Decentralized Deadline-Aware Coflow Scheduling for Datacenter Networks

Shouxi Luo, Hongfang Yu, Lemin Li
Key Laboratory of Optical Fiber Sensing and Communications, Ministry of Education
University of Electronic Science and Technology of China, Chengdu, P. R. China

***Abstract*—This paper presents D^2-CAS, a novel decentralized coflow scheduling system, to minimize the rate of deadline missed coflow for datacenter networks. To design D^2-CAS, we first formulate the deadline-missed coflow minimization problem and show its equivalence with the well-known problem of minimizing the late jobs in a *concurrent open shop*, which is NP-hard in ordinary sense. Inspired by *Moore-Hodgso's algorithm (MHA)*, the optimal solution for minimizing late jobs on a single machine, we design an efficient coflow schedule algorithm, CS-MHA, and further propose its decentralized implementation, D^2-CAS. Basically, each sender in D^2-CAS periodically runs a part of CS-MHA to get a local suggestion for flow priority assignment, and then multiple senders of a coflow negotiate for an orchestrated priority by leveraging their common data receivers. Via delivering each coflow's packets with its negotiated priority, senders finally carry out efficient deadline-aware coflow scheduling in a decentralized fashion.**

To the best of our knowledge, this is the first paper that theoretically investigates the deadline-aware coflow scheduling problem, while D^2-CAS is the first decentralized solution. Real parameter driven simulations imply that, with the simple yet efficient mechanism, D^2-CAS greatly outperforms all existing solutions on reducing the deadline-missed coflows (e.g., outperforms Varys more than $2\times$).

I. Introduction

Cluster computation applications, such as Hadoop, Spark, and search query platforms, are widely employed to process data- analytics and query in datacenters [1]–[5]. In these applications, a job is divided into multiple computation- or inquiry- stages, among which, a successive stage involves a group of parallel flows to fetch its input data from the previous stage's output. The term *coflow* is used to abstract such a group of parallel flows [3, 4], which share the common performance goal—This is because only completing all flows in the group can push forward a job's process. Due to the soft-real time nature of today's latency-sensitive jobs (e.g., web search, online retail, advertisement, etc.) [1]–[4], the involved coflows carry widely varying strict deadlines. That is to say, a coflow is useful and contributes to the *application goodput* if, and only if, it completes within the deadline. Accordingly, in the view of datacenter operators, scheduling flows to maximize the number of coflows that meet the deadline, becomes a crucial performance goal for flow scheduling [3, 4].

Although numerous flow scheduling proposals [1, 2, 5]–[12] have been proposed for DCNs over the past years, they either aim at optimizing the average coflow completion times (e.g., [5, 6, 9]–[12]), a quite different objective from deadline-aware scheduling, or reducing the deadline miss rate at flow-level (e.g., [1, 2, 7, 8]), which are proven to be inefficient for coflow-level optimizations [4, 9, 11]. To the best of our knowledge, Varys [4] is the only existing scheme designed for minimizing the missed deadlines for coflow. However, it suffers from two problems. Firstly, the centralized design makes it hard to scale for today's large-scale datacenters [5, 9]. Secondly, as this paper will show, Varys schedules coflows with a simple non-preemptive admission control, which results in serious performance loss.

In this paper, we attempt to find a solution that is high efficient and easy to scale up. To this end, we first look into the theoretical problem—How to schedule coflows is optimal for the minimization of missed deadlines? Theoretically, we realize that scheduling coflows to minimizing their deadline misses is equivalent to the well-known problem of minimizing late jobs in a *concurrent open shop*, which is strongly NP-hard in ordinary sense [13]. Motived by this, we borrow ideas from concurrent open shop scheduling for designing coflow scheduling schemes. More specially, we propose CS-MHA, an efficient deadline-aware coflow scheduling heuristics, based on *Moore-Hodgso's algorithm (MHA)* [14], which is known as the optimal algorithm for the deadline-aware job scheduling on a single machine.

To make a practical and scalable solution, we further design D^2-CAS, Decentralized, Deadline-driven, Coflow-Aware Schedule system, to carry out CS-MHA schedules in decentralized fashions. Basically, on handling these time-sensitive coflows, each sender in D^2-CAS periodically runs a part of CS-MHA, i.e., the *Moore-Hodgso's algorithm*, to compute a desired priority for every coflow that it hosts; then multiple senders of a coflow determine a final priority value for this coflow (i.e., all the involved flows) by using a simple, robust negotiation plane among them and their common data receivers; finally, each sender let packets belonging to this coflow carry the negotiated priority when delivering.

It is worth noting that, such a scheduling mechanism by design shares the same workflow with that of D-CAS, the state-of-the-art decentralized solution for average coflow completion time optimization [9, 11], except for how desired priorities are generated on sender—This is the key to achieve efficient deadline-aware coflow scheduling or average coflow completion optimization. With this compatible design, datacenter operators can easily upgrade a D-CAS enabled system to handle deadline-sensitive coflows by leveraging the MHA-

978-1-4799-6664-6/16/$31.00 ©2016 IEEE

based policy to generate the desired priority.

We develop a flow-level simulator based on that of Varys [4, 15] to evaluate the performance of D^2-CAS. Our results from traces (generated with real parameters [4]) driven simulations imply that, following the simple yet theoretical scheduling proposal, even as a decentralized solution, D^2-CAS significantly outperform centralized Varys, the state-of-the-art deadline-aware coflow scheduler.

In summary, we mainly make two contributions:

- We show that coflow scheduling is equivalent to the well-known *concurrent open shop* scheduling on minimizing the number of missed deadlines. Such a finding greatly extends that of prior art [11] and brings theoretical insights for scheduling latency-sensitive coflow.
- We propose an efficient deadline-aware coflow scheduling algorithm, CS-MHA, based on the findings of current open shop scheduling, and further design a practical coflow scheduling system, D^2-CAS, to implement CS-MHA in decentralized fashions. Trace-driven simulations indicate that D^2-CAS would let about $17\times$ more coflows meet their deadline compared with the case of no schedule scheme employed, and reduce the missed deadlines more than $2\times$ compared with Varys.

The rest of the paper proceeds as follows. Section II formulates the deadline-aware coflow scheduling problem and shows its equivalence with the concurrent job scheduling. Then, Section III sketches the design of D^2-CAS, and Section IV evaluates its performance. At last, Section V summarizes the paper.

II. Problem Statement and Theoretical Analysis

In this section, we present the formal model of minimizing the missed deadlines for a given set of coflows, and show that such a schedule problem is equivalent to the well-known NP-hard problem of minimizing the number of late jobs in a *concurrent open shop* (or *customer order scheduling*) [13].

A. Problem Formulation

In our analysis, for simplicity, we abstract the entire data center network fabric as a non-blocking switch [4, 9, 16] interconnecting all hosts, and each port (both ingress and egress) has one unit capacity. Corresponding, bandwidth competition only occurs at these ingress ports or egress ports.

Consider a data center with m hosts serving n coflows, labeled as $D^{(1)}, D^{(2)}, \ldots, D^{(n)}$, respectively. Without loss of generality, we suppose that each coflow involves $m \times m$ parallel flows between these hosts, and denote $d_{i,j}^{(k)}$ to be the size of flow from host i to host j belonging to coflow $D^{(k)}$. Following the notations of [11], if there are several flows from host i to host j belonging to coflow k, $d_{i,j}^{(k)}$ stands for their total volume; and if the coflow involves no flow from host i to host j, $d_{i,j}^{(k)}$ is set as 0. For coflow k, it has a deadline, L_k, implying the date by which it should be finished.

We further let C_k be the completion time of this coflow and introduce a binary variable x_k to denote whether the coflow completes within its deadline; then, for these coflows, maximizing the number of coflow meeting its deadline is to pursue the object of Equation (1), while subjecting to the constraint of bandwidth capacity at ingresses and egresses, as mathematical program $\{(1), (1a), (1b), (1c)\}$ says.

$$\text{Maximize } \sum_{k=1}^{n} x_k \tag{1}$$

subject to,

$$x_k = \begin{cases} 1 & C_k \le L_k \\ 0 & C_k > L_k \end{cases} \tag{1a}$$

$$\forall k, i: \sum_{\forall l: C_l \le C_k} \sum_{j=1}^{m} d_{i,j}^{(l)} \le C_k \tag{1b}$$

$$\forall k, j: \sum_{\forall l: C_l \le C_k} \sum_{i=1}^{m} d_{i,j}^{(l)} \le C_k \tag{1c}$$

B. Equivalency to Deadline-Aware Concurrent Open Shop Scheduling

Intuitively, coflow scheduling is quite similar to the well-known *concurrent open shop scheduling* problem [13, 17, 18].[1] For instance, we can treat the job in concurrent open shop scheduling as a specified type of coflow whose demand matrix is diagonal (i.e., $d_{i,j}^{(k)} = 0$ when $i \neq j$). Indeed, recent literature has proven that they do be equivalent if the objective is to optimize the average completion time [11]. In this part, we go further by showing this equivalence also exists on minimizing the missed deadlines in common sense.

We start by considering the schedule of a single coflow. Given a coflow, $D^{(k)}$, for example, let $\rho(D^{(k)})$ (or ρ_k for short) denote its maximum total loads on hosts, which can be calculated by Equation (2). In this paper, we assume that coexisting flows preempt the network bandwidth according to their priorities [8, 11], and flow senders perform ideally TCP-alike rate controls [11].[2] Consequently, as no other coflow exists, the coflow must be finished at time $\rho(D^{(k)})$. By letting $g_i^{(k)} = \sum_{j=1}^{m} d_{i,j}^{(k)}$ and $g_{i+m}^{(k)} = \sum_{j=1}^{m} d_{j,i}^{(k)}$, we get $\rho(D^{(k)}) = \max_{i=1,\ldots,2m} g_i^{(k)}$, which is exactly the case of scheduling one concurrent job on $2m$ machines, where $g_i^{(k)}$ stands for its task on machine i.

$$\rho(D^{(k)}) = \max\left\{ \max_i \left\{ \sum_{j=1}^{m} d_{i,j}^{(k)} \right\}, \max_j \left\{ \sum_{i=1}^{m} d_{i,j}^{(k)} \right\} \right\} \tag{2}$$

Then, let's consider the schedule of two coflows, saying $D^{(s)}$ and $D^{(t)}$. For ease of description, we use the "+" operation of coflow defined by literature [11] to represent the

[1] *Definition of Concurrent Open Shop Scheduling:* Suppose that there are m machines with each capable of processing one operation type. We have n jobs; each job involves multiple types of operations, which can be processed in parallel. The objective of job scheduling can be minimizing their total job completion times, or minimizing the number of late jobs when each job has a hard deadline (or due date) [13, 17, 18].

[2] This assumption is only used for theoretical analysis. Our proposed solutions, CS-MHA and D^2-CAS, do not require it to be hold anymore.

"merging" of coflows in this paper. For example, $D^{(s)}+D^{(t)}$ will produce another coflow $D^{(u)}$, whose demands from host i to host j (i.e., $d_{i,j}^{(u)}$) is defined by $d_{i,j}^{(s)}+d_{i,j}^{(t)}$. When treating $D^{(s)}$ and $D^{(t)}$ together as a "big" coflow, $D^{(u)}$, we obtain the similar conclusion for it—$D^{(u)}$ must be able to complete at $\rho(D^{(u)})$ as well. That is to say, both $D^{(s)}$ and $D^{(t)}$ can be finished within $\rho(D^{(u)})$ simultaneously, i.e., $\max\{C_s, C_t\} = \rho(D^{(s)}+D^{(t)})$. By substituting $D^{(s)}+D^{(t)}$ into Equation (2), we further obtain $\max\{C_s, C_t\} = \max_{i=1,\ldots,2m}(g_i^{(s)}+g_i^{(t)})$ as Equation (3) shows, which is exactly the case of scheduling two concurrent jobs on $2m$ machines, where $g_i^{(k)}$ stands for the task of job k on machine i.

$$\begin{aligned}
&\max\{C_s, C_t\} = \rho(D^{(s)}+D^{(t)}) \\
&= \max\left\{\max_i\left\{\sum_{j=1}^{m}(d_{i,j}^{(s)}+d_{i,j}^{(t)})\right\}, \max_j\left\{\sum_{i=1}^{m}(d_{i,j}^{(s)}+d_{i,j}^{(t)})\right\}\right\} \\
&= \max\left\{\max_i\left\{\sum_{j=1}^{m}d_{i,j}^{(s)}+\sum_{j=1}^{m}d_{i,j}^{(t)}\right\}, \max_j\left\{\sum_{i=1}^{m}d_{i,j}^{(s)}+\sum_{i=1}^{m}d_{i,j}^{(t)}\right\}\right\} \\
&= \max_{i=1,\ldots,2m}(g_i^{(s)}+g_i^{(t)})
\end{aligned} \tag{3}$$

Note that, we can treat any number of coflows as a single "big" coflow. That is to say, following the same approach proposed above, we obtain the way to construct relevant concurrent open job scheduling for every coflow scheduling. Accordingly, for the specific problem of scheduling n coflows to minimize the missed deadlines, by letting $g_i^{(k)}$ as job k's task on machine i, we convert it to an equivalent problem of scheduling n jobs on $2m$ machines to minimize the number of late jobs. After solving this late job minimization problem, we will get a set of admitted jobs and their permutation order towards optimal scheduling. Finally, by setting each coflow's flow priority according to this permutation order, we can carry out optimal deadline driven coflow scheduling as well.

At this point, we have sketched the equivalence between coflow scheduling and concurrent open shop scheduling on minimizing the number of missed deadline, and introduced a simple mechanism to construct solutions for coflow scheduling from that of the dual concurrent open shop scheduling.

C. Computational Analysis

Lots of studies have attempted to design efficient schedule schemes for minimizing the number of late jobs [13, 14, 17]. They imply that, in the case of single machine scheduling, one can find an optimal solution in $O(n\log n)$ time by using the Moore-Hodgso's algorithm [14]; however, once there are more than one machine, the problem is strongly NP-hard in the ordinary sense [13, 17]. As is known, today's data center generally involves hundreds of thousands of machines [19]. Thus, in ordinary sense, minimizing the number of missed deadlines for coflow is also strongly NP-hard.

It is worth noting that, if there is only one bottleneck port or the coflow loads on ports are in agreeable conditions [13],[3] the capacity constraints on multiple ports can be reduced into a single one; then Moore-Hodgso's algorithm is able to find the optimal schedule scheme for these cases as well.

III. Towards Practical Solutions

Inspired by the findings of concurrent open shop scheduling, we design CS-MHA, an efficient deadline-aware coflow schedule heuristics, and further proposed its decentralized implementation, D^2-CAS (Decentralized, Deadline-driven, Coflow-Aware Scheduling system). In rough, D^2-CAS (or CS-MHA) decomposes the m-hosts coflow scheduling problem into m independent single host schedule problems and employs Moore-Hodgso's algorithm for the schedule on each host. To erase the inconsistent of scheduling decisions between hosts, D^2-CAS leverages a simple negotiation mechanism to orchestrate the decisions for each coflow.

In the following, we firstly present the design of the schedule algorithm, CS-MHA, in Section III-A, then discuss how D^2-CAS enforces this strategy in decentralized fashions in Section III-B.

A. Schedule Algorithm Design: CS-MHA

Local Schedule on Each Port. For the schedule on each port/host, CS-MHA computes the schedule orders (standing for coflows' flow priorities) by using the optimal Moore-Hodgso's algorithm. For simplicity, we use MHA to denote this local schedule strategy on each port.

Suppose that there are n coflows on port i (an ingress or egress). MHA first sorts coflows in order of non-decreasing deadline (denote the order as U_i), and initializes the scheduled coflow set $S_i = \emptyset$, the excluded coflow set $E_i = \emptyset$, and maximum load $\lambda_i = 0$. Then, MHA repeatedly picks an unscheduled coflow $D^{(k)}$ from U_i, adds it into S_i, and updates $\lambda_i = \lambda_i + g_i^{(k)}$. If λ_i does not exceed this coflow's deadline, L_k, MHA continues to schedule the next unscheduled coflow; otherwise, MHA needs to remove the coflow with the heaviest load on this port (denoted as $\bar{k} = \arg\max_{k\in S_i}\{g_i^{(k)}\}$) from S_i, let $\lambda_i = \lambda_i - g_i^{(\bar{k})}$, and add it into the excluded set E_i, before handling the next unscheduled coflow.

After all coflows are checked, MHA has found the admitted coflow set S_i for port i. If port i happens to be the only bottleneck port, or coflow loads on ports are agreeable, by simply assigning decreasing flow priorities to coflows in S_i in increasing order of their deadlines, one can obtain the optimal result of minimizing missed deadlines.

Global Orchestration. In practice, coflow loads are generally disagreeable on ports [4]; consequently, different ports would have various admitted sets. A straightforward solution for this is to use $\cap_{\forall i}S_i$ as the global admitted coflow set, and deny all the coflows in set $\cup_{\forall i}E_i$. However, such a strategy is

[3] $\{g_i^{(k)}\}$ is agreeable means: for any port pair $\langle i_1, i_2\rangle$ and coflow ID pair $\langle k_1, k_2\rangle$, if $g_{i_i}^{(k_1)} \le g_{i_1}^{(k_2)}$ then $g_{i_2}^{(k_1)} \le g_{i_2}^{(k_2)}$.

conservative; the inconsistency between admitted sets might result in a coflow being denied even if it can meet the deadline—Because the reject of a prior coflow has released sufficient bandwidth for this coflow. As a remedy, we can let some "denied" coflows (i.e., those in $\cup_{\forall i} E_i$) use the remaining bandwidth, which might get finished within their deadlines. In this paper, we simply let coflows in $\cup_{\forall i} E_i$ preempt the residual bandwidth in increasing order of $\frac{\max_{\forall i} g_i^{(k)}}{L_k+\epsilon}$, the Minimum Bandwidth required on the Bottleneck port, i.e., MinBB for short, where ϵ is a tiny constant avoiding the divide-by-zero error. The design idea is, if a coflow has a smaller MinBB, it is more likely to catch up with its deadline by using the residual bandwidth. Finally, the entire process of how a set of coflows are scheduled is as Algorithm 1 sketches, which is $2m$-approximation as Theorem 1 shows.

Algorithm 1 CS-MHA: Centralized Scheduling Based on MHA

Input: number of coflows n; number of hosts m; flow demands $d_{i,j}^{(k)} \in \mathbb{R}_{\geq 0}$ for all $k \in N$, $i \in M$ and $j \in M$
Output: permutation order (i.e., priority) of coflows $\pi : \{1, \ldots, n\} \mapsto N$ ▷ N is the set of coflow IDs
1: $g_i^{(k)} \leftarrow \sum_{j \in M} d_{i,j}^{(k)}$ for all $k \in N$ and $i \in M$ ▷ Ingress
2: $g_{j+m}^{(k)} \leftarrow \sum_{i \in M} d_{i,j}^{(k)}$ for all $k \in N$ and $j \in M$ ▷ Egress
3: $S \leftarrow \emptyset$; $E \leftarrow \emptyset$
4: **for** $i \leftarrow 1, 2, \ldots, 2m$ **do**
5: Compute S_i, E_i with **MHA**(Moore-Hodgso's algorithm [14])
6: $S \leftarrow S \cap S_i$; $E \leftarrow S \cup E_i$
7: **end for**
8: Sort S in increasing order of L_k and denote the list as P
9: Sort E in increasing order of $\frac{\max_{\forall i} g_i^{(k)}}{L_k+\epsilon}$ and append them to the right of P
10: $\pi(i) \leftarrow P[i]$ for $i \leftarrow 1, 2, \ldots, |P|$

Theorem 1. *Algorithm 1 (CS-MHA) is $2m$-approximation.*

Proof. Denote $\hat{E}$ to be the set of missed deadlines in the schedule constructed by Algorithm 1, and *opt* to be the optimum result. Obviously, for any port i, $|E_i| \leq opt$ and $\hat{E} \subseteq \cup_{i=1}^{2m} E_i$; namely, $|\hat{E}| \leq |\cup_{i=1}^{2m} E_i| \leq \sum_{i=1}^{2m} |E_i| \leq 2m \times opt$. Thus, we obtain $\frac{|\hat{E}|}{opt} \leq 2m$. □

It is worth noting that, this theoretical bound is loose as the analysis does not take the effect of the remedy shown in Line 9 into account. Numerical results indicate that such a schedule strategy obtains very good results.

B. Decentralized Implementation: D²-CAS

Recent studies have reported that today's large-scale DCNs prefer decentralized coflow schedule systems [5, 9, 11]. Accordingly, we design D²-CAS to implement CS-MHA in a decentralized fashion.

In CS-MHA, by directly setting a coflow's priority with the interval to its deadline or its MinBB, we can avoid the sort operations for S and E (Line 8-10). Such a scheme also decouples the process of inter-coflow scheduling, so that each coflow can schedule its own flows individually. For the convenience of description, we design the flow priority of each coflow as a tuple $\langle P_1, P_2 \rangle$, where P_1 is 0 or 1, denoting whether that coflow belongs to set E or not, and P_2 is a positive real number denoting the interval to deadline or the MinBB, depending on the value of P_1.[4] Consequently, data senders only need to set the flow priority of coflow k with tuple $\langle 0, L_k \rangle$ if it is in set S, or $\langle 1, \frac{\max_{\forall i} g_i^{(k)}}{L_k+\epsilon} \rangle$ otherwise.

Inspired by D-CAS [9], the state-of-the-art decentralized coflow scheduling system for average completion time optimization, we design a similar robust negotiation mechanism within each coflow's data senders and receivers, to 1) test whether a coflow belongs to set $E = \cap_{\forall i} E_i$, and 2) compute $\max_{\forall i} g_i^{(k)}$ in decentralized fashions. Note that, in practice, it is hard for the data receiver to compute its S_i and E_i, because receivers could not figure out how many data it will get for each coflow in advance.[5] Consequently, D²-CAS does not take the constraint on receivers into account when making schedule decisions as well.

Algorithm 2 D²-CAS: Decentralized Scheduling Based on MHA

Operations on Sender i ▷ called every-δ
1: Get U_i, the set of unexpired coflows on this sender
2: Update L_k, the interval to coflow k's deadline, and $d_{i,j}^{(k)}$, the size of k's untransmitted data on this sender,for $k \in U_i$
3: Compute S_i and E_i for U_i with **MHA**
4: Announce $\langle 0, L_k \rangle$, coflow k's *desired-priority* (i.e., $pd_i^{(k)}$), to all k's receivers, for $k \in S_i$
5: Announce $\langle 1, \frac{\max_{\forall i} g_i^{(k)}}{L_k+\epsilon} \rangle$, coflow k's *desired-priority* (i.e., $pd_i^{(k)}$), to all k's receivers, for $k \in E_i$
6: Wait two RTTs, and denote $M_i^{(k)}$ as the set of coflow k's feedbacks; add $pd_i^{(k)}$ into $M_i^{(k)}$, for $k \in U_i$
7: Update the priority of coflow k's data transfers to $\max M_i^{(k)}$, for $k \in U_i$

Operations on Receiver j ▷ called on getting $pd_i^{(k)}$
8: Use $pd_i^{(k)}$ to update $B_j^{(k)}$, the bucket of received desired priorities this receiver cached for coflow k
9: Remove expired messages from $B_j^{(k)}$
10: Send $\max B_j^{(k)}$, coflow k's feedback-priority, to sender i

D²-CAS shares the same system framework and workflow with D-CAS. The operations data senders and receivers performing are sketched by Algorithm 2. Roughly, each data sender schedules a coflow by periodically adjusting its flow priority to a negotiated value. At each turn, the sender firstly computes its local S_i and E_i based on the

[4] In practice, it is easy to convert such tuple-style priorities to single-value priorities that data center switches support [5, 8], by using $f(\langle P_1, P_2 \rangle) = P_1 \times M_p + P_2$, where M_p is a big positive number larger than the maximum of P_2. If switches have limited priority queues [8], we can design mapping schemes to convert tuple-style priorities into the supported priority queues. We leave this to further work.

[5] Similar to prior work [4, 8], we assume that a flow's information such as total size and remaining size is available at its sender.

latest information of unexpired coflows (Line 1-3). For each unexpired coflow on it, the sender generates a *desired-priority* and announces this value to all the coflow's data receivers (Line 4-5). On getting coflow k's *desired-priority* message from host i, the receiver will choose the maximum *desired-priority* announced by the same coflow recently, and sends it back to sender i (Line 8-10). Finally, for coflow k, the sender computes the maximum priority value among *desired-priority* and feedbacks, and uses this maximum priority value to deliver k's data (Line 7). Note that, as tuple $\langle 1, x\rangle$ is always larger than tuple $\langle 0, y\rangle$ (the larger value the lower priority here), once a coflow is assigned into set E_i at a sender, all other colleague senders will know this by using such a $\max(\cdot)$ driven negotiation.

About System Overhead. The operation on each receiver is quite simple; so we mainly focus on sender operations here. Obviously, to perform deadline-aware coflow scheduling, a sender will recall MHA and announce the computed results to coflow receivers, every δ-interval. Recall that, the worst-case time complexity of MHA is $O(n \log n)$, where n is the number of coflows held by the sender. Such a computation task is quite easy for today's hardware. As for the communication overhead, each sender only needs to announce the desired priority computed by MHA to the hosts that are fetching (coflow) data from it. Compared with these coflow data transfers, such a priority information only occupies a few bytes, very trivial in today's data center networks.

IV. Performance Evaluation

In common with recent literature [4, 9], we develop a flow-level simulator to evaluate the performance of D^2-CAS by performing a detailed replay of coflow traces generated with real-world parameters [4, 15]. The preliminary numerical results implies: D^2-CAS would let about $17\times$ more coflows meet their deadline compared with per-flow fair-sharing, and reduces the missed deadlines about $2\times$ compared with Varys.

A. Setup

We use the generator (named CustomTraceProducer in [15]) and real parameters provided by Varys [4] to synthesize coflow traces. Without declaration, all tests are based on 400 coflows served by a cluster with 80 hosts, in which, both the ingress and egress of the DCN fabric are set with the capacity of 1 Gpbs. Coflows are assumed to arrive in a Poisson process with parameter λ, while all flows in a coflow are assumed to arrive simultaneously. To avoid overloading nor underloading the network, λ is set as $\frac{NetworkThroughput}{MeanOfCoflowSize}$ by defaults (Under this setting, $E(NetworkLoad) = 1$) [9]. In common with prior work [4, 8], for each coflow k, we set its deadline constraint to be $(1 + z) \times \rho_k$, where ρ_k is its minimum completion time in an empty network, and z is a randomly real number following an uniform distribution $U[0, 2x]$ [4], or an exponential distribution $Exp(x)$ [8]. Unless otherwise specified, the mean value of z is set as 1. For each set of test parameters, we generate 20 trials to calculate its average and [min, max] range (See errorbars in the result figures).

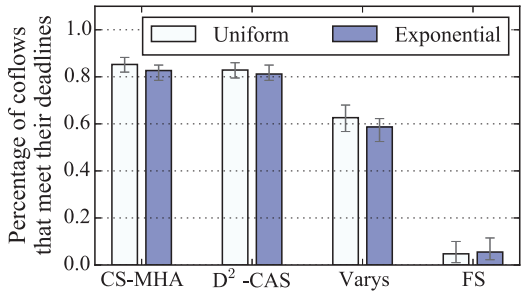

Fig. 1. D^2-CAS let more coflows meet deadlines than Varys [4] and FS, the default per-flow fair-sharing scheme.

B. Results

Fig. 1 shows that, under both uniform and exponential deadline distributions, D^2-CAS always increases the proportion of coflows that meet the deadline to 80%, which outperforms the result of the per-flow fair sharing scheme about $17\times$, and reduces the missed deadlines about $2\times$ compared with Varys—About 40% coflows miss their deadlines with Varys, while less than 20% coflows miss with D^2-CAS. The per-flow fairness performs poorly because it overlooks deadline requirements of flows when scheduling; in such cases, coflows with tight deadlines could not get finished in time. As for Varys, despite it admits and schedules coflows according to their deadlines, its naive non-preemptive schedule would works badly—Because the coflow size is heavy-tailed in practice [4, 10], the admission of a large coflow would result in many deadline-sensitive small coflows being denied.

Besides, we also observe the close performance between D^2-CAS and CS-MHA. In the test, CS-MHA performs online coflow scheduling with a centralized controller. On each coflow arrival, the controller calls Algorithm 1 to get a new priority assignment for all unexpired coflows; then senders uses the new priorities to delivery data. The small performance gap between CS-MHA and D^2-CAS suggests that neglecting the coflow loads on the receiver when making scheduling has a little effect on the schedule effectiveness.

To investigate the impact of network load and deadline requirement on schedule results, we vary the network load from 0.4 to 1.4 by controlling λ,[6] and vary the mean of the uniform/exponential distribution (i.e., x) from 0.25 to 5. Fig. 2 and Fig. 3 show how the percentage of deadline-met coflows changes. As expected, less percentage of coflows will meet their deadlines when the network is overloaded; while more coflows will meet their deadlines when the deadline distribution parameter x has a larger value. When the network is underloaded, each coflow gets more bandwidth; they are more likely to meet the deadline. Similarly, with the

[6]Note that, a coflow will automatically expire after its deadline; so the system's coflow queue always has a limited length even if the network load computed by λ is larger than 1.

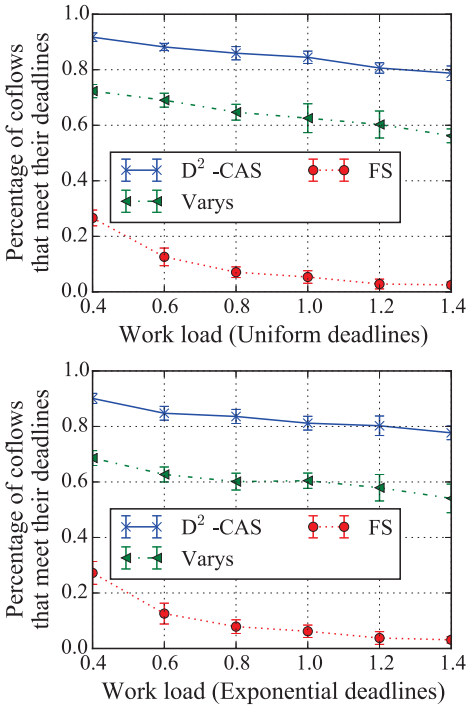

Fig. 2. Impact of network loads.

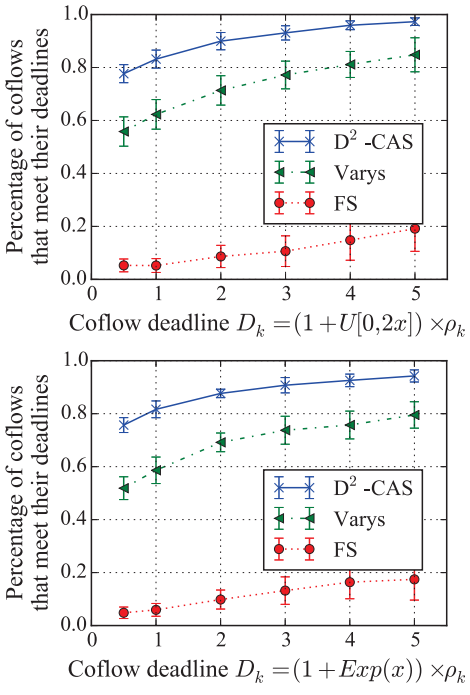

Fig. 3. Impact of deadline distributions.

mean value x increasing, coflows get looser deadlines; then more coflows can meet their deadlines.

All these results confirm the significant improvements of D^2-CAS over Varys and the per-flow fairness scheme under various settings. Specifically, the number of coflows that miss their deadlines under the schedule of D^2-CAS is always less than half of that of Varys.

V. Conclusion

This paper has shown the equivalence between coflow scheduling and concurrent job scheduling on minimizing the missed deadlines, and sketched the design of D^2-CAS, an efficient, decentralized, deadline-aware coflow scheduling solution. Preliminary simulations imply that, even as a decentralized solution, D^2-CAS is effect on letting more time-sensitive coflows meet their deadlines than centralized Varys.

Acknowledgements. This work was supported in part by the 973 Program under Grant No. 2013CB329103, the 863 Program under Grant No. 2015AA015702 and 2015AA016102, the National Natural Science Foundation of China under Grant No. 61271171, 61271165, and 61571098, the Ministry of Education - China Mobile Research Fund under Grant No. MCM20130131, and the China Postdoctoral Science Foundation under Grant No. 2015M570778.

References

[1] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM*, Aug. 2011, pp. 50–61.

[2] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware Datacenter TCP (D2TCP)," in *SIGCOMM*, Aug. 2012, pp. 115–126.

[3] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. 11th ACM HotNets*, 2012, pp. 31–36.

[4] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *SIGCOMM*, Aug. 2014, pp. 443–454.

[5] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *SIGCOMM*, Aug. 2014, pp. 431–442.

[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra." in *SIGCOMM*, Aug. 2011, pp. 98–109.

[7] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM*, Aug. 2012, pp. 127–138.

[8] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-optimal Datacenter Transport," in *SIGCOMM*, Aug. 2013, pp. 435–446.

[9] S. Luo, H. Yu, Y. Zhao, B. Wu, S. Wang, and L. Li, "Minimizing Average Coflow Completion Time with Decentralized Scheduling," in *IEEE ICC*, June 2015, pp. 307–312.

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

[11] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li, "Towards Practical and Near-optimal Coflow Scheduling for Data Center Networks," *IEEE Transactions on Parallel and Distributed Systems*, doi: 10.1109/TPDS.2016.2525767

[12] Y. Gao, H. Yu, S. Luo, and S. Yu, "Information-agnostic coflow scheduling with optimal demotion thresholds," in *IEEE ICC*, May 2016.

[13] B. Lin and A. Kononov, "Customer order scheduling to minimize the number of late jobs," *European Journal of Operational Research*, vol. 183, no. 2, pp. 944 – 948, 2007.

[14] J. M. Moore, "An n job, one machine sequencing algorithm for minimizing the number of late jobs," *Management Science*, vol. 15, no. 1, pp. 102–109, 1968.

[15] M. Chowdhury, "Flow-level simulator for coflow scheduling used in varys," https://github.com/coflow/coflowsim.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, Aug. 2008, pp. 63–74.

[17] T. A. Roemer, "A note on the complexity of the concurrent open shop problem," *J. of Scheduling*, vol. 9, no. 4, pp. 389–396, Aug. 2006.

[18] M. Mastrolilli et al., "Minimizing the sum of weighted completion times in a concurrent open shop," *Operations Research Letters*, vol. 38, no. 5, pp. 390–395, Sep. 2010.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *SIGCOMM*, Aug. 2009, pp. 51–62.