [5]. We will further investigate the setting of threshold in our future work.

Number of queues required. Next, we address the question of the number of queues required to ensure that our proposed method will achieve a good performance.

Theorem 1. The performance improvement has diminishing returns behavior as $k \to \infty$.

Proof. Let $K = \infty$ denote the number of priority queues and μ be the processing rate of a link. The waiting time w_k at queue with priority $k \in K$ is described by equation from [26].

$$w_k = \frac{1}{\mu} \frac{1}{f \prod_{j=1}^{k} \left(1 - \sum_{i=1}^{j} \rho_i\right)}, \quad (6)$$

where ρ_k denotes the traffic load k^{th} priority queue. Then, we have $\rho_k = \frac{\lambda_k}{\mu}$ [26], where λ_k denotes the arrival rate at k^{th} priority queue. Observe that, given priority $P_f^1 > P_f^2 > \cdots > P_f^K$, we have $w_1 \le w_2 \le \cdots \le w_K$. Let $U(w_k)$ be the utility function of w_k to evaluate the performance of the system. The performance evaluation can be formulized as follows. $Maximize \sum_{k=0}^{K} U(w_k)$, where $U(w_k) = \frac{1}{w_k}$. $\sum_{k=0}^{K} U(w_k)$ can also be expressed as $\sum_{k=0}^{K} U(w_k) = \frac{1}{w_1} + \frac{1}{w_2} + \frac{1}{w_3} + \cdots + \frac{1}{w_K}$. Notice $\lim_{k \to \infty} U(w_k) = 0$, which also implies that the utility of $U(w_k)$ diminishes as $k \to \infty$. Thus, the performance improvement follows the behavior of diminishing returns. ■

Theorem 1 implies that at some point the benefits of multiple queues diminish as the number of queues increases, which is consistent with findings in [5, 25] and confirmed by our testbed and simulation results. We utilize 4 queues in our experiments and achieve satisfactory outcomes.

Discussion. We acknowledge that the coflow patterns in datacenter may not always follow many-to-one or many-to-many, and further, it is not impossible that a coflow may consist of individual flows. In these scenarios, Stream behaves similar to existing scheduler like PIAS [25].

IV. Evaluation

The performance of Stream is evaluated through experiments in our testbed with 1G port switches and large-scale simulation using Facebook data trace from [4,5]. Our primary metric for comparison is the average CCT, and our performance improvement factor is described as follows.

$$Improvement = \frac{Compared\ CCTs}{Stream's\ CCTs}.$$

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

The main results are summarized as follows:

1. In testbed experiment, relative to TCP fair sharing, Stream improves the average CCT by up to 1.3× faster and the average mice coflow CCT by up to 1.87× faster.
2. Large-scale simulation shows that on average, Stream outperforms state of the art decentralized solution (Baraat) and per-flow fair sharing by up to 1.4× and 1.71× faster respectively, and only trailing by 0.87× compared to the centralized solution, Aalo. For mice coflows, Stream is 2.7× and 5.1× better in comparison to Baraat and per-flow fair sharing respectively, while achieving comparable outcomes to Aalo.
3. In multi-wave scenario, Stream outperforms Baraat and per-flow fair sharing by up to 1.7× and 2.8× faster. Compared to Aalo, Stream achieves similar performance.
4. In many-to-many, on average Stream improves the performance by up to 1.85× and 1.9× faster than Baraat abd per-flow fair sharing respectively, while achieving comparable performance to Aalo.

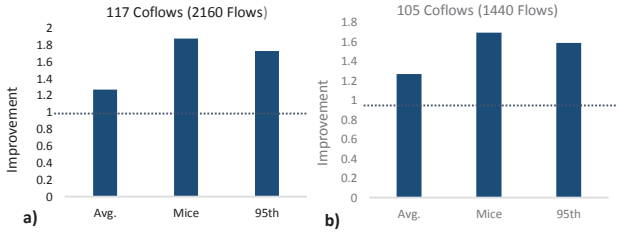

Fig. 5. Testbed Experiments with TCP and Stream of avg. CCT, avg. mice coflows CCT, and 95th percentile avg. CCT. (a) *Scenario one*: 117 coflows with 2160 flows. (b) *Scenario two*: 105 coflows with 1140 flows.

A. Testbed Experiment

Implementation: We build Stream prototype based on modifying the TCP kernel module in Linux operating system. Then, we implement client/server application to emulate senders and receivers in many-to-one scenario by utilizing socket programming. Here, client applications are the senders and server applications are the receivers. We assume coflow ID is provided by application layer in this implementation. Hence, Senders utilize setsockopt to pass down coflow ID from the application layer to the transport layer. This allows the application layer to insert coflow ID into IP option field

in TCP packet header. The ID is utilized to identify which packet belongs to which coflow. At the receiver's end, coflow ID is extracted from packet received from its senders.

To communicate priority decision, the receiver utilizes the reserve field in the TCP header of ACK to map the priority (e.g. priority 2) to Differentiated Services Code Point (DSCP) [29] bits of ACK packets that are sent to its senders. The 4 bits in Reserve field provides a range of integer 0 to 15, which is sufficient to represent 8 priority queues.

These coflow monitoring and priority notification schemes are accomplished by adding a few lines in TCP kernel in Linux. At last, threshold information can be stored in a file to allow thresholds to be adjusted without re-compilation.

To meet the required constraints described in problem formulation (1), capacity constraint in (1.a) can be addressed by utilizing Explicit Congestion Notification (ECN) [29] based protocol (DCTCP [11]), starvation constraint in (1.b) can be elevated by senders quickly performing the starvation mitigation when the timer expires at 10ms, which is TCP RTOmin [11]. To satisfy packet out of order constraint (1.c), Stream only deprioritizes coflows only if it is required.

Testbed: 8 servers connected to a Pica8 P-3297 48-port 1 Gigabit Ethernet, 4-port 1 Gigabit Ethernet commodity switch with 2MB shared memory, which supports strict priority queuing with at most 8 classes of services queue [1]. Each server is a Dell Server: PowerEdge R320 with CPU Intel(R) Xeon(R) CPU E5-1410 0 @ 2.80GHz, 8G memory, and Broadcom 5720 Dual Port 1Gb LOM Gigabit Ethernet NIC. Each server runs Ubuntu 14.04.2 LTS with Linux 4.0 kernel. In our switch, we enforce strict priority queuing and classify packet based on the DSCP field.

Experiment: To evaluate Stream, we create two experiment scenarios in which 6 machines are running senders and a machine running receivers. In the first scenario, the experiment is conducted with 2160 TCP flows that make up 117 coflows. In the second scenario, there are 1440 TCP flows which make up 105 coflows. In both scenarios, we added the 8^{th} server to generate background traffic of 500 Megabits per second (50% of the link capacity) using iperf, which is a common traffic characteristic in datacenter [16]. We compare the average CCT of Stream to the average CCT of TCP fair sharing. This set of experiments is conducted using 8 priority queues. Our heavy tailed traffic pattern is randomly generated according to traffic patterns from Facebook and Bing search (Microsoft) [4, 3], and is illustrated in Figure 1.

Experiment results. Our testbed experiment demonstrates that when compared to TCP fair sharing, Stream achieves better performance by 1.3× and 1.27× on average in the first and second scenario respectively, as illustrated in Figure 5. Also, as depicted in the same figure, in both scenarios Stream reduces the average CCT of mice coflows by up to 1.7× and 1.87× respectively. Moreover, Stream also has better performance by up 1.58× and 1.72× at 95^{th} percentile in comparison to scheduler with regular per-flow sharing in both scenarios. Through these instances, we demonstrate that Stream performs better than TCP fair sharing, especially in network with higher traffic load.

K Pods	# of Servers	# of Switches
k=8	128	80
k=16	1024	320
k=24	3456	720
k=32	8192	1280
k=48	27648	2880

Waves	1th	2nd	3th	4th
Single	100%			
Two	90%	10%		
Three	81%	9%	10%	
Four	81%	9%	4%	6%

Table 1 (left) and table 2 (right). *Table 1* describes network size of FatTree topology. *Table 2* describes flow distribution in multi-wave coflow.

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

Table 3. Five categories of coflow with different size in many-to-one pattern (size A) and many-to-many pattern (size B).

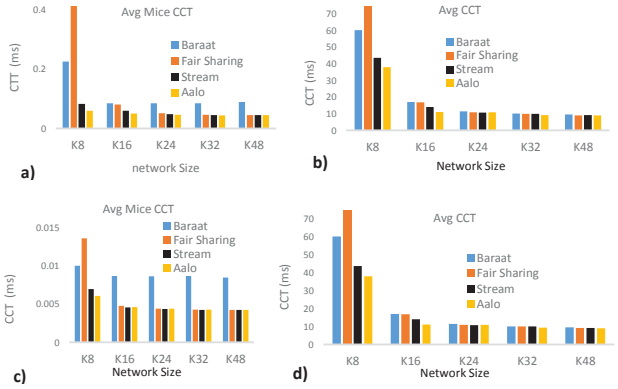

Fig. 6. Single wave in network in 1G switches (Figure a and b) and network in 10G switches (Figure c and d).

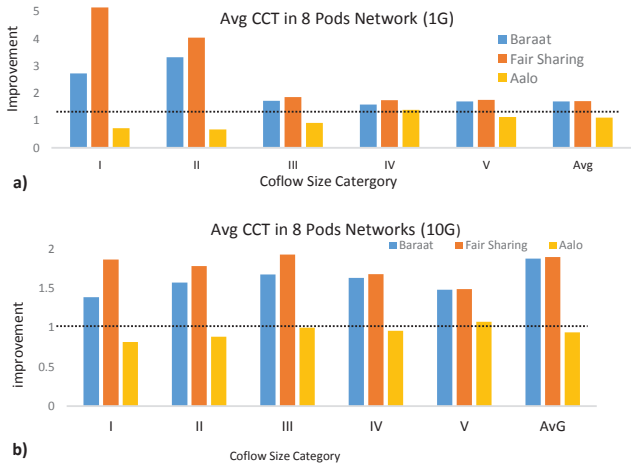

Fig 7. Average CCT improvement in 8 pods 1G and 10G networks according coflow categories described in table 3.

B. *Large-scale Simulations*

In this section, we evaluate Stream's performance in many-to-one and many-to-many scenarios. In many-to-one scenario, we consider trace-driven, bursty, and multi-wave traffic. In many-to-many, we utilize benchmarks from Cloudera [5] and Facebook [28,32]. In all our simulations, we use a production traffic trace collected from Facebook datacenter, specifically from 150-racks (3000 machines) [5].

Simulation setting: We develop a flow-level simulator and it accounts for the flow arrival and departure events, rather than packet sending and receiving events. It updates the rate and the remaining volume of each flow when event occurs. We employ FatTree network topology [30] with up to 27,648 hosts (48 pods). We conduct our simulation with 1 Gigabit (1G) switches to create a higher traffic load condition, as well as 10 Gigabit (10G) switches where delay in non-network edges is minimal. Our assumptions are: the switch has sufficient buffer to store incoming data, each flow traverses along one path, and coflow size follows heavy-tailed distribution.

In our simulations, we compare Stream to per-flow fair sharing, Baraat [3], and Aalo [5]. Per-Flow Fair-Sharing (FS) is a scheme that shares the capacity equally among flows traversing the same link. Baraat, a FIFO with limited multiplexing (FIFO-LM) scheduler, is the state of the art decentralized scheduler. To analyze how Stream performs against centralized solution, we compare our solution to Aalo. For simplicity, Aalo's additional delay from managing centralized system is not considered in the simulator and information on coflow is made available instantaneously to centralized controller. Additionally, based on findings in [5] and results from our testbed experiment, 4 priority queues provides the best outcome. Thus, Aalo and Stream employ 4 priority queues in their scheduling schemes. Moreover, in principle, all schemes assume that coflow characteristics are unknown ahead of time.

Traffic load. Stream is evaluated using traffic load by replaying production traces from Facebook clusters [4, 5]. Bursty traffic pattern of coflows arriving at the same interval, which is also common in datacenter [17, 32], is considered in our study. We also incorporate the commonly used Equal-cost multi-path routing (ECMP) [29] to route and load balance flows in the flow simulator. Additionally, since TCP is the common transport protocol in datacenter, we implement rate limiter that behaves like TCP for all schemes, except for Baraat where the rate limiter is implemented according to its design in [3].

Many-to-one pattern. Here we provide an overview of Stream's performances in different network sizes in 1G and 10G networks. We then analyze how Stream performs under heavier load. To evaluate Stream with different traffic loads while preserving the authenticity of the original trace, we increase the network size according as described in table 1.

In 1G network, on average, Stream achieves faster completion time than Baraat and FS, by up to 1.4× and 1.77× respectively (Figure 5b), but trailing by 0.87× compared to Aalo (within 13%). Stream achieves up to 2.7× and 5.1× faster for mice coflows compared to Baraat and FS respectively (Figure 6a). Compared to Aalo (centralized), Stream is trailing by 0.76× (within 24%).

In 10G networks, Stream on average achieves shorter completion time than Baraat and FS by up to 1.5× and 2.1× respectively, but trails 0.83× compares to Aalo (Figure 6d). For mice coflows, Stream outperforms Baraat and FS by up to 1.8× and 1.9× faster respectively; and within 13% of Aalo (Figure 6c).

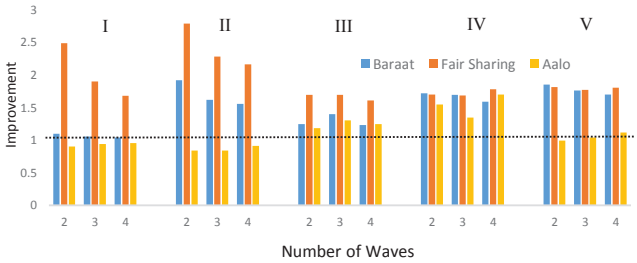

Fig. 8. The improvement with 2, 3, and 4 waves coflow in 8 pods 1G network. The evaluation is categorized into 5 groups described in table 3A.

Further, we break down Stream's performance according to different categories described in table 3 using 8 pods network with 1G and 10G switches. As illustrated in Figure 7, Stream outperforms Baraat and FS across all categories in both 1G and 10G networks. Stream's lower average CCT compared to FS results from the higher resource dedicated to higher priority coflow. Especially for smaller coflows, Stream outperforms FS by up to 5× faster, as depicted in Figure 7a. Also, Stream outperforms Baraat by up to 3× better in group I and II (Figure 7a). Baraat's performance suffers from lower priority mice coflows queuing behind higher priority larger coflows. Stream avoids this problem by allowing smaller coflows to jump ahead of the queue by deprioritizing larger coflows. On average, Stream performs comparably well to Aalo. Stream slightly trails behind Aalo for smaller coflows, an expected outcome for centralized system with complete information. This explanation does not address why Stream converges quicker than Baraat when the traffic load decreases (Figure 7). This question will be addressed later in this paper.

Notice in figure 5 that as network size scales up (k-pod is increased from 8 to 48), the average CCT improvement converges because there are more resources available and the traffic becomes more distributed from load balancing with ECMP.

Multi-wave scheduling. We modify the original trace by varying the maximum number of concurrent senders in each wave according to configuration provided by [4] as described in table 3. In Figure 8, we demonstrate the importance of coflow states across waves in 8 pods network. Stream outperforms FS across waves by 1.7× and up to 2.8× with smaller coflows. Stream outperforms Baraat up to 1.9× and shares similar performance with Aalo across waves and categories. Stream's ability to approximate the states of a coflow as a whole give it an advantage over FS. Stream allows mice coflows to jump ahead of large coflows even when they arrive later, while in Baraat mice coflow that come later may end up queuing behind higher priority large coflows.

Bursty traffic. We consider another scenario in datacenter [17,32] where coflows arrive at the same time. The simulation is performed in 8 pods 1G and 10G networks. The original trace is modified such that all coflows arrive within the same interval. Since Aalo and Baraat use FIFO in their

schemes, we keep the same coflow ID and FIFO setting as previous experiments. In 1G network, Stream outperforms Baraat and FS by at least 1.9× faster on average (Figure 9a). Notice that for coflow group II, Stream performs up to 4× better than both Baraat and FS. Stream again achieves similar outcomes with Aalo across the groups in this scenario. In 10G network, Stream outperforms both Baraat and FS by 1.6× and 1.7× (Figure 9b) respectively, while Stream is within 7% of Aalo across the groups.

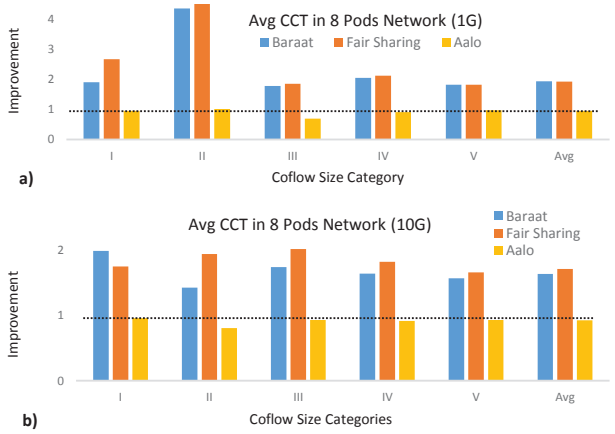

Fig. 9. Improvement average CCT in bursty traffic in 1G and 10G networks.

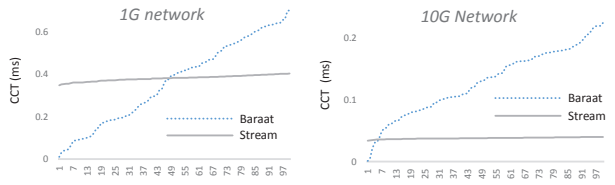

Fig. 10. The CCT of the first 100 completed coflows with Baraat and Stream in 1G and 10G networks.

In the following discussion we demonstrate why Stream outperforms Baraat. Notice that in Figure 10, CCTs of the first 100 coflows from Stream is flat, because they are processed almost simultaneously and they complete at almost the same time. In contrast, Baraat's CCTs of the first 100 coflows rise linearly. This is because in FIFO, coflow that is queued in the back must wait until all coflows ahead of it are processed. Thus when coflows all arrive within the same interval, those with lower priority end up with a longer wait in the queue. The waiting time is even longer when there are more high priority large coflows in the queue, because more network resource are allocated to large coflows. As shown in Figure 10, the higher the number of mice coflows, the longer is the waiting time for mice coflows in the back of the queue.

We refer this phenomenon as *LM-Effect* which occurs when there is more capacity allocated for limited multiplexing (LM) than FIFO. Furthermore, LM-Effect is propagated as flows traverse more queues, increasing the gap between Stream and Baraat. With this insight, the intersecting lines in Figure 10 can be interpreted as the limit of Baraat's improvement over Stream. Stream performs better than Baraat when there is a higher number of mice coflows, especially in datacenter where the majority (at least 90%) of the population is mice coflows.

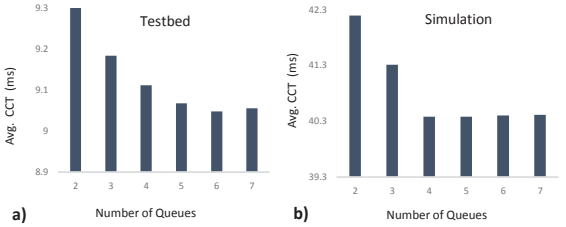

Fig. 11. Coflow scheduling with different number priority queues through testbed and simulation experiments.

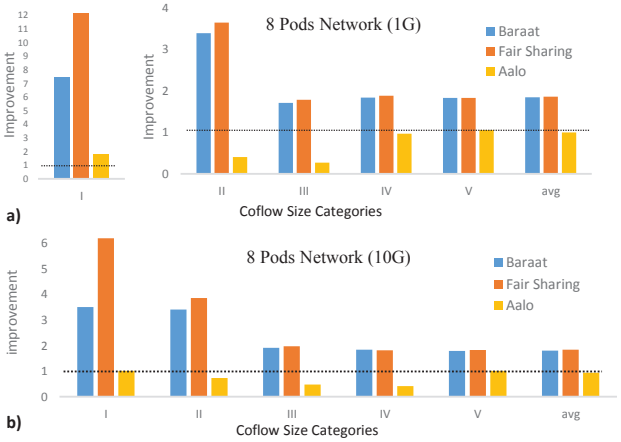

Fig. 12. Performance Improvement of Coflow with Many-to-many pattern using *TPC-DS query-42* benchmark in 8 pods 1G and 10G networks.

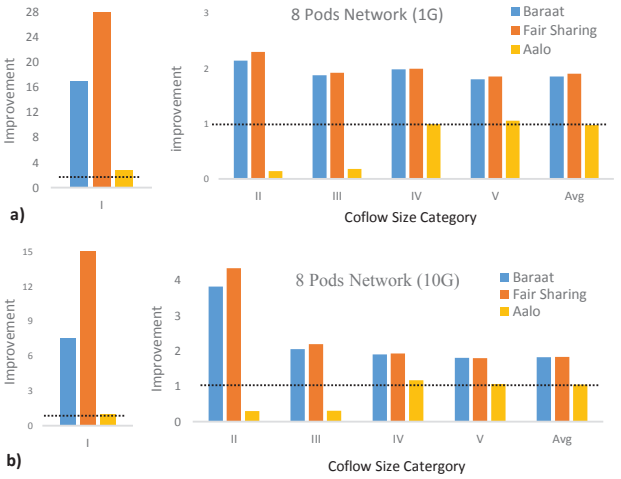

Fig. 13. Performance Improvement of Coflow with Many-to-many pattern using *Facebook-Tao structure* benchmark in 8 pods (a) 1G and (b) 10G networks.

Impact of number of queues: We conduct two experiments with 2 to 7 priority queues in our testbed using similar setup as in our previous testbed experiment with 30 coflows, and through a simulation with 8 pods network and 1000 coflows. The experiments are conducted in many-to-one scenario. Here, our results show that 4 queues is sufficient to achieve satisfactory result, similar to the findings in [5, 25]. We observe that the performance improvement affected by the number of queues follows the pattern of diminishing returns (Figure 11), which confirms Theorem 1. Here, we observe that the population of coflows in queue decreases as the number of queue increases, as expected in a heavy tail pattern.

Many-to-many pattern. We utilize Cloudera Industrial benchmark, TPC-DS query-42 (TPC-DS) [4], and Facebook Tao structure (FB-Tao) [28, 32] to evaluate Stream in many-to-many scenario (because Facebook trace only consists of coflow with many-to-one). We incorporate benchmarks and insights from [3, 4, 19, 21, 23, 24, 32] and reorganize the original trace to generate a more realistic trace according to DAG structure in Figure 2a and 2b. Each DAG structure is made up of sub-coflows that are actually exact replications of a coflow taken from the original trace; and each DAG structure is mapped to a different coflow from the original trace. The coflow size with many-to-many pattern is described in table 3. Overall, Stream performs better than Baraat and FS in both TPC-DS and FB-Tao structures, and performs on average comparable to Aalo.

With TPC-DS benchmark Figure 12 demonstrates that Stream is 1.85× better (on average) in comparison to Baraat and FS, while Stream and Aalo shares similar performance on average in both 1G and 10G networks. Also notice in Figure 12 that Stream outperforms Baraat, FS, and Aalo in category I by 7.43×, 12.12×, and 1.79× respectively in 1G network. In 10G network Stream performs better by 3.51×, 6.19×, and 1.02× than Baraat, FS, and Aalo respectively. In summary, relative to both Baraat and FS, Stream is at least 1.71× better in 1G network and 1.83× better in 10G network. Stream's performance is comparable to Aalo on average, except in the middle category in both 1G and 10G network.

With FB-Tao, on average, Stream outperforms Baraat and FS, by 1.75× and 1.833× faster respectively in 1G network (Figure 13), while Stream achieves a comparable outcome to Aalo. Stream also outperforms Baraat and FS by average 1.85× and 1.9× respectively in 10G network, and Stream is only within 2% to Aalo. Moreover, Stream also outperforms Baraat, FS, and Aalo with smaller coflow from category I by 16.9×, 28.79×, and 2.81× respectively in 1G network, and 7.53×, 15.68×, and 1.1× respectively in 10G network. In Summary, Stream outperforms both Baraat and FS by at least 1.7× in both 1G and 10G networks. In comparison to Aalo, Stream performance is comparable across category except in 1GB-10GB and 10GB-100GB categories.

Stream performs overall better than Baraat and FS in this scenario. By using WP, IR, and CP approaches, Stream is able to quickly gather information (e.g. number of sub-coflows in a coflow and sub-coflow state) and rapidly estimate coflow state. Therefore, Stream can quickly differentiate between small and large coflows and allocate the appropriate resources. In contrast, Baraat's scheduler only utilizes information that is available at the switch, which may result in less information for scheduling decision. As for FS, its performance is inferior caused by lack of coordination.

On average, Stream's performance is comparable to that of Aalo. Observe specifically category 1 (6MB-1GB), Stream outperforms Aalo by up to 2.8×. This is because in Aalo large and mice coflows may be processed together until a large coflow is detected when bytes received exceeds the threshold of mice coflow. This could lead to lower CCTs for mice coflows. On the other hand, Stream differentiates between small and large coflows at sub-coflow level because one of our assumptions is that a mice coflow may consist of small sub-coflows. Stream demotes large sub-coflows when their individual bytes received exceeds the threshold of mice sub-coflow. This way, a large coflow consisting of large sub-coflows can be deprioritized early, even before it exceeds the threshold of mice coflow. In the case of large coflow with many mice sub-coflows, it will be detected by the parents of mice sub-coflows with our Child-to-Parent scheme.

For categories II and III which makes up to 20% of total coflows, Aalo is more advantageous over Stream (0.4×) because Aalo is a centralized system with a global view, enabling it to be more precise in distinguishing coflows with similar characteristics, leading to better performance in these two categories. This slight disadvantage does not negate Stream's superior performance in all categories compared to other decentralized schemes.

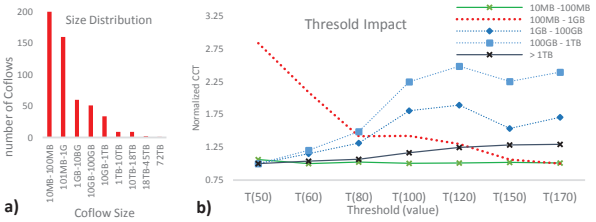

Fig. 14. The impact of threshold value for first priority queue in 1G network with Facebook TAO structure in Many-to-many scenario.

Trade-off. To evaluate how threshold selection may impact CCTs in Stream, we employ different values as the threshold for the highest priority queue in 8 pods 1G network of 4 priority queues with FB-Tao benchmark. As threshold value increases, Stream allows larger size coflows to be processed as mice coflows. While doing this improve the CCTs of some coflows in the highest priority queue, it degrades others in the same queue (Figure 14). This is because more coflows are competing for the resources. The other effect is that a longer processing delay in higher priority queue means a longer wait in lower priority queue. This finding is consistent with Kleinrock's Conservation Law for priority scheduling [26] which says that we cannot improve the response time of one class of task by increasing its priority without hurting the response time of at least one other class. Kleinrock's Conservation Law also applies to Baraat and Aalo where both schemes sacrifice the performance of mice coflows to resolve starvation of large coflows.

V. Related Work

One of the early works on coflow scheduling is Orchestra [6], where coflows are scheduled using FIFO. Varys [4] and Aalo [5] later improved the performance in [6] by prioritizing smallest-bottleneck-first and smallest-total-size-first in their scheduling mechanisms. In comparison to other approaches, Aalo [5] assumes coflow size is not known ahead of time. RAPIER [7] and OMCoflow [37] incorporate routing

algorithm into their schemes. Likewise, CORA [8] integrates resource allocation solution into its flow scheduling scheme. Following that, the authors of [9] consider coflows with different levels of importance and reformulate the problem into weighted CCTs minimization problem. CODA [36] is the first work to leverage machine learning techniques to infer and schedule coflows. These are all centralized approaches that may provide good performance. However, centralized approaches are generally hindered by the high overhead cost of managing a centralized system.

The other alternative is the decentralized approach. The current decentralized coflow scheduling scheme is pioneered by Baraat [3], a heuristic that adopts FIFO with some level of multiplexing that allows mice flows to be processed in the background in the presence of large coflows. Otherwise, mice flows are processed according to FIFO. However, this approach has a few drawbacks. Since the scheduling decision is made locally at switches, this makes gathering information on coflow more challenging for the scheduler if flows of a same coflow that do not traverse through the same switch. Additionally, the solution also requires switch source code modification, which is not deployable friendly. Optas [27] is the other decentralized scheduling, but is designed specifically for a special case of coflows of size 4MB or less. Different from these solutions, our proposal solves general coflow scheduling problem by opportunistically taking advantage of many-to-one and many-to-many patterns.

VI. Conclusion

Stream is a coflow scheduling scheme that minimizes CCT in decentralized fashion. It opportunistically takes advantage of the receiver in many-to-one and many-to-many communication patterns, utilizing C-SJF and WP-IR-CP approaches. The outcomes from both our testbed experiments and large-scale network simulation demonstrate that Stream is an effective and practical solution in improving network performance in datacenter, performing particularly well in heavier traffic. Finally, we also demonstrate that our solution is readily implementable.

Acknowledgements. This work is supported in part by the Hong Kong RGC ECS- 26200014, GRF-16203715, GRF-613113, CRF-C703615G, HKUST-PDF, and the China 973 Program No.2014CB340303. We thank our shepherd Javad Grader, the anonymous ICNP reviewers, and members from HKUST SING Lab for their valuable feedback. We thank Zhouwang Fu and Ge Chen for assisting our experiments.

Reference

[1] http://www.pica8.com/documents/pica8-datasheet-picos.pdf
[2] M. Alizadeh, , et al., “Less is More: Trading a little Bandwidth for Ultra-Low Latency in theDataCenter”, Usenix NSDI 2012.
[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)”, ACM 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] Y. Yu, et al., “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High Lelvel Language”, USENIX OSDI, 2012.
[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] L. Kleinrock, “Queuing Systems, Vol 2 Coomputer application”, New York, Wiley, 1976.
[27] Z. Li, et al, “OPTAS: Decentralized Flow Monitoring and Scheduling for Tiny Tasks”, IEEE INFOCOM, 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. Vahdat, et al,”Scale-Out Networking in the Data Center”, IEEE Micro, Vol. 30 , Issue 4, p. 29-41, 2010.
[32] A. Roy, et al, “Inside the Social Network’s (Datacenter) Network,” in ACM SIGCOMM 2015.
[33] G. Anantharnarayanan, “PACMan: Coordinated memory caching for parallel” in USENIX NSDI, 2012.
[34] R. Bifulco, et al.”Improving SDN with InSpired Switches”, ACM SOSR, 20016.
[35] H. Li, et al.,” Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks”, IEEE SOCC, 2014.
[36] H. Zhang, et al.,”CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark”, ACM SIGCOMM, 2016.
[37] Y. Li, et al.,”Efficient Online Coflow Routing and Scheduling”, ACM MOBICHOC, 2016.