

An Improved Bound for Minimizing the Total Weighted Completion Time of Coflows in Datacenters

Mehrnoosh Shafiee, *Student Member, IEEE*, and Javad Ghaderi, *Member, IEEE*

Abstract—In data-parallel computing frameworks, intermediate parallel data is often produced at various stages which needs to be transferred among servers in the datacenter network (e.g. the shuffle phase in MapReduce). A stage often cannot start or be completed unless all the required data pieces from the preceding stage are received. *Coflow* is a recently proposed networking abstraction to capture such communication patterns. We consider the problem of efficiently scheduling coflows with release dates in a shared datacenter network so as to minimize the total weighted completion time of coflows. Several heuristics have been proposed recently to address this problem, as well as a few polynomial-time approximation algorithms with provable performance guarantees. Our main result in this paper is a polynomial-time deterministic algorithm that improves the prior known results. Specifically, we propose a deterministic algorithm with approximation ratio of 5, which improves the prior best known ratio of 12. For the special case when all coflows are released at time zero, our deterministic algorithm obtains approximation ratio of 4 which improves the prior best known ratio of 8. The key ingredient of our approach is an improved linear program formulation for sorting the coflows followed by a simple list scheduling policy. Extensive simulation results, using both synthetic and real traffic traces, are presented that verify the performance of our algorithm and show improvement over the prior approaches.

Index Terms—Scheduling Algorithms, Approximation Algorithms, Coflow, Datacenter Network

I. INTRODUCTION

Many data-parallel computing applications (e.g. MapReduce [2], Hadoop [3], [4], Dryad [5], etc.) consist of multiple computation and communication stages or have machines grouped by functionality. While computation involves local operations in servers, communication takes place at the level of machine groups and involves transfer of many pieces of intermediate data (flows) across groups of machines for further processing. In such applications, the collective effect of all the flows between the two machine groups is more important than that of any of the individual flows. A computation stage often cannot start unless all the required data pieces from the preceding stage are received, or the application latency is determined by the transfer of the last flow between the groups [6], [7].

As an example, consider a MapReduce application. Each mapper performs local computations and writes intermediate

data to the disk, then each reducer pulls intermediate data from different mappers, merges them, and computes its output. The job will not finish until its last reducer is completed. Consequently, the job completion time depends on the time that the last flow of the communication phase (called shuffle) is finished. Such intermediate communication stages in a data-parallel application can account on average for about 56% of the job's runtime (see Appendix A in [8] for more detail), and hence can have a significant impact on application performance. Optimizing flow-level performance metrics (e.g. the average flow completion time) have been extensively studied before from both networking systems and theoretical perspective (see, e.g., [9]–[11] and references there.), however, these metrics ignore the dependence among the flows of an application which is critical for the application-level performance in data-parallel computing applications.

Recently Chowdhury and Stoica [12] have introduced the *coflow* abstraction to capture these communication patterns. A *coflow* is defined as a collection of parallel flows whose completion time is determined by the completion time of the last flow in the collection. Coflows can represent most communication patterns between successive computation stages of data-parallel applications [6]. Clearly the traditional flow communication is still a coflow with a single flow. Jobs from one or more data-parallel applications create multiple coflows in a shared datacenter network. These coflows could vary widely in terms of the total size of the parallel flows, the number of the parallel flows, and the size of the individual flows in the coflows (e.g., see the analysis of production traces in [6]). Classical flow/job scheduling algorithms do not perform well in this environment [6] because each coflow consists of multiple flows— whose completion time is dominated by its slowest flow— and further, the progress of each flow depends on its assigned rate at *both* its source and its destination. This coupling of rate assignments between the flows in a coflow and across the source-destination pairs in the network is what makes the coflow scheduling problem considerably harder than the classical flow/job scheduling problems.

In this paper, we study the coflow scheduling problem, namely, the algorithmic task of determining when to start serving each flow and at what rate, in order to minimize the weighted sum of completion times of coflows in the system. In the case of equal weights, this is equivalent to minimizing the average completion time of coflows.

The authors are with the Department of Electrical Engineering, Columbia University, New York, NY, 10027 USA (e-mail: s.mehrnoosh@columbia.edu, jghaderi@ee.columbia.edu). This work is supported by NSF Grants CNS-1652115 and CNS-1565774. A brief 3-page abstract of this paper has appeared in SPAA 2017 Conference [1].

A. Related Work

Several scheduling heuristics have been already proposed in the literature for scheduling coflows, e.g. [6], [7], [13], [14]. A FIFO-based solution was proposed in [7] which also uses multiplexing of coflows to avoid starvation of small flows which are blocked by large head-of-line flows. A Smallest-Effective-Bottleneck-First heuristic was introduced in Varys [6]: it sorts the coflows in an ascending order in a list based on their maximum loads on the servers, and then assigns rates to the flows of the first coflow in the list such that all its flows finish at the same time. The remaining capacity is distributed among the rest of the coflows in the list in a similar fashion to avoid under-utilization of the network. Similar heuristics without prior knowledge of coflows were introduced in Aalo [14]. A joint scheduling and routing of coflows in datacenter networks was introduced in [13] where similar heuristics based on a Minimum-Remaining-Time-First policy are developed.

Here, we would like to highlight three papers [15]–[17] that are more relevant to our work. These papers consider the problem of minimizing the total weighted completion time of coflows with release dates (i.e., coflows arrive over time.) and provide algorithms with provable guarantees. This problem is shown to be NP-complete through its connection with the concurrent open shop problem [6], [15], and then approximation algorithms are proposed which run in polynomial time and return a solution whose value is guaranteed to be within a constant fraction of the optimal (a.k.a., *approximation ratio*). These papers rely on linear programming relaxation techniques from combinatorial scheduling literature (see, e.g., [18]–[20]). In [15], the authors utilize an interval-indexed linear program formulation which helps partitioning the coflows into disjoint groups. All coflows that fall into one partition are then viewed as a single coflow, where a polynomial-time algorithm is used to optimize its completion time. Authors in [16] have recently constructed an instance of the concurrent open shop problem (see [21] for the problem definition) from the original coflow scheduling problem. Then applying the well-known approximation algorithms for the concurrent open shop problem to the constructed instance, an ordering of coflows is obtained which is then used in a similar fashion as in [15] to obtain an approximation algorithm. The deterministic algorithm in [16] has better a approximation ratio compared to [15], for both cases of with and without release dates. In [17], a linear program approach based on ordering variables is utilized to develop two algorithms, one deterministic and the other randomized. The deterministic algorithm gives the same bounds as in [16], while the randomized algorithm has better performance approximation ratios compared to [15], [16], for both cases of with and without release dates.

B. Main Contributions

In this paper, we consider the problem of minimizing the total weighted coflow completion time. Our main contributions can be summarized as follows.

- **Coflow Scheduling Algorithm.** We use a Linear Program (LP) approach based on ordering variables followed

TABLE I: Performance guarantees (Approximation ratios)

Case	Best known		This paper deterministic
	deterministic	randomized	
Without release dates	8 [16], [17]	2e [17]	4
With release dates	12 [16], [17]	3e [17]	5

by a simple list scheduling policy to develop a deterministic algorithm. Our approach improves the prior algorithms in both cases of with and without release dates. Even if we consider equal weights for all coflows (i.e., minimizing the average completion time), our algorithm has the best known performance guarantee. Table I summarizes our results in comparison with the prior best-known performance bounds. Performance of a deterministic (randomized) algorithm is defined based on approximation ratio, i.e., the ratio between the (expected) weighted sum of coflow completion times obtained by the algorithm and the optimal value. When coflows have release dates (which is often the case in practice as coflows are generated at different times), our deterministic algorithm improves the approximation ratio of 12 [16], [17] to 5, which is also better than the best known randomized algorithm proposed in [17] with approximation ratio of $3e$ (≈ 8.16). When all coflows have release dates equal to zero, our deterministic algorithm has approximation ratio of 4 while the best prior known result is 8 [16], [17] for deterministic and $2e$ (≈ 5.436) [17] for randomized algorithms¹.

- **Empirical Evaluations.** We evaluate the performance of our algorithm, compared to the prior approaches, using both synthetic traffic as well as real traffic based on a Hive/MapReduce trace from a large production cluster at Facebook. Both synthetic and empirical evaluations show that our deterministic algorithm indeed outperforms the prior approaches. For instance, for the Facebook trace with general release dates, our algorithm outperforms Varys [6], the algorithm proposed in [15], and the algorithm proposed in [17] by 24%, 40%, and 19%, respectively. Finally, we compare the fairness of various algorithms and propose couple of ideas to improve the fairness.

II. SYSTEM MODEL AND PROBLEM FORMULATION

Datecenter Network

Similar to [6], [15], we abstract out the datacenter network as one giant $N \times N$ non-blocking switch, with N input links connected to N source servers and N output links connected to N destination servers. Thus the network can be viewed as a bipartite graph with source nodes denoted by set \mathcal{I} on one side and destination nodes denoted by set \mathcal{J} on the other side (therefore, $\mathcal{I} \cap \mathcal{J} = \emptyset$). Moreover, there are capacity constraints on the input and output links. For simplicity, we assume all links have equal capacity (as in [15]); nevertheless, our method can be easily extended to the general case where

¹We have been recently informed of the paper [22] which has appeared after the original submission of our work and proposes an algorithm with the same approximation guarantee as our algorithm. The paper [22] uses a different linear programming, and our scheduling policy is much simpler than the policy they proposed. Moreover, we also study the performance of our algorithm through extensive simulations with synthetic and real traffic traces and compare its performance with other coflow scheduling algorithms.

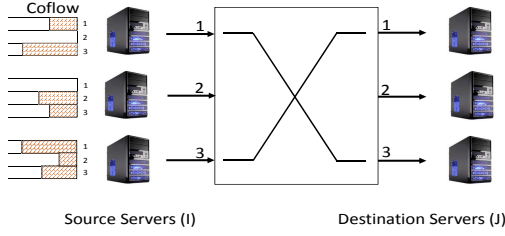


Fig. 1: A coflow in a 3×3 switch architecture.

the links have unequal capacities. Without loss of generality, we assume that all the link capacities are normalized to one.

Scheduling Constraints

We allow a general class of scheduling algorithms where the rate allocation can be performed continuously over time, i.e., for each flow, fractional data units can be transferred from its input link to its corresponding output link over time as long as link capacity constraints are respected. In the special case that the rate allocation is restricted to data units (packets), each source node can send at most one packet in every time unit (time slot) and each destination node can receive at most one packet in every time slot, and the feasible schedule has to form a matching of the switch's bipartite graph. In this case, our model reduces to the model in [15] and, as it is shown later, our proposed algorithm will respect the matching constraints, therefore, it is compatible with both models.

Coflow

A coflow is a collection of flows whose completion time is determined by the completion time of the latest flow in the collection. The coflow k can be denoted as an $N \times N$ demand matrix $D^{(k)}$. Every flow is a triple (i, j, k) , where $i \in \mathcal{I}$ is its source node, $j \in \mathcal{J}$ is its destination node, and k is the coflow to which it belongs. The size of flow (i, j, k) is denoted by d_{ij}^k , which is the (i, j) -th element of the matrix $D^{(k)}$. For simplicity, we assume that all flows within a coflow arrive to the system at the same time (as in [15]); however, our results still hold for the case that flows of a coflow are released at different times (which could indeed happen in practice [14]). A 3×3 switch architecture is shown in Figure 1 as an example, where a coflow is illustrated by means of input queues, e.g., the file in the j -th queue at the source link i indicates that the coflow has a flow from source server i to destination server j . For instance, in Figure 1, the illustrated coflow has 7 flows in total, while two of its flows have source server 1, one goes to destination server 1 and the other to destination server 3.

Total Weighted Coflow Completion Time

We consider the coflow scheduling problem with release dates. There is a set of K coflows denoted by \mathcal{K} . Coflow $k \in \mathcal{K}$ is released (arrives) at time r_k which means it can only be scheduled after time r_k . We use f_k to denote the finishing (completion) time of coflow k , which, by definition of coflow,

is the time when all its flows have finished processing. In other words, for every coflow $k \in \mathcal{K}$,

$$f_k = \max_{i \in \mathcal{I}, j \in \mathcal{J}} f_{ij}^k, \quad (1)$$

where f_{ij}^k is the completion time of flow (i, j, k) .

For given positive weights w_k , $k \in \mathcal{K}$, the goal is to minimize the weighted sum of coflow completion times: $\sum_{k=1}^K w_k f_k$. The weights can capture different priority for different coflows. In the special case that all the weights are equal, the problem is equivalent to minimizing the average coflow completion time.

Define

$$T = \max_{k \in \mathcal{K}} r_k + \sum_{k \in \mathcal{K}} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} d_{ij}^k. \quad (2)$$

Note that T is clearly an upper bound on the minimum time required for processing of all the coflows. We denote by $x_{ij}^k(t)$ the transmission rate assigned to flow (i, j, k) at time $t \in [0, T]$. Then the optimal rate control must solve the following optimal control problem

$$\text{minimize } \sum_{k=1}^K w_k f_k \quad (3a)$$

$$\text{subject to: } f_k \geq f_{ij}^k, \quad i \in \mathcal{I}, j \in \mathcal{J}, k \in \mathcal{K} \quad (3b)$$

$$d_{ij}^k = \int_0^{f_{ij}^k} x_{ij}^k(t) dt, \quad i \in \mathcal{I}, j \in \mathcal{J}, k \in \mathcal{K} \quad (3c)$$

$$\sum_j \sum_k x_{ij}^k(t) \leq 1, \quad i \in \mathcal{I}, t \in [0, T] \quad (3d)$$

$$\sum_i \sum_k x_{ij}^k(t) \leq 1, \quad j \in \mathcal{J}, t \in [0, T] \quad (3e)$$

$$x_{ij}^k(t) = 0, \quad \forall t < r_k, i \in \mathcal{I}, j \in \mathcal{J}, k \in \mathcal{K} \quad (3f)$$

$$x_{ij}^k(t) \geq 0, \quad i \in \mathcal{I}, j \in \mathcal{J}, k \in \mathcal{K}, t \in [0, T] \quad (3g)$$

In the above, the constraint (3b) indicates that each coflow k is completed when all its flows have been completed. Note that since the optimization (3) is a minimization problem, a coflow completion time is equal to the completion time of its latest flow, in agreement with (1). The constraint (3c) ensures that the demand (file size) of every flow, d_{ij}^k , is transmitted by its completion time, f_{ij}^k . Constraints (3d) and (3e) state the capacity constraints on source links and destination links, respectively. The fact that a flow cannot be transmitted before its release date (which is equal to release date of its corresponding coflow) is captured by the constraint (3f). Finally, the constraint (3g) simply states that the rates are non-negative.

Remark 1. An alternative formulation of (3) could be minimizing the weighted sum of delays, where delay of coflow k is defined as $f_k - r_k$. The two minimizations are equivalent as only the objectives differ in a constant term $\sum_k r_k w_k$, however in terms of approximation results they could be very different. In the case of zero release dates, the two formulations are trivially the same, and our algorithms yield the same approximation results for both formulations. However, for the general release dates, there is no constant ratio approximation algorithm for minimizing the weighted sum of delays. This can

be shown through its connection to the single machine scheduling for which finding a constant approximation algorithm for the delay-based formulation is NP-complete [23].

III. MOTIVATIONS AND CHALLENGES

The coflows can be widely different in terms of the number of parallel flows, the size of individual flows, the groups of servers involved, etc. Heuristics from traditional flow/task scheduling, such as shortest- or smallest-first policies [24], [25], do not have a clear equivalence in coflow scheduling. One can define a shortest or smallest-first policy based on the number of parallel flows in a coflow, or the aggregate flow sizes in a coflow, however these policies perform poorly [6], as they do not completely take all the characteristics of coflows into consideration.

Recall that the completion time of a coflow is dominated by its slowest flow (as described by (1) or (3b)). Hence, it makes sense to slow down all the flows in a coflow to match the completion time of the flow that will take the longest to finish. The unused capacity then can be used to allow other coexisting coflows to make progress and the total (or average) coflow completion time decreases. Varys [6] is the first heuristic that effectively implements this intuition by combining Smallest-Effective-Bottleneck-First and Minimum-Allocation-for-Desired-Duration policies. Before describing Varys, we present a few definitions that are used in the rest of this paper.

Definition 1 (Aggregate Size and Effective Size of a Coflow). Let

$$d_i^k = \sum_{j \in \mathcal{J}} d_{ij}^k; \quad d_j^k = \sum_{i \in \mathcal{I}} d_{ij}^k, \quad (4)$$

be respectively the aggregate flow size that coflow k needs to send from source node i and receive at destination node j . The effective size of coflow k is defined as

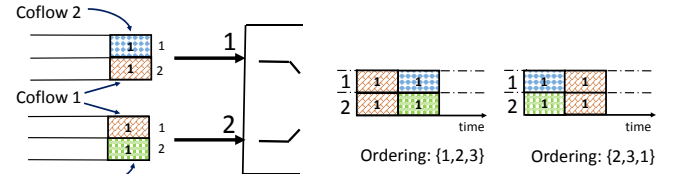
$$W(k) = \max\{\max_{i \in \mathcal{I}} d_i^k, \max_{j \in \mathcal{J}} d_j^k\}. \quad (5)$$

Thus $W(k)$ is the maximum amount of data that needs to be sent or received by a node for coflow k . Note that, due to normalized capacity constraints on links, when coflow k is released, we need at least $W(k)$ amount of time to process all its flows.

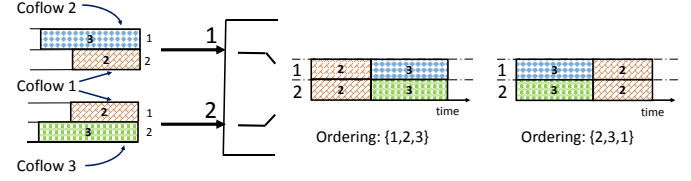
Overview of Varys. Varys [6] orders coflows in a list based on their effective size in an increasing order. Transmission rates of individual flows of the first coflow in the list are set such that all its flows complete at the same time. The remaining capacity of links are updated and iteratively distributed among other coflows in the list in a similar fashion. Formally, the completion time of coflow k , $k = 1, \dots, K$, is calculated as follows

$$\Gamma^k = \max\{\max_{i \in \mathcal{I}} \frac{d_i^k}{\text{Rem}(i)}, \max_{j \in \mathcal{J}} \frac{d_j^k}{\text{Rem}(j)}\},$$

where $\text{Rem}(i)$ (similarly, $\text{Rem}(j)$) is the remaining capacity of input link i (output link j) after transmission rates of all coflows $k' < k$ are set. Then for flow (i, j, k) , Varys assigns transmission rate $x_{ij}^k = d_{ij}^k / \Gamma^k$. In case that there is still idle



(a) All coflows have equal effective size. Both orderings are possible under Varys, with the total completion time of $1 + 2 + 2 = 5$ and $1 + 1 + 2 = 4$, for the left and right ordering respectively.



(b) Varys schedules coflow 1 first, according to the ordering $\{1, 2, 3\}$, which gives a total completion time of $2 + 5 + 5 = 12$. The optimal schedule is the ordering $\{2, 3, 1\}$ with a total completion time of $3 + 3 + 5 = 11$.

Fig. 2: Inefficiency of Varys in a 2×2 switch network with 3 coflows.

capacity, for each input link $i \in \mathcal{I}$, the remaining capacity is allocated to the flows of coflows subject to capacity constraints in corresponding output links. Once the first coflow completes, all the flow sizes and the scheduling list are updated and the iterative procedure is repeated to complete the second coflow and distribute the unused capacity. The procedure is stopped when all the coflows are processed.

While Varys performs better than traditional flow scheduling algorithms, it could still be inefficient. The main reason is that Varys is oblivious to the dependency among coflows that share a (source or destination) node. To further expose this issue, we present a simple example.

Example 1 (Inefficiency of Varys). Consider the 2×2 switch network illustrated in Figure 2 where there are 3 coflows in the system. In Figure 2a, the effective coflow sizes are $W(1) = W(2) = W(3) = 1$, therefore, Varys cannot differentiate among coflows. Scheduling coflows in the order $\{1, 2, 3\}$ or $\{2, 3, 1\}$ are both possible under Varys but they result in different total completion times, $1 + 2 + 2 = 5$ and $1 + 1 + 2 = 4$, respectively (assuming the weights are all one for all the coflows). Next, consider a slight modification of flow sizes, as shown in Figure 2b. In this example $W(1) = 2$ and $W(2) = W(3) = 3$. Based on Varys algorithm, coflow 1 is scheduled first during time interval $(0, 2]$ at rate 1. When coflow 1 completes, coflows 2 and 3 are scheduled in time interval $(2, 5]$; hence, the total completion time will be $2 + 5 + 5 = 12$. However, if we schedule coflows 2 and 3 first, the total completion times will reduce to $3 + 3 + 5 = 11$. Note that in both examples, coflow 1 completely blocks coflows 2 and 3, which is not captured by Varys. In fact, the negative impact of ignoring configuration of coflows and their shared nodes is much more profound in large networks with a large number of coflows (see simulations in Section VIII).

Overview of LP-based algorithms. The papers [15] and [17] use Linear Programs (LPs) (based on interval-indexed

variables or ordering variables) that capture more information about coflows and provide a better ordering of coflows for scheduling compared to Varys [6]. At the high level, the technical approach in these papers is based on partitioning jobs (coflows) into polynomial number of groups based on solution to a polynomial-sized relaxed linear program, and minimizing the completion time of each group by treating the group as a single coflow. Grouping can have a significant impact on decreasing the completion time of coflows. For instance, in view of examples in Figure 2, grouping coflows 2 and 3, and scheduling them first, decreases the total completion time as explained.

NP-hardness and connection to the concurrent open shop problem. The concurrent open shop problem [21] can be essentially viewed as a special case of the coflow scheduling problem *when demand matrices are diagonal* (in the jargon of concurrent open shop problem, the coflows are jobs, the flows in each coflow are tasks for that job, and the destination nodes are machines with unit capacities). It is known that it is NP-complete to approximate the concurrent open shop problem, when jobs are released at time zero, within a factor better than $2 - \epsilon$ for any $\epsilon > 0$ [26]. Although the model we consider for coflow scheduling is different from the model used in [15], similar reduction as proposed in [15] can be leveraged to show NP-completeness of the coflow scheduling problem. More precisely, every instance of the concurrent open shop problem can be reduced to an instance of coflow scheduling problem (see Appendix A for the details), hence it is NP-complete to approximate the coflow scheduling problem (without release dates) within $2 - \epsilon$, for any $\epsilon > 0$. There are 2-approximation algorithms for the concurrent open shop (e.g., [21]), however, these algorithms cannot be ported to the coflow scheduling problem due to the coupling of source and destination link capacity constraints in the coflow scheduling problem (see Appendix A for a counter example that shows the 2-approximation algorithm from concurrent open shop cannot be ported to the coflow scheduling problem).

Next, we describe our coflow scheduling algorithm. The algorithm is based on a linear program formulation for sorting the coflows followed by a simple list scheduling policy

IV. LINEAR PROGRAMING (LP) RELAXATION

In this section, we use *linear ordering variables* (see, e.g., [20], [21], [27], [28]) to present a relaxed integer program of the original scheduling problem (3). We then relax these variables to obtain a linear program (LP). In the next section, we use the optimal solution to this LP as a subroutine in our scheduling algorithm.

Ordering variables. For each pair of coflows, if both coflows have *some* flows incident at some node (either originated from or destined at that node), we define a binary variable which indicates which coflow finishes all its flows before the other coflow does so in the schedule. Formally, for any two coflows k, k' with aggregate flow sizes $d_m^k \neq 0$ and $d_m^{k'} \neq 0$ on some node $m \in \mathcal{I} \cup \mathcal{J}$ (recall definition (4)), we introduce a binary variable $\delta_{kk'} \in \{0, 1\}$ such that $\delta_{kk'} = 1$ if coflow k finishes all its flows before coflow k' finishes all its

flows, and it is 0 otherwise. If both coflows finish their flows at the same time (which is possible in the case of continuous-time rate control), we set either one of $\delta_{kk'}$ or $\delta_{k'k}$ to 1 and the other one to 0, arbitrarily.

Relaxed Integer Program (IP). We formulate the following Integer Program (IP):

$$(IP) \min \sum_{k=1}^K w_k f_k \quad (6a)$$

$$f_k \geq d_i^k + \sum_{k' \in \mathcal{K}} d_i^{k'} \delta_{k'k} \quad i \in \mathcal{I}, k \in \mathcal{K} \quad (6b)$$

$$f_k \geq d_j^k + \sum_{k' \in \mathcal{K}} d_j^{k'} \delta_{k'k} \quad j \in \mathcal{J}, k \in \mathcal{K} \quad (6c)$$

$$f_k \geq W(k) + r_k \quad k \in \mathcal{K} \quad (6d)$$

$$\delta_{kk'} + \delta_{k'k} = 1 \quad k, k' \in \mathcal{K} \quad (6e)$$

$$\delta_{kk'} \in \{0, 1\} \quad k, k' \in \mathcal{K} \quad (6f)$$

In the above, to simplify the formulation, we have defined $\delta_{kk'}$, for all pairs of coflows, by defining $d_m^k = 0$ if coflow k has no flow originated from or destined to node m .

The constraint (6b) (similarly (6c)) follows from the definition of ordering variables and the fact that flows incident to a source node i (a destination node j) are processed by a single link of unit capacity. To better see this, note that the total amount of traffic can be sent in the time period $(0, f_k]$ over the i -th link is at most f_k . This traffic is given by the right-hand-side of (6b) (similarly (6c)) which basically sums the aggregate size of coflows incident to node i that finish their flows before coflow k finishes its corresponding flows, plus the aggregate size of coflow k at node i itself, d_i^k . This implies constraint (6b) and (6c). The fact that each coflow cannot be completed before its release date plus its effective size is captured by constraint (6d). The next constraint (6e) indicates that for each two incident coflows, one precedes the other.

Note that this optimization problem is a relaxed integer program for the problem (3), since the set of constraints are not capturing all the requirements we need to meet for a feasible schedule. For example, we cannot start scheduling flows of a coflow when it is not released yet, while constraint (6d) does not necessarily avoid this, thus leading to a smaller value of finishing time compared to the optimal solution to (3). Further, release dates and scheduling constraints in optimization (3) might cause idle times in flow transmission of a node, therefore yielding a larger value of finishing time for a coflow than what is restricted by (6b), (6c), (6d). To further illustrate this issue, we present a simple example.

Example 2. Consider a 2×2 switch network as in Figure 3. Assume there are 4 coflows, each has one flow. Flow $(1, 1, 1)$ is released at time 0 with size 1, and the other three flows are released at time 1 with size 2. It is easy to check that the following values for the ordering variables and flow completion times satisfy all the constraints (6b)–(6f). For brevity, we only report the ordering variables for coflows that actually share a node. For example, it is redundant to consider ordering variables corresponding to coflow 1 and coflow 4 as

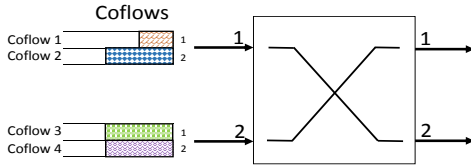


Fig. 3: 4 coflows in a 2×2 switch architecture, flow (1, 1) is released at time 0, and all the others are released at time 1.

they are not incident at any (source/destination) node and any value for their associated pairwise ordering variables does not have any impact on the optimal value for IP (6). Below, the ordering variables and coflow completion times are presented, and all the ordering variables which are not specified can be taken as zero.

$$\begin{aligned} \delta_{12} &= 1, & \delta_{34} &= 1, \\ \delta_{13} &= 1, & \delta_{24} &= 1, \\ f_1 &= 1, & f_2 &= 3, \\ f_3 &= 3, & f_4 &= 4. \end{aligned}$$

While these values satisfy (6b)–(6f), this is not a valid schedule since it requires transmission of flow (2, 2, 4) starting at time 0, while it is not released yet. To see this, note that $f_1 = 1$, so to finish processing of coflow 1 or equivalently flow (1, 1, 1) by time 1, we need to start its transmission at maximum rate at time 0. Then, due to the capacity constraints, the first time flows (1, 2, 2) and (2, 1, 3) can start transmission is at time 1, when flow (1, 1, 1) has been completed. Since we require to complete both of these flows at time 3, they need to be transmitted at maximum rate in the time interval (1, 3]. Therefore, the only way to finish flow (2, 2, 4) at time 4 is to send one unit of its data in time interval (0, 1] and its remaining unit of data in time interval (3, 4], but this flow has not been released before time 1. So the proposed IP does not address all the scheduling constraints.

Relaxed Linear Program (LP). In the linear program relaxation, we allow the ordering variables to be fractional. Specifically, we replace the constraints (6f) with the constraints (7b) below. We refer to the obtained linear problem by (LP).

$$\begin{aligned} \text{(LP)} \quad \min \quad & \sum_{k=1}^K w_k f_k & (7a) \\ \text{subject to:} \quad & (6b) - (6e), \\ & \delta_{kk'} \in [0, 1] \quad k, k' \in \mathcal{K} & (7b) \end{aligned}$$

We use \tilde{f}_k to denote the optimal solution to this LP for the completion time of coflow k . Also we use $\widetilde{\text{OPT}} = \sum_k w_k \tilde{f}_k$ to denote the corresponding objective value. Similarly we use f_k^* to denote the optimal completion time of coflow k in the original coflow scheduling problem (3), and use $\text{OPT} = \sum_k w_k f_k^*$ to denote its optimal objective value. The following lemma establishes a relation between $\widetilde{\text{OPT}}$ and OPT .

Lemma 1. *The optimal value of the LP, $\widetilde{\text{OPT}}$, is a lower bound on the optimal total weighted completion time OPT of coflow scheduling problem.*

Proof. Consider an optimal solution to the optimization problem (3). We set ordering variables so as $\delta_{kk'} = 1$ if coflow k precedes coflow k' in this solution, and $\delta_{kk'} = 0$, otherwise. If both coflows finish their corresponding flows at the same time, we set either one to 1 and the other one to 0. We note that this set of ordering variables and coflow completion times satisfies constraints (6b) and (6c) (by taking integral from both side of constraint (3d) and (3e) from time 0 to f_k) and also constraint (6d) (by combining constraints (3c) and (3f)). Furthermore, the rest of (LP) constraints are satisfied by the construction of ordering variables. Therefore, optimal solution of problem (3) can be converted to a feasible solution to (LP). Hence, the optimal value of LP, $\widetilde{\text{OPT}}$, is at most equal to OPT . \square

V. COFLOW SCHEDULING ALGORITHM

In this section, we describe our polynomial-time coflow scheduling algorithm and state the main results about its performance guarantees.

The scheduling algorithm is presented in Algorithm 1. It has three main steps:

- 1) solve the relaxed LP (7),
- 2) use the solution of the relaxed LP to order flows of coflows,
- 3) apply a simple list scheduling algorithm based on the ordering.

The relaxed LP (7) has $O(K^2)$ variables and $O(K^2 + KN)$ constraints and can be solved efficiently in polynomial time, e.g. using interior point method [29] (see Section VIII-E for more details about the complexity).

Then, the algorithm orders the coflows based on values of \tilde{f}_k (optimal solution to LP) in nondecreasing order. More precisely, we re-index coflows such that,

$$\tilde{f}_1 \leq \tilde{f}_2 \leq \dots \leq \tilde{f}_K. \quad (8)$$

Ties are broken arbitrarily. We emphasize that we do not need to round the values of the ordering variables in LP to obtain the ordering of coflows, instead we use the values of \tilde{f}_k (optimal solution to LP) which do not need to be integer.

At any time, the algorithm maintains a list for the flows in the system such that for every two flows (i, j, k) and (i', j', k') with $k < k'$ (based on ordering (8)), flow (i, j, k) is placed before flow (i', j', k') in the list. Flows of the same coflow are listed in an arbitrary order. The algorithm scans the list starting from the first flow and schedules a flow if both its corresponding source and destination links are idle at that time. Upon completion of a flow or arrival of a coflow, the algorithm preempts the schedule, updates the list, and starts scheduling the flows based on the updated list.

The main result regarding the performance of Algorithm 1 is stated in Theorem 1.

Theorem 1. *Algorithm 1 is a polynomial-time 5-approximation algorithm for the problem of minimizing total weighted completion time of coflows with release dates.*

When all coflows are released at time 0, we can improve the algorithm's performance ratio.

Algorithm 1 Deterministic Coflow Scheduling Algorithm

Suppose coflows $\left[d_{ij}^k, 1 \leq i, j \leq N \right]$, $k \in \mathcal{K}$, with release dates r_k , $k \in \mathcal{K}$, and weights w_k , $k \in \mathcal{K}$, are given.

- 1: Solve the linear program (LP) and denote its optimal solution by $\{\tilde{f}_k; k \in \mathcal{K}\}$.
- 2: Order and re-index the coflows such that:

$$\tilde{f}_1 \leq \tilde{f}_2 \leq \dots \leq \tilde{f}_K, \quad (9)$$

where ties are broken arbitrarily.

- 3: Wait until the first coflow is released.
- 4: **while** There is some incomplete flow, **do**
- 5: List the released and incomplete flows respecting the ordering in (9). Let L be the total number of flows in the list.
- 6: **for** $l = 1$ to L **do**
- 7: Denote the l -th flow in the list by (i_l, j_l, k_l) ,
- 8: **if** Both the links i_l and j_l are idle, **then**
- 9: Schedule flow (i_l, j_l, k_l) .
- 10: **end if**
- 11: **end for**
- 12: **while** No new flow is completed or released **do**
- 13: Transmit the flows that get scheduled in line 9 at maximum rate 1.
- 14: **end while**
- 15: **end while**

Corollary 1. *If all coflows are released at time 0, then Algorithm 1 is a 4-approximation algorithm.*

VI. PROOF SKETCH OF MAIN RESULTS

In this section, we present the sketch of proofs of the main results for our polynomial-time coflow scheduling algorithm. Before proceeding with the proofs, we make the following definitions.

Definition 2 (Aggregate Size and Effective Size of a List of Coflows). *For a list of K coflows and for a node $s \in \mathcal{I} \cup \mathcal{J}$, we define $W(1, \dots, k; s)$ to be the amount of data needs to be sent or received by node s in the network considering only the first k coflows. We also denote by $W(1, \dots, k)$ the effective size of the aggregate coflow constructed by the first k coflows, $k \leq K$. Specifically,*

$$W(1, \dots, k; s) = \sum_{l=1}^k d_s^l \quad (10)$$

$$W(1, \dots, k) = \max_{s \in \mathcal{I} \cup \mathcal{J}} W(1, \dots, k; s) \quad (11)$$

A. Bounded completion time for the collection of coflows

Consider the list of coflows according to the ordering in (8) and define $W(1, \dots, k)$ based on Definition 2. The following lemma demonstrates a relationship between completion time of coflow k obtained from (LP) and $W(1, \dots, k)$ which is used later in the proofs.

Lemma 2. $\tilde{f}_k \geq \frac{W(1, \dots, k)}{2}$.

Proof. The proof uses similar ideas as in Gandhi, et al. [28] and Kim [19]. Using constraint (6b), for any source node $i \in \mathcal{I}$, we have

$$d_i^l \tilde{f}_l \geq (d_i^l)^2 + \sum_{l'=1}^k d_i^l d_{i'}^{l'} \delta_{l'l'} \quad (12)$$

which implies that,

$$\begin{aligned} \sum_{l=1}^k d_i^l \tilde{f}_l &\geq \sum_{l=1}^k (d_i^l)^2 + \sum_{l=1}^k \sum_{l'=1}^k d_i^l d_{i'}^{l'} \delta_{l'l'} \\ &= \frac{1}{2} \left(2 \times \sum_{l=1}^k (d_i^l)^2 \right. \\ &\quad \left. + \sum_{l=1}^k \sum_{l'=1}^k (d_i^l d_{i'}^{l'} \delta_{l'l'} + d_i^{l'} d_{i'}^l \delta_{l'l'}) \right) \end{aligned} \quad (13)$$

We simplify the right-hand side of (13), using constraint (6e), combined with the following equality

$$\sum_{l=1}^k (d_i^l)^2 + \sum_{l=1}^k \sum_{l'=1}^k d_i^l d_{i'}^{l'} = \left(\sum_{l=1}^k d_i^l \right)^2, \quad (14)$$

and conclude that

$$\begin{aligned} \sum_{l=1}^k d_i^l \tilde{f}_l &\geq \frac{1}{2} \sum_{l=1}^k (d_i^l)^2 + \frac{1}{2} \left(\sum_{l=1}^k d_i^l \right)^2 \\ &\geq \frac{1}{2} \left(\sum_{l=1}^k d_i^l \right)^2 = \frac{1}{2} (W(1, \dots, k; i))^2 \end{aligned} \quad (15)$$

Where the last equality follows from Definition 10. Similar argument results in the following inequality for any destination node $j \in \mathcal{J}$, i.e.,

$$\sum_{l=1}^k d_j^l \tilde{f}_l \geq \frac{1}{2} (W(1, \dots, k; j))^2.$$

Now consider the node s^* which has the maximum load induced by the first k coflows, namely, $W(1, \dots, k) = W(1, \dots, k; s^*)$.

$$\begin{aligned} \tilde{f}_k W(1, \dots, k; s^*) &= \tilde{f}_k \sum_{l=1}^k d_{s^*}^l \\ &\geq \sum_{l=1}^k d_{s^*}^l \tilde{f}_l \\ &\geq \frac{1}{2} (W(1, \dots, k; s^*))^2 \end{aligned} \quad (16)$$

This implies that,

$$\tilde{f}_k \geq \frac{1}{2} W(1, \dots, k; s^*) = \frac{1}{2} W(1, \dots, k). \quad (17)$$

This completes the proof. \square

Note that $W(1, \dots, k)$ is a lower bound on the time that it takes for all the first k coflows to be completed (as a result of the capacity constraints in the optimization (3)). Hence, Lemma 2 states that by allowing ordering variables to be fractional, completion time of coflow k obtained from (LP) is still lower bounded by half of $W(1, \dots, k)$.

B. Proof of Theorem 1 and Corollary 1

Proof of Theorem 1. We use $\{\hat{f}_k\}_{k=1}^K$ to denote the actual coflow completion times under our deterministic algorithm. Suppose flow (i, j, k) is the last flow of coflow k that is completed. In general, Algorithm 1 may preempt a flow several times during its execution. For now, suppose flow (i, j, k) is not preempted and use t_k to denote the time when its transmission is started (the arguments can be easily extended to the preemption case as we show at the end of the proof). Therefore

$$\hat{f}_k = \hat{f}_{ij}^k = t_k + d_{ij}^k \quad (18)$$

From the algorithm description, t_k is the first time that both links i and j are idle and there are no higher priority flows to be scheduled (i.e., there is no flow (i, j, k') from i to j with $k' < k$ in the list). By definition of $W(1, \dots, k; s)$, node s , $s \in \{i, j\}$, has $W(1, \dots, k; s) - d_{ij}^k$ data units to send or receive by time t_k . Since the capacity of all links are normalized to 1, it should hold that

$$\begin{aligned} t_k &\leq r_k + W(1, \dots, k; i) - d_{ij}^k + W(1, \dots, k; j) - d_{ij}^k \\ &\leq r_k + 2W(1, \dots, k) - 2d_{ij}, \end{aligned}$$

where the last inequality is by Definition 11. Combining this inequality with equality (18) yields the following bound on \hat{f}_k .

$$\hat{f}_k \leq r_k + 2W(1, \dots, k)$$

Using Lemma 2 and constraint (6d), we can conclude that

$$\hat{f}_k \leq 5\tilde{f}_k,$$

which implies that

$$\sum_{k=1}^K w_k \hat{f}_k \leq 5 \sum_{k=1}^K w_k \tilde{f}_k.$$

This shows an approximation ratio of 5 for Algorithm 1 using Lemma 1. Finally, if flow (i, j, k) is preempted, the above argument can still be used by letting t_k to be the starting time of its last piece and d_{ij}^k to be the remaining size of its last piece at time t_k . This completes the proof. \square

Proof of Corollary 1. When all coflows are released at time 0, $t_k \leq W(1, \dots, k) - d_{ij}^k + W(1, \dots, k) - d_{ij}^k$. The rest of the argument is similar to the proof of Theorem 1. Therefore, the algorithm has approximation ratio of 4 when all coflows are release at time 0. \square

VII. EXTENSION TO ONLINE ALGORITHM

Similar to previous work [15], [16], Algorithm 1 is an offline algorithm, and requires the complete knowledge of the flow sizes and release dates. While this knowledge can be learned in long running services, developing online algorithms that deal with the dynamic nature and unavailability of this information is of practical importance. One natural extension of our algorithm to an online setting, assuming that the coflow information revealed at its release date, is as follows: Upon each coflow arrival, we re-order the coflows by re-solving the (LP) using the remaining coflow sizes and the newly arrived coflow, and update the list. Given the updated list, the

scheduling is done as in Algorithm 1. To reduce complexity of the online algorithm, we may re-solve the LP once in every T seconds, for some T that can be tuned, and update the list accordingly. We leave theoretical and experimental study of this online algorithm as a future work.

VIII. EMPIRICAL EVALUATIONS

In this section, we present our simulation results and evaluate the performance of our algorithm for both cases of with and without release dates, under both synthetic and real traffic traces. We also simulate the deterministic algorithms proposed in [15], [17] and Varys [6] and compare their performance with the performance of our algorithm. Finally, we comment on the fairness issues of the algorithm.

A. Workload

We evaluate algorithms under both synthetic and real traffic traces.

Synthetic traffic: To generate synthetic traces we slightly modify the model used in [30]. We consider the problem of size $K = 160$ coflows in a switch network with $N = 16$ input and output links. We denote by M the number of non-zero flows in each coflow. We consider two cases:

- **Dense instance:** For each coflow, M is chosen uniformly from the set $\{N, N + 1, \dots, N^2\}$. Therefore, coflows have $O(N^2)$ non-zero flows on average.
- **Combined instance:** Each coflow is sparse or dense with probability 1/2. For each sparse coflow, M is chosen uniformly from the set $\{1, 2, \dots, N\}$, and for each dense coflow M is chosen uniformly from the set $\{N, N + 1, \dots, N^2\}$.

Given the number M of flows in each coflow, M pairs of input and output links are chosen randomly. For each pair that is selected, an integer flow size (processing requirement) d_{ij} is randomly selected from the uniform distribution on $\{1, 2, \dots, 100\}$. For the case of scheduling with release dates, we generate the coflow inter-arrival times uniformly from $[1, 100]$. We generate 100 instances for each case and report the average algorithms' performance.

Real traffic: This workload was also used in [6], [15], [17]. The workload is based on a Hive/MapReduce trace at Facebook that was collected from a 3000-machine cluster with 150 racks. In this trace, the following information is provided for each coflow: arrival time of the coflow in millisecond, locations of mappers (rack number to which they belong), locations of reducers (rack number to which they belong), and the amount of shuffle data in Megabytes for each reducer. We assume that shuffle data of each reducer in a coflow is evenly generated from all mappers specified for that coflow. The data trace consists of 526 coflows in total from very sparse coflows (the most sparse coflow has only 1 flow) to very dense coflows (the most dense coflow has 21170 flows.). Similar to [15], we filter the coflows based on the number of their non-zero flows, M . Apart from considering all coflows ($M \geq 1$), we consider three coflow collections filtered by the conditions $M \geq 10$, $M \geq 30$, and $M \geq 50$. In other words, we use the following 4 collections:

- All coflows,

- Coflows with $M \geq 10$,
- Coflows with $M \geq 30$,
- Coflows with $M \geq 50$.

Furthermore, the original cluster had a 10:1 core-to-rack oversubscription ratio with a total bisection bandwidth of 300 Gbps. Hence, each link has a capacity of 128 MBps. To obtain the same traffic intensity offered to our network (without oversubscription), for the case of scheduling coflows with release dates, we need to scale down the arrival times of coflows by 10. For the case of without release dates, we assume that all coflows arrive at time 0.

B. Algorithms

We simulate four algorithms: the algorithm proposed in this paper, Varys [6], the deterministic algorithm in [15], and the deterministic algorithm in [17]. We briefly overview these algorithms and also elaborate on the backfilling strategy that has been combined with the deterministic algorithms in [15], [17] to avoid under utilization of network resources.

1. Varys [6]: Scheduling and rate assignments under Varys were explained in detail in Section III. There is a parameter δ in the original design of Varys that controls the tradeoff between fairness and completion time. Since we focus on minimizing the total completion time of coflows, we set δ to 0 which yields the best performance of Varys. In this case, upon arrival or completion of a coflow, the coflow ordering is updated and the rate assignment is done iteratively as described in Section III.

2. Interval-Indexed-Grouping (LP-II-GB) [15]: The algorithm requires discrete time (i.e., time slots) and is based on an interval-indexed formulation of a polynomial-time linear program (LP) as follows. The time is divided into geometrically increasing intervals. The binary decision variables x_{lk} are introduced which indicate whether coflow k is scheduled to complete within the l -th interval $(t_l, t_{l+1}]$. Using these binary variables, a lower bound on the objective function is formulated subject to link capacity constraints and the release date constraints. The binary variables are then relaxed leading to an LP whose solution is used for ordering coflows. More precisely, the relaxed completion time of coflow k is defined as $f_k = \sum_l t_l x_{lk}$, where t_l is the left point of the l -th interval and $x_{lk} \in [0, 1]$ is the relaxed decision variable. Based on the optimal solution to this LP, coflows are listed in an increasing order of their relaxed completion time. For each coflow k in the list, $k = 1, \dots, K$, we compute effective size of the cumulated first k coflows in the list, $W(1, \dots, k)$. All coflows that fall within the same time interval according to value of $W(1, \dots, k)$ are grouped together and treated as a single coflow and scheduled so as to minimize its completion time. Scheduling of coflows within a group makes use of the Birkhoff-von Neumann decomposition. If two data units from coflows k and k' within the same group use the same pair of input and output, and k is ordered before k' , then we always process the data unit from coflow k first. For backfilling, when we use a schedule that matches input i to output j , if there is no more service requirement on the pair of input i and output j for some coflow in the current partition, we backfill in order

from the flows on the same pair of ports in the subsequent coflows. We would like to emphasize that this algorithm needs to discretize time and is based on matching source nodes to destination nodes. We select the time unit to be 1/128 second as suggested in [15] so that each port has a capacity of 1 MB per time unit. We refer to this algorithm as ‘LP-II-GB’, where II stands for Interval-Indexed, and GB stands for Grouping and Backfilling.

3. Ordering-Variable-Grouping (LP-OV-GB) [17]: We implement the deterministic algorithm in [17]. Linear programming formulation is the same as LP in (7). Coflows are then grouped based on the optimal solution to the LP. To schedule coflows of each group, we construct a single aggregate coflow denote by D and schedule its flows to optimize its completion time. We assign transmission rate $x_{ij} = d_{ij}/W(D)$ to the flow from source node i to destination node j until its completion. Moreover, the continuous backfilling is done as follows: After assigning rates to aggregate coflow, we increase x_{ij} until either capacity of link i or link j is fully utilized. We continue until for any node, either source or destination node, the summation of rates sum to one. We also transmit flows respecting coflow order inside of each partition. When there is no more service requirement on the pair of input i and output j for coflows of current partition, we backfill (transmit) in order from the flows on the same pair of ports from the subsequent coflows. We refer to this algorithm as ‘LP-OV-GB’, where OV stands for ordering variables, and GB stands for Grouping and Backfilling.

4. Algorithm 1 (LP-OV-LS): We implement our algorithm as described in Algorithm 1, and refer to it as ‘LP-OV-LS’, where OV stands for ordering variables, and LS stands for list scheduling.

C. Evaluation Results

Performance of Our Algorithm. We report the ratios of total weighted completion time obtained from Algorithm 1 and the optimal value of relaxed linear program (7) (which is a lower bound on the optimal value of the coflow scheduling problem) to verify Theorem 1 and Corollary 1. We only present results of the simulations using the real traffic trace, with equal weights and random weights. For the case of random weights, the weight of each coflow is chosen uniformly at random from the interval $[0, 1]$. The results are more or less similar for other collections and for synthetic traffic traces and all are consistent with our theoretical results.

Table II shows the performance ratio of the deterministic algorithm for the cases of with and without release dates. All performances are within our theoretical results indicating the approximation ratio of at most 4 when all coflows release at time 0 and at most 5 when coflows have general release dates. In fact, the approximation ratios for the real traffic trace are much smaller than 4 and 5 and very close to 1.

Performance Comparison with Other Algorithms. Now, we compare the performance of Algorithm 1 (LP-OV-LS) with LP-II-GB, LP-OV-GB, and Varys. We set all the weights of coflows to be equal to one.

TABLE II: Performance Ratio of Algorithm 1

Case	Equal weights	Random weights
Without release dates	1.05	1.06
With release dates	1.034	1.038

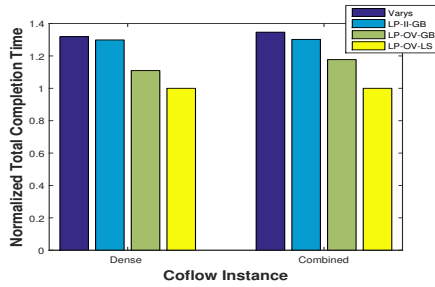


Fig. 4: Performance of Varys, LP-II-GB, LP-OV-GB, and LP-OV-LS when all coflows release at time 0 for 100 random dense and combined instances, normalized with the performance of LP-OV-LS.

1. *Performance evaluation under synthetic traffic:* For each of the two instances explained in Section VIII-A, we randomly generate 100 different traffic traces and compute the average performance of algorithms over the traffic traces.

Figure 4 and 5 depict the average result of our simulations (over 100 dense and 100 combined instances) for the zero release dates and general release dates, respectively. As we see, Algorithm 1 (LP-OV-LS) outperforms Varys and LP-II-GB by almost 30%, and LP-OV-GB by almost 11% in dense instance for both general and zero release dates. In combined instance, the improvements are 35%, 30%, and 17% when all coflows are released at time 0, and 28%, 29%, and 17% for the case of general release dates over Varys, LP-II-GB, and LP-OV-GB, respectively.

This workload is more intensive in the number of non-zero flows; however, more uniform in the flow sizes and source-destination pairs in comparison to the real traffic trace. The real traffic trace (described in Section VIII-A) contains a large number of sparse coflows; namely, about 50% of coflows have less than 10 flows. Also, it widely varies in terms of flow sizes and source-destination pairs in the network. We now present evaluation results under this traffic.

2. *Performance evaluation under real traffic:* We ran simulations for the four collections of coflows described in Section VIII-A. We normalize the total completion time under each algorithm by the total completion time under Algorithm 1 (LP-OV-LS).

Figure 6 shows the performance of different algorithms for different collections of coflows when all coflows are released at time 0. LP-OV-LS outperforms Varys by almost 112–117% in different collections. It also constantly outperforms LP-II-GB and LP-OV-GB by almost 74–78% and 63–68%, respectively.

Figure 7 shows the performance of different algorithms for different collections of coflows for the case of release dates. LP-OV-LS outperforms Varys by almost 24%, 65%, 91%, and 99% for all coflows, $M \geq 10$, $M \geq 30$, $M \geq 50$, respectively. It also outperforms LP-II-GB for 40%, 62%, 71%, and 82%,

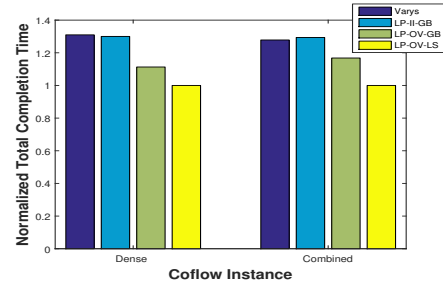


Fig. 5: Performance of Varys, LP-II-GB, LP-OV-GB, and LP-OV-LS in the case of release dates for 100 random dense and combined instances, normalized with the performance of LP-OV-LS.

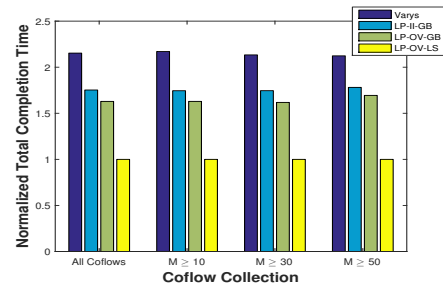


Fig. 6: Performance of Varys, LP-II-GB, LP-OV-GB, and LP-OV-LS when all coflows release at time 0, normalized with the performance of LP-OV-LS, under real traffic trace.

and LP-OV-GB by 19%, 54%, 64%, and 73%, respectively.

Figure 8 depicts the CDF plots of coflow completion time for all four algorithms when all coflows are considered, for both cases of with and without release dates. Based on the plots, 95% of all coflows have completion time less than 100 seconds under our algorithm, while this is 220 seconds for Varys, when all release dates are zero. Also the CDF plots under our algorithm are quite sharp which means that the variance of completion times under our algorithm is smaller than the other algorithms.

D. Incorporating Fairness

So far, we focused on minimizing the total weighted completion time of coflows, without considering any fairness among the rates allocated to different coflows. In this section, we propose a simple adjustment to our algorithm to provide a trade-off between fairness and optimality, and provide simulation results to study the effect of the fairness adjustment.

We use a simple metric to quantify fairness (or equivalently unfairness) among coflows. Define $p_t(k)$, the progress of coflow k by time t , to be the amount of decrease in its effective size by time t , formally,

$$p_t(k) = W(k) - W_t(k), \quad (19)$$

where $W(k)$ is the original effective size of coflow k (its effective size at its release date) and $W_t(k)$ is its effective size at time t after possibly partial transmission of some of its flows (recall (5) for the definition of coflow's effective size).

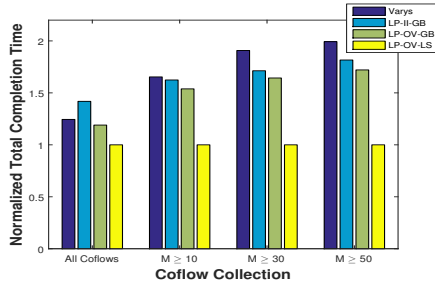
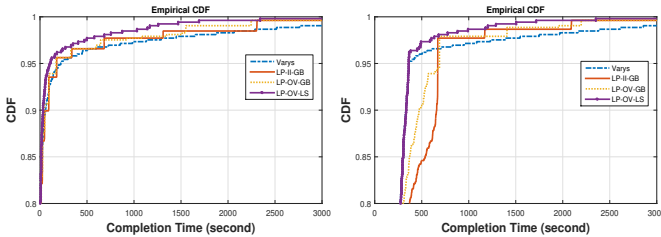


Fig. 7: Performance of Varys, LP-II-GB, LP-OV-GB, and LP-OV-LS in the case of release dates, normalized with the performance of LP-OV-LS, under real traffic trace.



(a) All release dates are 0.

(b) General release dates.

Fig. 8: CDF of coflow completion time under Varys, LP-II-GB, LP-OV-GB, and LP-OV-LS for real traffic trace a) when all coflows release at time 0, b) in the case of release dates.

Ideally, for fairness issues, we might want to have an equal progress among the coflows in the system.²

Hence, we use the standard deviation among progress of coflows that are in the system as our unfairness metric in rate allocation to the current coflows, i.e., the larger the standard deviation of progresses is, the more unfair the algorithm is. Formally, the unfairness at time t is defined as

$$SD_t = \sqrt{\frac{\sum_{k \in \mathcal{K}_t} (p_t(k))^2}{K_t} - \left(\frac{\sum_{k \in \mathcal{K}_t} p_t(k)}{K_t}\right)^2}, \quad (20)$$

where \mathcal{K}_t is the set of coflows in the network at time t and K_t denotes its cardinality.

To measure unfairness throughout the entire schedule, we compute the progress of remaining coflows in the system upon departure of a coflow at time t and calculate the corresponding standard deviation according to (20). We then take the average of all computed standard deviations as a measure for unfairness. Based on this definition, note that a coflow that is completed earlier contributes less in the described unfairness metric because its progress is not counted in future standard deviations. Such a coflow probably has smaller effective size and its flows block less number of flows of other coflows; therefore, scheduling this coflow does not cause severe starvation for other coflows. Given this intuition, the metric captures unfairness reasonably well.

²There are other notions of fairness such as max-min fair, proportional fair, and alpha-utility fair, proposed in rate allocation for flow scheduling, e.g., see [31], [32]. The situation is more complicated in coflow scheduling, since coflows have different number of flows with different overlapping structures. Extending such notions of fairness to coflow scheduling could be an interesting future research.

To incorporate fairness in our algorithm, we introduce two tunable parameters τ and δ ($\tau \geq \delta$) and alternate between time intervals of length τ and δ as follows. The algorithm maintains *two* lists, one is the original list in which coflows are sorted respecting inequality (8) and is updated upon arrival and departure of flows, and the other sorts the coflows in non-decreasing order of their progresses, as defined in (19). We refer to the latter list as the progress list. For a time period of length τ , we use our algorithm to schedule flows of coflows; namely, we list schedule flows according to the original list (i.e., based on optimal solution to LP). At the end of this time interval, we compute progress of coflows and update the progress list. Denote by \bar{p}_t the average progress over the progress list at time t and assume that coflow k is the first coflow in the progress list. The goal of scheduling over the period δ is to decrease the gap between the progress of starved coflows and the average progress. Toward this end, we schedule the flows for a time period of length $\Delta = \min\{W_t(k), \bar{p}_t - p_t(k)\}$, where $W_t(k)$, current effective size of coflow k , is the time needed to complete coflow k ignoring other coflows in the system, and $\bar{p}_t - p_t(k)$ is the gap between its progress and the average progress. Keeping the scheduling policy simple, we use the list scheduling using the progress list for Δ amount of time. We then update the progress list, compute the average progress, current effective size of the first coflow in the progress list, and Δ , and continue in the same manner until either the total amount of time spent in this scheduling phase reaches δ or the progress of all coflows becomes equal. Afterwards, we preempt the schedule, update the original coflow list, and resume our list scheduling for another τ amount of time, and so on. Setting the parameter δ to 0 will produce our original scheduling algorithm. By choosing $\delta > 0$, we can avoid coflow starvations at the cost of an increased total completion time. Varys [6] also uses a two phase procedure, however the way that we compensate for fairness, by list scheduling based on the progress list, is different from Varys.

To examine the performance of the proposed scheme, we consider all coflows of the real traffic trace when they release at time zero and look at the total completion time of coflows and the unfairness metric (average of standard deviations measured upon departure of coflows) for different values of τ and δ . For practical consideration, as suggested by Varys [6], we set δ to be $O(100)$ milliseconds and T to be $O(1)$ second. Figure 9 shows the total completion time for different values of δ and τ , normalized with the performance of LP-OV-LS (our original algorithm) which is when $\delta = 0$ for any value of τ . For a fixed δ , total completion time decreases as τ increases because the algorithm schedules flows based on the list that is formed to optimize total completion time for larger fraction of time. Also, fixing τ , total completion time increases as δ increases. Figure 10 depicts the corresponding unfairness metric for different values of δ and τ . We can see that, as δ increases, average of progress standard deviations decreases, which means that the scheduling algorithm allocates rates in a more fair manner. Moreover, fixing δ , unfairness increases as we increase τ , as expected.

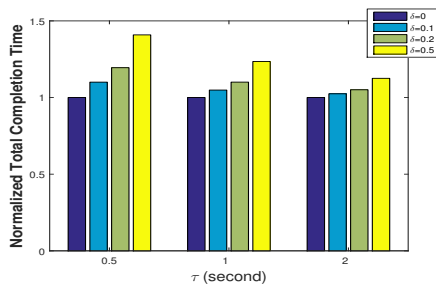


Fig. 9: Total coflow completion time for different values of δ and τ (both in second), normalized with the performance of LP-OV-LS ($\delta = 0$ for any τ).

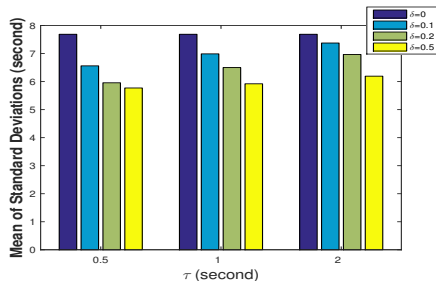


Fig. 10: Average standard deviation for the progress of coflows, for different values of δ and τ (both in second).

E. Discussion on Algorithm's Complexity

In this section, we provide a discussion on complexity of our algorithm which is mainly determined by the step of finding appropriate ordering of coflows. The scheduling step is the simple list scheduling policy where complexity of computing the schedule—upon arrival or departure of a flow—is at most the length of the list, which is equal to the number of incomplete flows. The relaxed LP (7) that is used to obtain ordering of coflows has $O(K^2)$ variables and $O(K^2 + KN)$ constraints and can be solved in polynomial time, e.g. using interior point method [29]. On a desktop PC, with 8 Intel CPU core *i7* – 4790 processors @ 3.60 GHz and 32.00 GB RAM, it took 101.93 seconds to solve the LP for the Facebook trace, when all coflows are considered for the case of general release dates. In this case, the maximum coflow completion time under our algorithm is 3492 seconds and the average completion time is 183.7 seconds. For the collection with $M \geq 50$, it took 24.40 seconds to solve the LP for the case of general release dates. In this case, the maximum coflow completion time under our algorithm is 3447 seconds and the average completion time is 194.23 seconds. We note that solving the LP can be done much faster using the powerful computing resources in today's datacenters. The computation overhead as well as communication overhead (i.e., sending the rates to servers) might still be an issue for smaller coflows—the same issue as in other algorithms such as Varys [6].

IX. CONCLUDING REMARKS

In this paper, we studied the problem of scheduling of coflows with release dates to minimize their total weighted completion time, and proposed an algorithm with improved

approximation ratio. We also conducted extensive experiments to evaluate the performance of our algorithm, compared with three algorithms proposed before, using both real and synthetic traffic traces. Our experimental results show that our algorithm in fact performs very close to optimal.

As future work, other realistic constraints such as precedence requirement or deadline constraints need to be considered. Also, theoretical and experimental evaluation of the performance of the proposed online algorithm is left for future work. While we modeled the datacenter network as a giant non-blocking switch (thus focusing on rate allocation), the routing of coflows in the datacenter network is also of great importance for achieving the quality of service.

REFERENCES

- [1] M. Shafiee and J. Ghaderi, "Brief announcement: A new improved bound for coflow scheduling," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2017.
- [2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE, 2010, pp. 1–10.
- [4] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.
- [5] M. Isard, M. Buidu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.
- [6] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 443–454.
- [7] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 431–442.
- [8] N. M. K. Chowdhury, *Coflow: A networking abstraction for distributed data-parallel applications*. University of California, Berkeley, 2015.
- [9] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 503–514.
- [10] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 465–478, 2015.
- [11] M. Shafiee and J. Ghaderi, "A simple congestion-aware algorithm for load balancing in datacenter networks," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.
- [12] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 31–36.
- [13] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapiet: Integrating routing and scheduling for coflow-aware data center networks," in *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2015, pp. 424–432.
- [14] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 2015, pp. 393–406.
- [15] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, pp. 294–303.
- [16] S. Khuller and M. Purohit, "Brief announcement: Improved approximation algorithms for scheduling co-flows," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2016, pp. 239–240.

- [17] M. Shafiee and J. Ghaderi, "Scheduling coflows in datacenter networks: Improved bound for total weighted completion time," *ACM SIGMETRICS, Poster Paper*, 2017.
- [18] K. Jurcik, "Open shop scheduling to minimize makespan," *Department of Mathematical Sciences Lakehead University Thunder Bay, Ontario*, 2009.
- [19] Y.-A. Kim, "Data migration to minimize the total completion time," *Journal of Algorithms*, vol. 55, no. 1, pp. 42–57, 2005.
- [20] L. A. Hall, D. B. Shmoys, and J. Wein, "Scheduling to minimize average completion time: Off-line and on-line algorithms," in *SODA*, vol. 96, 1996, pp. 142–151.
- [21] M. Mastrolilli, M. Queyranne, A. S. Schulz, O. Svensson, and N. A. Uhan, "Minimizing the sum of weighted completion times in a concurrent open shop," *Operations Research Letters*, vol. 38, no. 5, pp. 390–395, 2010.
- [22] S. Ahmadi, S. Khuller, M. Purohit, and S. Yang, "On scheduling coflows," in *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 2017, pp. 13–24.
- [23] H. Kellerer, T. Tautenhahn, and G. Woeginger, "Approximability and nonapproximability results for minimizing total flow time on a single machine," *SIAM Journal on Computing*, vol. 28, no. 4, pp. 1155–1166, 1999.
- [24] M. Pinedo, *Scheduling*. Springer, 2015.
- [25] L. Schrage, "Letter to the editor—a proof of the optimality of the shortest remaining processing time discipline," *Operations Research*, vol. 16, no. 3, pp. 687–690, 1968.
- [26] S. Sachdeva and R. Saket, "Optimal inapproximability for scheduling problems via structural hardness for hypergraph vertex cover," in *Computational Complexity (CCC), 2013 IEEE Conference on*. IEEE, 2013, pp. 219–229.
- [27] C. Potts, "An algorithm for the single machine sequencing problem with precedence constraints," in *Combinatorial Optimization II*. Springer, 1980, pp. 78–87.
- [28] R. Gandhi, M. M. Halldórsson, G. Kortsarz, and H. Shachnai, "Improved bounds for scheduling conflicting jobs with minsum criteria," *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 1, p. 11, 2008.
- [29] J. Renegar, "A polynomial-time algorithm, based on newton's method, for linear programming," *Mathematical Programming*, vol. 40, no. 1-3, pp. 59–93, 1988.
- [30] Z. Qiu, C. Stein, and Y. Zhong, "Experimental analysis of algorithms for coflow scheduling," in *International Symposium on Experimental Algorithms*. Springer, 2016, pp. 262–277.
- [31] R. Srikant, *The mathematics of Internet congestion control*. Springer Science & Business Media, 2012.
- [32] D. Nace and M. Pióro, "Max-min fairness and its applications to routing and load-balancing in communication networks: a tutorial," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, 2008.
- [33] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li, "Towards practical and near-optimal coflow scheduling for data center networks," 2016.

APPENDIX

NP-COMPLETENESS AND COUNTER EXAMPLE

NP-Completeness of Optimization (3): We first show NP-completeness of the coflow scheduling problem as formulated in optimization (3). This is done through reduction from the concurrent open shop problem, in which a set of K jobs and a set of N machines are given. Each job consists of some tasks where each task is associated with a size and a specific machine in which it should be processed. We convert each job to a coflow by constructing a diagonal demand matrix [15]. By this construction, the constraints (3d) and (3e) are equivalent. The optimal solution to optimization (3) consists of non-negative transmission rates $x_{j,j}^{k,*}(t)$ that sum to at most one on destination node j at each time $t \in [0, T]$. However, in the concurrent open shop problem each machine can work on one task at a time which can be translated to zero and one transmission rates in the jargon of the coflow scheduling problem. Now, we show that given an optimal solution with rates $x_{j,j}^{k,*}(t)$ to optimization (3) for the converted coflow

scheduling problem, we can always transform it to a feasible solution for the original concurrent open shop problem. To do so, we consider destination node j (machine j) and start from the last flow (task) that completes on this node. If there are multiple last flows, we choose one arbitrarily. We denote by $f_{j,j}^{k,*}$ its optimal finishing time and by $d_{j,j}^k$ its size. We then set all transmission (processing) rates of this flow (task) to zero from time 0 to $f_{j,j}^{k,*} - d_{j,j}^k$, and to one from time $f_{j,j}^{k,*} - d_{j,j}^k$ to $f_{j,j}^{k,*}$. We adjust rates of other flows such that transmission rates sum to at most one at every time while all the flows are guaranteed to be processed before their completion time (which is given by the optimal solution). This can be easily done by increasing $x_{j,j}^{k'}(t)$ for $t \in [0, f_{j,j}^{k,*} - d_{j,j}^k]$ by $\Delta x_{j,j}^{k'}(t)$ determined as follows

$$\Delta x_{j,j}^{k'}(t) = \frac{\int_{f_{j,j}^{k,*} - d_{j,j}^k}^{f_{j,j}^{k,*}} x_{j,j}^{k'}(\tau) d\tau}{d_{j,j}^k} \times x_{j,j}^{k'}(t)$$

By doing so, finishing time of the last flow does not change, and finishing time of other flows may decrease. The iterative procedure is repeated until processing rates of all flows converted to zero or one on node j . Therefore, we end up with possibly better solution in terms of total completion times of flows for node j with zero-one rates. We apply this mechanism to all nodes; hence, the total completion time of the transformed solution is as good as the optimal solution. Thus, if an algorithm can solve the coflow scheduling problem in polynomial time, it can do so for concurrent open shop problem which contradicts with its NP-completeness. This completes the argument and NP-completeness of coflow scheduling problem is concluded.

2-approximation algorithms from the concurrent open shop cannot be directly applied to coflow scheduling: As we discussed in Section III, the 2-approximation algorithms for the concurrent open shop problem cannot be directly applied to achieve 2-approximation algorithms for the coflow scheduling problem. This is because given an ordering of K coflows, there does not always exist a schedule in which the first coflow completes at time $W(1)$, the second coflow completes at time $W(1, 2)$, and so on, until the last coflow completes at time $W(1, \dots, K)$ (recall Definition 2 for definition of $W(1, \dots, k)$). We provide a counter example to show this.

Example 3 (Counter Example). Consider a 3×3 network with 2 coflows as shown in Figure 11. One can force the ordering algorithm to output orange coflow as the first coflow and the green coflow as the second one in the list (e.g., by means of assigning appropriate weight to coflows). To finish the first coflow (orange coflow) in $W(1)$, transmission rates are assigned as shown in Figure 12a. To avoid under-utilization of network resources, the remaining capacities are dedicated to flows of coflow 2 (green coflow). After $W(1) = 2$ units of time, coflow 1 completes and the remaining flows of coflow 2 is as shown in Figure 12b, therefore, one needs 2 more units of time to complete remaining flows of coflow 2. Hence, coflow 2 completes at time $4 > W(1, 2) = 3$.

In fact, the 2-approximation algorithm in [33], for coflow scheduling when all the release dates are zero, relies on the

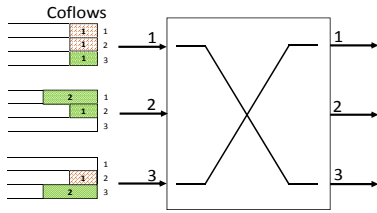
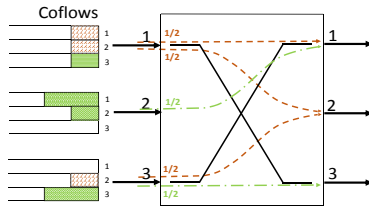
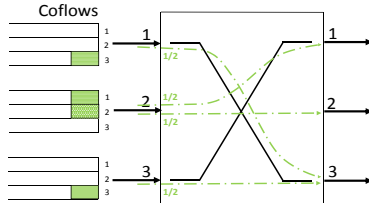


Fig. 11: Two coflows in a 3×3 switch architecture. Flow sizes are depicted inside each flow.

assumption that such a schedule exists which, as we showed by the counter example, is not always true and hence the 4-approximation algorithm proposed in this paper is the best known approximation algorithm in this case.



(a) Transmission rates so as to complete orange coflow at time 2.



(b) Remaining flows of green coflow at time 2 and rate assignment to complete its flows at time 4.

Fig. 12: Inaccuracy of proposed algorithm in [33].



Mehrnoosh Shafiee joined M.Sc./ Ph.D. program of the Department of Electrical Engineering at Columbia in August 2014. She is interested in the analysis and design of resource allocation algorithms for large-scale distributed systems. She did her B.Sc. in EE department of the Sharif University of Technology, Tehran, Iran.



Javad Ghaderi joined the Department of Electrical Engineering at Columbia University in July 2014. His research interests include network algorithms and network control and optimization. Dr. Ghaderi received his B.Sc. from the University of Tehran, Iran, in 2006, his M.Sc. from the University of Waterloo, Canada, in 2008, and his Ph.D. from the University of Illinois at Urbana-Champaign (UIUC), in 2013, all in Electrical and Computer Engineering. He spent a one-year Simons Postdoctoral Fellowship at the University of Texas at Austin before joining Columbia. He is the recipient of the Mac Van Valkenburg Graduate Research Award at UIUC, Best Student Paper Finalist at the 2013 American Control Conference, Best Paper Award at ACM CoNEXT 2016, and NSF CAREER Award in 2017.