Bottleneck-Aware Coflow Scheduling Without Prior Knowledge

Libin Liu[†] Hong Xu[†] Chengxi Gao[‡] Peng Wang[§]
[†]City University of Hong Kong [‡]SIAT, CAS [§]Huawei

***Abstract*—Coflow scheduling is critical to the communication efficiency of data-parallel applications in data centers. While schemes like Varys can achieve optimal performance, they require a priori information about coflows which is hard to obtain in practice. Existing non-clairvoyant solutions like Aalo generalize least attained service (LAS) scheduling discipline to address this issue. However they fail to identify the bottleneck flows in a coflow and tend to allocate excessive bandwidth to the non-bottleneck flows within the coflow, leading to bandwidth wastage and inferior overall performance.**

To this end, we present Fai that strives to improve the overall coflow performance by accelerating the bottleneck flow without prior knowledge. Fai employs bottleneck-aware scheduling for coflows. Fai adopts loose coordination to update coflow priority and flow rates based on total bytes sent. In addition, Fai detects bottleneck flows based on a flow's rate and bytes sent, and de-allocates bandwidth for other flows to match the bottleneck rate without affecting the coflow completion time (CCT). The saved bandwidth is then distributed among coflows according to their priority to improve overall performance. Both testbed deployments and trace-driven simulations show that Fai outperforms Aalo substantially.

I. Introduction

Today's data centers host many data-intensive applications (e.g. MapReduce [1], Spark [2], and Dryad [3]) to meet the increasing demand for data processing and analytics [4], [5]. Studies have shown that the intermediate data transfer during shuffling accounts for a substantial part of job processing [6]–[9]. For example Facebook reports that transferring data between successive stages occupies 33% of the Hadoop jobs' running time during the reduce phase [6].

A data analytics job is composed of many stages including communication and computation [1]–[3], [10], [11]. Often a communication stage cannot finish until all its flows have completed [6], [7], [9]. *Coflow* is proposed as a new abstraction that captures this unique communication pattern of data-intensive applications [10]. Instead of considering individual flows and their performance, coflow considers the application-level semantics and applies to the *all-or-nothing* property of data analytics jobs [12]–[14]: a coflow includes all flows of the same communication stage and is completed only when all its flows finish. Thus better coflow completion time (CCT) directly leads to faster job completion time [6]–[9], [15].

Coflow scheduling emerges as an important research problem in our community. Varys [9] proposes heuristics such as smallest-bottleneck-first and smallest-total-size-first to minimize CCT, by assuming that *complete* information of coflows is known in advance. However, information like the coflow size and arrival times of its member flows is difficult to obtain a priori in practice [8], [15]–[17]. Take multi-stage jobs as an example: usually data are transferred as soon as they have been generated, making it almost impossible to obtain the flow size before the transmission ends. Aalo [8] and CODA [15], therefore, turn to information-agnostic scheduling. Aalo leverages the least attained service (LAS) scheduling discipline [18] and uses a coflow's total bytes sent across all its flows to prioritize it periodically. Then scheduling can be done independently at each end-host: coflows are dispatched according to weighted fair queueing, and within each priority simple FIFO is adopted. CODA [15] on the other hand adopts machine learning to identify coflows without application modifications.

Intuitively, CCT is determined solely by a coflow's slowest flow, i.e. the bottleneck flow; the other flows that finish earlier does not contribute to this coflow's CCT improvement, which is essentially a form of bandwidth wastage. Specifically, since existing information-agnostic schedulers deal with the member flows of a coflow without coordination, though these flows carry the same priority, the local resource contention they experience at each end-host can be vastly different. As a result, some flows may obtain more bandwidth and finish sooner as their hosts do not have coflows with higher priorities, while other flows may have less bandwidth and become stragglers. This is clearly inefficient since the extra bandwidth for the fast flows has no contributions to CCT and can be re-balanced to improve the performance of the stragglers for other coflows, or make the coflows with lower priorities run earlier at the same host, and improve the overall average CCT.

To address this issue, we present a novel bottleneck-aware non-clairvoyant coflow scheduler called Fai.[1] The central challenge is, how to detect a coflow's bottleneck flow and allocate bandwidth to avoid wastage, without any prior information? The flow with the smallest bandwidth in one scheduling period may not be the bottleneck since its size is unknown, and it may have much bandwidth in previous periods and have few bytes left to send. Another strawman approach is to pick the flow with the smallest total bytes sent so far as the bottleneck. This does not work well when the flows of a coflow have rather different sizes, which are common in practice. Fai relies on a simple and robust heuristic that combines the two metrics: it selects the ones with the smallest total bytes sent so far and the smallest bandwidth allocated currently as

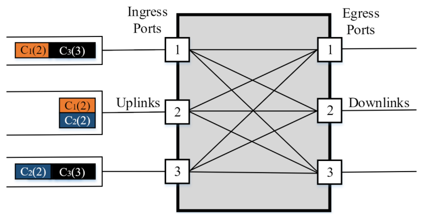
[1]Fai means fast in Cantonese.

the bottleneck flow(s). Fai then reduces the sending rates of other flows (of the same coflow) to the bottleneck flow rate to minimize the bandwidth wastage without degrading the coflow's performance. The reclaimed bandwidth is allocated to other coflows following their priority levels, to improve the network utilization and overall performance, e.g., minimizing CCT and coflow makespan[2]. When no flow satisfies the above criterion, Fai does nothing and continues to use the original rates computed by existing coflow schedulers.

II. Motivation

In this section, we use a toy example to illustrate the performance loss of existing non-clairvoyant schedulers in §II-A, thereby motivating the need to consider bottleneck flows. We further demonstrate the potential gain of bottleneck-aware non-clairvoyant scheduling using a Facebook workload in §II-B.

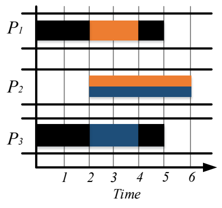
A. A Toy Example



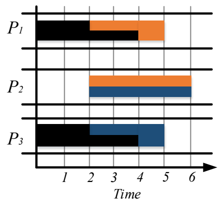
(a) A non-blocking data center fabric

C1	C2	C3
2	2	0

(b) Arrival times



(c) Aalo



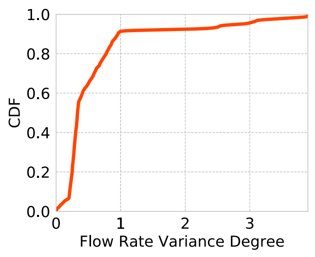
(d) Bottleneck-Aware

Fig. 1: (a) A non-blocking data center fabric with 3 ingress ports and 3 egress ports. There are 3 coflows in this example, C_1 in orange/light, C_2 in blue/dark and C_3 in black. The numbers in the brackets on flows represent the units of data on the corresponding link. Each port can transfer one unit of data in one time unit. (b) Coflow arrival times for the three coflows. (c) Aalo scheduling. (d) Bottleneck-Aware allocation.

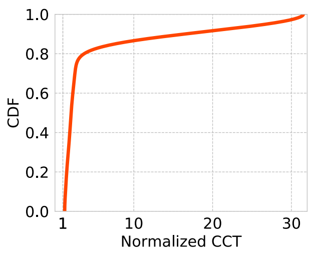
We now use a toy example to demonstrate why state-of-the-art non-clairvoyant coflow scheduler, Aalo, may not work well due to the lack of consideration of bottleneck flows. In the example, we abstract the network as a large *non-blocking* switch with uplinks and downlinks connected to the end hosts as in Fig. 1(a) and bandwidth contention only occurs at the edge (egress and ingress ports).

Suppose there are 3 coflows in the 3-machine data center fabric. The sizes of their flows on each link are shown in Fig. 1(a), and the arrival times are shown in Fig. 1(b). Fig. 1(c) shows the scheduling results of Aalo. When C_1 first arrives at time 2 (C_3 has already sent 4 units of data), it has the highest

[2]Makespan is defined as the time elapsed to complete all submitted coflows.



(a) CDF of FRVD



(b) CDF of Optimal NCCT

Fig. 2: CDFs of flow rate variance degree (FRVD) in Aalo and optimal normalized CCT for Aalo against Fai.

priority, and its flow preempts C_3 on P_1 to fully occupy the link. Similarly, when C_2 arrives, its flow preempts C_3 on P_3 and uses the whole link. C_1 and C_2 share the link on P_2 since they both have the highest priority. At time 4, C_1's flow on P_1 completes and C_2's flow on P_3 completes. Then C_3 uses the whole link until time slot 5 when its flows complete on P_1 and P_3. C_1 and C_2 continue to share the link to P_2 until time slot 6. As a result, the CCTs for C_1, C_2 and C_3 are 4, 4, and 5, respectively.

In fact, when C_1 and C_2 arrive, their flows get 0.5 unit of bandwidth on P_2 and 1 unit of bandwidth on P_1 and P_3. Thus, the bottleneck for C_1 and C_2 is on P_2. Though their flows on P_1 and P_3 finish earlier, it does not improve their CCT at all. Hence it is wasteful to allocate so much bandwidth to them on P_1 and P_3. Instead, we can allocate bandwidth to C_1 and C_2 by 0.5 on P_1 and P_3, so that they have the same bandwidth as their bottleneck flows on P_2. Fig. 1(d) shows the corresponding scheduling results. C_3's CCT is improved by 20% to 4 without hurting the other coflows at all.

B. Empirical Analysis

We now empirically quantify the degree of non-uniform bandwidth allocation for flows within a coflow in Aalo as a result of its local per-flow bandwidth allocation on different hosts. We conduct a simulation run using *Coflow-FB* workload and record each flow's size and each coflow's completion time in Aalo. More details about the trace workload and settings can be found in §IV-A. Based on the results, we obtain each flow's average rate and the *flow rate variance degree (FRVD)* for each coflow, which is defined as the difference between the largest and smallest flow rates for all flows within a coflow normalized by its median flow rate. Fig. 2(a) shows the CDF of FRVDs. We observe the average, median, 90%ile, and 99%ile FRVD are 0.63, 0.33, 0.96, and 3.90, respectively. Note that the *Coflow-FB* workload [8] reports that the member flows have the same size for about 80% of its coflows. Thus the FRVD results clearly demonstrate that it is common in Aalo to allocate varying bandwidth to the flows of the same coflow. Naturally, bottleneck flow always exists for each coflow and excessive allocation for the non-bottleneck flows leads to severe wastage.

We further quantify the performance gain that can be obtained by allocating bandwidth in a bottleneck-aware manner. To answer this, we use the same *Coflow-FB* workload and run an *ideal* non-clairvoyant bottleneck-aware scheduler as follows: Firstly Aalo is used to schedule coflows in a non-clairvoyant way, and after all coflows finish we identify each coflow's true bottleneck flow and its average rate. We then re-adjust the rates of all other flows to the bottleneck rate of this coflow without affecting its CCT, and allocate the reclaimed bandwidth to other coflows following their priority levels to reduce their completion time. This represents the potential CCT gain for Fai. Fig. 2(b) shows the ideal scheduler's normalized CCT against Aalo (CCT in Aalo / CCT in ideal). Compared to Aalo, ideal bottleneck-aware scheduling can improve the average, 50%ile, 90%ile, and 99%ile CCT by 4.98×, 2.16×, 20.39×, and 40.25×, respectively. The results indicate that there is much performance gain for Fai to realize compared to state of the art.

III. Design

In general, Fai builds upon LAS scheduling, and relies on a bottleneck detection mechanism and bandwidth allocation algorithms to schedule coflows in addition.

A. Overview

Fig. 3 shows the overview of Fai. There are two main components in the system:

- **Coordinator**. The coordinator is a logically centralized entity; in practice it can be an independent process on a dedicated CPU core, or multiple processes on machines to manage a large-scale data center. It performs coflow scheduling every $O(10)$ milliseconds [8], [9], [15], which is called a scheduling epoch (or period). At each epoch, the coordinator collects the coflow information from the Fai daemons, updates coflow status, and assigns them to different priority queues based on their total bytes sent. It then computes per-flow bandwidth allocation and dispatches the decisions to the Fai daemons on each end host.
- **End hosts.** Each end host in Fai is a coflow sender and receiver. It runs a local daemon to monitor the runtime status of active coflows and reports the information to the coordinator at each scheduling epoch. They also enforce bandwidth allocation by using rate limiters on individual flows. Scalable software rate limiters have been deployed in production [19] and we omit the related discussion in this paper.

The Fai coordinator has a small number of n queues in total, from the highest priority queue Q_1 to the lowest priority queue Q_n as shown in Fig. 3. We adopt the coflow size thresholds experimentally determined based on a production workload [8]. The threshold to demote coflows is $Q_i = Q_1 \times 10^i$ where $i \in [1, n-1]$ and $Q_1 = 10MB$. There are 10 priority queues.

B. Bandwidth Allocation

With the coflow priority, bandwidth allocation in Fai is performed in three steps, including (1) initial allocation, (2) bottleneck detection and bandwidth update, and (3) remaining bandwidth reallocation.

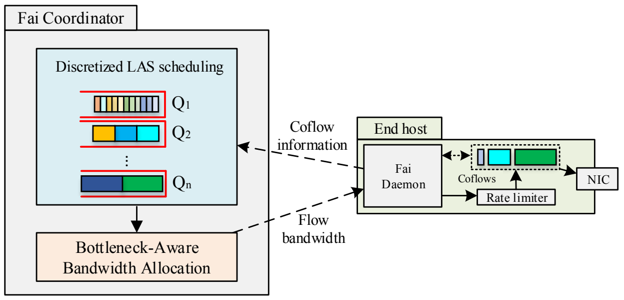


Fig. 3: Fai overview

Algorithm 1 Initial Allocation

```
1:  for $L \in \{L_I\} \bigcup \{L_E\}$ do
2:      $L_{bw} = L_{capacity}$
3:  Update total weight: $W = \sum Q_i.weight$
4:  for $i = 1$ to $n$ do
5:      $Q_i.bw = L_{bw} * Q_i.weight/W$
6:  for $i = 1$ to $n$ do
7:      for $C_j \in Q_i$ do
8:          for $f_k \in C_j$ do
9:              $f_k.bw =$ Fair-sharing $Q_i.bw$
10:             $L^I_{bw} = L^I_{bw} - f_k.bw$
11:             $L^E_{bw} = L^E_{bw} - f_k.bw$
12:             $Q_i.bw = Q_i.bw - f_k.bw$
```

Step 1: Initial Allocation. Fai first allocates bandwidth according to weighted fair sharing (WFQ) across all priority queues. Within each queue, it uses FIFO scheduling. Algorithm 1 shows the detail. Fai first sets the available bandwidth on each sender and receiver link to the link capacity (lines 1 and 2). Then it allocates bandwidth according to WFQ across all priority queues (lines 3 to 5). Next, starting from the highest priority queue Q_1, Fai picks a coflow according to FIFO and on each link, fair-shares this queue's available bandwidth among all flows of this coflow (lines 6 to 9). It finally updates the queue's available bandwidth and the link's remaining bandwidth (lines 10 to 12).

Step 2: Bottleneck Detection and Bandwidth Update. As quantified in §II-B, flows of a coflow can be assigned different bandwidths on different links from the initial allocation in Step 1. For a clairvoyant scheduler [9], it assumes that it has complete prior information including the number of coflows, the number of flows inside each coflow, and per-flow size. This makes it easy to find the bottleneck flow which requires the longest time to complete. However, for non-clairvoyant coflow scheduling, none of the prior information is known.

An intuitive method to determine the bottleneck is to select the flow(s) with the smallest total bytes sent until the current epoch. This would work when all flows of the coflow have the same size. This depends on the workloads. Although *Coflow-FB* workload reports that about 80% coflows have their flows with the same size, in practice it is not uncommon that flows have varying sizes. For example, Varys [9] shows that flow

Algorithm 2 Bottleneck Detection and Bandwidth Update

```
1: for $i = 1 \ to \ n$ do
2:     for $C_j \in Q_i$ do
3:         $min\_bytes = \infty$, $min\_bw = \infty$, $F_{min} = \varnothing$
4:         for $f_k \in C_j$ do
5:             if $f_k.bytes\_sent < min\_bytes$ then
6:                 $min\_bytes = f_k.bytes\_sent$
7:             if $f_k.bw < min\_bw$ then
8:                 $min\_bw = f_k.bw$
9:         for $f_k \in C_j$ do
10:            if $f_k.bytes\_sent == min\_bytes$ then
11:                $F_{min} = F_{min} \bigcup \{f_k\}$
12:        for $f_k \in F_{min}$ do
13:            if $f_k.bw == min\_bw$ then
14:                for $f \in C_j$ do
15:                    $f.bw = min\_bw$
16:                    Update $L^I_{bw}$ and $L^E_{bw}$
17:                break
```

Algorithm 3 Remaining Bandwidth Reallocation

```
1: for $i = 1 \ to \ n$ do
2:     for $C_j \in Q_i$ do
3:         for $f \in C_j$ do
4:             for $f_k \in f$ do
5:                 $L_{left} = min\{L^I_{bw}, L^E_{bw}\}$
6:             Find the minimum $L^{min}_{left}$ for $f$ among all $L_{left}$
7:             for $f_k \in f$ do
8:                 $f_k.bw = f_k.bw + L^{min}_{left}$
9:                 $L^I_{bw} = L^I_{bw} - L^{min}_{left}$
10:                $L^E_{bw} = L^E_{bw} - L^{min}_{left}$
```

sizes in some coflows are highly skewed. In such cases, the *Min-Bytes* approach does not work well. Another strawman approach is to pick the flow(s) with the least bandwidth after Step 1 as the bottleneck. Such a minimum bandwidth method does not work well because the flow may have much bandwidth in the previous epochs and have progressed a lot with few bytes left to send.

Fai adopts a *minimum-bytes-and-bandwidth* (*MBAB*) approach that jointly considers the two metrics to identify the bottleneck, which outperforms the two strawman approaches significantly as we empirically show in our evaluation (§IV-B). Effectively *MBAB* is less prone to false positives and false negatives from using *Min-Bytes* and *Min-BW*. Algorithm 2 shows the logic of *MBAB*: Starting from the highest priority queue Q_1, Fai finds the least bytes sent and the least bandwidth of all flows inside each coflow (lines 4–8), and updates the set F_{min} whose flows have the least bytes sent (lines 9–11). If any flow in F_{min} has the least bandwidth, we set the bandwidth of other flows in this coflow to the bottleneck flow bandwidth, and the remaining bandwidth of the corresponding sender and receiver links is updated as well (lines 12–17). Otherwise, all flows' bandwidth allocation remains unchanged.

Step 3: Remaining Bandwidth Reallocation. From Step 2 there is extra bandwidth saved from withdrawing excessive allocation to the non-bottleneck flows. We thus need to allocate this saved bandwidth to coflows to improve their CCT. For this purpose we utilize Algorithm 3 based on updated bandwidth information from Algorithm 2. The bandwidth reallocation adopts strict priority scheduling. For each coflow, Fai iterates through all its flows, checks the remaining bandwidth on each local ingress and egress link (lines 4 and 5), and finds the minimum remaining bandwidth L^{min}_{left} (line 6). As a result, all member flows receive L^{min}_{left} bandwidth and the corresponding ingress and egress links' remaining capacities are updated (lines 8 to 10).

IV. Trace-Driven Simulation

In this section, we evaluate Fai through trace-driven simulation. We implement our scheduling logic in *CoflowSim* [20], a coflow simulator written by authors of Aalo [8] and Varys [9]. Our simulation addresses the following questions:

- How efficient is our *MBAB* heuristic for bottleneck detection and bandwidth update? (§IV-B)
- How well does Fai perform compared to existing non-clairvoyant coflow scheduler ? (§IV-C)
- How far away is Fai from a clairvoyant coflow scheduler with complete information? (§IV-C)

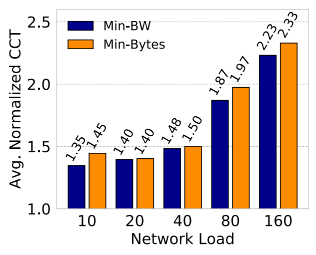
A. Setup

Topology. The data center fabric is modeled as a 150×150 non-blocking switch, where an ingress (egress) port corresponds to a 1Gbps uplink (downlink) of a rack [8], [9], [15], [21].

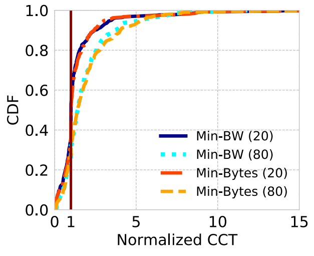
Methodology. To emulate realistic scenarios, we use the production workload from Facebook, which we call *Coflow-FB*. It has 526 coflows and is based on a one-hour Hive/MapReduce trace collected from a 3000-machine, 150-rack cluster. The workload contains coflow arrival information and we directly generate coflows according to it. To simulate various network loads, we vary the number of coflows arriving to the network in one time slot (10ms in our simulation) from 10, 20, 40, 80, to 160. Coflows arrive according to a Poisson process continuously instead of only appearing at the beginning of the time slot. Note that the original Coflow-FB workload has coflow arrival time information, to simulate various network loads, we modify the coflow arrival times accordingly.

Schemes Compared. First, to demonstrate the efficiency of the *MBAB* bottleneck detection heuristic, we compare it against other design choices (*Min-BW* and *Min-Bytes*) explored in §III-B. Subsequently, we compare Fai with two well-known coflow schedulers: Aalo [8] and Varys [9]. As explained, Aalo is a non-clairvoyant scheduler, and Varys is a clairvoyant one that uses complete knowledge of a coflow's individual flows. Aalo, therefore, serves as the performance baseline, while Varys the performance upper-bound.

Metrics Used. Our primary metric for comparison is the improvement in coflow completion time (CCT). We define it

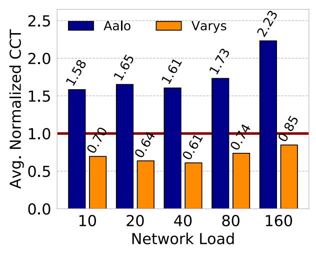


(a) Average normalized CCT

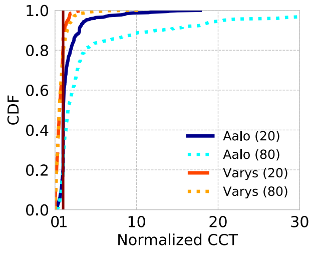


(b) CDF of normalized CCT

Fig. 4: Comparison of the average normalized CCT and CDF of the normalized CCT with network load 20 and 80.



(a) Average normalized CCT



(b) CDF of normalized CCT

Fig. 5: Normalized CCT comparison against Aalo and Varys and CDF of the normalized CCT with network load 20 and 80.

as the CCT under the compared scheme normalized by that under Fai.

$$\text{Normalized CCT} = \frac{\text{Compared CCT}}{\text{CCT under Fai}}.$$

Clearly, if the normalized CCT of a scheme is greater than one, Fai is faster than that scheme.

B. *Effectiveness of MBAB*

We first investigate the effectiveness of our key design choice, the *MBAB* heuristic for bottleneck detection in §III-B. We look at the normalized CCT, which is now defined as the CCT under the compared heuristic normalized by that under *MBAB*. Fig. 4 depicts the comparison results.

Fig. 4(a) shows *MBAB*'s average normalized CCT improvement compared to *Min-BW* and *Min-Bytes*. We observe that *MBAB* performs much better in all network loads. *MBAB* reduces the average normalized CCT by up to 2.23× over *Min-BW*, and by up to 2.33× over *Min-Bytes*. Across all network loads, on average *MBAB* reduces normalized CCT by 37.72% compared to *Min-BW* and 39.85% compared to *Min-Bytes*. We also explore the distributions of normalized CCT. In Fig. 4(b), we select the network load 20 (low load) and 80 (high load), and plot the CDF of normalized CCT for them. We can see that for the *majority* of coflows, *MBAB* provides improvements over the other two bottleneck detection methods. Fai improves performance of 52.85% and 51.33% of the coflows for network load 20 and 80, respectively, compared to *Min-BW* and *Min-Bytes*.

To summarize, *MBAB* performs more effectively than the two strawman strategies, which justifies its effectiveness in our design.

C. *Overall Performance*

We now investigate Fai's overall performance.

We first look at the average CCT reduction provided by Fai as shown in Fig. 5(a). Fai reduces the average normalized CCT by up to 2.23×, compared to Aalo. On average across the loads, Fai reduces normalized CCT by 1.76×. Clearly, Fai significantly outperforms state-of-the-art Aalo. This is expected as Fai detects the bottleneck flows and re-distribute the abundant bandwidth that are otherwise used without improving CCT in Aalo. We also evaluate the performance gap of Fai with respect to clairvoyant scheduling represented by Varys. Fig. 5(a) shows that Fai performs at most 64% and at least 18% worse than Varys. On average Fai is 40% and 41% worse.

Fig. 5(b) depicts the CDF of the normalized CCT under different schemes for network load 20 and 80. Observe that Fai substantially speeds up coflow completion over Aalo, and delivers similar CCT with Varys for more than half of the coflows. More than 84.38% and 73.39% coflows finish faster than Aalo at the two loads, respectively. Specially, more than 7.81% and 12.81% coflows finish faster in Fai than Varys at network load 20 and 80, respectively. This is because elephant flows are largely delayed in presence of mice ones in Varys compared to Fai.

V. Testbed Experiments

We prototype Fai in Scala and Python based on the open source Aalo prototype [8].

A. *Setup*

Our testbed experiments are deployed in a cluster with 40 machines connected to a 1GbE switch. Each machine has two 2.4GHz Intel Xeon 8-Core E5-2630 v3 processors, a 64GB DDR4 RAM, a 200GB SSD, and a quad-port Intel i350 GbE NIC.

Workload. We generate three types of coflows as the workload for testbed experiments. Each coflow has endpoints on all 40 machines with different communication patterns. Table I summarizes the information of three coflows. In particular, for coflow-A, we evenly divide its endpoints into 10 groups and each group does an all-to-all shuffle, which performs 4×4 pattern communication. As a result, coflow-A consists of 160 flows. Coflow-B follows a pairwise one-to-one communication pattern between machine i and machine $i+20$, where $1 \leq i \leq 20$. Thus, it consists of 40 flows. Coflow-C also includes 40 flows following the same communication pattern, in which the communication happens between machine j and machine $j+10$, where $1 \leq i \leq 10$ and $21 \leq i \leq 30$. The three coflows together have 240 flows, each of which is generated with a random size between 50MB and 100MB. In addition, coflow-A, coflow-B, and coflow-C arrive at 0s, 0.5s and 1s sequentially.

Coflows	Comm. Pattern	# of Flows	Arrival Time (s)
Coflow-A	all-to-all	160	0
Coflow-B	pairwise one-to-one	40	0.5
Coflow-C	pairwise one-to-one	40	1

TABLE I: Coflow information in our testbed.

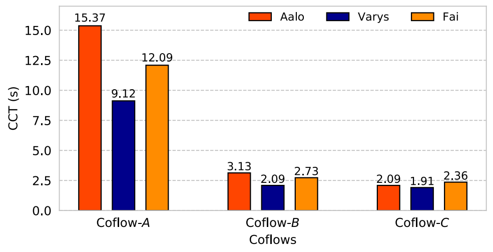


Fig. 6: CCTs of three coflows in our 40-machine cluster.

B. Experiment Results

Fig. 6 depicts the CCTs of three coflows using Aalo, Varys, and Fai. Fai is around 22.36% worse on average than Varys. More importantly, compared to Aalo Fai reduces the CCTs of coflow-A and coflow-B by 21.34% and 12.78%, respectively. Besides, it reduces the makespan by 21.34%. Fai performs worse than Aalo and Varys for coflow-C. We find that it is because Fai's *MBAB* bottleneck detection wrongly recognizes non-bottleneck flows as bottleneck, and the true bottleneck flow's bandwidth is reduced in the subsequent bandwidth adjustment and reallocation steps. This further demonstrates that correctly finding the real bottleneck is inherently difficult for a non-clairvoyant coflow scheduler. Therefore, designing a more efficient bottleneck flow detection strategy is an important future direction.

VI. Related Work

There has been much work on coflow scheduling. Here we discuss related work other than Aalo [8] and Varys [9] which have been discussed throughout this paper.

A series of work [7], [22] assumes that they know information about coflows a priori. Stream [21] is a decentralized scheduler, which utilizes many-to-one coflow patterns to coordinate coflows in a distributed manner. Although recent work has shown that it is possible to identify coflows and their properties with reasonable accuracy for some applications [15], in most production scenarios such prior information is impossible to obtain as explained in §I. Like Fai, several non-clairvoyant coflow schedulers have been developed to work in more practical scenarios. Other than Aalo, CODA [15] attempts to identify and schedule coflows without application modifications and designs an error-tolerant scheduler, making it applicable to many practical cases.

None of the above work considers adjusting the excessive bandwidth allocation according to the bottleneck flow, which is the focus of our work in this paper.

VII. Conclusion

We have presented Fai, a non-clairvoyant coflow scheduler. Fai improves the performance of a coflow's bottleneck without affecting other flows. We implement Fai and evaluate it on a 40-machine cluster as well as using large scale trace-driven simulations with production workloads. Both testbed experiments and trace-driven simulations show that Fai outperforms Aalo substantially. In the future, we will explore how well Fai performs in more complicated scenarios.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. USENIX OSDI*, 2004.

[2] 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 *Proc. USENIX NSDI*, 2012.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *Proc. ACM EuroSys*, 2007.

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proc. USENIX NSDI*, 2011.

[5] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proc. ACM SoCC*, 2013.

[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," in *Proc. ACM SIGCOMM*, 2011.

[7] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized Task-aware Scheduling for Data Center Networks," in *Proc. ACM SIGCOMM*, 2014.

[8] M. Chowdhury and I. Stoica, "Efficient Coflow Scheduling Without Prior Knowledge," in *Proc. ACM SIGCOMM*, 2015.

[9] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient Coflow Scheduling with Varys," in *Proc. ACM SIGCOMM*, 2014.

[10] M. Chowdhury and I. Stoica, "Coflow: A Networking Abstraction for Cluster Applications," in *Proc. ACM HotNets*, 2012.

[11] M. Grzegorz, H. A. Matthew, J. C. B. Aart, C. D. James, H. Ilan, L. Naty, and C. Grzegorz, "Pregel: A System for Large-Scale Graph Processing," in *Proc. ACM SIGMOD*, 2010.

[12] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *Proc. USENIX OSDI*, 2008.

[13] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *Proc. USENIX NSDI*, 2012.

[14] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters using Mantri," in *Proc. USENIX OSDI*, 2010.

[15] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "CODA: Toward Automatically Identifying and Scheduling COflows in the DArk," in *Proc. ACM SIGCOMM*, 2016.

[16] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein, "MapReduce Online," in *Proc. USENIX NSDI*, 2010.

[17] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a Compiler and Runtime for Heterogeneous Systems," in *Proc. ACM SOSP*, 2013.

[18] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack, "Analysis of LAS Scheduling for Job Size Distributions with High Variance," in *Proc. ACM SIGMETRICS*, 2003.

[19] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable Traffic Shaping at End Hosts," in *Proc. ACM SIGCOMM*, 2017.

[20] "CoflowSim," https://github.com/coflow/coflowsim.

[21] H. Susanto, H. Jin, and K. Chen, "Stream: Decentralized Opportunistic Inter-Coflow Scheduling for Datacenter Networks," in *Proc. IEEE ICNP*, 2016.

[22] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: Near-Optimal Network Design for Coflows," in *Proc. ACM SIGCOMM*, 2018.