Towards Practical and Near-optimal Coflow Scheduling for Data Center Networks*

Shouxi Luo, Hongfang Yu, Yangming Zhao, Sheng Wang, Shui Yu, and Lemin Li

Abstract—In current data centers, an application (e.g., MapReduce, Dryad, search platform, etc.) usually generates a group of parallel flows to complete a job. These flows compose a *coflow* and only completing them all is meaningful to the application. Accordingly, minimizing the average Coflow Completion Time (CCT) becomes a critical objective of flow scheduling. However, achieving this goal in today's Data Center Networks (DCNs) is quite challenging, not only because the schedule problem is theoretically NP-hard, but also because it is tough to perform practical flow scheduling in large-scale DCNs. In this paper, we find that the optimal result of minimizing the average CCT of a set of coflows is bounded by that of the well-known problem of minimizing the sum of completion times in a *concurrent open shop*. As there are abundant existing solutions for concurrent open shop, we open up a variety of techniques for coflow scheduling. Inspired by the best known result, we derive the 2-approximation algorithm from concurrent open shop for coflow scheduling, and further develop a *decentralized* coflow scheduling system, D-CAS, which avoids the system problems associated with current centralized proposals while addressing the performance challenges of decentralized suggestions. Trace-driven simulations indicate that D-CAS achieves a performance close to Varys, the state-of-the-art centralized method, and outperforms Baraat, the only existing decentralized method, significantly.

Index Terms—Coflow, datacenter networks, decentralized, scheduling

1 Introduction

Today's data centers widely employ cluster computation frameworks (e.g., MapReduce [1], Dryad [2], CIEL [3], and Spark [4]) to deal with the increasing data process and analysis demands. In these frameworks, data-intensive jobs are divided into multiple successive data-parallel computation stages, and a succeeding computation stage cannot start until *all* its required inputs are in place, which are the outputs of the previous stage. Recent studies [5]–[7] have shown that the intermediate data transmission is not a negligible phase in the process of a job. For example, it accounts for 33% of the job running time in Facebook's system [5]. Accordingly, speeding up data transfers between computation stages will accelerate the job completions and increase resource utilizations [5]–[7].

In general, an inter-stage data transfer of a job involves a group of parallel flows. Such a collection of flows sharing the same performance goal like minimizing the latest flow's completion time or ensuring that all flows meet a common deadline are defined as a *coflow* [6, 8]. To optimize the completion times of jobs, the hypervisor needs to schedule flows at coflow level rather than individual ones, because only completing all flows in a coflow will let the job computation enter its next stage. Such a feature makes the flow scheduling in Data Center Networks (DCNs) quite challenging as the "coflow-level" relations between flows need to be considered. As a result, the problem of optimizing job completions is to find the best schedule to minimize the average Coflow Completion Time (CCT).

Several recent works [5]–[7, 9, 10] have attempted to investigate this schedule problem of minimizing average CCTs. To the best of our knowledge, Varys [6] and Baraat [7] are the state-of-the-art schemes in centralized and decentralized fashions, respectively. However, centralized schemes like Varys suffer from the scalability problem. On one hand, it is difficult for a central controller to collect all the real-time coflow information in a large scale DCN; on the other hand, calculating schedule schemes for the entire network also involves lots of computation. What's worse, the computed schedule schemes need be executed by flow senders (or switches), yet it is impossible to accurately synchronize all senders (or switches, respectively) in real time.

Different from Varys, Baraat [7] is a decentralized solution scheduling coflows according to their arrival orders. Baraat shows that FIFO based schemes achieve good performance if coflows are homogeneous, i.e., their sizes vary in a limited range and do not follow a heavy-tail distribution. However, this is not always the case in practice, where coflows would be heterogeneous as both their sizes and numbers of flows vary widely [6]. Accordingly, *non-preemptive* policies like FIFO suffer from the *head-of-line blocking problem* [6, 7]. Despite Baraat detects large size coflows online and adopts fair sharing to mitigate head-of-line blocking, it still badly underperforms (even worse than the naive per-flow fair sharing) on heterogeneous coflows as we will show.

To make effective coflow schedules, we firstly studied the minimization of CCTs in theory. By formulating the schedule of coflows as the management of their flow priorities, we showed that the optimal result of minimizing the average CCT for a set of coflows is bounded by that of minimizing the sum of completion times in a *concurrent open shop*, which is a well-known NP-hard problem [11]. Then, based on the best known result of concurrent open shop [11], we reformed the 2-approximation

- *S. Luo, H. Yu, Y. Zhao, S. Wang, and L. Li are with the Key Laboratory of Optical Fiber Sensing and Communications, Ministry of Education, University of Electronic Science and Technology of China, Chengdu 611731, P.R.China. E-mail: rithmns@gmail.com, {yuhf, zhaoyangming, wsh_keylab, lml}@uestc.edu.cn*
- *S. Yu is with the School of IT, Deakin University, Victoria, 3125, Australia. E-mail: syu@deakin.edu.au.*

**This version corrects some errors in the version published in TPDS.*

schedule algorithm for minimizing average CCTs.

To perform practical coflow scheduling, we deeply analyzed the properties that a practical scheduling system should have, and proposed D-CAS, a *Decentralized*, *Coflow-Aware* scheduling System, based on the 2-approximation algorithm. Since the original 2-approximation algorithm is offline and requires a central controller to make complicated schedules, we reformed it to an online, decentralized algorithm, which just schedules each coflow by dynamically setting its packet priorities according to the maximum remaining size (i.e., the size of untransmitted data) on senders. Following this, D-CAS avoids the system problems associated with centralized Varys (i.e., scalability, fault-tolerance, etc.) while addressing the performance challenges of decentralized Baraat (i.e., underperforming on heterogeneous coflows).

We developed a simulator based on that of Varys [12], and used coflow traces generated with real parameters [6, 12] to evaluate the performance of D-CAS under various settings. Extensive simulations confirmed the effectiveness of D-CAS: it improves the average CCTs about 2–3 times over per-flow fairness; the performance improvements are close to that of the state-of-the-art centralized scheme Varys – the gap is always less than 15%; and it outperforms the only existing decentralized scheme Baraat by 1.4–4 times. In addition, even when switches have limited priority queues and senders are agnostic of remaining sizes, D-CAS still obtains good performances – it outperforms the per-flow fairness more than 2.5 times.

In summary, the contributions of this paper are twofold.

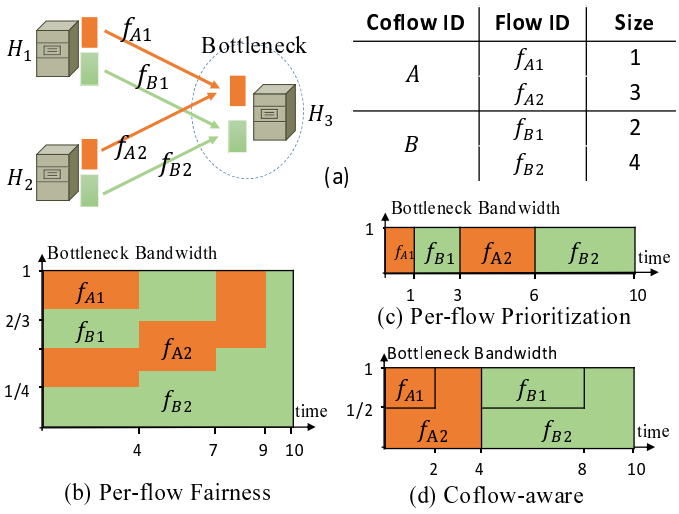
- We analyzed the connection between coflow scheduling and concurrent open shop on the object of minimizing the average completion times, and showed how the findings from the latter motivates the schedule of coflow.
- We proposed D-CAS, a simple, flexible, effective, practical, and readily-deployable decentralized coflow scheduling system, to optimize average CCTs in DCNs.

The rest of the paper is organized as follows. Section 2 briefly overviews the background and motivation of our work. Section 3 relaxes the coflow scheduling into the problem of *concurrent open shop*. Section 4 designs the practical schedule algorithm and Section 5 further develops D-CAS. After that, extensive simulations are presented in Section 6. Finally, related work and conclusions follow in Section 7 and Section 8, respectively.

2 Background and Motivation

In this section, we present three key desirable properties of flow schedulers for minimizing average CCTs in DCNs, which motivate our design of schedule algorithms and D-CAS.

By defaults, the term *flow* is used to indicate a sequence of packets from a source computer to a destination belonging to the same data transport task, such as a worker reads the input data from another [1], or replicates the result to a replica node of the cluster file systems [13]. At the application-level, those flows with associated semantics and a collective objective (e.g., belonging to the same MapReduce job's data shuffle phase, or result replicating phase [1, 8]) compose a *coflow*.



Coflow ID	Flow ID	Size
A	f_{A1}	1
	f_{A2}	3
B	f_{B1}	2
	f_{B2}	4

Fig. 1. Motivating example[1]. (a) Four concurrent flows from two coflows competing for a single bottleneck link; (b) Fair sharing, $avg(CCT)$=9.5; (c) Smallest-flow-first, $avg(CCT)$=8; (d) Smallest-coflow-first, $avg(CCT)$=7.

2.1 Why Coflow-aware Scheduling

The goal of flow scheduling is to optimize CCTs. Conventional coflow-agnostic scheduling methods designed to minimize the average Flow Completion Time (FCT), e.g., pFabric [16] and PDQ [15], cannot reach the optimum solution.

Consider the example illustrated in Fig. 1. There are four concurrent flows belonging to two coflows (Coflow A: f_{A1}, f_{A2}; Coflow B: f_{B1}, f_{B2}.). All flows arrive simultaneously and their demands are shown in Fig. 1a. With per-flow max-min fairness–the *de facto* standard of today's bandwidth allocation schemes [14], the result is illustrated in Fig. 1b. In this case, all the unfinished flows in the network obtain the same bandwidth, and flows $\langle f_{A1}, f_{A2}, f_{B1}, f_{B2}\rangle$ will complete at time $\langle 4, 9, 7, 10\rangle$. Hereby, coflow A and coflow B complete at time 9 and 10, respectively. Their average CCT is $\frac{9+10}{2} = 9.5$. Similarly, the smallest-flow-first scheme [15, 16] that minimizes the average FCT can only derive the average CCT $\frac{6+10}{2} = 8$ as Fig. 1c shows. If the coflow-aware flow scheduling scheme, such as *smallest-coflow-first* (i.e., the coflow with the minimum total flow volume is scheduled first) [6] is adopted, the average CCT is only $\frac{4+10}{2} = 7$ as shown in Fig. 1d. There is a saving of $\sim 26\%$ compared to fair sharing scheme and a saving of $\sim 13\%$ compared to flow level scheduling scheme.

In a word, *the coflow-aware scheduling policy can bring benefits to the average CCT in DCNs.*

1. In the case of per-flow max-min fairness [14], initially, four concurrent flows compete the ingress bandwidth of H3 and each flow get 1/4 unit of bandwidth. After 4 units of time, the flow with the smallest demand (i.e., f_{A1}) completes; then each of the remaining 3 flows obtains 1/3 bandwidth as Fig. 1b shows; and so on. After 10 units of time, all flows complete and their FCTs are 4, 9, 7, 10, respectively.

Differently, in the case of smallest-flow-first scheduling [15, 16], flows preempt the bottleneck bandwidth according to their flow-level demand sizes, 1, 3, 2, 4, resulting in the FCTs of 1, 6, 3, 10 as Fig. 1c demonstrates.

Similarly, with smallest-coflow-first mechanism, the priority numbers of f_{A1}, f_{A2}, f_{B1}, and f_{B2} are respectively set to 1, 1, 2, and 2, according to their coflows' demand sizes. As a result, they preempt the bottleneck bandwidth as Fig. 1d demonstrates, which leads to FCTs 4, 4, 10, and 10.

2.2 Why Decentralized Scheduling

Intuitively, if all coflow information is available, one can schedule coflows following the coflow-level *minimum-remaining-time-first* (MRTF) policy by using a central scheduler like Varys [6]. However, this idea is not practical in reality. Firstly, current data centers have hundreds of thousands of hosts and millions of concurrent flows [17], it is difficult to collect all flow information (such as the sources, destinations and remaining sizes) to compute a global scheduling scheme. Moreover, it is hard to enforce the computed scheduling actions to all flows from different senders in real time. Accordingly, the control delay of the centralized scheduling is too big for pervasive small coflows since these coflows may complete within a few RTTs (Round-Trip Time) [7]. On the other hand, as a centralized scheduling system, the central scheduler also bears other problems such as scalability and fault-tolerance.

Accordingly, *the decentralized scheduling is preferred in a system to improve the average CCT in DCNs*.

2.3 Why Preemptive Scheduling

To minimize the average CCT, a flow scheduler should pursue the coflow-level MRTF principle in decentralized fashions. Namely, on hosts, smaller coflows should be scheduled before larger ones. In an online system, we cannot suppose smaller coflows always arrive earlier than the larger ones. Therefore, preemptive schedule schemes are necessary. Otherwise, the system would suffer from the head-of-line blocking problem, i.e., small coflows arriving later being blocked by those early arrived large coflows.

Baraat is a non-preemptive system addressing the head-of-line blocking problem by deploying a Limited Multiplexing (LM) scheme on switches. LM would dynamically change the level of multiplexing and let flows with the lower priority to be served when the current coflow is detected as large. Due to Baraat's non-preemptive property, it is subjected to two major shortcomings. First, LM scheme performs badly when coflow sizes are heterogeneous. In some cases, it is even worse than the naive fair sharing scheme. Second, with the increasing of multiplexing level, the performance of Baraat approaches that of the fair sharing, which is demonstrated to be unsuitable for minimizing average completion times [6, 15, 16].

Hereby, *preemption is a necessary property to minimize the average CCT in DCNs.*

3 A Theoretical Analysis of the Average CCT Minimization

In this section, we theoretically analyze the minimization of average CCTs. By formulating the schedule of coflows as the management of their flow priorities, we make a low bound analysis of minimizing the average CCT for a given set of coflows, which in turn gives insights for designing coflow schedule algorithms.

3.1 Network Model

We start by introducing two basic assumptions that we make in our theoretical analysis as well as algorithm design, named *Prioritized Fairness* and *Ideal Rate Control*, respectively.

- *Prioritized Fairness:* When multiple flows compete on an ingress, egress, or link, flows with the higher priority will preempt those with the lower priority, and flows with the same priority share the bottleneck's bandwidth fairly (i.e., bandwidth allocation follows the TCP style per-flow max-min fairness [14, 18] within each priority).
- *Ideal Rate Control:* Each flow sender performs a TCP-alike rate-control. All rate adjustments complete immediately and the network bandwidth is always fully used in the prioritized fairness fashion.

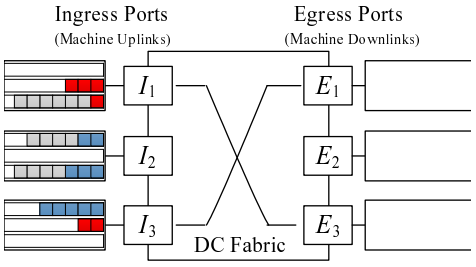


Fig. 2. Conceptual view of the network: the fabric is non-blocking and packets (or flows or coflows) preempt the bandwidth according to their priorities.

These two assumptions capture the common key feature of current data center transport schemes [15, 16, 19, 20] and make the theoretical analysis of coflow scheduling easy. We note that yet there is a gap between the point that currently deployed transport schemes could achieve and the ideal one obtained under the assumptions. Especially, current congestion control algorithms are not *ideal* [19]–[21]—They might take time to get convergence; accordingly, the network might not always be fully used. Fortunately, recent studies [16] have shown that, by setting flow priorities dynamically and explicitly (the design that our final proposal employs), the need for rate control is minimal and simple rate-control schemes would achieve near-optimal results. Thus, our final proposed solution does not rely on the assumptions strongly and would work well in common scenes. Additionally, many recent research contributions have been made to achieve quick convergences for rate control [15, 21], with which, the proposed approach could reach a point closer to the ideal one in future.

For simplicity and in common with Varys [6], we further abstract the entire data center network as a non-blocking fabric [6, 16, 18, 22], in which all ports have the same normalized unit capacity, and bandwidth competition only appears in ingresses or egresses. Such an abstraction is reasonable and matches with recent full bisection bandwidth topologies widely used in current production data centers [17, 23]—To provide uniform high capacity as well as other targets like equidistant endpoints with network core, unlimited workload mobility, etc., today's production data center networks widely adopt non-blocking Clos topologies by design [17, 22, 23]. As an example, Fig. 2 shows a case of a small non-blocking DCN connected with 3 hosts, in which tree coflows preempt the network according to their priorities (with different colors).

3.2 Problem Statement

We consider the offline scheduling of n coflows on a non-blocking DCN with m hosts, i.e., there are m ingresses and m egresses, and their capacity is 1. Without loss of generality, we assume that each coflow involves $m \times m$ parallel flows between the m ingresses and m egresses. Then, the coflow indexed by k is a collection of flows $D^{(k)} = \{d_{i,j}^{(k)} | 1 \leq i \leq m, 1 \leq j \leq m\}$, where $d_{i,j}^{(k)}$ is the size of the flow to be transferred from host i to host j. Note that,

for any coflow k, if it involves multiple parallel flows from i to j, we merge their demands and treat them as a single big flow. On the contrary, if the coflow does not involve any flow from host i to host j, we just set $d_{i,j}^{(k)} = 0$.

Let $r_{i,j}^{(k)}(t)$ be the rate of coflow $D^{(k)}$ from host i to host j at time instance t, and C_k be its completion time; then, for a given set of coflows, labeled $1, \ldots, n$, minimizing their average CCT is equivalent to minimizing their sum of CCTs, while subjecting to the capacity constraints on senders (i.e., Constrains (2b)) and receivers (i.e., Constrains (2c)), as the mathematical program $\{$(1), (2a), (2b), (2c), (2d)$\}$ denotes. During our following analysis, we use Equation (1) as the optimization objective that coflow scheduling methods pursue.

$$\min \sum_{k=1}^{n} C_k. \tag{1}$$

Subject to:

$$\forall(k,i,j) : \int_0^{C_k} r_{i,j}^{(k)}(t)dt = d_{i,j}^{(k)} \tag{2a}$$

$$\forall(t,i) : \sum_{k=1}^{n}\sum_{j=1}^{m} r_{i,j}^{(k)}(t) \leq 1 \tag{2b}$$

$$\forall(t,j) : \sum_{k=1}^{n}\sum_{i=1}^{m} r_{i,j}^{(k)}(t) \leq 1 \tag{2c}$$

$$\forall(k,i,j,t) : 0 \leq r_{i,j}^{(k)}(t) \leq 1 \tag{2d}$$

3.3 A Low-bound Analysis

In this subsection, we show that, on the target of minimizing the total completion times for a given set of coflow, the result obtained from the optimal order permutation of its associated concurrent open shop scheduling problem gives a bound of the original coflow scheduling, i.e., Proposition 1. Based on this, we get the guideline for the schedule of coflows.

As a brief introduction, we sketch the *concurrent open shop* scheduling problem as following: Consider that there are n^* jobs and each job involves at most m^* types of concurrently operations, which should be processed by m^* specific machines, respectively. The objective of concurrent open shop scheduling is to find a job execution order that could minimize the total job completion times, or the number of late jobs when each job has a hard deadline, or other objectives [11, 24].

Proposition 1. *The optimal solution of minimizing the sum of job completion times in a concurrent open shop gives a low bound of minimizing the total CCTs for a given set of coflows.*

Firstly, let's consider the minimization of a single coflow's completion time. For any coflow $D^{(k)}$, denote $\rho(D^{(k)})$ (or ρ_k for short) to be its maximum load on hosts calculated by Equation (3). Then, obviously, the completion time of $D^{(k)}$ (i.e., C_k) must not be smaller than $\rho_k/1 = \rho_k$. That is to say, ρ_k gives a low bound of a coflow's completion time. Actually, as Lemma 1 says, this bound is exactly the optimum of $\min C_k$ and it is easy to achieve.

$$\rho(D^{(k)}) = \max\left\{\max_i\left\{\sum_{j=1}^{m} d_{i,j}^{(k)}\right\}, \max_j\left\{\sum_{i=1}^{m} d_{i,j}^{(k)}\right\}\right\} \tag{3}$$

Lemma 1. *For coflow k, we can achieve its optimal completion time $\rho(D^{(k)})$ by simply setting all its flows with the highest and exclusive priority.*

Proof. The proof is quite simple. If coflow k does not complete within ρ_k, it must not get the smallest and exclusive priority value, or the bottleneck port must not be fully used. These two cases contradict with the assumption of *Prioritized Fairness* or *Ideal Rate Control*, respectively. □

Lemma 1 discusses the case of minimizing a single coflow's completion time. Next, we look into the schedule of two coflows. As a technical convenience, thereafter, we use "+" to present the operation of treating two coflows as a single one. That is to say, for two $m \times m$ coflows, $D^{(k_1)}$ and $D^{(k_2)}$, $D^{(k_1)} + D^{(k_2)}$ will produce another $m \times m$ coflow $D^{(k_3)}$, whose demand from host i to host j is $d_{i,j}^{(k_3)} = d_{i,j}^{(k_1)} + d_{i,j}^{(k_2)}$. Then, based on Lemma 1, we get the following lemma for minimizing the sum of two coflows' completion times.

Lemma 2. *For two coflows, say $D^{(k)}$ and $D^{(l)}$, the optimum of $\min\max\{C_k, C_l\}$ is $\rho(D^{(k)} + D^{(l)})$, which can be achieved by setting their flows with the same highest and exclusive priority.*

Proof. We first prove that $\rho(D^{(k)} + D^{(l)})$ is the low bound of $\min\max\{C_k, C_l\}$. Obviously, with any scheduling schemes, $\max\{C_k, C_l\}$ must not be smaller than $\rho(D^{(k)} + D^{(l)})$. This is because, if $\max\{C_k, C_l\} < \rho(D^{(k)} + D^{(l)})$, it means both coflow k and coflow l can complete before $\rho(D^{(k)} + D^{(l)})$, i.e., coflow $D^{(k)} + D^{(l)}$ can complete before $\rho(D^{(k)} + D^{(l)})$—There is a contradiction.

Then, we show how to get $\rho(D^{(k)} + D^{(l)})$. Similar to Lemma 1, if these two coflows have the same smaller flow priorities than all others, they must complete within $\rho(D^{(k)}+D^{(l)})$. □

Lemma 2 indicates that, on making the optimal schedule of two coflows, say $D^{(k)}$ and $D^{(l)}$, the minimized completion time of the latter one must be $\rho(D^{(k)} + D^{(l)})$. If both the first and the second coflow can simultaneously achieve their minimized completion times, the optimum of $\min(C_k + C_l)$ must be either $\rho(D^{(k)}) + \rho(D^{(k)} + D^{(l)})$, or $\rho(D^{(l)}) + \rho(D^{(k)} + D^{(l)})$, corresponding to the permutations of the two coflows. Unfortunately, in some cases, their optimal values can not be reached at the same time. As an example, consider the schedule of $\{d_{1,1}^{(1)} = 1, d_{2,2}^{(1)} = 1\}$ and $\{d_{1,3}^{(2)} = 1, d_{2,3}^{(2)} = 1\}$, in which, $\rho(D^{(1)}) + \rho(D^{(1)} + D^{(2)})$ and $\rho(D^{(2)}) + \rho(D^{(2)} + D^{(1)})$ are 3 and 4, respectively, while their minimized sum of completion times is 4. So, this permutation-based computation gives a lower bound of the two coflow's schedule.

With mathematical induction, it is easy to generalize this conclusion to the optimal schedule of n coflows. According, we get the fact that this low bound analysis of total completion time's minimization given by optimal priority permutation always exists for coflow scheduling as Lemma 3 states, in which, $\pi[k]$ denotes the priority number (i.e., permutation index) of coflow k, $T(\pi)$, calculated by Equation (4), stands for the sum of the ideal coflow completion times under priority permutation π.

$$T(\pi) = \sum_{i=1}^{n} \rho(\sum_{j=1}^{i} D^{(\pi[j])}) \tag{4}$$

Lemma 3. *For a set of coflows, $\min_{\pi\in\Pi} T(\pi)$ gives a low bound of their total CCT, where Π is the set of all their permutations.*

So far, we have transferred the low bound of minimizing total CCTs for a given set of coflows into the problem of finding an optimal order permutation for them, which is quite similar to the well-known concurrent open shop scheduling. Indeed, this $T(\pi)$'s minimization problem is exactly the optimization of minimizing the total completion times in a concurrent open shop as Theorem 1 says. By Lemma 3 and Theorem 1, we finally obtain Proposition 1.

Theorem 1. *Finding coflows' priority permutation π to minimize their $T(\pi)$, is equivalent to the problem of minimizing the sum of job completion times in a concurrent open shop.*

Proof. For each coflow $D^{(k)}$, let $d_i^{(k)} = \sum_{j=1}^{m} d_{i,j}^{(k)}$ for ingress $i = 1 \dots n$, and $d_{j+m}^{(k)} = \sum_{i=1}^{m} d_{i,j}^{(k)}$ for egress $j = 1 \dots n$. By substituting $d_i^{(k)}$ and $d_{j+m}^{(k)}$ into Equation (4), we obtain the fact that finding the optimal permutation π to minimize coflows's $T(\pi)$ is exactly the case of making permutation schedules to minimize the total job completion time in a concurrent open shop. In this constructed concurrent open shop problem, there are $2m$ machines, and $d_i^{(k)}$ (or $d_{j+m}^{(k)}$, resp.) denotes the operations of job k that need be executed by machine i (or $j+m$, resp.). □

Given a set of coflows, let *opt* be the optimal result of their sum of completion times, and Π be the set of all their permutations. From Proposition 1, we know that $\min_{\pi\in\Pi} T(\pi) \leq opt$. For each permutation π, suppose $T^*(\pi)$ to be the sum of their completion times under the setting that flows belonging to the i-th coflow (i.e., $D^{(\pi[i])}$) carry with the packet priority value i. Then, *opt* must be bounded by $\min_{\pi\in\Pi} T(\pi) \leq opt \leq \min_{\pi\in\Pi} T^*(\pi)$. That is to say, we can obtain good total completion time optimization with a good permutation order. Indeed, recent literature has implied that the permutation order suggested by the results of relaxed concurrent open shop schedule could also derive bounded solutions for coflow scheduling [10]. That inspires us to "borrow" excellent proposal from optimal concurrent open shop scheduling for coflow scheduling, as Guideline 1 says.

GUIDELINE 1. *The permutation order suggested by the optimal result of the corresponding concurrent open shop problem would derive a good schedule for the optimization of average CCT.*

3.4 Hardness

Recent literature [6, 10] has shown the NP-hardness of the minimization of average CCT by reducing it to the well-known concurrent open shop schedule problem. And [11, 24] have shown that minimizing the sum of weighted completion times in a concurrent open shop is strongly NP-hard and inapproximable within a factor strictly less than 6/5 if P≠NP.

4 SCHEDULING ALGORITHM DESIGN

Inspired by the findings of Section 3, we design effective and practical coflow schedule algorithms in this section.

We firstly introduce a 2-approximation schedule algorithm for offline coflow schedules based on the best known result of concurrent open shop [11, 25]. Then, as this 2-approximation offline algorithm works in a centralized fashion and performs complicated scheduling, we realistically reform it to an online algorithm that is much simpler and easier to operate in decentralized fashions. We use coflow traces generated with real-world parameters to evaluate the impacts of our simplification on effectiveness. Results indicate that our simplified approach, SOA-II, achieves a close performance with the original 2-approximation algorithm. Consequently, we choose to schedule coflows with SOA-II. The detail of how SOA-II drives our practical scheduling system, D-CAS, follows in the next section.

4.1 The 2-approximation Solution

The 2-approximation solution (or *2-approx* for short) is a permutation based algorithm [11]. Within $O(n(m+n))$ elementary operations, it outputs a schedule order of coflows, standing for their flow priorities that could achieve the 2-approximation average CCT minimization.

Algorithm 1 shows how *2-approx* schedules a set of coflows. It starts by finding the last coflow to complete, then the second to the last (i.e., the next-to-last), the third to the last, and so on (Line 7). At each turn, it first observes the port with the maximum load (Line 8) (either an ingress from a sender or an egress from a receiver), picks out the coflow with the minimum weight-to-processing time ratio on that port (i.e., $w_k/g_\mu^{(k)}$), and sets it as the last coflow to complete (Line 9)—I.e., this coflow will get the lowest priority. Then the algorithm adjusts the weights of other coflows (Line 11), subtracts this coflow's loads from the ports' loads (Line 12), and proceeds in determining the next-to-last coflow in a similar way (Line 7). In the optimization of average CCTs, all coflows have the same unit weight; accordingly, the adjusted weights are initialized as 1s at the beginning (Line 6).

By setting coflow priority values with the output π, one can achieve near-optimal offline coflow schedule in theory.

Algorithm 1 2-approximation Approach (2-approx for short)

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 schedule (i.e. priority) of coflows $\pi : \{1, \dots, n\} \mapsto N$.

1: $U \leftarrow \{1, 2, \dots, n\}$ ▷ unscheduled coflows
2: $P \leftarrow \{1, 2, \dots, 2m\}$ ▷ the index of port
3: $g_i^{(k)} \leftarrow \sum_{j\in M} d_{i,j}^{(k)}$ for all $k \in N$ and $i \in M$ ▷ Ingress
4: $g_{j+m}^{(k)} \leftarrow \sum_{i\in M} d_{i,j}^{(k)}$ for all $k \in N$ and $j \in M$ ▷ Egress
▷ $g_p^{(k)}$ denotes coflow k's aggr. load on port p.
5: $L_i \leftarrow \sum_{k\in N} g_i^{(k)}$ for all $i \in P$ ▷ total load of port i.
6: $w_k \leftarrow 1$ for all $k \in N$ ▷ initialize adjusted weights
7: **for** $z \leftarrow n, n-1, \dots, 1$ **do**
8: $\mu \leftarrow \arg\max_{i\in P} L_i$
▷ determine the "bottleneck" port for coflow $\pi(z)$
9: $\pi(z) \leftarrow \arg\min_{k\in U} w_k/g_\mu^{(k)}$ ▷ determine priority
10: $\theta \leftarrow w_{\pi(z)}/g_\mu^{(\pi(z))}$
11: $w_k \leftarrow w_k - \theta \cdot g_\mu^{(k)}$ for all $k \in U$ ▷ adjust weights
12: $L_i \leftarrow L_i - g_i^{(\pi(z))}$ for all $i \in P$ ▷ update port loads
13: $U \leftarrow U \setminus \{\pi(z)\}$ ▷ update unscheduled coflows
14: **end for**

4.2 From the 2-Approximation Approach to SOA-II

The *2-approx* is proven to be effective; however, it is impractical for coflow scheduling in DCNs due these reasons:

- *Offline v.s. Online: 2-approx* only makes offline schedules, while coflows occur dynamically in practice [5]–[8];
- *Centralized v.s. Decentralized: 2-approx* is a centralized solution performing schedules based on the global coflow

information; however, as Section 2.2 discusses, large scale datacenters prefer decentralized scheduling systems.

In this part, we realistically simplify *2-approx* to one that is much simpler and easier to work in decentralized fashions.

4.2.1 Simplify the Scheduling with Two Relaxations

As Algorithm 1 indicates, the key of *2-approx* is to set the lowest priority to the coflow with the minimum weight-to-processing time ratio on the port with the maximum load, and then handles the next-to-last coflow. In practice, coflows occurs online and the load of each port varies with time. On average, all ports would have the same load since today's datacenters generally assign jobs with load balancing [1, 7]. If so, the schedule process does not need to take the load diversity of ports into account; a straightforward relaxation of *2-approx* is to directly set the coflow with the maximum per-port load with the lowest priority at each round of the *for-loop* in Algorithm 1 (**1st relaxation**). Even so, in a decentralized manner, it is quite hard for a data receiver to figure out a coflow's loads on its port before that coflow completes. On the contrary, a data sender (i.e., the ingresses in the DCN fabric) generally knows a coflow's load on their ports, because it holds all the data to be transferred in advance[2]. Accordingly, we just consider the load information on senders[3] (**2nd relaxation**).

With these two relaxations[4], we reduce *2-approx* to one scheduling coflows based on their maximum loads on senders as Algorithm 2 shows. But yet, such a procedure is still centralized and offline, as it has to compute a global priority sequence for all coflows once a new coflow arrives (Line 3).

Algorithm 2 SOA-I: Simplified Offline Approach

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 schedule (i.e. priority) of coflows $\pi : \{1, \ldots, n\} \mapsto N$.

1: $g_i^{(k)} \leftarrow \sum_{j \in M} d_{i,j}^{(k)}$ for all $k \in N$ and $i \in M$ ▷ Ingress
2: $g_k \leftarrow \max_{i \in M} g_i^{(k)}$ for all $k \in N$
3: Sort $[g_1, g_2, \ldots, g_n]$ in non-decreasing order and store their indexes in π.

Furthermore, consider the case where switches support arbitrary priority values [16], then senders can just use the maximum load value of each coflow (i.e., $\max_i \sum_{j=1}^{m} d_{i,j}^{(k)}$ for coflow k) as its priority value when transmitting. This mechanism decouples the process of inter-coflow scheduling, so that each coflow can individually handle the schedule of its own flows. Even if switches only support a restricted number of priorities (e.g., hardware switches only has several priority queues), we can design schemes to map arbitrary priority values to the supported priority levels (we propose two mapping schemes and make evaluations in Section 6.3). By using a coflow's maximum load value as its flow priority, once a new coflow comes, data senders only need to recheck each coflow's remaining load on them, and set each coflow's priority with its maximum load value.

In our solution, senders periodically fetch the remaining size of each coflow and use the maximum value to update their priorities as Algorithm 3 shows. Ideally, SOA-II should be executed on every flow arrival or completion event.

2. In some cases like streaming flows, data senders also do not know the accurate information of a flow remaining size. As Section 6.3 will show, our final schedule system, D-CAS, can handle this effectively as well.

3. Note that, a host might act as a receiver of a coflow and a sender of another simultaneously. For each coflow, we just consider its own senders.

4. To illustrate how the two relaxations help, revisit the toy case of Fig. 1 as an example: With global coflow knowledge, the original *2-approx* knows that the maximum per-port loads of coflow A and B are $\max(1, 3, 4) = 4$ and $\max(2, 4, 6) = 6$, respectively. Following the complicated Algorithm 1, *2-approx* finally obtains the priority assignment $\pi(coflow\ A) = 1; \pi(coflow\ B) = 2$. Differently, with the relaxations, SOA-I only needs the load knowledge at ingresses; by directly sorting $\{\max(1, 3), \max(2, 4)\}$, it gets $\pi(coflow\ A) = 1; \pi(coflow\ B) = 2$ as well.

TABLE 1
Proportion of the effectiveness loss of SOA-I and SOA-II over *2-approx*.

Loss of effectiveness	$\leqslant .00$	$\leqslant .05$	$\leqslant .10$	$\leqslant .15$
with SOA-I (Algorithm 2)	.0	3.5%	37%	82.5%
with SOA-II (Algorithm 3)	.0	24.5%	87.5%	100%

Algorithm 3 SOA-II: Simplified Online Approach

Note: this procedure is recalled periodically
Input: number of *online* coflows n; number of hosts m; *remaining* flow demands $rem_{i,j}^{(k)} \in \mathbb{R}_{\geq 0}$ for all $k \in N$, $i \in M$ and $j \in M$
Output: priority for each coflow $\pi_p : \{1, \ldots, n\} \mapsto \mathbb{R}_{\geq 0}$.

1: $g_i^{(k)} \leftarrow \sum_{j \in M} rem_{i,j}^{(k)}$ for all $k \in N$ and $i \in M$ ▷ Ingress
2: $\pi_p[k] \leftarrow \max_{i \in M} g_i^{(k)}$ for all $k \in N$

4.3 How Far is SOA-II From the Optimal?

Obviously, with reasonable relaxations, we have simplified the *2-approx* to one that is online and easy to operate in decentralized manners. Then, a following doubt is *"how much does the simplification affect the schedule effectiveness?"*

To answer this, we investigate the completion times of a set of coflows under three scheduling schemes—the original *2-approx*, SOA-I, and SOA-II. We consider the schedule of 200 coflows served by a small-scale cluster consisting of 40 hosts. These hosts are connected with a non-blocking network fabric, where each port (both ingresses and egresses) has the capacity of 1 Gbps. Coflow traces are synthesized using the generator of Varys with the same real-world parameters detailed in Section 6. As for the schedule of SOA-II, we assume that the operation of priority update shown in Algorithm 3 is called every 0.01 second and the simulator runs at 0.01 second decision intervals.

Since coflows are generated randomly (refer to the generator's code [12]), we repeat the numerical simulation 200 times. Table 1 shows the proportions of $\frac{AvgCCT\ with\ SOA\text{-}I}{AvgCCT\ with\ 2\text{-}approx} - 1$ and $\frac{AvgCCT\ with\ SOA\text{-}II}{AvgCCT\ with\ 2\text{-}approx} - 1$, respectively. It indicates that SOA-II achieves better average CCTs than that of SOA-I. By denoting $\frac{AvgCCT\ with\ SOA\text{-}x}{AvgCCT\ with\ 2\text{-}approx} - 1$ to be the effectiveness loss of SOA-x over *2-approx*, the statistics also imply that the gap between SOA-II and *2-approx* is less than 10% in most cases, and might not exceed 15%.

As Fig. 3 shows, we further calculate the distributions of coflow completion times. The curves indicates that SOA-II achieves close effectiveness to *2-approx* while slightly prolonging completion times of small coflows.

In summary, with realistic assumptions, we have simplified the complex, offline, and centralized *2-approx* to the simple, online, and decentralized SOA-II, at a little cost of effectiveness, as Table 2 says. Accordingly, we choose to schedule coflows with

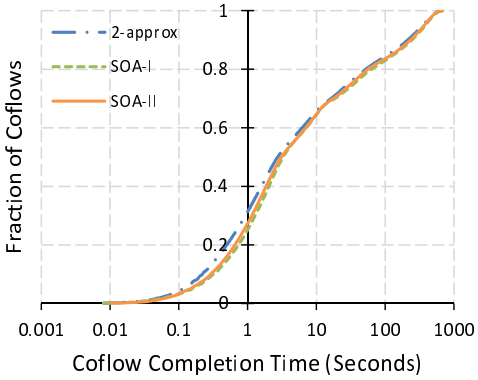


Fig. 3. CCT distributions for *2-approx*, SOA-I, and SOA-II. Note that the X-axes are in logarithmic scale.

SOA-II and the detail of how it drives practical coflow scheduling follows in next section.

TABLE 2
Comparison of the three approaches.

#Scheme	Sched-Mode	Work-Mode	Procedure	Performance
2-approx	Offline	Centralized	Complex	High
SOA-I	Offline	Centralized	Simple	High
SOA-II	**Online**	**Decentralized**	**Simple**	**High**

5 Towards Practical Scheduler

In this section, we present the detail of how we design D-CAS, a practical decentralized coflow scheduling system. Basically, driven by SOA-II, D-CAS natively performs *preemptive* and *work-conserving* schedules. Nevertheless, in practical terms, D-CAS must also be able to handle small coflows and avoid perpetual starvations. Accordingly, we devise approaches to make D-CAS support small coflows and *starvation-free*.

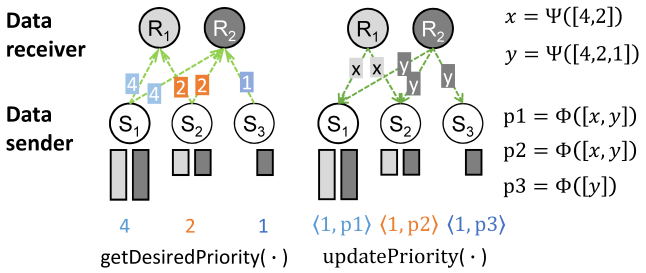
To clearly present D-CAS, we first give some key definitions D-CAS uses in Section 5.1, then introduce the core designs in Section 5.2, explain the algorithm details that senders and receivers perform in Section 5.3, and finally discuss the overheads and scalability of D-CAS in Section 5.4.

5.1 Key Definitions

Coflow (and its property): A coflow is a set of flows with associated application-level semantics and only completing them all could push forward the application process. We say a coflow is the *parent coflow* of all its flows. The *length* of a coflow is defined as the volume of its largest flow, while the *width* is the number of flows in it. By summing up the volume of all its flows, we get the *size* of this coflow.

Subcoflow: A subcoflow S consists of all the flows in a coflow (say C) that stem from the same *source host*. For simplicity, we call C as the *parent coflow* of subcoflow S. Thus, a subcoflow can be identified by the tuple of its parent coflow and the source host. Similarly, the *size* of a subcoflow is the total volume of all its flows.

Priority: In D-CAS, each source host (i.e., data sender) dynamically sets priorities to packets to realize the coflow scheduling and small coflows always preempt large coflows. For convenience, we design the priority number as a tuple $\langle T, P\rangle$, and call the two items, T and P, as *main priority* and *secondary priority*,



(a) Announce *desired-priority* (b) Get priority feedbacks then update

Fig. 4. Framework overview: how a coflow's subcoflows (S_1, S_2, S_3) implement SOA-II using a negotiation mechanism with the help of its own data receivers (R_1, R_2).

respectively. We say, $\langle T_1, P_1\rangle$ is a higher priority than $\langle T_2, P_2\rangle$, iff $T_1 < T_2$, or $T_1 = T_2$ and $P_1 < P_2$. In Section 6.3, we will discuss how to encode such tuple-based priorities to the practical priorities supported by today's commercial switches.

5.2 Core Design Overview

5.2.1 Implementing Decentralized SOA-II

Recall that, in SOA-II, each coflow uses its maximum remaining load on senders as its flow priority. Consequently, for each coflow, D-CAS needs to 1) find out its maximum remaining subcoflow size value, and 2) notify this value to all its data senders. Motived by the specific communication patterns of coflow [8], D-CAS implement this with a simple decentralized negotiation mechanism as Fig. 4 shows[5]. Roughly, the negotiation mechanism works as follows:

1) For each coflow, each of its senders (i.e., subcoflows) announces a *desired-priority* presenting the remaining subcoflow size on this hosts to all its data receivers. See the example shown in Fig. 4(a), the *desired-priorities* of the three subcoflows are 4, 2, and 1, respectively, based on their remaining sizes.
2) On getting a *desired-priority* message, the data receiver replies a feedback that is derived by a predefined function $\Psi(\cdot)$ and received messages. As shown in Fig. 4(b), R_1 and R_2 get feedbacks to the senders with $x = \Psi([4, 2])$ and $y = \Psi([4, 2, 1])$, respectively.
3) At last, when the sender collects feedbacks from data receivers, it gets the target priority by computing $\Phi([x, y])$.

Obviously, by letting both $\Psi(\cdot)$ and $\Phi(\cdot)$ be $\max(\cdot)$, D-CAS implements SOA-II in a decentralized fashion.

5.2.2 Handling Small Coflows

In practice, the feedback mechanism of D-CAS needs a few RTTs to take effect, while small coflows might have completed within such a delay. Therefore, when a subcoflow is smaller than a predefined threshold, D-CAS does not schedule it according to SOA-II any more. Prior work of Baraat [7] has shown that FIFO policy achieves good performances for minimizing the average CCT when the flow size (i.e., coflow *length*) is distributed in a small range. Accordingly, D-CAS schedules small coflows following FIFO like Baraat [7].

5. In the case where there is no common data receivers between multiple subcoflows, D-CAS randomly chooses one subcoflow's host as a *virtual* receiver for negotiating.

5.2.3 Avoiding Perpetual Starvation

If small coflows are arriving at a high rate, the preemption of SOA-II would let large coflows cannot make any progress. To avoid perpetual starvation, D-CAS introduces a simple adjustment to the schedule: Once a subcoflow has not got any service during the last T, it will obtain a higher priority in the next schedule intervals δ (see Section 5.3 for details). By fixing tunable parameters T and δ ($T \gg \delta$), D-CAS is able to trade the average completion times for the impacts of starvation.

5.3 D-CAS Details

This subsection explains the operations that each sender and receiver perform in detail. At the end, we also discuss how to upgrade D-CAS to perform other schedule policies, e.g., per-flow fair sharing, by simply using different functions $\Psi(\cdot)$, $\Phi(\cdot)$, and $getDesiredPriority(\cdot)$ at end hosts.

5.3.1 Sender

In D-CAS, each sender periodically (every δ-interval) detects subcoflows' remaining sizes and updates data transfers' priorities, as Algorithm 4 shows. Consider a subcoflow S whose parent coflow is c; suppose cp is the Coflow-ID of c representing its arrival order (same to the Task-ID in [7]). If S's remaining size (rem in Line 4) on the sender is less than the predefined $thresholdVolume$ (e.g., the bandwidth delay production–BDP), its flows will be sent with priority value $\langle 0, cp\rangle$, pursuing the FIFO scheduling (Line 7). Otherwise, S is identified as a large subcoflow; its main priority of its flows should be set to 1. Then, if this subcoflow has not received any service during the last T-interval ($T \gg \delta$), its secondary priority will be set to be 0 to prevent starvation (Line 8). In this case, all the unserved subcoflows share the network with per-flow fairness mechanism. If not, the negotiation mechanism of SOA-II is launched for the secondary priority. To this end, each sender announces its desired priority to the receivers and waits several RTTs (Line 11). Regularly, the sender uses the received feedbacks to calculate the secondary priority with $\Psi(\cdot)$ (Line 15). However, if all the feedbacks get lost due to network congestion or failure, the sender just sets the secondary priority to its desired priority (Line 17).

Algorithm 4 Coflow-aware Flow Scheduling in Sender

```
1:  procedure SCHEDULE(Subcoflows 𝕊)                    ▷ recall every-δ
2:      for all S ∈ 𝕊 do
3:          c ← S.coflowID
4:          rem ← getLocalRemSize(S)
5:          d ← getDesiredPriority(S)
6:          if rem < thresholdVolume then
7:              p ← ⟨0, getCoflowArriveOrder(c)⟩        ▷ FIFO
8:          else if S.waitTime() > T then
9:              p ← ⟨1, 0⟩                              ▷ Avoid Starvation
10:         else
11:             Announce d to all S's data receivers.
12:             Wait two RTTs.
13:             M ← getFeedbackPrioritySet(c)
14:             if M is not empty then
15:                 p ← ⟨1, Φ(M)⟩
16:             else
17:                 p ← ⟨1, d⟩
18:             end if
19:         end if
20:         Update the priority of S's data transfers to p.
21:     end for
22: end procedure
```

Algorithm 5 Reply Feedbacks in Receiver

```
1: procedure REPLY(msg m)                  ▷ message from the sender
       ▷ m stores the subcoflow's desired priority and information
2:     c ← m.coflowID
3:     B ← getCachedMsgs(c)
4:     Update B using m and remove expired messages.
5:     p ← Ψ(B.desiredPriorityValues())
6:     Send the feedback-priority of c (p) to m.src.
7: end procedure
```

5.3.2 Receiver

In D-CAS, each receiver maintains a cache for the latest desired-priority of each subcoflow. When receiving a desired-priority, it operations as Algorithm 5 illustrates: (i) get (Line 3) and update the cache (Line 4, messages older than $expiredTime$ will be removed); (ii) derive the feedback using its cached desired-priority information from the same coflow (Line 5), and (iii) send it back (Line 6).

5.3.3 About $\Psi(\cdot)$, $\Phi(\cdot)$, and $getDesiredPriority(\cdot)$

To follow SOA-II, both $\Psi(\cdot)$ and $\Phi(\cdot)$ are set as $\max(\cdot)$. As for $getDesiredPriority(\cdot)$, if all data senders have the same upstream capacity, we can set $getDesiredPriority(S)$ to simply return the remaining size of S. However, the homogeneity of hosts do not always exist in practice [26]. For example, in virtualized data centers, such as Amazon's Elastic Compute Cloud (EC2), different types of instances (i.e., virtual machines) might have various bandwidth capacities. To eliminate the ill effects of bandwidth heterogeneity, D-CAS heuristically designs $\frac{getLocalRemSize(\cdot)}{NICLineRate}$ as the return value, where $NICLineRate$ denotes the capacity of the host's upstream link.

Additionally, it is worth noting that, one can easily upgrade D-CAS to other existing system by adopting different $\Psi(\cdot)$, $\Phi(\cdot)$, and $getDesiredPriority(\cdot)$. For example, if $getDesiredPriority(\cdot) \equiv 1$ (while $\Psi(\cdot)$ and $\Phi(\cdot)$ are $max(\cdot)$), D-CAS turns into a per-flow fair-sharing system. As well, if $getDesiredPriority(\cdot)$ returns the interval to the coflow's deadline, D-CAS then performs deadline-aware coflow scheduling.

5.4 Complexity and Overheads

In D-CAS, the operations that senders as well as receivers perform are quite simple; the complexities are linear with the amount of parallel flows the host is serving. In practice, to achieve fast job computations while making efficient resource utilizations, the maximum number of simultaneous tasks that a host is capable of running in parallel is tuned according to its hardware capabilities such as CPU, memory, Disk and network I/O [27, 28], and the default setting in Hadoop is four [26]. That is to say, the number of concurrent coflows that a host serves would be a small constant determined by its hardware capability. Moreover, the amount of concurrent flows that a coflow node involves is generally defined by the job's size and task splitting scheme, which are independent from the total number of coflows as well as the data center scales [26]–[28]. For instance, both Facebook and

Microsoft have reported that work nodes in their production data centers involve tens of concurrent flows on average [19, 29]. Thus, the computation overheads of performing coflow scheduling for each host in D-CAS would be $O(1)$. As for the traffic overheads introduced by decentralized negotiations, two hosts periodically exchange coflows' desired-priority information only when they have data to transmit. Compared with the size of the coflow's payloads, such a priority information only occupies a few bytes and is very trivial in bandwidth usage.

Thus, D-CAS is efficient, lightweight, and highly scalable.

6 SIMULATION

We have implemented a Python-based simulator to evaluate the performance of D-CAS by comparing it with Varys, Baraat, and the per-flow fairness mechanism. Our simulator shares the same design with that of Varys [12], and performs a detailed replay of the similar coflow traces as well. Extensive simulation results demonstrate: 1) D-CAS improves the average CCT about 2–3$\times$ over per-flow fairness; 2) D-CAS achieves a performance very close to Varys—the performance gap between D-CAS and Varys is less than 15%; 3) D-CAS outperforms Baraat by about $4\times$ when coflows are heterogeneous, and about $1.4\times$ when coflows are homogeneous; 4) even when switches have limited priority queues and senders are agnostic of remaining sizes, D-CAS still outperforms the per-flow fairness more than $2.5\times$.

6.1 Methodology

6.1.1 Setup

We employ the generator provided by Varys [6] (i.e., the CustomTraceProduce in [12]) as well as the corresponding real-world parameters obtained from Facebook's data centers [6] to synthesize coflow traces. Respecting to the statistics in Varys, all the coflows are categorized to be four types in terms of their length (the size of the largest flow in bytes for a coflow) and width (the number of parallel flows in a coflow): *Narrow&Short*, *Narrow&Long*, *Wide&Short*, and *Wide&Long*, where a coflow is considered to be *short* if its length is less than 10 MB, and *narrow* if it involves at most 50 flows.

By defaults, coflows are assumed to arrive in a *Poisson* process with parameter λ, and each above type of coflows constitutes 52%, 16%, 15%, 17% of the coflow stream, respectively (according to the statistics reported by [6]). Without declaration, simulation results are based on 400 coflows served by an 80-hosts cluster with following settings:

1). All the flows in a coflow arrive at the same time [5, 6], and the upper bound of flow volume (i.e., the upper bound of coflow length) is 500 MB.
2). The network model (i.e., the entire datacenter fabric) is abstracted out as one non-blocking switch [6, 16] interconnecting all the machines, and we only focus on its ingress and egress ports (e.g., machine NICs); both the ingress and egress are set with the capacity of 1 Gbps.
3). The arrival rate of coflow is set to $\lambda = \frac{NetworkThroughput}{MeanOfCoflowSize}$; this is to make the network neither be overloaded nor underloaded since the network's average load (i.e., $E(NetworkLoad)$[6]) is 1 within such a setting.
4). As for D-CAS, we use T=1 s, δ=0.1 s, $expiredTime$=2 s, and $thresholdVolume$=1 MB.
5). As for Varys, since it only reschedules flows when a coflow arrives and completes, it cannot fully utilize the network resource [6]. To maximize its potentiality regardless of control overheads, we let it reschedule flows on every flow arrival and completion by defaults. As a supplement, we also implement the original task-level event driven version, which is named as Varys* in Section 6.3.3.
6). As for Baraat, we set its threshold of large-coflow identifying to be 80th percentile of the coflow size. In fact, we repeat the tests multi-times and observe that the results are insensitive to the threshold setting in our simulation.

6. Note that, $E(NetworkLoad) = \frac{\lambda \times MeanOfCoflowSize}{NetworkThroughtput}$

6.1.2 Metrics

We use the improvement in average CCT as our primary metric and the improvement factor is defined as

$$\text{Factor of Improvement} = \frac{\text{Current Average CCT}}{\text{Modified Average CCT}} \quad (5)$$

In all tests, we use the duration of per-flow fair sharing as a baseline (i.e., it is the current duration by defaults), since per-flow fair sharing represents the operation of current transport protocols (e.g., TCP, DCTCP) in DCNs. It should be noted that, as coflow traces are generated randomly, for each set of test parameters, we generate 20 test instances. We observe that the absolute value of average CCT varies with different input traces. However, the conclusions they imply are consistent. Accordingly, each improvement factor shown in the paper, except those shown in Fig. 5(a), is the average value of 20 trials.

6.2 Detailed Results on Various Inputs

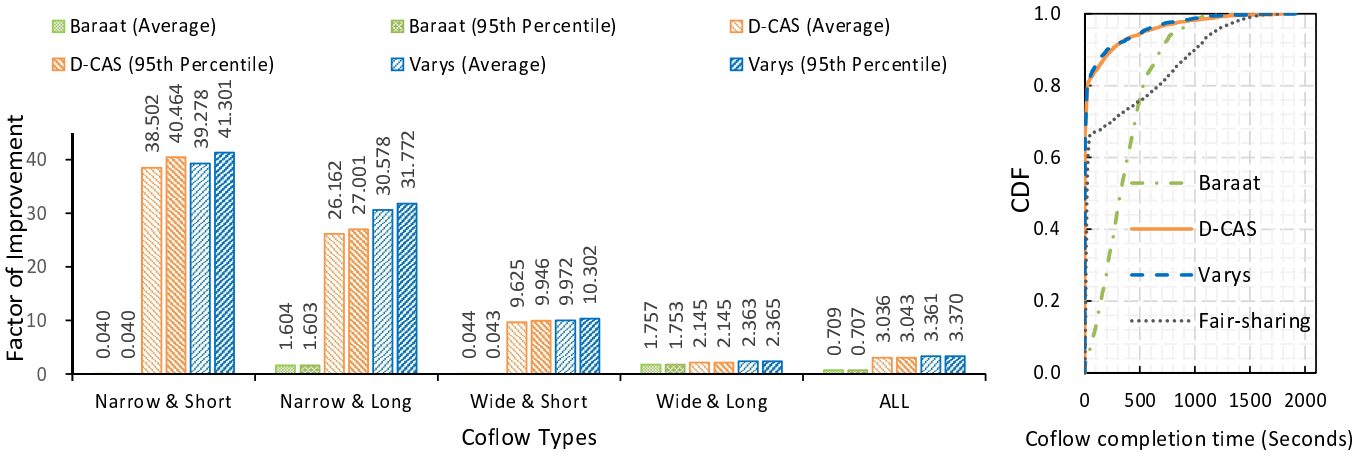
We firstly investigate the performance of D-CAS under different *network load*s, *cluster/network scale*s and *coflow type*s.

6.2.1 The Performance of D-CAS

Fig. 5(a) shows the detailed improvements of Baraat, D-CAS, and Varys w.r.t. per-flow fair-sharing, respectively. It implies that both D-CAS and Varys greatly reduce the average and 95th percentile CCT across all coflow types. For example, the improvement factors of D-CAS on average CCTs are 38.502 (Narrow&Short), 26.162 (Narrow&Long), 9.625 (Wide&Short), 2.145 (Wide&Long), and 3.036 (ALL), while that of Varys are 39.278 (Narrow&Short), 30.578 (Narrow&Long), 9.972 (Wide&Short), 2.363 (Wide&Long), and 3.361 (ALL). We note that, the performances of D-CAS and Varys are very close, and the gap between their improvements is less than 10%. This is due to the fact that the negotiation mechanism in D-CAS helps each coflow using its maximum-subcoflow-remaining-size[7] as its priority. Thus the coflow with the smaller remaining size on all its senders would be more likely to use the network. This makes the scheduling scheme in D-CAS approach the SOA-II policy.

The results also imply the bad performance of Baraat on scheduling heterogeneous coflows. Baraat only speeds up the completions of long (both *Narrow&Long* and *Wide&Long*) coflows a little, while degrading the completions of short coflows. It is inferior to the baseline on the overall average CCT. This is because, the FIFO policy Baraat using leads to serious head-of-line blocking problem and its limited multiplexing (LM) scheme

7. When all hosts have the same $NICLineRate$s, $getDesiredPriority(\cdot)$ is equivalent to $getLocalRemSize(\cdot)$.



(a) Improvements w.r.t. the default per-flow fairness mechanism in the average and 95th percentile.

(b) CCT distributions.

Fig. 5. Detailed performance of Baraat, D-CAS, Varys and the default per-flow max-min fairness mechanism on coflow traces generated with real-world parameters [6].

cannot solve this problem absolutely, especially when facing heterogeneous coflows.

Fig. 5(b) shows the CDFs of CCT under different scheduling schemes. It implies that, though the preemptive schemes like D-CAS and Varys reduce the average CCTs, they prolong the CCT tails. That is to say, in heavily loaded data centers, these applications involving large coflows would experience longer durations for data transmissions (compared with default per-flow fairness scheme). This is trivial in cooperative environments such as private datacenters. However, in multi-tenant environments like public clouds, applications from different tenants can be non-cooperative and fair sharing as well as performance isolation is preferred. Then, non-preemptive schemes like Baraat [7] and HUG [30] are more suitable; they cut the long tails of CCT with the cost of enlarging average CCTs. As further work, D-CAS would be extended to support average CCT optimizations with performance isolation guarantees.

6.2.2 Impact of Network Load

To study the impact of network load, we vary the arrival rate of coflow and investigate the performance of different scheduling schemes. Note that, as each trial only involves 400 coflows, all these coflows will finally get served even if their (peak) arrival rate is larger than 1. Fig. 6(a) shows the simulation results. It indicates: 1) regardless of the network load, the improvement factor of D-CAS is always close to that of Varys and outperforms Baraat a lot; 2) the improvement factors of D-CAS and Varys slightly increase with the network load, while that of Baraat is stable. This is because, the heavier the network load is, the larger optimization space there is for preemptive scheduling schemes.

6.2.3 Impact of Cluster/Network Scale

To explore the impact of cluster/network scale, we investigate their improvement factors under different sizes of clusters. The result in Fig. 6(b) shows both the improvement factors of D-CAS and Varys grow with the network size (coincides with the simulation results in Varys [6]), while that of Baraat is relatively stable. Such a phenomenon is mainly caused by the setting that we always adjust λ to make $E(NetworkLoad) = 1$ in simulations. Under such a setting, $NetworkThroughtput$ would grow linearly with the cluster size and there will be more concurrent coflows in a larger cluster. Accordingly, preemptive coflow-aware scheduling methods like D-CAS and Varys get more optimization spaces than non-preemptive methods like per-flow fairness and Baraat.

6.2.4 Impact of Coflow Type

We now study the impact of coflow types. To highlight the comparison, we investigate the performance of different scheduling schemes when there is only one type of coflows in the network. In each simulation, we also hold $E(NetworkLoad) = 1$. From the results shown in Fig. 6(c), we make two important observations. First, Baraat outperforms per-flow fairness in all the four cases. This is because the FIFO scheduling policy gets good performances when coflows are homogeneous. Second, all three coflow-aware scheduling schemes perform better when the coflow is longer and wider, and their increments of improvement factors are more sensitive to coflow *width* than coflow *length*. This is due to the fact that, the larger *width* coflows have, the more likely they may interleave with each other, in which condition, the performance of coflow-agnostic per-flow fair sharing mechanism falls increasingly further behind as it is coflow-agnostic.

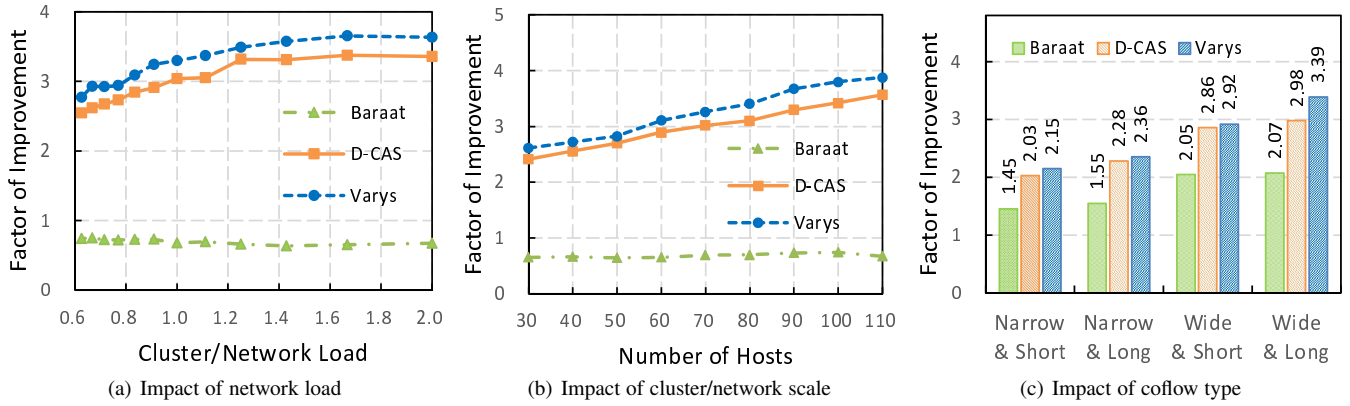
Besides, we also observe that: 1) D-CAS always outperforms Baraat, about $4\times$ when coflows are heterogeneous, and about $1.4\times$ when coflows are homogeneous; 2) D-CAS achieves a performance very close to that of Varys, their performance gaps are always less than 15% in simulations.

6.3 Performances under Real-world Settings

The above subsection has studied the performance of D-CAS under ideal conditions, where data senders are able to get the real-time remaining size of each subcoflow precisely, switches support arbitrary priority values like pfabric [16] and Baraat [7], and all children flows of a coflow appear simultaneously. Unfortunately, these assumptions are usually unsatisfied in practice. In this subsection, we investigate the performance of D-CAS when these assumptions are removed.

6.3.1 Errors in $getLocalRemSize()$

In real data centers, flows (or transfers) are transmitted online; it is difficult to estimate the exact remaining size of a flow. Therefore,



(a) Impact of network load (b) Impact of cluster/network scale (c) Impact of coflow type

Fig. 6. Improvements in the average CCT under different settings. Note that all the factor of improvements use per-flow fairness as baseline.

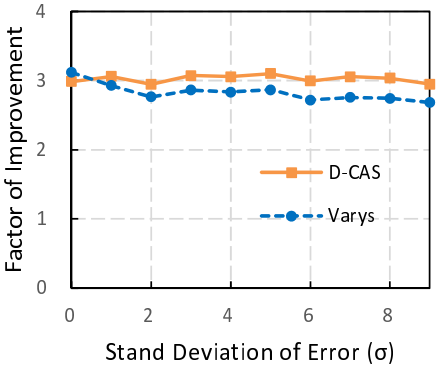


Fig. 7. Impact of errors in $getLocalRemSize()$.

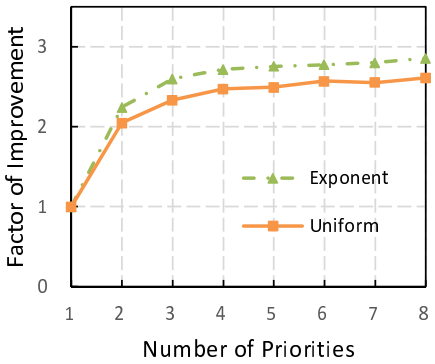


Fig. 8. Impact of limited priority queues.

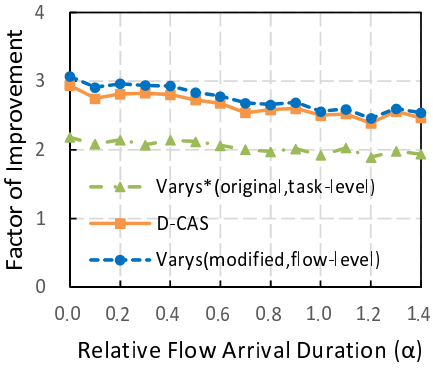


Fig. 9. Impact of flow arrival interval.

we study how the error in the remaining size estimation would impact D-CAS's effectiveness. To this end, we randomly add estimation errors, following normal distributions with parameter $\mathcal{N}(0, \sigma)$, into the estimated remaining size. For example, if a subcoflow's remaining size is rem, its estimated remaining size would be $rem \times \max(1 + x, 0.1)$, where $X \sim \mathcal{N}(0, \sigma)$. Fig. 7 shows how the system performance changes with the variance of errors (i.e., σ). From this figure, we find that D-CAS's performance is not sensitive to errors in the remaining size estimation. In contrast, when estimation errors exist, the performance of Varys will reduce, which makes it underperforming D-CAS. Roughly, the larger estimation error is, the larger performance gap there will be between D-CAS and Varys. This means that the performance of Varys heavily relays on the accuracy of remaining size estimation. As errors are inevitable, D-CAS is a better choice in practice.

In addition, we also investigate the case where each sender is totally agnostic of the remaining size of subcoflow. By simply regarding the original size and the sent size as a subcoflow's remaining size, D-CAS would still get $2.85\times$ and $2.39\times$ improvements, respectively.

6.3.2 Limited Priority Number

In D-CAS, we assume that switches support arbitrary priorities in common with others [7, 16]; however, this is not always the case in real systems. For instance, today's commodity switches typically support 4-8 queues per port [16]. In these cases, D-CAS has to assign multiple coflows with the same priority during the schedule. Suppose the remaining subcoflow size ranges in $[0, rem^{\dagger}]$; then, there exists multiple schemes that divide $[0, rem^{\dagger}]$ into K intervals, where K is the number of priorities supported by switches. Fig. 8 shows how D-CAS's performance changes with the available priority number in the system, under two simple priority assignment schemes, named "Exponent" and "Uniform", respectively[8]. Under "Exponent" scheme, the length of each interval is $\tau, 2\tau, \ldots, 2^{K-1}\tau$, respectively, where $\tau = \frac{rem^{\dagger}}{2^K - 1}$; while all the intervals have the same size (i.e., $\frac{rem^{\dagger}}{K}$) under the "Uniform" scheme. With priority converting, the subcoflows whose sizes are in the i-th interval are assigned with the priority value i. From Fig. 8, we observe that more priorities can bring more performance improvements to the system, and "Exponent" priority assignment scheme outperforms "Uniform" scheme.

The results also imply that, even with a small set of priority numbers (e.g., 4) and a simple interval division scheme (e.g., "Exponent"), D-CAS still outperforms per-flow fairness more than $2.5\times$. Moreover, we further test the cases of each sender simply regarding a subcoflow's original size or sent size as its remaining size, the improvement factors are larger than 2.5 as well. Therefore, D-CAS is efficient and readily-deployable by using existing commodity switches.

6.3.3 Flow Arrival Interval

Usually, flows belonging to a coflow do not arrive at the same time; this may impact the performances of schedulers. Fig. 9 shows how the performances of D-CAS, Varys*, and Varys, change

8. In this case, all small subcoflows, as well as the subcoflows falling into the first interval, will be transmitted with the same highest priority, 1.

with the flow arrival interval in a coflow. For each coflow, we assume that its flow arrival times have an exponential distribution with parameter $\frac{1}{\mu_f}$, where μ_f is the mean duration between the first flow and other flows. Recall that coflows arrive in a Poisson process with parameter λ (see Section 6.1.1), we let $\mu_f = \alpha\frac{1}{\lambda}$, $\alpha \in \mathbb{R}_{\geq 0}$ in our tests. We call α as the *relative flow arrival duration*, as it stands for how long a flow will appear after its parent coflow arrivals. Obviously, the larger α is, the longer duration should a coflow's flow arrivals will take. The difference between Varys* and Varys is that Varys performs reschedules whenever a flow arrives or completes, while Varys* performs only on coflow arrivals and completions, which is the setting of the original Varys [6].

Fig. 9 implies two important observations. First, the performances of Varys (both Varys and Varys*) and D-CAS are decreasing with the increase of flow arrival durations. This is due to the online nature of the experiments: when coflows have long flow arrival durations, a large coflow will be regarded as a smaller coflow and preempt the network before its large children flow appears. As a result, the effects of coflow-aware schedule will decrease with the duration growing. Second, D-CAS gets a performance close to Varys, while much better than that of Varys*. Varys* underperforms because it is not work-conserving at every point in time [6]; on the contrary, both D-CAS and our modified "flow-level" Varys always perform work-conserving schedules. However, in large-scale data centers, as Section 6.4 will show, it is impractical for a centralized scheduling system to trigger reschedules at every flow arrival and completion event like our simulator performs. Thus, D-CAS will be more effective.

6.4 About Scalability

At last, we analyze the scalability of D-CAS and Varys*/Varys as Table 3 illustrates. Basically the result also suits for the comparisons between other decentralized (e.g., Baraat [7]) and centralized (e.g., Rapier [9]) solutions.

We do not list the modified version of Varys (referred as Varys) in Table 3 because it only differs from the original one (referred as Varys*) on schedule frequency—The modified Varys needs a reschedule upon each flow arrival or departure. In the following, without specific declarations, the conclusion of Varys is true for both versions. Recall that, once a rescheduling is needed, the central Varys master first computes a global rate allocation scheme for *all* flows based on the current knowledge of *all* coflows such as the number of flows, their sizes, and endpoints, then modifies the involved senders' traffic shapers to enforce the scheme. Nowadays, production data centers have hundreds of thousands of hosts and millions of concurrent flows and the scale is continuing to grow [17, 29]. It is hard for Varys to handle the case since its single master has to compute and assign bandwidth allocations for all concurrent flows. Even worse, such operations have to be executed on every coflow (or worse, on every flow, in the case of modified version) arrival and completion event. As the event intervals are likely in the magnitudes of tens of microseconds to tens of milliseconds [29], the control overhead is non-trivial in real systems. Besides, as a centralized system, Varys also suffers from the problem of single point of failure. Accordingly, Varys (as well as other centralized approaches like Rapier [9]) is with poor scalability.

In contrast, D-CAS adopts decentralized designs. With a tunable interval, each sender computes new priorities only for flows from it based its local coflow knowledge and data receivers' feedbacks. As Section 5.4 has discussed, D-CAS is by design robust, lightweight, and highly scalable.

7 Related Work

Our work is to optimize data transfers in DCNs. There is a large body of prior work; we broadly categorize them into four classes.

Routing in DCNs Hedera [31] and MicroTE [32] employ the partial predictability of data center traffic to optimize flow routing in a central fashion. Differently, CONGA [33] switches the flows in the flowlet level to pursue a better load balance. By using feedback mechanisms to exchange congestion information between remote switches, CONGA achieves network-wide load balancing in a distributed manner. As these literatures focus on how to fully utilize the network resource and do not take semantics among flows into account, they are not optimal in terms of the average CCT.

Per-flow Prioritization Many literatures like pFabric [16] and PDQ [15] focus on scheduling flows to minimize the average flow completion time (FCT), or to reduce the number of flows that miss their deadlines. As Section 2.1 shows, these flow level mechanisms lose some opportunities for minimizing average CCT since they are coflow-agnostic. However, methods for minimizing average FCTs and for minimizing average CCTs share the basic idea in scheduling: One is pursuing the smallest-flow-first principle, the other is pursuing the smallest-subcoflow-first principle.

Coflow Scheduling The problem of coflow scheduling is quite similar to the concurrent open shop problem [11, 24, 25], which is proven to be strongly NP-hard [11]. This connection has been reported by Yuan Zhong *et al.* [6, 10] recently and they developed a 64/3-approximation heuristics based on LP-relaxations. Here, we analyzed their connections on minimizing the average completion times. Based on this, we borrow schedule algorithm designs from [11], and further develop D-CAS (How D-CAS works was first sketched in an earlier paper [34].).

Orchestra [5], Varys [6], and Rapier [9] are three centralized coflow schedulers. Due to their centralized property, they have scalability problem to be deployed in large scale data centers. On the other hand, due to the control delays in centralized system, they are less useful for small coflows which complete their transfers in a few RTTs [6, 7] (Rapier simply treats small coflows as background traffic without performing coflow-aware schedules.). As far as we know, Baraat [7] is the only decentralized coflow-aware scheduling system in DCNs. However, it suffers from the head-of-line blocking problem and may result in a worse performance than the per-flow fair sharing mechanism.

Other approaches Basically, above approaches focus on accelerating existing transfers whose endpoints are fixed and predetermined by the applications. We also note that, there are numbers of proposals optimizing data transfers from other dimensions. For example, Sinbad [13] and KMN [35] seek to reduce cross-rack traffic and congested links by choosing endpoints for replica placement, or by co-locating tasks with their input data. Aggregator [36] tries to reduce the sizes as well as amounts of transfers by aggregating them on the fly. These solutions are orthogonal to flow scheduling approaches and seem to be useful for data-intensive applications such as Big Data analytics [37].

TABLE 3
Comparison of scalability

	Required knowledge	Schedule/control overhead	Schedule frequency	Reliability
D-CAS (decentralized)	Real-time, local	- (each sender manages its own flows)	Every δ (tunable)	-
Varys* (centralized)	Real-time, global	High (sync. computed rates to all senders)	On coflow arrival/departure	Single point of failure

8 Conclusion and Further Work

This article showed that minimizing the average CCT for a set of coflows is bounded by the results of minimizing the total job complete time of concurrent open shop. Based on this, it borrowed the 2-approximation from the schedule of concurrent open shop for offline coflow scheduling and further design a practical decentralized online coflow scheduling system, D-CAS. Numerical simulation results demonstrated that D-CAS is efficient and readily-deployable. It achieved a performance close to the state-of-the-art centralized scheduling method, Varys, and significantly outperformed the state-of-the-art decentralized scheduling system, Baraat. Currently, D-CAS focuses on optimizing the average CCTs for independent coflows. In our further work, we plan to extend D-CAS to handle coflow dependencies and support fairness-aware coflow scheduling.

Acknowledgments

The authors would like to thank the reviewers and editors for their helpful feedback. 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, and the Ministry of Education - China Mobile Research Fund under Grant No. MCM20130131. H. Yu is the corresponding author.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, Mar. 2007.
[3] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: A Universal Execution Engine for Distributed Data-flow Computing," in *NSDI*, 2011, pp. 113–126.
[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.
[5] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra." in *SIGCOMM*, 2011, pp. 98–109.
[6] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient Coflow Scheduling with Varys," in *SIGCOMM*, Aug. 2014, pp. 443–454.
[7] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized Task-aware Scheduling for Data Center Networks," in *SIGCOMM*, 2014, pp. 431–442.
[8] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. 11th ACM HotNets*, 2012, pp. 31–36.
[9] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks," in *INFOCOM*, April 2015, pp. 424–432.
[10] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proc. 27th ACM SPAA*, 2015, pp. 294–303.
[11] M. Mastrolilli and M. Q. *et al.*, "Minimizing the sum of weighted completion times in a concurrent open shop," *Operations Research Letters*, vol. 38, no. 5, pp. 390 – 395, 2010.
[12] M. Chowdhury, "Flow-level simulator for coflow scheduling used in varys," https://github.com/coflow/coflowsim.
[13] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *SIGCOMM*, 2013, pp. 231–242.
[14] D. Nace and M. Pioro, "Max-min fairness and its applications to routing and load-balancing in communication networks: a tutorial," *IEEE Communications Surveys Tutorials*, vol. 10, no. 4, pp. 5–17, Fourth 2008.
[15] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM*, Aug. 2012, pp. 127–138.
[16] 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.
[17] 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.
[18] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: Sharing the network in cloud computing," in *SIGCOMM*, 2012, pp. 187–198.
[19] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *SIGCOMM*, 2010, pp. 63–74.
[20] M. Alizadeh and A. Javanmard *et al.*, "Analysis of DCTCP: Stability, Convergence, and Fairness," in *SIGMETRICS*, 2011, pp. 73–84.
[21] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti, "High speed networks need proactive congestion control," in *Proc. 14th ACM HotNets*, 2015, pp. 14:1–14:7.
[22] H. Ballani and P. Costa *et al.*, "Towards Predictable Datacenter Networks," in *SIGCOMM*, Aug. 2011, pp. 242–253.
[23] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008, pp. 63–74.
[24] 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.
[25] A. Kumar, R. Manokaran, M. Tulsiani, and N. K. Vishnoi, "On LP-based Approximability for Strict CSPs," in *Proc. 22nd ACM-SIAM SODA*, 2011, pp. 1560–1573.
[26] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI*, 2008, pp. 29–42.
[27] E. Sammer, *Hadoop Operations*, 1st ed. O'Reilly Media, Inc., 2012.
[28] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analytics*, 1st ed. O'Reilly Media, Inc., 2015.
[29] A. Roy and H. Zeng *et al.*, "Inside the Social Network's (Datacenter) Network," in *SIGCOMM*, 2015, pp. 123–137.
[30] M. Chowdhury, Z. Liu, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *NSDI*, 2016.
[31] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *NSDI*, 2010, pp. 281–296.
[32] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proc. 7th CoNEXT*, 2011, pp. 8:1–8:12.
[33] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *SIGCOMM*, 2014, pp. 503–514.
[34] S. Luo, H. Yu, Y. Zhao, B. Wu, S. Wang, and L. Li, "Minimizing Average Coflow Completion Time with Decentralized Scheduling," in *IEEE International Conference on Communications (ICC)*, June 2015, pp. 307–312.
[35] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *OSDI*, Oct. 2014, pp. 301–316.
[36] H. Ke, P. Li, S. Guo, and I. Stojmenovic, "Aggregation on the fly: reducing traffic for big data in the cloud," *IEEE Network*, vol. 29, no. 5, pp. 17–23, September 2015.
[37] D. Zeng, L. Gu, and C. Guo, *Cloud Networking for Big Data*. Springer International Publishing, 2015.