Coflow Scheduling of Multi-stage Jobs with Isolation Guarantee

Zifan Liu, Haipeng Dai and Wanchun Dou
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China
zifanliu@smail.nju.edu.cn, haipengdai,douwh@nju.edu.cn

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—keyword, keyword, keyword**

I. 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, in which intermediate data are 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 disk or network and produce 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 transfer from mapper machines to reducer machines, which is called as *shuffle* phase. These flows are reported may account for over 50% of job completion time, which has an significant impact on job performance.

The *coflow* abstraction was proposed to help improve shuffle performance. A coflow means a collection of parallel flows [4]. In the MapReduce case, flows in one *shuffle* phase are termed a coflow. Traditional network performance measurements, such as average flow-completion-time (FCT), neglect application-level communication pattern of data-parallel jobs. Not until all parallel flows have finish transmission will the MapReduce shuffle phase complete, thus only the slowest flow (not the average FCT) in a coflow dominates the start time of reducer tasks. Hence, for the job-completion-time (JCT) improvement, the optimization scheduling framework should schedule flows in coflow level rather than flow level. Many recent work focus on achieving optimal performance, e.g., minimizing average coflow-completion-time [5]–[9]. In the meantime, many network schedulers settle for fair network sharing to offer optimal isolation among coflows [10]–[13].

However, achieving both targets above 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 not always results in faster jobs because the dependencies in a job should be considered about. Tian *et al.* [14] have taken the first step on minimizing multi-stage job-completion-time, but does not give concern to isolation of jobs. Second, schedulers providing optimal isolation usually allocate fair share network bandwidth to all coflows, giving the completion of coflows predictable CCTs. However, fair scheduling has been found short of coflow performance (i.e., minimizing average CCT) [10]. Performance and fairness have been considered as conflict objective for long time. Wang *et al.* [15] have made an attempt for achieving both objective, but does not consider about the context of multi-stage jobs.

In the context multi-stage jobs, We call 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 [2], 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 [16]. 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 aimed at addressing the challenge of achieving the best of both world of optimal performance and isolation guarantee in the context of multi-stage jobs while making the following contributions.

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

Second, We present a formal mathematical formulation for this probelm (Section III) and propose a novel coflow scheduler for multi-stage jobs (Section IV). The key idea of our approach is to allocate bandwidth to coflows in the priority order depending on the completion times under fair schedulers (e.g., DRF [11] or HUG [10]). We find this idea can led to the best of both world for two reasons. First, since fair schedulers offer coflows the same progress, small coflows seem to complete faster than large ones. Like performance optimal schedulers, such as Varys [5], small coflows have higher priority than large ones to minimize the average CCT. Second, isolation serve can only be observed by applications when coflow completes. our approach tends to complete

coflows earlier than them would have in a fair scheduler, thus a *long-term isolation guarantee* is provided to applications. To deal with the multi-stage jobs, we will set the priorities in two level: *inter-job priority* and *intra-job priority*. *Inter-job priority* means the faster a job completes in a fair scheduler, the higher priority it is given. *Intra-job priority* means within a job, earlier completed coflows in a fair scheduler have higher priority than the later ones.

(evaluation related parts to be completed later!!!!)

II. Related Work

Existing work 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 as a secondary objective. We will discuss these schedulers in the aspects primary objectives.

Performance-optimal: many schedulers focus on minimizing average CCT. Orchestra [17] first presents the coflow abstraction and shows that even a simple FIFO algorithm can significantly improve coflow performance. Varys [5] 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 [7] aims at scheduling information-agnostic coflows by automatically collecting coflow information. Barrat [8] and Stream [9] focus on decentralized coflow scheduling. These schedulers cannot handle the dependencies and minimize average JCT for multi-stage jobs. The only schedulers that take multi-stage jobs for concern is Aalo [6] and an approximation algorithm presented by Tian *et al.* [14]. Aalo uses only one short section to discuss a simple heuristic for minimizing average JCT. Tian's algorithm cannot guarantee isolation between jobs.

Isolation-optimal: There are also some schedulers aimed at providing isolation guarantee to coflows. HUG [10] seeks a Dominant Resource Fairness (DRF) [11] to allocate fair share of bandwidth to coflows. Chen *et al.* [18] assume coflows have respective utility fuctions and develops a approximation algorithm to resolve the optimal max-min fairness problem. NC-DRF [13] provides isolation guarantee to non-clairvoyant coflows with the coflow correlation information. These schedulers result in longer average CCT thus fail to provide near-optimal performance.

Both two objectives: Coflex [19] and Utopia [15] are the only two schedulers that consider both two objectives with the same importance. Coflex is proposed as a middle ground between Varys and HUG, by navigating the tradeoff space between performance and isolation with a elastic *fairness knob* in the range of 0 to 1. However such knob is hard to determined in real-world systems. Utopia uses super-coflow scheduling to provide isolation guarantee with near-optimal performance, but cannot handle the context of multi-stage jobs.

III. Model and Objective

In this section, we firstly delineate 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.

TABLE I
Key Terms and Descriptions

Terms	Description
M	The number of total jobs.
N	The number of total coflows.
K	The number of machines.
$F_i = \langle f_i^1, \ldots, f_i^{2K} \rangle$	Demand vector of coflow-i.
$d_i = \langle d_i^1, \ldots, d_i^{2K} \rangle$	Correlation vector of coflow-i.
$c_i = \langle c_i^1, \ldots, c_i^{2K} \rangle$	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.

A. 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. Hence we focus sorely on bandwidth of each port. In our analysis, all links are assumed of equal capacity normalized to one.

The coflow abstraction presents the communication demand between stages of parallel computing model. 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 [5], [6], [14]. In this paper, we assume all jobs are released at time 0 and all coflows with no dependencies start at time 0 too.

Specifically, the coflow *demand vector* $F_i = \langle f_i^1, \ldots, f_i^{2K} \rangle$ 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* $d_i = \langle d_i^1, \ldots, d_i^{2K} \rangle$ 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 sends 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 the the *bandwidth allocation vector* $c_i = \langle c_i^1, \ldots, c_i^{2K} \rangle$ 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 measured 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 a multi-stage job-m has N active coflows, i.e., $\{F_{m,1}, \ldots, F_{m,N}\}$. Given the bottleneck demand $\{\overline{f}_{m,1}, \ldots, \overline{f}_{m,N}\}$ and progress $\{P_{m,1}, \ldots, P_{m,N}\}$ 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 $F_{m,n}$. Like above, 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.

B. Objective

In common consensus [4], [15], [19], a coflow scheduler focuses primarily on two objectives, average CCT and isolation guarantee. Under the multi-stage coflow scheduling problem, we should concern the average JCT instead.

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 settled as 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 define such guarantee as the *minimum progress* across coflows [10], 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 [11] or HUG [10]) as a baseline, as long as an application observes its jobs finish no later than 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 which enforce 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 of both targets in a long run, which is formalized as

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

IV. Algorithm and Analysis

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

A. Coflow Sorting

Before we allocate bandwidth to coflows, the priorities of each coflow should be determined. Consider all coflows are scheduled by the fair scheduler DRF, which enforces the equal progress across coflows [10], [11]. On account of the completion time of each coflow under such scheduling, we can obtain a priority order for all coflows. If a coflow completes faster in DRF scheduling, it has a higher priority than the lowers.

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 times of coflows 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'. It is noticed that in popular parallel frameworks, the number of coflows in different stages stays similar [3], [21]. We assume that P^* keeps across stages, then the completion time of coflow $F_i \notin F'$ is estimated as $t_i = P^* \overline{f_i}$.

we assume the job J_i is the i-th job to complete data transferring, i.e., when J_i completes, $J_1, \ldots, J_{i-1}$ have already completed. Let $(F_{i,1}, \ldots, F_{i,N_i})$ be the coflows constituting job J_i in the topological order and let super-coflow $S_{i,j}$ be the bonding super-coflow of $F_{i,j}$ when it finishes. To simplify our explanation,

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 times. Then we can get the *estimate completion time* of coflow-j $F_{i,j}$ as the maximum total weight along DAG edges from roots. 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. Like previous, if a job completes faster, it has a higher priority. Within a job, the priority order of coflows is the ascending order of estimate completion tiome. Therefore we get a prioritized queue of all coflows $\mathbf{F} = (F_{1,1}, \ldots, F_{1,N_1}, \ldots, F_{M,1}, \ldots, F_{M,N_M})$. 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.
$F' \leftarrow$ coflows have no dependencies in F.
$P^* \leftarrow$ the minimum progress of F' under DRF.
for $i = 1 \rightarrow |F|$ **do**
  if $F_i \in F'$ **then**
    $t_i \leftarrow$ the completion time of F_i under DRF.
  else
    $t_i \leftarrow P^* \overline{f_i}$.
  if F_i has dependencies **then**
    $t_i \leftarrow t_i + \max\{t_j | F_i \text{ depends on } F_j\}$
$T_m \leftarrow$ 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 $m = 1 \rightarrow |J|$ **do**
  $\mathbf{F} \leftarrow \mathbf{F} + \overline{F_m}$

B. Bandwidth Allocation

After obtaining the prioritized queue of coflows, we can allocated bandwidth to the active coflows one by one. Specifically, we introduce the *super-coflow* conception which was proposed in [15] 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 $F_1, \cdots, F_i$. Additionally, we call S_i is the *bonding super-coflow* of F_i. Formally, the demand vector $D_i = \langle D_i^1, \ldots, D_i^{2K} \rangle$ of S_i is the accumulation of the first i coflows, i.e., $D_i = \sum_{j=1}^{i} d_j$. Super-coflow S_1 trivially degrades into coflow F_1 as a special case.

In previous works, coflows are mostly scheduled individually. We take the minimum-allocation-for-desired-duration (MADD) algorithm, which is a common approach for sequential coflow bandwidth allocation, as an example [5], [17]. Under 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 $d_i = \langle d_i^1, \ldots, d_i^{2K} \rangle$ 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 acquiring the maximum possible progress, the bandwidth allocation of coflow-i on port k is at least $P' d_i^k$. Under 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 motivation example in Fig. 1. Three coflows(A, B and C) transfer data through three ports whose bandwidths are uniformed to 1. Coflow-A has demand vector $F_A = \langle 3, 0, 0 \rangle$; coflow-$B$ has demand vector $F_B = \langle 1, 4, 2 \rangle$; coflow-$C$ has demand vector $F_C = \langle 0, 6, 5 \rangle$. Since coflow-$A$ has the smallest bottleneck demand 3, it has higher priority than others, followed by coflow-B. The priority order of three coflows is $A > B > C$. Fig. 1a shows the bandwidth allocation under MADD scheme. At time 0, coflow-B cannot get any bandwidth on port 1, i.e., it can get no progress, so no bandwidth is allocated to coflow-B but to coflow-C on port 2 and 3. Coflow-B is preempted by coflow-C, whereas in a optimal scheme, coflow-B still should gain bandwidth on port 2 and 3 as shown in Fig. 1b.

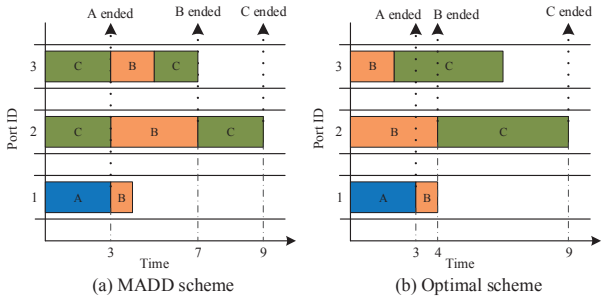

Fig. 1. Priority inversion under MADD. Coflows transfer data through three ports whose bandwidths are uniformed to 1. Three coflows have the demand vectors $F_A = \langle 3, 0, 0 \rangle$, $F_B = \langle 1, 4, 2 \rangle$, $F_C = \langle 0, 6, 5 \rangle$. The priority order of three coflows is $A > B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

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 under MADD.

Assume the active coflow priority queue is $F' = (F'_1, \ldots, F'_{N'})$ and the super-coflow queue is $S = (S_1, \ldots, S_{N'})$. 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. It is noted 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

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

After then, 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} d_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 coflows in the order of F by FIFO.
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.

C. 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. We formalize this theorem as following:

Theorem 1 (Long-term Isolation Guarantee): Assume that jobs J are released at time 0. For all 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

$$T_i \leq T^* + \overline{d}_i. \quad (8)$$

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, T_i, the JCT of J_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 finish before F_i under DRF. We have $F(i) \subseteq F_0(i)$ because all coflows in $\{J_1, \ldots, J_i\}$ 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 under DRF. When J_i completes, previous $i-1$ jobs $J_1, \ldots, J_{i-1}$ must have all completed, so does S_i. It is noticed 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

$$T_i \leq \overline{D}_i \leq T_i^*. \quad (9)$$

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 is 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 $k \in C(B_i)$ finished data transmission of all the coflows in $F(i)$. Under our algorithm, the lacking of bandwidth utilization on B_i can only occurs when bandwidth of ports in $C(B_i)$ are all fully used, otherwise our algorithm will tempt to allocate the spare bandwidth to a flow between such coupled port and B_i. Therefore we have

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

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 flow transferring between port k and port l at time t. In particular,

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

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

We notice that during the transmission of S_i, the actual bandwidth allocation on some ports may continues 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 *shifts* 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

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

where $d_i^{B_i}$ 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)(16), we can transform Eq.(13) into

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

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

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

Second 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)(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)(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)(22) we transform Eq.(17) into

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

Sub-Case 2 The bottleneck port of S_i *shifts* to $B_i^1, B_i^2, \ldots, B_i$ 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

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

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

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

■

V. Evaluation

A. 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 coflows in multi-stage jobs, we compare our algorithm with three coflow scheduling schemes: HUG [10], Aalo [6] and Utopia [15]. HUG is the isolation-optimal scheme that enforce 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 coflows.

workload: We take the realistic coflow traces [22] synthesized from real-world MapReduce workloads collected from a 3000-machine, 150-rack Facebook cluster, which are widely used as a benchmark for coflow scheduling analysis [5], [6], [13], [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 that the transmitted bytes are recorded in reducer-level instead of flow-level, thus we apply two transformations to these traces. First, we partition the bytes of each reducer pseudo-randomly to each mappers. Second, we combine these coflows randomly into jobs that each job is expected to have β coflows. Dependencies between coflows within a job are also randomly generated.

Setup: We set that there exists 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 a

uplink (downlink) port of a machine with 1 Gbps bandwidth. Therefore, the total available bandwidth in this fabric is 2α Gbps.

B. isolation guarantee

In the 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, 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 lagerer the normalized JCT, the slower jobs will complete under the compared scheduler. If the normalized JCT is larger (smaller) than 1, the job completes faster (slower) in our scheduler.

C. Performance

We next evaluate the performance (i.e., average JCT) of our scheduler with other baseline schedulers.

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

ACKNOWLEDGMENT

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 7, pp. 107–113, Jan. 2008.

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

[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 the 9th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2012, pp. 2–2.

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

[5] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 443–454.

[6] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 393–406.

[7] 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 the 2016 ACM SIGCOMM Conference*, 2016, pp. 160–173.

[8] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 431–442.

[9] H. Susanto, H. Jin, K. Chen *et al.*, "Stream: Decentralized opportunistic inter-coflow scheduling for datacenter networks," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, 2016, pp. 1–10.

[10] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 407–424.

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *Nsdi*, vol. 11, no. 2011, 2011, pp. 24–24.

[12] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: sharing the network in cloud computing," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 187–198, 2012.

[13] L. Wang and W. Wang, "Fair coflow scheduling without prior knowledge," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018.

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

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

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS operating systems review*, vol. 41, no. 3, 2007, pp. 59–72.

[17] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[18] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *INFOCOM 2016-The 35th Annual 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 *INFOCOM 2017-IEEE 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 *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. 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 the 27th ACM symposium on Parallelism in Algorithms and Architectures*, 2015, pp. 294–303.