Efficient Coflow Scheduling of Multi-stage Jobs with Isolation Guarantee

Zifan Liu, Haipeng Dai, Bingchuan Tian, Wajid Rafique, and Wanchun Dou

Abstract—Coflow scheduling is critical for data-parallel computing performance in datacenters. Performance and isolation guarantee have become two major objectives for coflow scheduling. However, in the context of multi-stage jobs, existing coflow scheduling frameworks only focus on minimizing the average job completion time (JCT) while overlooking the isolation guarantee. To address this problem, in this paper we propose the first coflow scheduling scheme that aims to achieve both objectives. We show that our scheduler outperforms existing alternatives significantly in minimizing average JCT while guaranteeing that no job will be delayed beyond a constant time than its JCT in a fair scheme. Our evaluation results show that our scheduler reduces the average JCT by at least 86% compared with state-of-the-art schedulers.

Index Terms—datacenter network, coflow scheduler, multi-stage jobs

1 Introduction

Data-parallel frameworks, such as MapReduce [1], Hadoop [2] and Spark [3], are widely deployed in modern datacenters. Various distributed computing jobs (*e.g.*, data mining or querying) are run by these frameworks, where intermediate data is transferred between a group of machines. For example, a MapReduce job distributes mapper tasks and reducer tasks on two sets of machines (not necessarily be disjoint) determined by a master process. Each mapper task reads input data from local disks or networks and produces intermediate key/value pairs. Each reducer task accepts a set of keys and values for those keys, merges values together, and writes the final output data to local files. The intermediate data needs to be transferred from mapper machines to reducer machines and such process is called as a *shuffle* phase. These flows are reported to account for over 50% of job completion time, which has a significant impact on job performance [4].

The *coflow* abstraction was proposed to help improve shuffle performance. A coflow means a collection of parallel flows with associated semantics and a collective objective [5]. In the MapReduce case, flows in one *shuffle* phase are termed a coflow. Not until all parallel flows have finished transmission will the MapReduce shuffle phase complete, thus the slowest flow in a coflow critically affects the completion time of reducer tasks. Hence, to reduce the coflow-completion-time (CCT), the optimization scheduling framework should schedule flows in coflow level rather than flow level. Many recent works focus on achieving optimal performance, *e.g.*, minimizing average coflow-completion-time [6], [7], [8], [9], [10]. In the meantime, many network schedulers settle for offering optimal isolation among coflows [11], [12], [13], [14]. These schedulers guarantee fair bandwidth sharing for all coflows so that the predictable CCT can be provided.

- *Z. Liu, H. Dai, B. Tian, W. Rafique, and W. Dou are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210023, China. E-mail: {zifanliu, bctian, rafiqwajid}@smail.nju.edu.cn, {haipengdai, douwc}@nju.edu.cn.*

However, achieving both the above targets simultaneously is challenging in the multi-stage job scenario. First, for jobs containing only one shuffle phase, since there is only one coflow in a job, minimizing average CCT usually leads to faster jobs. However, for the multi-stage jobs, minimizing average CCT does not always result in faster jobs because the dependencies in a job should be considered. Tian *et al.* [15] took the first step on minimizing multi-stage job-completion-time (JCT), but did not give attention to the isolation between jobs. Second, schedulers providing optimal isolation usually allocate fair share network bandwidth to all coflows. However, fair scheduling may fall short in poor coflow performance (*i.e.*, minimizing average CCT) [11]. Performance and fairness have been considered as conflict objectives for long time. Wang *et al.* [16] attempted to achieve both objectives, but they did not consider multi-stage jobs.

In the context of multi-stage jobs, we say that coflow C_2 is dependent on another coflow C_1 if the computation stage of C_1 is the producer of C_2. There are two types of dependencies: *Start-After* and *Finishes-Before*. *Start-After* indicates the entity of explicit barriers [7], which means coflow C_2 cannot start before coflow C_1 completes. *Finishes-Before* means coflow C_2 can coexist with coflow C_1 but cannot complete until C_1 has completed, which is common for the pipeline of continuing stages [17]. We focus on scheduling jobs with *Start-After* type dependencies in the rest parts of this paper, and leave the *Finishes-Before* dependencies to future work.

In this paper, we aim to address the challenge of achieving both of optimal performance and isolation guarantee in the context of multi-stage jobs. Generally, we make the following major contributions.

First, to the best of our knowledge, this is the first work for the studies: *how to schedule dependent coflows of multi-stage jobs to provide long-term isolation guarantee while minimizing the average JCT* (Section 2).

Second, We present a formal mathematical formulation for this probelm (Section 3) and propose a novel coflow

scheduler for multi-stage jobs (Section 4). The key idea of our approach is to allocate bandwidth to coflows in the priority order depending on their completion time under fair schedulers (*e.g.*, DRF [12] or HUG [11]). We find this idea can lead to the best of both objectives for the following two reasons. First, since fair schedulers offer coflows the same progress, small coflows seem to complete earlier than large ones. Second, the isolation service can only be observed by applications when coflows complete. Our approach tends to complete coflows earlier than they would have for a fair scheduler, thus a *long-term isolation guarantee* is provided to applications.

Finally, we evaluate the performance of our scheduler by comparing it with three state-of-the-art schedulers: HUG [11], Aalo [7], and Utopia [16]. Results indicate that our scheduler can provide guaranteed isolation between jobs and the job slowdown can be reduced by at least 86% compared with the state-of-the-art schedulers (Section 5.1).

2 Related Work

Existing works mostly focus on scheduling coflows in single-stage jobs. These schedulers usually settle on one goal (performance or isolation) as the primary objective, while ignoring or treating the other one as a secondary objective. We will discuss these schedulers in the aspects of primary objectives.

Performance-optimal: Many schedulers focus on minimizing average CCT. Orchestra [4] first presents the coflow abstraction and shows that even a simple FIFO algorithm can significantly improve coflow performance. Varys [6] uses the *smallest-effective-bottleneck-first* (SEBF) heuristic to sort coflows and the *minimum-allocation-for-desired- duration* (MADD) heuristic to preferentially allocate least bandwidth to coflows for minimizing average CCT and deadline missing rate. CODA [8] aims at scheduling information-agnostic coflows by automatically collecting coflow information. Barrat [9] and Stream [10] focus on decentralized coflow scheduling. These schedulers cannot handle the dependencies and minimize average JCT for multi-stage jobs. To the best of our knowledge, only [7] and [15] consider multi-stage jobs and attempt to minimize average JCT.

Isolation-optimal: There are also some schedulers aiming at providing isolation guarantee to coflows. HUG [11] seeks Dominant Resource Fairness (DRF) [12] to allocate fair share of bandwidth to coflows. Chen *et al.* [18] assumed that coflows have respective utility fuctions and developed an approximation algorithm to address the obtained optimal max-min fairness problem. NC-DRF [14] provides isolation guarantee to non-clairvoyant coflows with the coflow correlation information. These schedulers fall short in longer average CCT and cannot provide near-optimal performance.

Both two objectives: Coflex [19] and Utopia [16] are the only two schedulers that consider both the two objectives with same importance. Coflex is proposed as a middle ground between Varys and HUG, which navigates the tradeoff space between performance and isolation with an elastic *fairness knob* in the range of 0 to 1. However, such knob is hard to be determined in real-world systems. Utopia uses super-coflow scheduling to provide isolation guarantee with near-optimal performance, but cannot handle the multi-stage jobs.

TABLE 1
Key Terms and Descriptions

Terms	Description
M	Number of total jobs
N	Number of total coflows
K	Number of machines
$F_i = \langle f_i^1, \cdots, f_i^{2K} \rangle$	Demand vector of coflow-i
$d_i = \langle d_i^1, \cdots, d_i^{2K} \rangle$	Correlation vector of coflow-i
$c_i = \langle c_i^1, \cdots, c_i^{2K} \rangle$	Bandwidth allocation of coflow-i
$\overline{f}_i = \max_k f_i^k$	Bottleneck demand of coflow-i
P_i	Progress of coflow-i
Γ_i	Progress of job-i
T_i	JCT of job-i
T_i^*	JCT of job-i under a fair scheduler

3 Model and Objective

In this section, we firstly introduce the model of datacenter networks and coflow, and then discuss two objectives. To simplify the discussion, key terms used in our model are summarized in Table 1.

3.1 Model

Given the full bisection bandwidth, which has been well developed in modern datacenter [20], we treat the datacenter network as a big non-blocking switch connecting K machines. Each machine has one uplink port and one downlink port, thus the whole fabric has $2K$ ports. In this simplified model, the ports are the only congestion points. Therefore, we focus sorely on bandwidth of each port. In our analysis, all links are assumed to be of equal capacity that is normalized to one.

The coflow abstraction presents the communication demand between stages of parallel computing frameworks. A coflow is composed of a collection of flows across a group of machines sharing a common performance requirement. The completion time of the latest flow defines the completion time of this coflow. In many data-parallel frameworks like MapReduce/Hadoop, the job and coflow properties, such as source, destination, amount of data transferred of each flow, coflow dependencies are known as a priori (*e.g.*, after the mapper phase in MapReduce) [6], [7], [15]. In this paper, we assume that all jobs are released at time 0 and all coflows with no dependencies start at time 0 too.

Specifically, the coflow *demand vector*, say $F_i = \langle f_i^1, \cdots, f_i^{2K} \rangle$, captures the data demand of coflow-i, where f_i^k denotes the amount of data transferred on port k. Additionally, f_i^{hl} denotes a flow transferring data from port h to port l. Among all flows in coflow-i, we name the port with the largest traffic as bottleneck port. Let the data demand on this port be the *bottleneck demand*, defined as $\overline{f}_i = \max_k f_i^k$. To simplify our analysis, the *correlation vector*, say $d_i = \langle d_i^1, \cdots, d_i^{2K} \rangle$, is engaged to describe the demand correlation across ports, where d_i^k is the normalized data demand on port k by the bottleneck demand, *i.e.*, $d_i^k = f_i^k / \overline{f}_i$. This vector indicates that for every byte coflow-i sent on bottleneck port, at least d_i^k bytes should be transferred on port k.

Coflows have elastic bandwidth demands on multiple ports, which means that the *bandwidth allocation vector*, say $c_i = \langle c_i^1, \cdots, c_i^{2K} \rangle$, of coflow-$i$, where c_i^k is the bandwidth

share on port k, is not necessarily in the same ratio of demand vector d_i. Given the bandwidth allocation vector c_i for each coflow-i calculated by coflow scheduler given the demand vectors, the coflow progress is restricted by the worst-case port. Formally, *progress* of coflow-i is defined as the minimum demand-normalized rate allocation across ports, *i.e.*,

$$P_i = \min_{i:d_i^k>0} \frac{c_i^k}{d_i^k}. \tag{1}$$

Intuitively, progress of coflow-i means the smallest demand satisfaction ratio across all ports, which determines the CCT of coflow-i.

Assume that a multi-stage job-m has N active coflows, *i.e.*, $\{F_{m,1}, \cdots, F_{m,N}\}$. Given the bottleneck demand $\{\overline{f}_{m,1}, \cdots, \overline{f}_{m,N}\}$ and progress $\{P_{m,1}, \cdots, P_{m,N}\}$ of each coflow, the progress of job-m can be computed as the weighted average progress of active coflows in job-m, *i.e.*,

$$\Gamma_m = \frac{\sum_{n=1}^{N} \overline{f}_{m,n} P_{m,n}}{\sum_{n=1}^{N} \overline{f}_{m,n}}, \tag{2}$$

where $\overline{f}_{m,n}$ is the weight of coflow $F_{m,n}$. Similarly, progress of job-m indicates the collectivity smallest demand satisfaction ratio of all coflows belonging to it, which has significant effect on the JCT of job-m.

3.2 Objective

For the multi-stage coflow scheduling problem, we are concerned with the average JCT and isolation guarantee.

1) *Average JCT*: To speed up data-parallel application completion time, as many jobs as possible should be finished in their fastest possible ways. Therefore, minimizing the average JCT is a critical objective for an efficient coflow scheduler.
2) *Isolation Guarantee*: In a shared datacenter network, all tenants expect *performance isolation guarantees*. Existing work has defined such guarantee as the *minimum progress* across coflows [11], *i.e.*, $\max_i P_i$. For multi-stage jobs, we define the isolation guarantee as the minimum progress across active jobs, *i.e.*, $\max_m \Gamma_m$. To optimize the isolation guarantee, a coflow scheduler should look for an allocation to maximize the minimum progress.

However, on the application level, the effect of isolation guarantee cannot be perceived until the job is finished. If we take the fair scheme (*e.g.*, DRF [12] or HUG [11]) as a baseline, as long as an application observes that its jobs finish no later than the time point at which they would have finished in the baseline algorithm, the isolation guarantee is provided in long run. Thus, we introduce the job *long-term isolation guarantee* to our model.

Definition 1 (Long-term Isolation Guarantee): Consider a multi-stage job-i, let T_i be its JCT by coflow scheduler S. T_i^* is its JCT by a fair scheduler that enforces a minimum instantaneous progress of all active coflows. We call the scheduler S provides the job long-term isolation guarantee if all jobs complete no later than $T_i^* + D$, where D is a constant delay, *i.e.*,

$$T_i \leq T_i^* + D. \tag{3}$$

In this paper, our objective is to gain the best performance regarding both targets in a long run, which is formalized as

$$\begin{aligned} \text{minimize} \quad & \sum_i T_i, \\ \text{s.t.} \quad & T_i \leq T_i^* + D, \text{ for all job-}i. \end{aligned} \tag{4}$$

4 Algorithm and Analysis

In this section, we design a coflow scheduling algorithm of multi-stage jobs with isolation guarantee. We first elaborate on how to sort coflows based on their priorities. Then, we propose our bandwidth allocation algorithm.

4.1 Coflow Sorting

Before we allocate bandwidth to coflows, the priority of each coflow should be determined. Suppose all coflows are scheduled by the fair scheduler DRF, which enforces the equal progress across coflows [11], [12]. Based on the completion time of each coflow under such scheduling, we can obtain a priority order for all coflows. If a coflow completes earlier in DRF scheduling, it has a higher priority than the others. Since DRF enforces the same progress to all coflows, the completion time can be calculated by multiplying progress and bottleneck demands of these coflows.

However, directly performing DRF is not suitable in the context of multi-stage jobs, because some coflows that have dependencies are not released at the start. At first, we can only calculate the completion time of each coflow in F' with no dependencies, which are released at time 0. Then, the minimum progress of these coflows $P^* = \min_i P_i'$ is produced, where P_i' is the progress of F_i'. Note that in popular parallel frameworks, the numbers of coflows in different stages are similar [3], [21]. We assume that P^* keeps unchanged across stages, then the completion time of coflow $F_i \notin F'$ is estimated as $t_i = P^* \overline{f}_i$.

We assume that the job J_i is the i-th job to complete data transferring, *i.e.*, when J_i completes, $J_1, \cdots, J_{i-1}$ have already completed. Let $(F_{i,1}, \cdots, F_{i,N_i})$ be the coflows constituting job J_i in the topological order.

Besides, the completion time of single coflow is not necessarily the priority of the job containing it. In multi-stage jobs, coflow pipelines are developed using directed acyclic graph (DAG) pattern to show the dependencies between coflows. In job-i, we set the weights of coflows in DAG as their completion time. Then, we can get the *estimate completion time* of coflow-j $F_{i,j}$ as the maximum total weight along DAG edges from roots to coflow-j. A coflow with shorter estimate completion time has a higher priority. Additionally, the estimate completion time of job-m is the maximum estimate completion time of all coflows it contains. Similarly, if a job completes earlier, it has a higher priority. Within a job, the priority order of coflows is the ascending order of estimate completion time. Therefore we get a prioritized queue of all coflows $\mathbf{F} = (F_{1,1}, \cdots, F_{1,N_1}, \cdots, F_{M,1}, \cdots, F_{M,N_M})$. The entire procedure is summarized in Algorithm 1.

Algorithm 1 Coflow Sorting Algorithm

Input: Data demand set of all coflows F and the job set J.
Output: An ordered queue of all coflows $\mathbf{F}$.

```
Sort F in the topological order of DAG.
F' ← coflows have no dependencies in F.
P* ← the minimum progress of F' under DRF.
for i = 1 → |F| do
    if F_i ∈ F' then
        t_i ← the completion time of F_i under DRF.
    else
        t_i ← P* f̄_i.
    if F_i has dependencies then
        t_i ← t_i + max{t_j | F_i depends on F_j}.
T_m ← the maximum total time along DAG edges of J_m.
Sort J in the ascending order of T.
Initialize F as an empty queue.
for m = 1 → |J| do
    F̄_m ← coflows of J_m in the ascending order of t.
    F ← F + F̄_m.
```

4.2 Bandwidth Allocation

After obtaining the prioritized queue of coflows, we can allocate bandwidth to the active coflows one by one. Specifically, we introduce the *super-coflow* conception which was proposed in [16] for our solution.

Super-coflow. Given a queue of coflows $\mathbf{F}$, the super-coflow S_i is defined as the sequential aggregation of the first i coflows $F_1, \cdots, F_i$. Formally, the demand vector $D_i = \langle D_i^1, \cdots, D_i^{2K} \rangle$ of S_i is the accumulation of the first i coflows, *i.e.*, $D_i = \sum_{j=1}^{i} f_j$. Super-coflow S_1 trivially degrades into coflow F_1 as a special case.

In previous works, coflows are mostly scheduled individually. We take as an example the minimum-allocation-for-desired-duration (MADD) algorithm, which is a common approach for sequential coflow bandwidth allocation [4], [6]. In MADD, the least amount of bandwidth was allocated to a coflow to obtain the maximum possible progress. Let R_k be the remaining bandwidth on port k and $d_i = \langle d_i^1, \cdots, d_i^{2K} \rangle$ be the correlation vector of coflow-i. Thus, the maximum possible progress P' of coflow-i can be calculated as

$$P' = \min_{1 \le k \le 2K} \frac{R_k}{d_i^k}. \tag{5}$$

To acquire the maximum possible progress, the bandwidth allocation of coflow-i on port k is at least $P' d_i^k$. In MADD, the allocation is exactly $P' d_i^k$.

However, MADD may lead to the priority inversion problem. If a coflow is not able to get any progress, then it gets no bandwidth, even if allocating it some bandwidth will accelerate its completion. To further explain priority inversion, we consider a motivating example in Fig. 1. Three coflows (A, B, and C) transfer data through three ports whose bandwidths are normalized to 1. Coflow-A, coflow-B, and coflow-C have demand vectors $F_A = \langle 3, 0, 0 \rangle$, $F_B = \langle 1, 4, 2 \rangle$, and $F_C = \langle 0, 6, 5 \rangle$, respectively. Since coflow-A has the smallest bottleneck demand 3, it has the highest priority, followed by coflow-B. The priority order of the three coflows is $A > B > C$. Fig. 1(a) shows the bandwidth allocation for the MADD scheme. At time

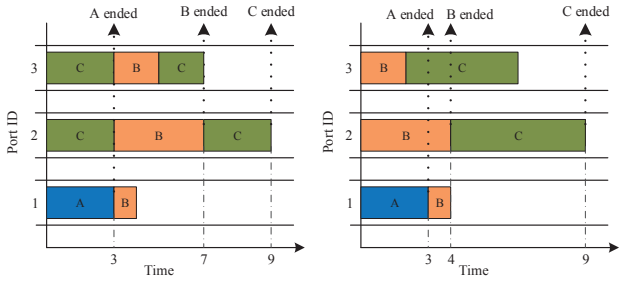


Fig. 1. Priority inversion in MADD. Coflows transfer data through three ports whose bandwidths are normalized to 1. Three coflows have demand vectors $F_A = \langle 3, 0, 0 \rangle$, $F_B = \langle 1, 4, 2 \rangle$, and $F_C = \langle 0, 6, 5 \rangle$. The priority order of the three coflows is $A > B > C$.

0, coflow-B cannot get any bandwidth on port 1, *i.e.*, it cannot get any progress, so no bandwidth is allocated to coflow-B but to coflow-C on ports 2 and 3. Coflow-B is preempted by coflow-C, whereas in an optimal scheme, coflow-B still should gain bandwidth on ports 2 and 3 as shown in Fig. 1(b).

On the contrary, our super-coflow conception eliminates the priority inversion situation as much as possible. When computing the bandwidth allocation, coflow F_i is justified if it contributes to super-coflow S_i. As long as F_i affects the progress of S_i, a bandwidth allocation to F_i is always guaranteed, even when it gains no progress alone, which easily triggers a priority inversion.

Assume that the active coflow priority queue is $F' = (F'_1, \cdots, F'_{N'})$. Our bandwidth allocation algorithm runs in turns. In the i-th turn, we will allocate the least bandwidth to F'_i, while the allocations of earlier turns stay unchanged, in order to achieve the minimum possible completion time of S_i. To easily describe our algorithm, we next focus on the bandwidth allocation of F'_i.

To achieve the minimum possible completion time of S_i, we first calculate the bottleneck data demand of S_i, *i.e.*,

$$\overline{D}_i = \max_{1 \le k \le 2K} D_i^k. \tag{6}$$

Let D_i^{hl} be the data amount transferred from port h to port l in S_i. Then, at least $D_i^{hl}/\overline{D}_i$ bandwidth should be allocated for the flows between port h and port l. Note that the first $i-1$ coflows have received some bandwidth in the earlier turns. We denote the bandwidth received by flow f_k^{hl} in coflow-k as u_k^{hl} for $k < i$, thus we can distribute at most $(D_i^{hl}/\overline{D}_i - \sum_{k=1}^{i-1} u_k^{hl})^+$ bandwidth to f_i^{hl}, where $(x)^+ = \max(0, x)$. Given the remaining bandwidth R_h on port h and R_l on port l, which limit the actual bandwidth available for f_i^{hl}, we finally get u_i^{hl} as

$$u_i^{hl} = \min[(D_i^{hl}/\overline{D}_i - \sum_{k=1}^{i-1} u_k^{hl})^+, R_h, R_l]. \tag{7}$$

After that, the remaining bandwidth on these two ports are updated. When all turns have finished, unused bandwidth is distributed to coflows. For each ingress port h, remaining bandwidth R_h is distributed to corresponding flows according to ratio of their current received bandwidth

u_i^{hl}, restricted by the corresponding egress port l. We summarize the whole procedure as SuperflowAllocation(F) in Algorithm 2.

Algorithm 2 Bandwidth Allocation Algorithm

```
1: procedure SUPERFLOWALLOCATION(Coflows F)
2:     Initialize unused bandwidth R_k ← 1 on port-k
3:     for i = 1 → |F| do
4:         Assemble demands D_i = Σ_{k=1}^{i} f_k.
5:         D̄_i ← max_k D_i^k.
6:         for all flow f_i^{hl} ∈ F_i do
7:             u_i^{hl} ← min[(D_i^{hl}/D̄_i − Σ_{k=1}^{i−1} u_k^{hl})^+, R_h, R_l].
8:             R_h ← R_h − u_i^{hl}.
9:             R_l ← R_l − u_i^{hl}.
10:    Allocate remaining bandwidth to all coflows.
11: procedure BANDWIDTHALLOCATION(Coflows F)
12:    F ← sorted F by Algorithm 1.
13:    Initialize active coflows F' ← ∅.
14:    while True do
15:        F' ← released coflows in the order of F.
16:        if F' is changed then
17:            Update remaining transfer demand of F'.
18:            SuperflowAllocation(F').
19:        if F' == ∅ then
20:            break.
```

When an old coflow is finished or a new coflow is released, the active coflow set F' is changed and the bandwidth allocations will be rescheduled. All the active coflows are sorted as the order we get by Algorithm 1 to maintain the priorities of all jobs through our scheduling. The main procedure is summarized as BandwidthAllocation(F) in Algorithm 2.

4.3 Long-term Isolation Guarantee

Our algorithm provides the job long-term isolation guarantee of Definition 1. For each job-i, the completion time T_i is guaranteed not to exceed a constant time over its completion time in DRF. Thus, we have the following theorem:

Theorem 1 (Long-term Isolation Guarantee): Assume that jobs J are released at time 0. For each job $J_i \in J$, let T_i be the JCT of J_i in Algorithm 2, and let T_i^* be the completion time of J_i in DRF. The completion time delay is bounded as

$$T_i \leq T^* + \overline{f}_i. \tag{8}$$

Proof: To simplify our proof, we abbreviate F_{i,N_i} and S_{i,N_i} to F_i and S_i through this proof and let $F(i)$ be the coflows before F_i in our priority order. Since F_i is the last completed coflow of J_i, the JCT of J_i (*i.e.*, T_i) equals to the completion time of F_i. Additionally, we let $F_0(i)$ be the coflows that has shorter estimate completion time than F_i, *i.e.*, coflows in $F_0(i)$ are the ones finished before F_i in DRF. We have $F(i) \subseteq F_0(i)$ because all coflows in $\{J_1, \cdots, J_i\}$ except F_i have shorter estimate completion time than F_i. To discuss T_i, we consider the following two cases.

Case 1. The bottleneck port of super-coflow S_i is kept using full bandwidth during the data transmission. In this case, the completion time of S_i is simply the time of transferring data on bottleneck port, *i.e.*, $\overline{D}_i$. When S_i completes, F_i must have completed. Therefore, we have $T_i \leq \overline{D}_i$. Then, we turn to DRF. According to our algorithm, all jobs are sorted by their JCT in DRF. When J_i completes, previous $i-1$ jobs $J_1, \cdots, J_{i-1}$ must have all completed, so does S_i. Note that $\overline{D}_i$ is the minimum possible completion time of S_i, *i.e.*, $\overline{D}_i \leq T_i^*$, thus we finally have

$$T_i \leq \overline{D}_i \leq T_i^*. \tag{9}$$

Case 2. The bottleneck port of super-coflow S_i is not fully used at some time during the data transmission. Recall that the bandwidth allocation on bottleneck port is constrained by the available bandwidth on the coupled ports due to Line 7 in Algorithm 2. In particular, we define port k as a coupled port of port l if there are flows in S_i transferring data between these two ports but getting less bandwidth allocation than $D_i^{kl}/\overline{D}_i$. Let B_i be the bottleneck port of S_i and let $C(B_i)$ be the set of coupled ports of B_i. Additionally, let $C_0(B_i)$ be the ports that have data transmission with B_i in S_i.

Let t_B be the time when port B_i starts to get full bandwidth allocation until the completion of S_i. Specifically, let t_k be the time when port $k \in C(B_i)$ finishes data transmission of all the coflows in $F(i)$. In our algorithm, the lacking of bandwidth utilization on B_i can only occur when bandwidth of each port in $C(B_i)$ is fully used, otherwise our algorithm will attempt to allocate the spare bandwidth to a flow between such coupled port and B_i. Therefore, we have

$$t_B \leq \max_{k \in C(B_i)} t_k. \tag{10}$$

Since the bandwidth allocation of each port varies throughout the data transmission because of completions of coflows, we denote $a_i^{kl}(t)$ and $b_i^{kl}(t)$ as the expected and actual bandwidth allocation of the flows of S_i transferring between port k and port l at time t. In particular,

$$a_i^{kl}(t) = D_i^{kl}(t)/\overline{D}_i(t); \tag{11}$$

$$b_i^{kl}(t) = u_i^{kl}(t) + \sum_{j \in F(i)} u_j^{kl}(t). \tag{12}$$

Note that during the transmission of S_i, the actual bandwidth allocation on some ports may continue being less than its expected bandwidth when S_i is running alone under DRF. In some extreme situations, the bottleneck port of S_i may change from B_i to another port k if port k keeps getting less bandwidth allocation than expected, which we call the bottleneck port of S_i *shifts*. To bound our JCT of J_i, we next discuss two sub-cases, differentiated by whether the bottleneck port of S_i shifts.

Sub-Case 1. The bottleneck port B_i of S_i does not *shift* during the data transmission. In the worst case, bottleneck port B_i can only get full bandwidth allocation when all previous coflows finish their data transmissions on the coupled ports of B_i, which means $t_B = \max_{k \in C(B_i)} t_k$. We have the JCT constraint of J_i as

$$T_i \leq \overline{D}_i + \sum_{k \in C_0(B_i)} \int_0^{t_k} (a_i^{kB_i}(t) - b_i^{kB_i}(t))dt. \tag{13}$$

Since S_i is the aggregation of $F(i)$ and F_i, we can split $\overline{D}_i$ into two parts as

$$\overline{D}_i = f_i^{B_i} + \mathbf{D}_i^{B_i}, \tag{14}$$

where $f_i^{B_i}$ is the data demand of F_i on port B_i and $\mathbf{D}_i^{B_i}$ is the sum of data demands of coflows in $F(i)$ on B_i. Thus, we have

$$\mathbf{D}_i^{B_i} = \sum_{k \in C_0(B_i)} \int_0^{t_k} (b_i^{kB_i}(t) - u_i^{kB_i}(t))dt. \tag{15}$$

Additionally, we have

$$u_i^{kB_i}(t) \geq 0. \tag{16}$$

By combining Eq. (14), (15), and (16), we can transform Eq. (13) into

$$T_i \leq f_i^{B_i} + \sum_{k \in C_0(B_i)} \int_0^{t_k} a_i^{kB_i}(t)dt. \tag{17}$$

Then, we first focus on the ports $k \in C(B_i)$. In DRF, when coflow F_i finishes, all coflows in $F_0(i)$ must have finished, so do coflows in $F(i)$. Since port k keeps using full bandwidth, F_i will not finish before t_k. We thus have

$$\max_{k \in C(B_i)} t_k \leq T_i^*. \tag{18}$$

Next, we discuss the ports $k \in C_0(B_i) - C(B_i)$. On these ports, the actual bandwidth allocations are not greater than the expected allocations, *i.e.*,

$$a_i^{kB_i}(t) \leq b_i^{kB_i}(t). \tag{19}$$

Combining Eq. (9), (11), and (19), we have

$$\max_{k \in C_0(B_i) - C(B_i)} t_k \leq \overline{D}_i \leq T_i^*. \tag{20}$$

Combining Eq. (18) and (20), we can bound the $t_i^{\max}$ as

$$t_i^{\max} = \max_{k \in C_0(B_i)} t_k \leq T_i^*. \tag{21}$$

Note that the bandwidth allocation on B_i cannot exceed 1, *i.e.*,

$$\sum_{j \in C_0(B_i)} a_i^{kB_i}(t) \leq 1. \tag{22}$$

According to Eq. (21) and (22) we transform Eq. (17) into

$$\begin{aligned} T_i &\leq f_i^{B_i} + T_i^* \\ &\leq \overline{f}_i + T_i^*. \end{aligned} \tag{23}$$

Sub-Case 2. The bottleneck port of S_i *shifts* to $B_i^1, B_i^2, \cdots, B_i$ in order. We denote the time when bottleneck port shifts to B_i as t^*. Consider the worst case that before t^* there exists no bandwidth allocation on B_i. Similar to Eq. (17), we have

$$\begin{aligned} T_i &\leq t^* + \overline{D}_i + \sum_{k \in C_0(B_i)} \int_{t^*}^{t^k} (a_i^{kB_i}(t) - b_i^{kB_i}(t))dt \\ &\leq t^* + (f_i^{B_i} + \mathbf{D}_i^{B_i}) + \sum_{k \in C_0(B_i)} \int_{t^*}^{t^k} (a_i^{kB_i}(t) - b_i^{kB_i}(t))dt \\ &\leq f_i^{B_i} + t^* + \sum_{k \in C_0(B_i)} \int_{t^*}^{t^k} a_i^{kB_i}(t)dt. \end{aligned} \tag{24}$$

By combining Eq. (22), similar to the previous analysis, we finally transform Eq. (24) to

$$\begin{aligned} T_i &\leq f_i^{B_i} + t^* + (t_i^{\max} - t_*) \\ &\leq f_i^{B_i} + t_i^{\max} \\ &\leq \overline{f}_i + T^*. \end{aligned} \tag{25}$$

This completes the proof. □

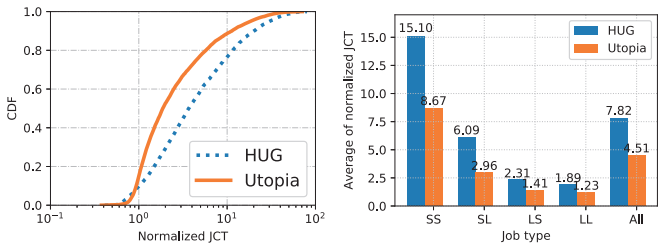
5 Evaluation

We evaluate our scheduler through trace-driven simulations. Followings are two highlights of our evaluations:

- Our scheduler provides long-term fairness with guaranteed isolation between jobs. It dominates HUG [11] and Utopia [16] in our simulations and only few jobs are delayed for a short time in our scheduler.
- Our Scheduler minimizes the average JCT and outperforms HUG, Utopia and Aalo by 92%, 86%, and 89%, which shows that our scheduler leads to significant performance improvement compared with existing schemes.

5.1 Methology

Simulator: We develop an event-based flow simulator to evaluate our scheduler. Since there is no prior work focusing on both of minimizing average JCT and guaranteeing isolation of multi-stage jobs, we compare our scheduler with three coflow scheduling schemes: HUG [11], Aalo [7], and Utopia [16]. HUG is the isolation-optimal scheme that enforces same progress among coflows. Aalo is a performance-optimal scheme that considers the context of multi-stage jobs. Utopia can provide the near-optimal performance with isolation guarantee but it cannot handle the multi-stage jobs.
Workload: We take the realistic coflow traces [22] synthesized from real-world MapReduce workloads collected from a 3000-machine and 150-rack Facebook cluster. These traces are widely used as a benchmark for coflow scheduling analysis [6], [7], [14], [23]. Specifically, all mappers (reducers) in the same rack are combined to one mapper (reducer). However, the original trace data captures single-stage coflows and is incomplete because the transmitted bytes are recorded in reducer-level instead of flow-level. Therefore, we apply two transformations to these traces. First, we partition the bytes of each reducer pseudo-randomly to each mapper. Second, we combine these coflows randomly into jobs such that each job is expected to have β coflows. Dependencies between coflows within a job are also randomly generated.
Setup: There are α machines in this virtual network. The datacenter fabric is abstracted as a $\alpha \times \alpha$ non-blocking switch. Each ingress (egress) port is connected with an uplink (downlink) port of a machine with 1 Gbps bandwidth. We will evaluate long-term isolation and performance of our scheduler and other three baseline schedulers. We will also investigate the influence of the number of machines (α) and the average number of coflows in a job (β). The default parameters are chosen as $\alpha = 30$ and $\beta = 20$. All results in our evaluation are the average of 100 individual simulations with standard error.



(a) Distribution of normalized JCT (b) Average normalized JCT

Fig. 2. Characteristics of long-term isolation guarantee. (a) Distribution of normalized JCT. (b) Average normalized JCT of different job types.

5.2 isolation guarantee

In Theorem 1 we have shown the delay bound of JCTs with our scheduler. Now we evaluate long-term isolation guarantee of our scheduler with other schedulers which guarantee isolation among coflows. Specifically, we compared our scheduler with two schedulers, *i.e.*, HUG and Utopia, and use a metric called *nornalized JCT* through the evaluation. Normalized JCT is defined for each job as the JCT of compared scheduler normalized by that under our scheduler, *i.e.*,

$$\text{Normalized JCT} = \frac{\text{Compared JCT}}{\text{JCT under our scheduler}}.$$

Intuitively, the larger the normalized JCT, the later jobs will complete for the compared schedulers. If the normalized JCT is larger (smaller) than 1, the job completes earlier (later) in our scheduler.

Since each job is randomly combined by coflows in our Facebook traces with random dependencies, the distributions of normalized JCT of two compared schedulers are necessary to evaluate. As shown in Fig. 2(a), most jobs complete earlier in our scheduler than they do in HUG and Utopia. Specifically, only 10% of jobs in HUG and 15% in Utopia complete earlier than that in our scheduler. Additionally, more than 70% of jobs for our scheme are more than $2\times$ faster than that for HUG, and more than 47% faster than that of Utopia.

For better understanding of the performance impact on different jobs, we further divide all jobs into four bins based on their total size and number of dependencies. Specifically, we consider a job *small* (*large*) if its total size is less (more) than half of jobs in specific time of simulation, and *short* (*long*) if it contains less (more) dependencies than $\beta/2$ where β is the average number of coflows in a job.

The average normalized JCTs of the two baseline schedulers in four different types are compared as shown in Fig. 2(b). A general trend can be observed that the normalized JCT of a *small* (*short*) job is much likely higher than that of a *large* (*long*) job. This trend is reasonable for our algorithm based on a completion time order specified by a fair scheduler. Because of the dominance of small jobs (more than 70% in our simulation), speeding up their completion significantly increases the average performance. On average, our scheduler can speed up a job's completion by $7.82\times$ ($4.51\times$) in expectation compared with HUG (Utopia).

5.3 Performance

We next evaluate the performance (*i.e.*, average JCT) of our scheduler with other baseline schedulers given different parameters (α and β). Because of small jobs are in the majority of all jobs (60% of all), we define a new metric called average *job slowdown* instead of average JCT in our evaluation by borrowing the idea from *coflow slowdown* defined in [16]. Job slowdown is defined as

$$\text{Job slowdown} = \frac{\text{Compared JCT}}{\text{Minimum JCT if running alone}}.$$

As shown in Fig. 3(a), the JCT of our scheduler is much smaller than those of the three baseline schedulers overall.

First, we investigate the influence of α, *i.e.*, the number of machines, as shown in Fig. 3(b). As discussed in Section 5.1, the total bandwidth grows linearly as α increases. However, the average job slowdown does not increase in reciprocal proportion of α because of the non-uniformity of coflow space distribution. Compared with HUG, Utopia, and Aalo, we reduce the average slowdown by up to 92%, 88%, and 90%, respectively.

Second, we investigate the influence of β, *i.e.*, the average number of coflows in each job, as shown in Fig. 3(c). We can see that all four lines decrease with similar gradient. Compared with HUG, Utopia, and Aalo, we reduce the average slowdown by up to 92%, 86%, and 89%, respectively.

Overall, our scheduler outperforms baseline schedulers significantly. We believe the main reason is that baseline schedulers focus on the coflow-level scheduling while ours focuses on job-level scheduling.

6 Conclusion

In this paper, a coflow scheduler of multi-stage jobs is proposed. To the best of our knowledge, our work is the first coflow scheduler that aims to simultaneously minimize the average JCT and serve isolation guarantee in the context of multi-stage jobs. We prove that our scheduler can provide long-term isolation guarantee. The trace-driven evaluations have confirmed that our scheduler achieves near-optimal performance compared with the state-of-the-art schedulers.

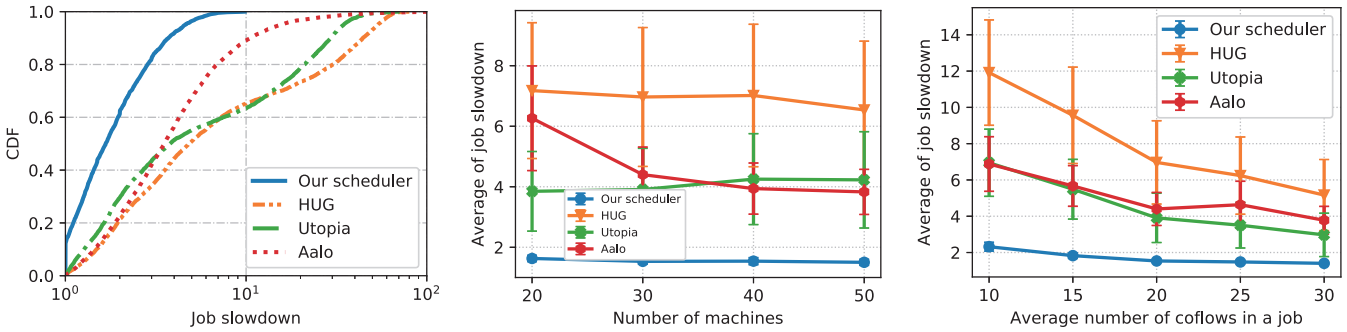
Acknowledgments

This work is supported in part by the National Science Foundation of China under Grants (No. 61672276, No. 61502229, No. 61872178), the National Key Research and Development Program of China under Grant (No. 2017YFB1400601), the Fundamental Research Funds for the Central Universities (No. 021014380079), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "Apache Hadoop." [Online]. Available: http://hadoop.apache.org/



(a) CDF of job slowdown (b) The # of machines (α) (c) Average # of coflows in each job (β)

Fig. 3. Characteristics of performance. (a) CDF of job slowdown. (b) Average job slowdown in different number of machines. (c) Average job slowdown in different average number of coflows in each job.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of USENIX Conference on Networked Systems Design and Implementation*. USENIX, 2012, pp. 15–28.

[4] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proceedings of Conference of the ACM Special Interest Group on Data Communication*. ACM, 2011, pp. 98–109.

[5] M. Chowdhury and I. Stoica, "Coflow:a networking abstraction for cluster applications," in *Proceedings of ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 31–36.

[6] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of Conference of the ACM Special Interest Group on Data Communication*. ACM, 2014, pp. 443–454.

[7] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proceedings of Conference of the ACM Special Interest Group on Data Communication*. ACM, 2015, pp. 393–406.

[8] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proceedings of Conference of the ACM Special Interest Group on Data Communication*. ACM, 2016, pp. 160–173.

[9] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proceedings of Conference of the ACM Special Interest Group on Data Communication*. ACM, 2014, pp. 431–442.

[10] H. Susanto, H. Jin, K. Chen *et al.*, "Stream: Decentralized opportunistic inter-coflow scheduling for datacenter networks," in *Proceedings of IEEE International Conference on Network Protocols*. IEEE, 2016, pp. 1–10.

[11] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *Proceedings of USENIX Conference on Networked Systems Design and Implementation*. USENIX, 2016, pp. 407–424.

[12] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of USENIX Conference on Networked Systems Design and Implementation*. USENIX, 2011, pp. 323–336.

[13] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: sharing the network in cloud computing," in *Proceedings of Conference of the ACM Special Interest Group on Data Communication*. ACM, 2012, pp. 187–198.

[14] L. Wang and W. Wang, "Fair coflow scheduling without prior knowledge," in *Proceedings of IEEE Conference on Distributed Computing Systems*. IEEE, 2018, pp. 22–32.

[15] B. Tian, C. Tian, H. Dai, and B. Wang, "Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time," in *Proceedings of IEEE International Conference on Computer Communications*. IEEE, 2018.

[16] B. Li, L. Wang, and W. Wang, "Utopia: Near-optimal coflow scheduling with isolation guarantee," in *Proceedings of IEEE International Conference on Computer Communications*. IEEE, 2018.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of ACM SIGOPS/EuroSys Conference on Computer Systems*. ACM, 2007, pp. 59–72.

[18] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *Proceedings of IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[19] W. Wang, S. Ma, B. Li, and B. Li, "Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling," in *Proceedings of IEEE International Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[20] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *Proceedings of Conference of the ACM Special Interest Group on Data Communication*. ACM, 2015, pp. 183–197.

[21] S. Wang, J. Zhang, T. Huang, J. Liu, T. Pan, and Y. Liu, "A survey of coflow scheduling schemes for data center networks," *IEEE Communications Magazine*, 2018.

[22] "Coflow-benchmark." [Online]. Available: https://github.com/coflow/coflow-benchmark

[23] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proceedings of ACM symposium on Parallelism in Algorithms and Architectures*, 2015, pp. 294–303.