Bottleneck-Aware Non-Clairvoyant Coflow Scheduling with Fai

Libin Liu, Chengxi Gao, Peng Wang, Hongming Huang, Jiamin Li, *Member, IEEE*, Hong Xu, *Senior Member, IEEE*, and Wei Zhang, *Member, IEEE*

Abstract—Coflow scheduling is critical to 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, 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 flows without priori knowledge. Fai employs bottleneck-aware scheduling. It 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. Testbed evaluation on a 40-node cluster shows that Fai improves average (P95) CCT by 1.73$\times$ (3.43$\times$), compared to Aalo. Large-scale trace-driven simulations also show that Fai outperforms Aalo substantially.

Index Terms—Coflow scheduling, Coflow completion time, Bottleneck-aware, Datacenter networks.

1 Introduction

TODAY'S data centers host many data-intensive applications (e.g., MapReduce [2], Spark [3], and Dryad [4]) to meet the increasing demand for data analytics [3]–[7]. Many studies [7]–[13] have shown that the intermediate data transfer during shuffling accounts for a substantial part of job processing. For example Facebook reports that data transmission between successive stages occupies 33% of the Hadoop jobs' running times during the reduce phase [8].

A data analytics job is composed of many stages including both communication and computation [2]–[4], [9], [14]. A communication stage usually cannot finish until all its flows have completed [8], [9], [11]. *Coflow* is thus proposed as an abstraction that captures this unique communication pattern of data-intensive applications [14]. Instead of considering individual flows, coflow considers the application-level semantics and applies to the *all-or-nothing* property of data analytics jobs [15]–[20]: a coflow includes all flows of the same communication stage and is completed only when all its flows finish. This way, better coflow completion time (CCT) directly leads to faster job completion [8]–[11], [21].

As a result, coflow scheduling emerges as an important research problem. Varys [11] 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 coflow size and arrival times of its member flows is difficult to obtain a priori in practice [10], [21]–[26]. 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. Thus, Aalo [10], Saath [27], and CODA [21] turn to information-agnostic scheduling. Aalo leverages the least attained service (LAS) scheduling discipline [28] and uses a coflow's total bytes sent across all flows (discretized into several levels) to prioritize it periodically. Then scheduling can be done independently at each host: coflows are dispatched according to weighted fair queueing, and within each priority simple FIFO is adopted. In addition, Saath [27] uses the same coflow priority structure as Aalo [10], yet takes into account the spatial dimension of coflow scheduling. It adopts *all-or-none* policy to avoid *out-of-sync* problem of flows within a coflow, and uses *least contention first* scheduling for the same coflow priority queue instead of FIFO. CODA [21] 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 do not contribute to this coflow's CCT improvement, which is essentially a form of bandwidth wastage. Specifically, existing information-agnostic schedulers deal with the member flows in-

- *This work was supported in part by the National Natural Science Foundation of China (No. 61802233), the Pilot Project for Integrated Innovation of Science, Education and Industry of Qilu University of Technology (Shandong Academy of Sciences) under Grant 2020KJC-ZD02, and a General Research Fund from Hong Kong Research Grants Council (No. 11209520).*
- *Libin Liu and Wei Zhang are with Shandong Provincial Key Laboratory of Computer Networks, Shandong Computer Science Center (National Supercomputer Center in Jinan), Qilu University of Technology (Shandong Academy of Sciences). Email: liulib@sdas.org, wzhang@sdas.org.*
- *Chengxi Gao is with the Institute of Advanced Computing and Digital Engineering Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China. Email: chengxi.gao@siat.ac.cn.*
- *Peng Wang is with Theory Lab, Huawei Hong Kong Research Center, Hong Kong SAR, China. Email: wang.peng6@huawei.com.*
- *Hongming Huang and Jiamin Li are with Department of Computer Science, City University of Hong Kong, Hong Kong SAR, China. Email: {honhuang7-c, jiaminli8-c}@my.cityu.edu.hk.*
- *Hong Xu is with Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR, China. Email: hongxu@cuhk.edu.hk.*
- *Part of this work was presented in the IEEE INFOCOM 2020 Workshop on Intelligent Cloud Computing and Networking (ICCN) [1]. The corresponding authors are Hong Xu and Wei Zhang.*

dependently without coordination. Though these flows carry the same priority, the local resource contention they experience at each 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 others 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 of Fai is, how can it detect a coflow's bottleneck and allocate bandwidth to avoid wastage without any prior information? Since the flow size is unknown, the flow with the smallest bandwidth in one scheduling period may not be the bottleneck since 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 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 this coflow's performance. The reclaimed bandwidth is allocated to other coflows following their priority levels in order to improve the network utilization and overall performance, e.g., 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.

We make several contributions in this work.

- We identify the need to consider bottleneck flows for better performance in non-clairvoyant scheduling and analyze the potential gain of re-distributing bandwidth in §2.
- We design Fai algorithms in §3 to schedule coflows and allocate bandwidth in a bottleneck-aware fashion. In each period, a centralized coordinator first updates coflow priority and flow rate based on LAS. Then it identifies the bottleneck flow of each coflow using the *minimum-bytes-and-bandwidth* (*MBAB*) heuristic, reduces other flows' rates to the bottleneck rate, and allocates the saved bandwidth to other coflows according to their priorities.
- We implement Fai and experimentally evaluate its performance in §4 and §5. Trace-driven simulations show that compared to state-of-the-art Aalo, Fai reduces the average CCT by up to 1.71× and 2.23×, respectively, for two production workloads. Testbed experiments on a 40-node cluster demonstrate that Fai also improves the average CCT by up to 2.11× over Aalo.

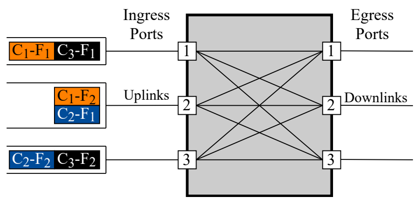
In the following, we first introduce the background for the non-blocking data center fabric and coflow scheduling, and motivation for the bottleneck-aware coflow scheduling in §2. Then, we describe system design, as well as the algorithms in §3. We discuss implementation of Fai in §4, and performance evaluation with extensive testbed experiments and large-scale simulations in §5. Finally, we discuss scalability in §6 and related work in §7, and conclude in §8.

1. Fai means fast in Cantonese.
2. Makespan is the time elapsed to complete all submitted coflows.

2 Background and Motivation

In this section, we first introduce the background about data center network, coflow, and coflow scheduling in §2.1. Then, we use a toy example to illustrate the performance loss of existing non-clairvoyant schedulers in §2.2, thereby motivating the need to consider bottleneck flows. We further demonstrate the potential gain of bottleneck-aware non-clairvoyant scheduling using a Facebook trace in §2.3.

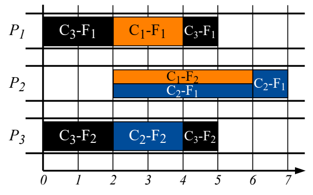
2.1 Background



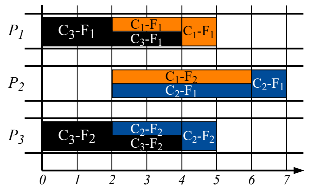
(a) A non-blocking data center fabric

Coflows	C_1		C_2		C_2	
Flows	F_1	F_2	F_1	F_2	F_1	F_2
Size	2	2	3	2	3	3
Arrival Time	2	2	2	2	0	0

(b) Sizes and arrival times



(c) Aalo



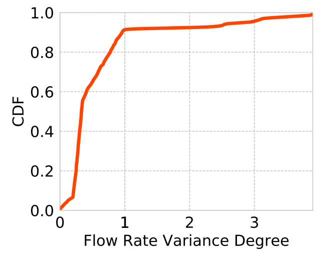
(d) Bottleneck-Aware

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. Each coflow has two flows, and C_x-F_y represents flow F_y of coflow C_x. The flows are on the corresponding links as shown, such as C_1-F_1 and C_3-F_1 on link 1 (P_1), C_1-F_2 and C_2-F_1 on link 2 (P_2), and C_2-F_2 and C_3-F_2 on link 3 (P_3). Each port can transfer one unit of data in one time unit. (b) The sizes and arrival times of flows within the three coflows. (c) Aalo scheduling. (d) Bottleneck-Aware bandwidth allocation. Noted that the priority queue transition threshold is 4.

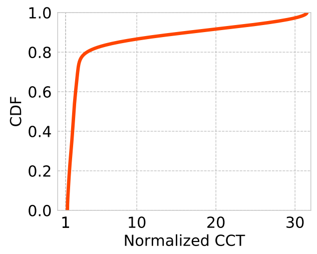
Data center network. Current data center network provides full bisection bandwidth [29]–[33]. Like previous work [10], [11], [21], we abstract the network as a large *non-blocking* switch with uplinks and downlinks connected to the end hosts. Bandwidth contention only occurs at the edge (the egress and ingress ports). Fig. 1(a) shows an 3×3 non-blocking data center fabric with 3 machines. All links have the same capacity.

Coflow. A coflow refers to a collection of parallel flows across a set of machines for one particular task [10], [11], [14], [21]. A coflow completes only when all its flows have completed.

Coflow Scheduling. For applications like MapReduce, the amount of data to transmit is known when a coflow starts. It is feasible to schedule coflows optimally using a clairvoyant scheduler such as Varys [11]. However, in most cases coflow characteristics are unknown a priori, e.g. in cases with multi-stage jobs using pipelining [10], a single stage consisting of multiple waves [16], and task failures resulting in redundant flows. We focus on practical non-clairvoyant coflow scheduling.



(a) CDF of FRVD



(b) CDF of optimal NCCT

Fig. 2: CDFs of flow rate variance degree (FRVD) in Aalo and optimal normalized coflow completion time (NCCT) for Aalo against Fai using *Coflow-FB* workload [10], [36].

2.2 A Toy Example

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 [34], [35] 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 and arrival times of their flows on each link 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 sent 4 units of data in total.), it has the highest priority, and thus flow C_1-F_1 preempts flow C_3-F_1 on P_1 to fully occupy the link. Similarly, when C_2 arrives, its flow C_2-F_2 preempts C_3-F_2 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 C_1-F_1 on P_1 completes and C_2's flow C_2-F_2 on P_3 completes. Then C_3 uses the whole link until time slot 5 when its flows C_3-F_1 and C_3-F_2 complete on P_1 and P_3. C_1-F_2 and C_2-F_1 continue to share the link to P_2 until time slot 6. Finally, C_2-F_1 uses the whole link until time slot 7, and then C_2 finishes. As a result, the coflow completion times (CCTs) for C_1, C_2 and C_3 are 4, 5, and 5, respectively.

In fact, when C_1 and C_2 arrive, their flows C_1-F_2 and C_2-F_1 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.

2.3 Empirical Analysis

FRVD	Avg.	50th	90th	99th
Aalo	0.63	0.33	0.96	3.90

TABLE 1: Statistical analysis of flow rate variance degree (FRVD) in Aalo using *Coflow-FB* workload [10], [36].

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 §5.1. Then 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 and Table 1 summarizes the corresponding statistical analysis. 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 [10], [36] reports that 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. Note that though our analysis here is based on *Coflow-FB* workload, the results hold for other trace data as we will show in §5.6.

NCCT	Avg.	50th	90th	99th
Aalo vs. *Ideal*	4.98	2.16	20.39	40.25

TABLE 2: Statistical analysis of normalized CCT comparison against Aalo using *Coflow-FB* workload [10], [36].

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: First 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 times. 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), and Table 2 summarizes the corresponding statistical analysis. 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, which motivates us to explore its design in §3 and evaluate its performance in §5.

3 Design

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

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

- **Fai coordinator**. The coordinator is the core of Fai. It 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 similar to other production schedulers [10], [11], [21]. We further demonstrate its efficiency in §5.2. More details about the scheduling epoch are explained in §4. At each epoch, the coordinator collects coflow information from the Fai slaves, updates coflow status,

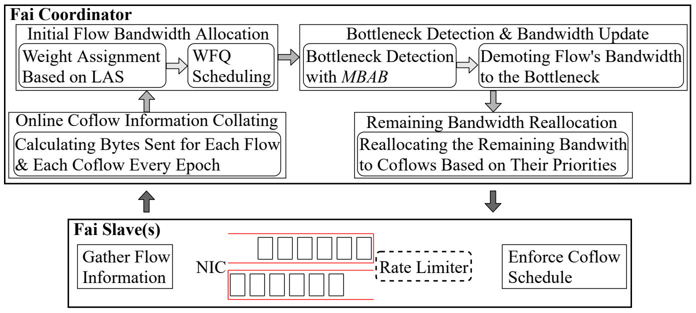

Fai Coordinator
Initial Flow Bandwidth Allocation
Weight Assignment Based on LAS
WFQ Scheduling
Bottleneck Detection & Bandwidth Update
Bottleneck Detection with MBAB
Demoting Flow's Bandwidth to the Bottleneck
Online Coflow Information Collating
Calculating Bytes Sent for Each Flow & Each Coflow Every Epoch
Remaining Bandwidth Reallocation
Reallocating the Remaining Bandwith to Coflows Based on Their Priorities
Fai Slave(s)
Gather Flow Information
NIC
Rate Limiter
Enforce Coflow Schedule

Fig. 3: Fai overview: Fai slaves collect flow-level information, and Fai coordinator periodically updates coflow schedules using our scheduling mechanisms.

assigns them to different priority queues based on their total bytes sent at initial flow bandwidth allocation stage, and computes per-flow bandwidth allocation by adopting weighted fair sharing. Next, it detects bottleneck flow(s) for each coflow with *MBAB* and demotes other flows' bandwidth to the bottleneck flow's bandwidth, which will be further explained in §3.2. Finally, the coordinator reallocates saved bandwidth according to coflows' priorities and dispatches the decisions to the Fai slaves on each end host.

- **Fai slaves.** Each slave is a coflow sender and receiver. It runs a local daemon to monitor the runtime status of active flows, calculates total bytes sent of each coflow, and reports it to the coordinator in each scheduling epoch. They also receive flow bandwidth allocation messages from the coordinator and enforce bandwidth allocation by using rate limiters on individual flows. Scalable software rate limiters have been deployed in production [37] and we omit the related discussion in this paper.

3.1 LAS Scheduling

The Fai coordinator utilizes the discretized LAS as the basic scheduling discipline. There are a small number of n queues in total, from the highest priority queue to the lowest priority queue. In general, a coflow experiences three types of events during its lifetime:

- **Arrival**. A coflow arrives to Fai when the data analytics application registers the coflow to the coordinator using its API. We use the same API implementation in Aalo [10] since we do not require any additional information from applications. The new coflow is enqueued into the highest priority queue as its total bytes sent is zero. Before the next epoch starts, flows of the new coflow simply fair-share the local remaining bandwidth on the links.
- **Demotion**. At a given epoch, once the coflow's total bytes sent exceeds the threshold of the queue, it is demoted to the next queue with a lower priority (similar to multi-level feedback queue), until it is enqueued to the lowest priority queue if it is large enough. Each flow is guaranteed to be allocated non-zero bandwidth and have progress at each epoch as Fai uses weighted fair queuing.
- **Completion**. Once a coflow completes, the analytics application de-registers it from Fai and it is removed from the coordinator immediately.

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:             $L_{bw}^{I_{f_k}} = L_{bw}^{I_{f_k}} - f_k.bw$
11:             $L_{bw}^{E_{f_k}} = L_{bw}^{E_{f_k}} - f_k.bw$
12:             $Q_i.bw = Q_i.bw - f_k.bw$
```

Notation	Explanation
L	Network link
L_I	Ingress network link
L_E	Egress network link
L_{bw}	The bandwidth of network link L
$L_{capacity}$	The capacity of network link L
W	The sum of weights for all priority queues
Q_i	The ith priority queue
$Q_i.bw$	The bandwidth of priority Q_i
$Q_i.weight$	The weight of priority Q_i
C_j	Coflow j
f_k	Flow k
$f_k.bw$	The allocated bandwidth of flow k
$L_{bw}^{I_{f_k}}$	The bandwidth of ingress network link L where flow f_k is on
$L_{bw}^{E_{f_k}}$	The bandwidth of egress network link L where flow f_k is on
min_bytes	The minimum bytes sent of a flow
min_bw	The minimum bandwidth obtained of a flow
F_{min}	The flow set for flows with the minimum bytes sent
$f_k.bytes\_sent$	The bytes sent of flow k
L_{left}	The remaining bandwidth of network link L
L_{left}^{min}	The minimum remaining bandwidth for all network links

TABLE 3: Notations in Fai's algorithms

We adopt the coflow size thresholds experimentally determined based on a production workload [10]. 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. We follow the setup for coflow priority queues in Aalo [10], which has been demonstrated that works well for *Coflow-FB* and *Coflow-320* workloads.

3.2 Scheduling Algorithm

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.

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 §2.3, flows of a coflow can be assigned different bandwidths on different links from the initial allocation in Step 1. For a clairvoyant scheduler [11], it assumes that it has complete

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 $f_k \in C_j$ do
15:                     $f_k.bw = min\_bw$
16:                     Update $L_{bw}^{I_{f_k}}$ and $L_{bw}^{E_{f_k}}$
17:                 break
```

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 [11] shows that flow 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 (§5.3). 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,

Algorithm 3 Remaining Bandwidth Reallocation

```
1:  $L_{left}^{min} = L_{capacity}$
2:  for $i = 1\ \ to\ \ n$ do
3:      for $C_j \in Q_i$ do
4:          for $f_k \in C_j$ do
5:              $L_{left} = min\{L_{bw}^{I_{f_k}}, L_{bw}^{E_{f_k}}\}$
6:              if $L_{left} < L_{left}^{min}$ then
7:                  $L_{left}^{min} = L_{left}$
8:          for $f_k \in C_j$ do
9:              $f_k.bw = f_k.bw + L_{left}^{min}$
10:             $L_{bw}^{I_{f_k}} = L_{bw}^{I_{f_k}} - L_{left}^{min}$
11:             $L_{bw}^{E_{f_k}} = L_{bw}^{E_{f_k}} - L_{left}^{min}$
```

checks the remaining bandwidth on each local ingress and egress link (lines 4 and 5), and finds the minimum remaining bandwidth L_{left}^{min} (lines 6 and 7). As a result, all member flows receive L_{left}^{min} bandwidth and the corresponding ingress and egress links' remaining capacities are updated (lines 8 to 11).

4 Implementation

Our prototype adopts the master-slave structure as shown in Fig. 3 and is based on the Aalo prototype [10]. Therefore, it natively supports Aalo's coflow API. The coordinator runs on the master machine with coflow information collected from slaves at the end-hosts. Its decisions are also enforced by the slaves.

We introduce the main components of our prototype now.

Fai Coordinator on the Master. The coordinator's main logic includes registering coflows, collecting coflow status, updating coflow priorities (LAS scheduling), bottleneck flow detection, bandwidth demotion, and bandwidth reallocation. It runs as a separate process. To implement our *MBAB* heuristic, we modify the bandwidth allocation function *getSchedule()* inside the *Dark-Scheduler* of Aalo prototype. We write three new functions, *InitRates()*, *BottleneckDetection()*, and *AllocRemains()*, to implement the three steps (Agorithms 1, 2, and 3) in §3.2.

Fai Slaves on End Hosts. Fai slaves monitor flows locally and send out flow status to the coordinator on each heartbeat. Besides, when the local daemon receives messages for bandwidth allocation, it enforces the specified flow rate by using Linux's *tc* and HTB *qdisc* tools. Specifically, we use the two-level HTB: the leaf nodes enforce per-flow rates and the root node classifies outgoing packets to their corresponding leaf nodes.

Choice of Scheduling Epoch. Fai slaves are more closely in sync as the epoch interval decreases. Based on our measurement in §5.2 (Fig. 8(a)) that a 40-machine CloudLab cluster can re-synchronize within 5.37 milliseconds on average, we recommend that the epoch period is O(10) milliseconds, which is similar to Aalo [10]. We show Fai's overall performance with different coordination periods in §5.2 (Fig. 8(b)).

Implementation Overhead of Fai Slaves. To measure the CPU and memory overheads of Fai slaves, we saturate the 1GbE NIC of our server with two 2.4GHz Intel Xeon 8-Core E5-2630 v3 processors and 64GB DDR4 RAM. The CPU overhead is around 1% compared with the Aalo daemons. Throughput remains the same in both cases.

5 Evaluation

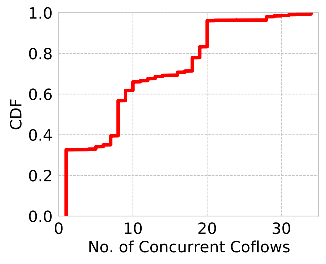
In this section, we evaluate Fai through a series of testbed experiments on a 40-machine cluster in CloudLab [38] using traces from both production clusters and an industrial benchmark. For large-scale evaluation, we use a trace-driven simulator that performs a detailed replay of task logs. Our evaluation seeks to answer the following questions:

- **How well does Fai perform compared to existing non-clairvoyant coflow schedulers, and what is its performance gap compared to a clairvoyant scheduler with complete information?** In §5.2, using *Coflow-320* and *Coflow-FB*, we show that Fai's improvements in average and 95th percentile CCTs over Aalo are at least 1.87× and 2.99×, respectively. Compared against clairvoyant scheduler Varys, Fai performs 11% worse on average, and 12.50% on 95th percentile.
- **How well does Fai scale in practice?** In §5.2, we show that Fai achieves reasonable scalability with slave daemons. We also show that the choice of 10ms coordination period achieves a good balance between performance and overhead.
- **How efficient is our *MBAB* heuristic for bottleneck detection and bandwidth update?** In §5.3, we can see that *MBAB* reduces the average normalized CCT by up to 1.66× and 2.23× over *Min-BW*, and by up to 1.71× and 2.33× over *Min-Bytes* for *Coflow-320* and *Coflow-FB* workloads, respectively.
- **How does Fai perform in large-scale simulations?** In §5.4, we observe that, on average across network loads, Fai reduces normalized CCT by 1.58× for *Coflow-320* workload and 1.76× for *Coflow-FB* workload.
- **How does Fai perform with various flow size distributions within a coflow?** In §5.6, we show that with varying flow sizes, the normalized CCT stabilizes 1 with at most 8% difference.

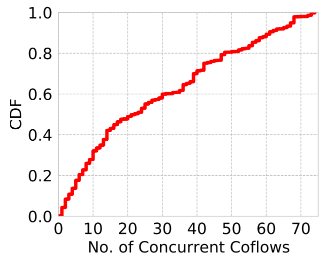
5.1 Evaluation Settings

Testbed. Our testbed experiments are deployed in a cluster in CloudLab with 40 machines connected to a 1GbE switch. Each machine has two 2.4GHz Intel Xeon 8-Core E5-2630 v3 processors, 64GB DDR4 RAM, 200GB SSD, and a quad-port Intel i350 GbE NIC. All machines run Ubuntu 16.04.2 LTS. We use the same compute engine as both Aalo [10] and Varys [11].

Testbed Workloads. We use two workloads from production environments. The first one we adopt is the same as the one used in prior work [10], [11], [21]. It is based on a one-hour Hive/MapReduce trace collected from a 3000-machine, 150-rack Facebook production cluster [36] and includes 526 coflows. Fig. 4(a) shows the distribution of the number of concurrent coflows for this *Coflow-FB* workload. The original cluster had a 10:1 core-to-rack oversubscription ratio and a total bisection bandwidth of 300Gbps. In our experiments, we scale down jobs accordingly to match the maximum possible 40Gbps bisection bandwidth of our deployment while preserving their communication characteristics. Besides, the second includes 320 coflows generated by the *CustomTraceProducer* in *CoflowSim* [39] according to a production trace collected from Microsoft cluster. We call it *Coflow-320* in the following. *Coflow-320* is based on a hybrid trace of web search and other services collected from 6000-machine, 150-rack Microsoft production clusters. In these clusters, each server connects to a Top of Rack switch (ToR) via 1Gbps Ethernet. Fig. 4(b) shows the distribution of the number of concurrent coflows for this *Coflow-320* workload. We also scale *Coflow-320* to match with the 40Gbps bisection bandwidth.



(a) *Coflow-FB*



(b) *Coflow-320*

Fig. 4: Workload characteristics for both *Coflow-FB* and *Coflow-320* workloads.

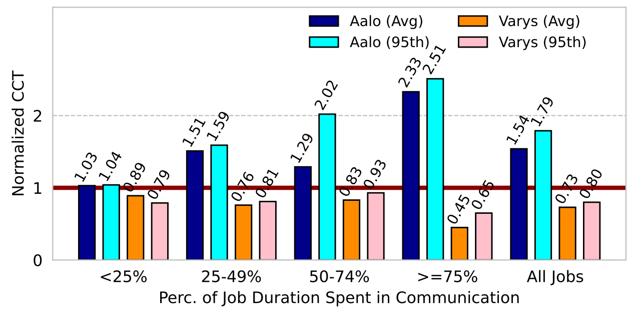
% of Coflows	SN	LN	SW	LW
Coflow-320	12	38	12	38
Coflow-FB	60	16	12	12

TABLE 4: Coflows binned by their length (Short or Long) and width (Narrow or Wide) for both *Coflow-FB* and *Coflow-320* workloads. SN represents short and narrow coflows, LN long and narrow coflows, SW short and wide coflows, and LW long and wide coflows.

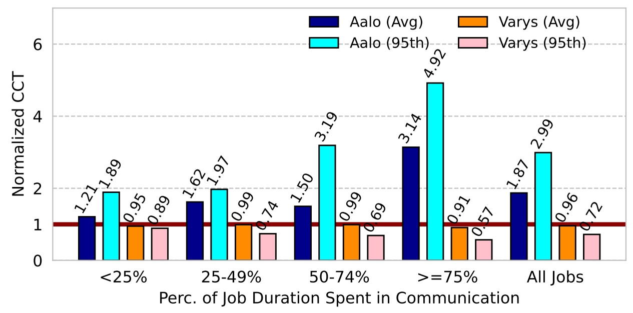
Simulator. For large-scale simulations, we used a trace-driven flow-level simulator that performs a detailed task-level replay of the traces. It preserves input-to-output ratios of tasks, locality constraints, and inter-arrival times between jobs, and it runs at 1s decision intervals. Besides, the data 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. This is commonly used in existing work [10], [11], [21], [40]–[42].

Simulation Workloads. Our simulations use the same two workloads used in the testbed experiments. To simulate various loads, we vary the number of coflows arriving to the network in one time slot (10ms in our simulation). Coflows arrive according to a Poisson process continuously instead of only appearing at the beginning of the time slot. Rather than directly using *Coflow-FB*, we manually modify the coflow arrival times to produce various traffic loads. *Coflow-FB* contains the concrete coflow arrival time for each load which we directly use for generating coflows. For *Coflow-320*, we generate five traces for each load and launch coflows accordingly. Thus for each scenario we conduct one individual run for *Coflow-FB* and conduct five independent runs and report the average results for *Coflow-320*. Both workloads combine all mappers (resp. reducers) under the same rack into one rack-level mapper (resp. reducer), as production clusters are oversubscribed in core-rack links [43], [44] and simulating rack-level is sufficient for them. Noted that flow sizes of 20% coflows in *Coflow-FB* workload are skewed [11].

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*) we explored in §3.2. Subsequently, we compare Fai with two well-known coflow schedulers: Aalo [10] and Varys [11]. As explained Aalo is a non-clairvoyant scheduler, and Varys is clairvoyant and uses complete knowledge of a coflow's individual flows. Therefore, Aalo serves

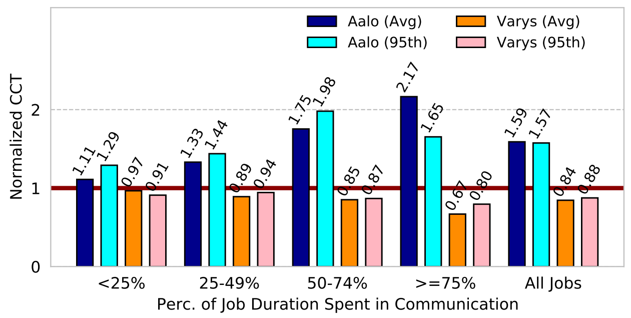


(a) Improvements in job completion times (JCTs)

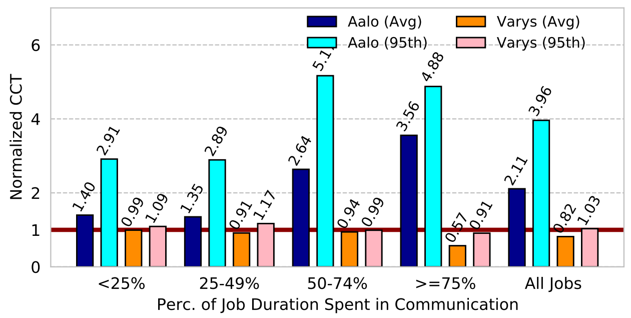


(b) Improvements in time spent on communication

Fig. 5: [Testbed] Average and 95th percentile improvements in job and communication completion times using Fai over Aalo and Varys with *Coflow-320* workload. Job completion time (JCT) is the end-to-end time including communication time and computation time.



(a) Improvements in job completion times (JCTs)



(b) Improvements in time spent on communication

Fig. 6: [Testbed] Average and 95th percentile improvements in job and communication completion times using Fai over Aalo and Varys with *Coflow-FB* workload. Job completion time (JCT) is the end-to-end time including communication time and computation time.

as the baseline while Varys the performance upper-bound. We do not compare against Sincronia [45] either as it is a clairvoyant scheduler and Varys can represent it.

Metrics Used. Our primary comparison metric is the improvement in coflow completion time (CCT). We define it as the CCT under the compared scheme normalized by that under Fai.

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

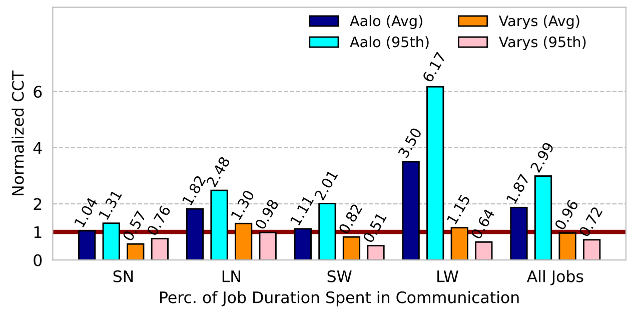
Clearly, if the normalized CCT of a scheme is greater than one, Fai is faster than that scheme.

5.2 Testbed Experiments

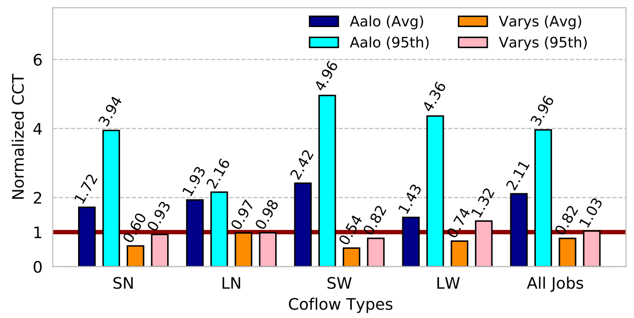
Overall Performance. Figs. 5 and 6 present the results by grouping jobs based on their time spent in communication using both *Coflow-320* and *Coflow-FB* workloads. For *Coflow-320*, Fig. 5(a) shows that compared to Aalo, Fai reduces the average and 95th percentile job completion times (JCTs) by at least 1.03× and 1.04× and at most 2.33× and 2.51×, respectively. Fai improves average JCT by 1.54× and 95th percentile JCT by 1.79×. Besides, Fig. 5(b) shows Fai's improvements in terms of average and 95th percentile CCT over Aalo are at least 1.21× and 1.89×, and are at most 1.87× and 2.99×, respectively, across the jobs. For *Coflow-FB*, Fig. 6(a) shows that compard to Aalo, Fai reduces the average and 95th percentile JCT by at least 1.11× and 1.29× and at most 2.17× and 1.98×, respectively. Fai improves average JCT by 1.59× and 95th JCT by 1.57×. In addition, Fig. 6(b) shows Fai's corresponding improvements in terms of average and 95th percentile CCT over Aalo are at least 1.35× and 2.89×, and are 2.11× and 3.96×, respectively, across the jobs. We can see that for jobs with longer relative communication time Fai can deliver higher gains.

Fig. 7 illustrates the gains by differentiating coflows according to their lengths and widths. The coflow distinguishments (bins) are shown in Table 4 for both *Coflow-FB* and *Coflow-320* workloads. *SN* represents short and narrow coflows, *LN* long and narrow ones, *SW* short and wide, and *LW* long and wide. As Fig. 7(a) shows, using *Coflow-320* workload, across the bins, average and 95th percentile CCTs are improved at least 1.04× and 1.31×, and at most 3.50× and 6.17× by Fai, respectively, compared to Aalo. Using *Coflow-FB* workload, across the bins, average and 95th percentile CCTs are improved 1.43× and 2.16×, and 2.42× and 4.96× by Fai, respectively, compared to Aalo. We observe that Fai performs best for LW coflows for *Coflow-320* and SW coflows for *Coflow-FB*, because Fai can quickly detect bottleneck flows and improve performance more pertinently.

To understand the performance gap between Fai and clairvoyant solutions, we compare Fai with Varys, which schedules coflows using complete prior information. For *Coflow-320*, Fig. 5(a) shows that the gap is at most 55% for average JCT and 35% for 95th percentile JCT. And for *Coflow-FB*, Fig. 6(a) shows that the gap is at most 16% for average JCT and 12% for 95th percentile JCT. Fig 5(b) and Fig. 6(b) further show that Fai performs worse whenever the fraction of time spent on communication is higher. Furthermore, Fig. 7 helps explain where Fai performs worse than Varys. For the large coflows (LN and LW), Fai performs similarly as Varys. For coflows from LW bins, Fai even performs better than Varys on average for *Coflow-320* and at 95th percentile for *Coflow-FB*, since Varys schedules short flows first to minimize CCT.

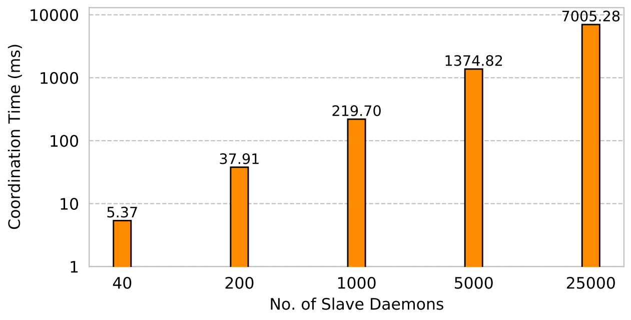


(a) *Coflow-320*

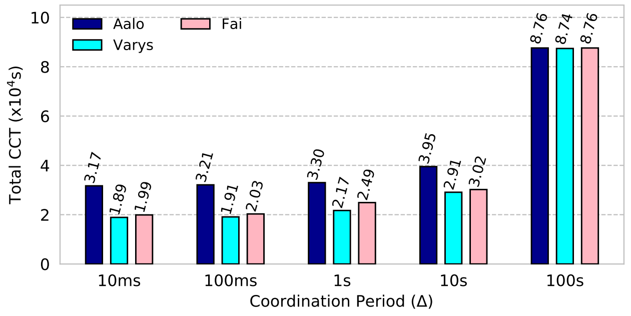


(b) *Coflow-FB*

Fig. 7: [Testbed] Improvements in the average and 95th percentile CCTs using Fai over Aalo and Varys. Here coflows are binned by their length and width as shown in Table 4. SN represents short and narrow coflows, LN long and narrow coflows, SW short and wide coflows, and LW long and wide coflows.



(a) Overheads at scale



(b) Impact of △

Fig. 8: [Testbed] Fai scalability: (a) more slave daemon processes require longer coordination periods (Y-axis is in log scale), and (b) longer coordination period hurts overall performance (measured as sum of CCT). Noted that we use *Coflow-FB* workload here.

Scalability. To evaluate Fai's scalability, we run different scales of slave daemons. Fig. 8(a) shows the average coordination time for various number of slave daemons. Here, n slaves imply that there are $n/40$ slave daemons one each machine of our testbed. We get the coordination time by transferring information for 100 coflows from each daemon to coordinator and send the coordination information back. We find that Fai's scalability is good but slightly worse than Aalo as expected. This is because of its bottleneck flow detection procedure.

In addition, we vary the coordination period (Δ) to see the impact on performance and support our choice of its value in our experiments. Fig. 8(b) shows the total CCT with different Δ values. Fai, Aalo, and Varys all perform worse with a larger Δ. When $\Delta > 100$s, they all perform similarly. Thus 10ms is a reasonable choice to achieve a good balance between performance and overhead.

5.3 Effectiveness of *MBAB*

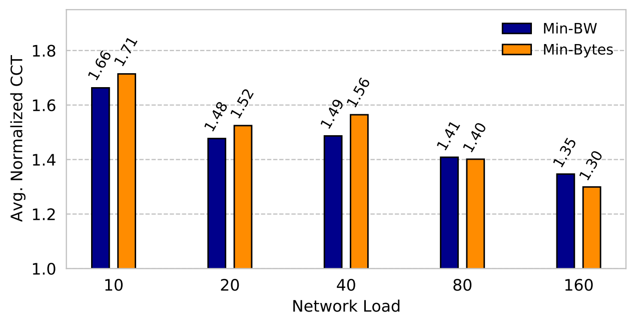
We first investigate the effectiveness of our key design choice, the *MBAB* heuristic for bottleneck detection in §3.2. We look at the normalized CCT, which is now defined as the CCT under the compared heuristic normalized by that under *MBAB*. Fig. 9 and 10, and Table 5 and 6 depict the comparison results.

Normalized CCT. Fig. 9 shows *MBAB*'s average normalized CCT improvement compared to *Min-BW* and *Min-Bytes* using the two workloads. We observe that *MBAB* performs much better in all network loads. For *Coflow-320*, *MBAB* reduces the average normalized CCT by up to 1.66× over *Min-BW*, and by up to 1.71× over *Min-Bytes*, as shown in Fig. 9(a). For *Coflow-FB*, *MBAB* reduces the average normalized CCT by up to 2.23× over *Min-BW*, and by up to 2.33× over *Min-Bytes*, as shown in Fig. 9(b). Across all network loads, on average *MBAB* reduces normalized CCT by 32.02% compared to *Min-BW* and 32.66% compared to *Min-Bytes* using *Coflow-320* workload, and 37.72% and 39.85% correspondingly using *Coflow-FB* workload.

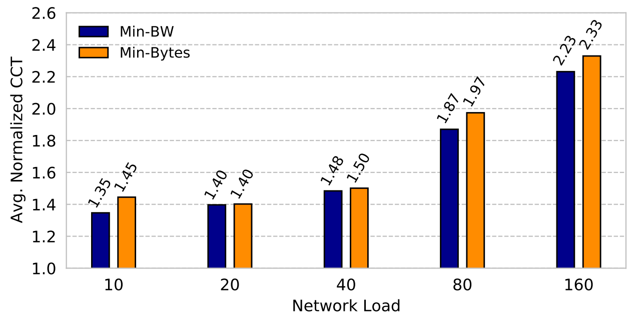
Besides, we can see Fig. 9(a) and Fig. 9(b) present different trends, where normalized CCT decreases with the increase of network loads in Fig. 8(a) while it increases as the network load increases in Fig. 8(b). This is because the flow size ditributions are quite different for *Coflow-320* and *Coflow-FB* workloads. From Table 4, we can see that for *Coflow-320*, 76% of all the coflows are long coflows, and for *Coflow-FB*, only 28% are long coflows. The benefit of more accurate bottleneck detection by *MBAB* for long coflows lowers, as long flows can tolerate inaccurate bottleneck detection for some epochs with network load increasing. However, short flows become more sensitive to the accuracy of bottleneck detection with network load increasing. Thus Fig. 9 shows diffetent trands for both workloads.

Normalized CCT Distribution. We also explore the distributions of normalized CCT in the simulation. In Fig. 10, we select the network load 20 (low load) and 80 (high load), and plot the CDFs of normalized CCT for them. We can see that for the *majority* of coflows, *MBAB* provides improvements over the other two bottleneck detection methods. For the *Coflow-320* workload at network load 20, Fai outperforms *Min-BW* for more than 67.81% of the coflows, and outperforms *Min-Bytes* for more than 71.25% coflows. For *Coflow-FB* workload, Fai improves performance of 52.85% and 51.33% of the coflows, respectively, compared to *Min-BW* and *Min-Bytes*.

Tables 5 and 6 further present the median, 90th, 95th, and

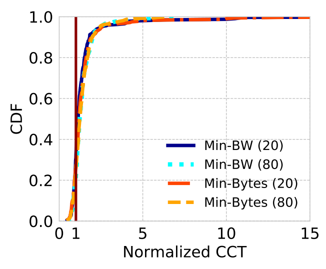


(a) *Coflow-320*

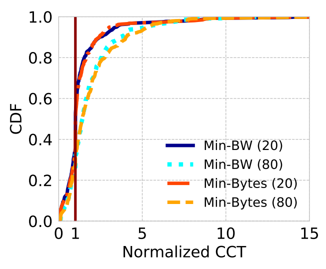


(b) *Coflow-FB*

Fig. 9: [Simulation] Comparison of the average normalized CCT using *Coflow-320* and *Coflow-FB*. We report the average results of five traces for *Coflow-320* workload.



(a) *Coflow-320*



(b) *Coflow-FB*

Fig. 10: [Simulation] CDFs of the normalized CCT with network load 20 and 80 using *Coflow-320* and *Coflow-FB* workloads respectively. We report the average results of five traces for *Coflow-320* workload.

99th percentile normalized CCT statistics for the two workloads. We can see that, for *Coflow-320* workload, *MBAB* consistently outperforms *Min-BW* and *Min-Bytes*; for *Coflow-FB* workload, *MBAB* outperforms *Min-BW* and *Min-Bytes* except at network load 10 and 20. On average across all loads, *MBAB* outperforms *Min-BW* and *Min-Bytes* by 1.20× and 1.23× at the median, 2.05× and 2.11× at the 90th percentile, 2.78× and 2.83× at the 95th percentile, and 6.93× and 7.07× at the 99th percentile for *Coflow-320* workload. The corresponding improvements are 1.28× and 1.28× at the median, 3.14× and 3.28× at the 90th percentile, 4.19× and 4.14× at the 95th percentile, and 8.15× and 8.41× at the 99th percentile in *Coflow-FB* workload.

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

Stats	10		20		40		80		160	
	BW	Byte	BW	Byte	BW	Byte	BW	Byte	BW	Byte
Median	1.10	1.13	1.13	1.19	1.21	1.29	1.23	1.25	1.34	1.31
90%	2.52	2.59	1.83	2.0	1.84	1.95	2.02	1.99	2.04	2.03
95%	4.53	4.33	2.58	2.83	2.16	2.28	2.39	2.46	2.26	2.24
99%	12.53	12.21	10.52	10.0	4.20	5.72	4.71	4.45	2.67	2.96

TABLE 5: [Simulation] Statistics of the normalized CCT using *Coflow-320* compared to *Min-BW* and *Min-Bytes*. Results are averaged over five runs.

Stats	10		20		40		80		160	
	BW	Byte	BW	Byte	BW	Byte	BW	Byte	BW	Byte
Median	1.0	1.0	1.0	1.0	1.13	1.10	1.42	1.40	1.85	1.90
90%	2.0	1.99	2.50	2.44	3.05	3.30	3.75	4.04	4.41	4.62
95%	3.16	2.85	3.50	3.12	3.50	3.75	5.24	5.36	5.56	5.67
99%	10.21	11.67	8.93	8.93	6.0	5.60	8.13	8.01	7.46	7.86

TABLE 6: [Simulation] Statistics of the normalized CCT using *Coflow-FB* compared to *Min-BW* and *Min-Bytes*.

5.4 Overall Performance

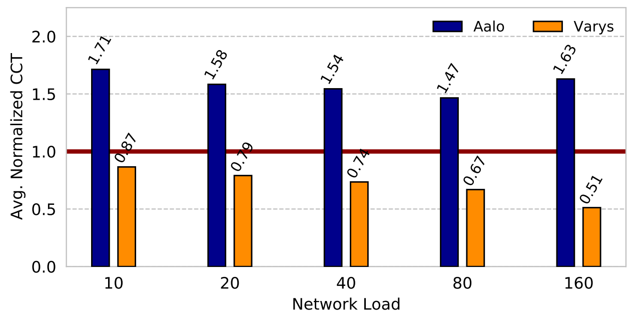
We now investigate Fai's overall performance in terms of normalized CCT, average link utilization, and flow completion time (FCT) variance within a coflow.

Normalized CCT. We first look at the average CCT reduction provided by Fai as shown in Fig. 11. In Fig. 11(a), for *Coflow-320* workload, Fai reduces average CCT by 1.71×, 1.58×, 1.54×, 1.47×, and 1.63×, respectively at different loads compared to Aalo. For *Coflow-FB* workload, Fig. 11(b) shows that Fai reduces the average CCT by 1.58×, 1.65×, 1.61×, 1.73×, and 2.23×, respectively compared to Aalo. On average across the loads, Fai reduces normalized CCT by 1.58× for *Coflow-320* workload and 1.76× for *Coflow-FB* workload. 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.

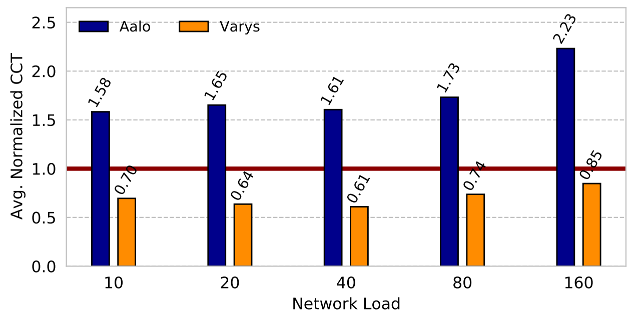
We also show the performance gap of Fai with respect to clairvoyant scheduling represented by Varys. Fig. 11(a) shows that Fai performs at most 96% and at least 15% worse than Varys for the *Coflow-320* workload. Fig. 11(b) shows that Fai performs at most 64% and at least 18% worse than Varys for the *Coflow-FB* workload. On average Fai is 40% and 41% worse than Varys.

Normalized CCT Distribution. Fig. 12 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. For *Coflow-320*, more than 82.50% and 74.38% coflows performs better in Fai than Aalo at network load 20 and 80, respectively, as shown in Fig. 12(a). More than 12.81% and 7.81% 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. For *Coflow-FB*, more than 84.38% and 73.39% coflows finish faster than Aalo at the two loads, respectively, as shown in Fig. 12(b).

Tables 7 and 8 show the statistics of normalized CCT in different network loads. Fai outperforms Aalo significantly across all four statistical metrics. Specificially, across all loads, the median is improved by 1.33×; the 90th percentile improvement is 2.28×, 95th percentile 3.07×, and 99th percentile 5.76×, for the *Coflow-320* workload. For the *Coflow-FB* workload, the numbers are 1.41×, 6.76×, 10.48×, and 22.88×. In addition, we see that Fai performs worse than Varys in terms of median normalized CCT because Varys has the complete coflow knowledge. In terms of 90th, 95th, and 99th percentiles, Fai actually reduces CCT by

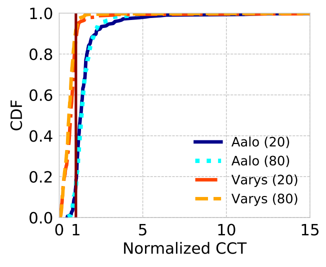


(a) *Coflow-320*

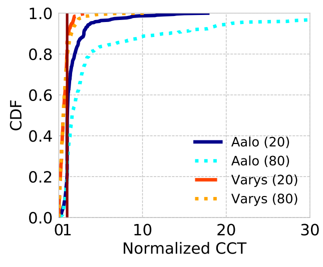


(b) *Coflow-FB*

Fig. 11: [Simulation] Normalized CCT comparison against Aalo and Varys.



(a) *Coflow-320*



(b) *Coflow-FB*

Fig. 12: [Simulation] CDFs of the normalized CCT with network load 20 and 80 using *Coflow-320* and *Coflow-FB* workloads respectively. We report the average results of five traces for *Coflow-320* workload.

Stats	10		20		40		80		160	
	Aalo	Varys	Aalo	Varys	Aalo	Varys	Aalo	Varys	Aalo	Varys
Median	1.22	0.73	1.29	0.69	1.32	0.69	1.33	0.58	1.52	0.53
90%	2.71	1.07	2.21	1.05	2.05	1.07	2.08	0.97	2.33	1.03
95%	4.69	1.53	3.25	1.16	2.41	1.16	2.48	1.10	2.54	1.11
99%	11.0	10.33	6.41	3.22	4.0	1.74	4.28	1.53	3.10	1.51

TABLE 7: [Simulation] Statistics of the normalized CCT compared to Aalo and Varys in *Coflow-320*. All results are averaged over five runs.

Stats	10		20		40		80		160	
	Aalo	Varys	Aalo	Varys	Aalo	Varys	Aalo	Varys	Aalo	Varys
Median	1.01	0.71	1.02	0.56	1.14	0.50	1.61	0.53	2.29	0.61
90%	2.10	1.07	3.0	1.17	3.32	1.23	13.27	1.49	12.10	1.55
95%	3.39	1.39	3.79	1.59	3.87	1.51	20.71	2.02	20.63	2.31
99%	12.60	2.09	11.71	2.41	6.10	2.29	45.20	4.12	38.77	6.46

TABLE 8: [Simulation] Statistics of the normalized CCT compared to Aalo and Varys in *Coflow-FB*.

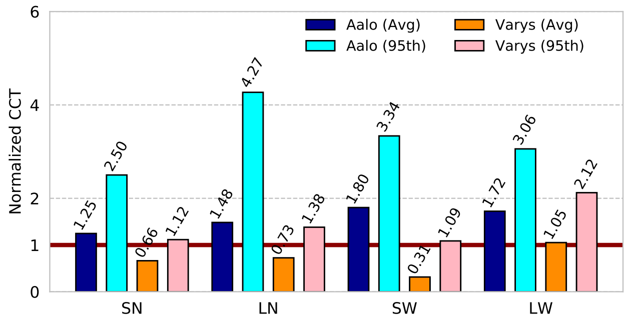


Fig. 13: [Simulation] Average normalized CCT across different coflow bins using *Coflow-320* (network load 20).

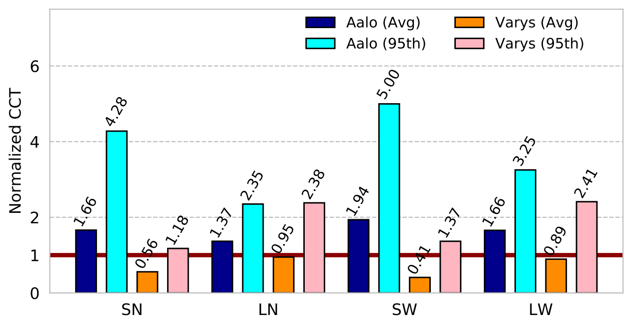


Fig. 14: [Simulation] Average normalized CCT across different coflow bins using *Coflow-FB* (network load 20).

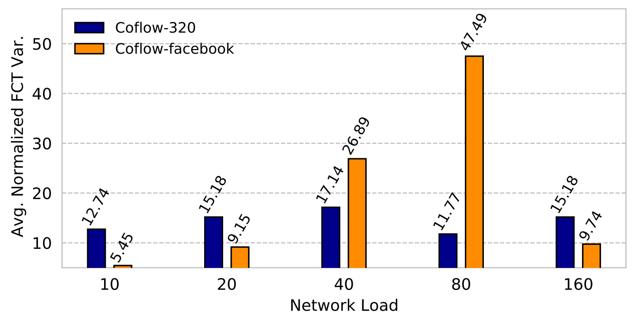
large margins compared to Varys. This is because Varys always tries to minimize the average CCT while sacrificing the fairness.

Coflow Bins. To better understand the performance impact of Fai on different coflow workloads, we categorize coflows into four bins based on their shuffle types. Specifically, we say a coflow is *small* (*long*) if its largest flow is less (greater) than 5MB, and *narrow* (*wide*) if it consists of less (more) than 50 flows [10], [11], [46]. Table 4 summarizes the distribution of binned coflows. We then plot the average and 95th percentile normalized CCT in four coflow bins as shown in Table 4 in Fig. 13 for *Coflow-320* and in Fig. 14 for *Coflow-FB*. We see that Fai consistently outperforms Aalo across all bins, Fai's average and 95th percentile CCT is up to 1.80× and 1.94× better for *Coflow-320* and 4.27× and 5× better for *Coflow-FB*, respectively. Note that the coflows in the SW bin have the largest CCT gap between Fai and Varys. This is because coflows with large width and small length tend to be scheduled poorly using non-clairvoyant schedulers. In contrast, Varys performs worse than Fai in terms of 95th percentile normalized CCT, especially for LN and LW bins. This is because Varys delays large coflows in presence of small ones. The results also explain why Fai has different performance gains in different workloads each with a unique combination of shuffle types.

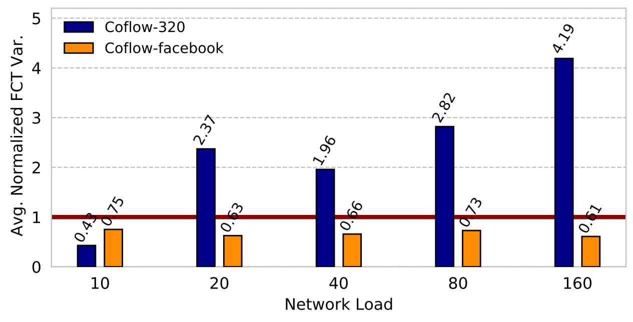
Network Utilization. We now evaluate the average link utilizations in Fai, Aalo, and Varys. Table 9 summarizes the results across the entire simulation time. We can clearly see that all three schemes have the same link utilization essentially: the maximum difference is 7%, which happens with load 10 between varys and aalo in *Coflow-FB* workload. This demonstrates that, Fai does not improve the CCT performance by using more bandwidth resources. Instead, it improves the CCT performance by better utilizing the bandwidth resource compared to Aalo which may allocates excessive bandwidth to non-bottleneck flows.

Network Load (%)		10	20	40	80	160
Coflow-320	Fai	74	76	77	78	79
	Aalo	73	75	76	77	78
	Varys	77	79	80	81	82
Coflow-FB	Fai	75	75	75	75	74
	Aalo	73	74	75	75	74
	Varys	80	79	81	79	79

TABLE 9: [Simulation] Average link utilization across time in Fai, Aalo, and Varys.



(a) Aalo



(b) Varys

Fig. 15: [Simulation] Average normalized FCT variance using two workloads. We report the average results of five traces for *Coflow-320* workload.

FCT Variance. We now look at the flow completion time (FCT) variance within a coflow under different schedulers. FCT variance is defined as the difference between the largest and smallest FCTs of flows within a coflow. A small FCT variance indicates the flows within a coflow finish at around the same time, which is preferred, and a large FCT variance means some flows lag behind significantly. We log each flow's completion time, and use the normalized FCT variance as the comparison metric, which is defined for each coflow as the FCT variance under the compared scheme divided by that under Fai. Fig. 15 shows the average normalized FCT variance across different loads using *Coflow-320* and *Coflow-FB* workloads. Observe that Fai always outperforms Aalo in the two workloads, since the normalized FCT variance is much larger than 1. Fai also outperforms Varys across all loads excluding load 10 for *Coflow-320* workload. This is interesting as Varys also tries to reduce the FCT variance. Yet, this phenomenon appears because of the coflow distributions of *Coflow-320* (LW bin has 38% coflows). Specifically, in Fig. 15(a), on average Fai can improve by 14.40× for *Coflow-320* and 19.74× for *Coflow-FB* across all loads compared to Aalo. For *Coflow-320* workload, the maximum normalized FCT variance is 17.14; for *Coflow-FB* workload, it is 47.49. In Fig. 15(b), for *Coflow-320* worload, on average Fai can improve the normalized FCT variance by 2.35× compared to Varys. Besides, Fai can improve the normalized FCT by up to 4.19×. Though Fai does not always perform better than Varys, especially in the *Coflow-FB* workload, it only performs worse than Varys by 0.32× on average.

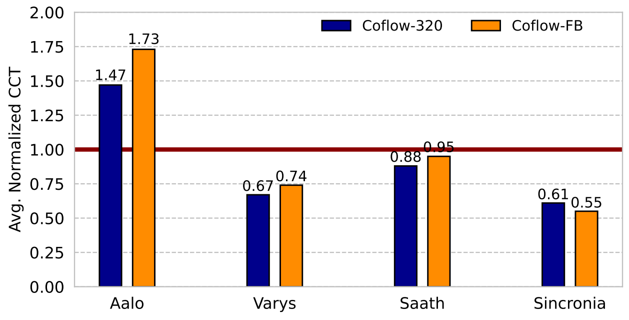


Fig. 16: [Simulation] Average normalized CCT using Fai over other scheduling policies with *Coflow-320* and *Coflow-FB* workloads. Noted that the network load is 80.

5.5 Comparing Fai against Aalo, Varys, Saath, and Sincronia in One Figure

In order to investigate the performance gap between Fai and other scheduling policies including Aalo, Varys, Saath [27], and Sincronia [45], we evaluate the average normalized CCT. Saath is a non-clairvoyant coflow scheduler and Sincronia is a clairvoyant one. Fig. 16 depicts the simulation results. We observe that, for *Coflow-320*, Fai improves average normalized CCT by 47% compared to Aalo, and performs 33%, 12%, and 39% worse compared to Varys, Saath, and Sincronia, respectively. For *Coflow-FB*, Fai improves average normalized CCT by 73% compared to Aalo, and performs 26%, 5%, and 45% worse compared to Varys, Saath, and Sincronia, respectively.

The results further confirms Fai's effectiveness over Aalo. Fai performs similarly as Saath, and their designs may be integrated to further improve coflows' performance.

5.6 Impact of Flow Size Variance within a Coflow

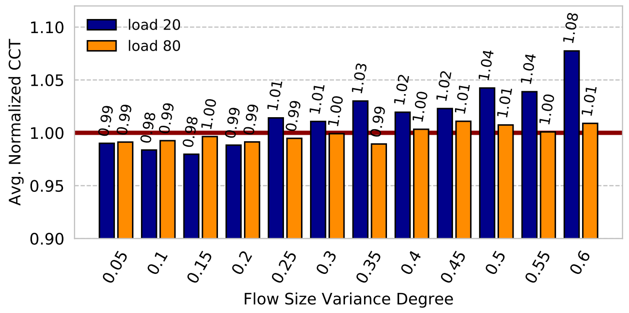


Fig. 17: [Simulation] Average normalized CCT with varying flow size variance in the *Coflow-320* workload.

In the previous experiments, flows within a coflow have the same size. One may wonder if Fai is still able to provide benefits when flows within a coflow have different sizes. To evaluate this, we vary the flow sizes while keeping the total shuffle bytes of the coflow unchanged in each trace. Specifically, we vary the degree of flow size variance which is the average size difference between the largest and smallest flows within a coflow normalized by its median flow size. Normalized CCT is now defined as CCT in Fai with the varying flow sizes normalized by CCT when all flows have identical size. We show the results using *Coflow-320* workload with loads of 20 and 80 in Fig. 17, when the degree of flow size variance changes from 0.05 to 0.6 (that is, on average the largest/smallest flow is 30% larger/smaller than the median flow of the coflow). Results using the *Coflow-FB* workload are quantitatively the same and omitted for brevity. We see that flow size variance hardly impacts Fai's CCT performance: the normalized CCT stabilizes at 1 with at most 8% difference. Thus, it demonstrates that Fai consistently delivers favorable performance gains even when flows within a coflow have fairly different sizes.

The reason for Fai's robustness to flow size distribution is its ability to identify bottleneck flows. As we analyzed before in §3.2, our *MBAB* algorithm is designed to deal with non-uniform flow sizes within a coflow by jointly considering flows' total bytes sent until current epoch and the allocated bandwidth. In §5.3, the simulation results also demonstrate *MBAB*'s effectiveness in identifying bottleneck flows.

6 Discussion

Scalability. Fig. 8 shows Fai's scalability running as a centralized system with 25000 machines, whose coordination time is 7005.28ms ($<$10s). To cope with even larger scale, one can deploy Fai in a distributed manner. As topologies of data center networks are based on multi-rooted trees [47], we can partition a large network into multiple smaller autonomous regions [48], each of which is handled by one Fai coordinator. Besides, we can also simplify it by using sampling based methods to estimate coflow and flow size, and infer the global bottleneck flow(s) of a coflow using SJF scheduling like Philae [49], in order to further improve scalability.

7 Related Work

There has been much work on coflow scheduling. We present Fai first in the conference paper [1]. Here we revise the design to make each component modular (§3), re-implement it to make it run in our testbed (§4), and evaluate it using testbed experiments and large-scale simulations (§5).

We discuss related work other than Aalo [10] and Varys [11] which have been discussed throughout this paper.

Clairvoyant Scheduling. A series of work [9], [45], [50] assumes that they know information about coflows a priori. Some work such as [51]–[53] proposes approximation algorithms to improve the performance of schedulers. Utopia [40] focuses on isolation and providing fairness guarantee in coflow scheduling. Stream [41] is a decentralized scheduler, which utilizes many-to-one coflow patterns to coordinate coflows in a distributed manner. Sunflow [54] adopts both intra-coflow and inter-coflow scheduling to approach the optimal for optical circuit switched network.

Although recent work has shown that it is possible to identify coflows and their properties with reasonable accuracy for some applications [21], in most production scenarios such prior information is impossible to obtain as explained in §1.

Non-Clairvoyant Scheduling. Like Fai, several non-clairvoyant coflow schedulers have been developed to work in more practical scenarios. Other than Aalo, CODA [21] attempts to identify and schedule coflows without application modifications and designs an error-tolerant scheduler, making it applicable to many practical cases. Besides, Saath [27] and Fai both adopt the same coflow priority structure as Aalo [10], and take into account the spatial dimension of coflow scheduling as well. Fai differs from Saath in the following aspects. Firstly, Saath uses *all-or-nothing* policy to avoid the out-of-sync problem of flows. In contrast, Fai utilizes weighted fair sharing across queues and fair sharing for the same queue, which guarantees flows of a coflow at different ports can get scheduled at the same time. Second, Saath adopts least contention first scheduling for the same priority queue and decides the coflows to be scheduled first, while Fai schedules all the coflows and tries to reallocate bandwidth based on coflow priorities and explicitly restricts the allocated bandwidth of each flow to solve the contention at each port. Graviton [55] aims to improve the coflow performance by adopting different scheduling policies in different queues. NC-DRF [42] and Coflex [56] try to provide isolation guarantees between contending coflows. While the above work has to modify applications for coflow schedulers.

Necessity of Centralized Solutions. Similar to [10], [11], [45], Fai requires a centralized coordinator to implement LAS scheduling. In contrast, Baraat [9] is a FIFO-based decentralized coflow scheduler focusing on small coflows. It uses fair sharing to avoid head-of-line blocking and does not support deadlines. Philae [49] proposes to explicitly learn coflow sizes online by sampling and reports to the SJF scheduler using the estimated coflow sizes. Fai can use its design philosophy to improve the scalability. While the amount of computation required by Fai's centralized coordinator is much lower according to practical experiments. It is still an open problem to understand how well a centralized scheduler performs in the large-scale production clusters.

Coflows with Routing. Fai, similar to [9], [11], [45], assumes the routing of flows within and across coflows are decided by the network layer (e.g., specific per-flow routing scheme or packet spraying). Yet, it is a priori conceivable that better performance can be achieved by co-designing coflow scheduling with routing. This problem has been studied in much recent work [50], [57]–[59]. DBA [60] proposes a distributed bottleneck-aware algorithm to schedule coflows to deal with in-network bottlenecks and approximate the minimum remaining time first. Fai can also extend to cope with in-network bottlenecks.

Other Performance Metrics. Finally, Fai is designed to improve the average coflow completion time, which is similar to the most of existing work on coflow scheduling [10], [11], [21], [45], [61]. There are many other interesting questions to explore other coflow metrics, such as considering the case of deadline-sensitive coflows with minimizing tail coflow completion time and fairness for coflows.

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.

8 Conclusion

We have presented Fai, a non-clairvoyant coflow scheduler. Fai is based on the LAS scheduling discipline and improves the performance of a coflow's bottleneck without affecting other flows. We implemented and evaluated Fai on a 40-machine cluster and using large scale trace-driven simulations with production workloads. Both testbed experiments and trace-driven simulations showed that Fai outperforms Aalo substantially.

This work takes a new step towards finding the local bottleneck flow within each coflow for distributed coflow scheduling. Although *MBAB* in Fai works well, it also opens up exciting research challenges on the theoretical underpinning, such as analysis about bottleneck flow detection and on characterizing the performance gap between non-clairvoyant and clairvoyant schedulers.

References

[1] L. Liu, H. Xu, C. Gao, and P. Wang, "Bottleneck-Aware Coflow Scheduling Without Prior Knowledge," in *Proc. Workshop on Intelligent Cloud Computing and Networking (ICCN), IEEE INFOCOM*, 2020.

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

[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 *Proc. USENIX NSDI*, 2012.

[4] 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.

[5] "Apache Hadoop," http://hadoop.apache.org.

[6] "Apache Hive," http://hive.apache.org.

[7] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker, "Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks," in *Proc. ACM SOSP*, 2017.

[8] 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.

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

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

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

[12] K. Morton, M. Balazinska, and D. Grossman, "ParaTimer: A Progress Indicator for MapReduce DAGs," in *Proc. ACM SIGMOD*, 2010.

[13] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *Proc. USENIX NSDI*, 2015.

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

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

[16] 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.

[17] 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.

[18] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proc. EuroSys*, 2013.

[19] "Apache Storm," http://storm.apache.org.

[20] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics," in *Proc. USENIX NSDI*, 2016.

[21] 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.

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

[23] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "PIAS: Practical information-agnostic flow scheduling for data center networks," in *Proc. USENIX NSDI*, 2015.

[24] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing Flows Quickly with Preemptive Scheduling," in *Proc. ACM SIGCOMM*, 2012.

[25] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling Mix-flows in Commodity Datacenters with Karuna," in *Proc. ACM SIGCOMM*, 2016.

[26] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proc. USENIX NSDI*, 2010.

[27] A. Jajoo, R. Gandhi, Y. C. Hu, and C.-K. Koh, "Saath: Speeding up Coflows by Exploiting the Spatial Dimension," in *Proc. ACM CoNEXT*, 2017, pp. 439–450.

[28] 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.

[29] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proc. ACM SIGCOMM*, 2008.

[30] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *Proc. ACM SIGCOMM*, 2009.

[31] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A Fault-Tolerant Engineered Network," in *Proc. USENIX NSDI*, 2013.

[32] A. Andreyev, "Introducing data center fabric, the next-generation Facebook data center network," https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, November 2014.

[33] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *Proc. ACM SIGCOMM*, 2015.

[34] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. M. B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-Optimal Datacenter Transport," in *Proc. ACM SIGCOMM*, 2013.

[35] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards Predictable Datacenter Networks," in *Proc. ACM SIGCOMM*, 2011.

[36] "Coflow-Benchmark," https://github.com/coflow/coflow-benchmark.

[37] 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.

[38] "CloudLab," https://www.cloudlab.us/.

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

[40] L. Wang, W. Wang, and B. Li, "Utopia: Near-optimal Coflow Scheduling with Isolation Guarantee," in *Proc. IEEE INFOCOM*, 2018.

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

[42] L. Wang and W. Wang, "Fair Coflow Scheduling without Prior Knowledge," in *Proc. IEEE ICDCS*, 2018.

[43] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical Network Performance Isolation at the Edge," in *Proc. USENIX NSDI*, 2013.

[44] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric," in *Proc. ACM CoNEXT*, 2015.

[45] 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.

[46] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-Resource Fairness for Correlated and Elastic Demands," in *Proc. USENIX NSDI*, 2016.

[47] C. COLS, "A Study of Non-blocking Switching Network," *Bell System Technology Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[48] J. A. Hawkinson and T. J. Bates, "Guidelines for Creation, Selection, and Registration of An Autonomous System (AS)," RFC 1930, Mar. 1996. [Online]. Available: https://rfc-editor.org/rfc/rfc1930.txt

[49] A. Jajoo, Y. C. Hu, and X. Lin, "Your Coflow has Many Flows: Sampling Them for Fun and Speed," in *Proc. USENIX ATC*, 2019, pp. 833–848.

[50] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks," in *Proc. IEEE INFOCOM*, 2015.

[51] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li, "Towards Practical and Near-optimal Coflow Scheduling for Data Center Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3366–3380, 2016.

[52] J. Jiang, S. Ma, B. Li, and B. Li, "Adia: Achieving High Link Utilization with Coflow-Aware Scheduling in Data Center Networks," *IEEE Transactions on Cloud Computing*, 2016.

[53] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the Total Weighted Completion Time of Coflows in Datacenter Networks," in *Proc. ACM SPAA*, 2015.

[54] X. S. Huang, X. S. Sun, and T. E. Ng, "Sunflow: Efficient Optical Circuit Scheduling for Coflows," in *Proc. ACM CoNEXT*, 2016, pp. 297–311.

[55] A. Jajoo, R. Gandhi, and Y. C. Hu, "Graviton: Twisting Space and Time to Speed-up CoFlows," in *Proc. USENIX HotCloud*, 2016.

[56] W. Wang, S. Ma, B. Li, and B. Li, "Coflex: Navigating the Fairness-Efficiency Tradeoff for Coflow Scheduling," in *Proc. IEEE INFOCOM*, 2017.

[57] J. Jiang, S. Ma, B. Li, and B. Li, "Tailor: Trimming Coflow Completion Times in Datacenter Networks," in *Proc. IEEE ICCCN*, 2016.

[58] Y. Li, S. H.-C. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou, and F. Lau, "Efficient Online Coflow Routing and Scheduling," in *Proc. ACM MobiHoc*, 2016.

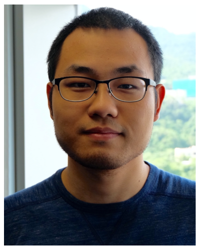
[59] H. Tan, S. H.-C. Jiang, Y. Li, X. Li, C. Zhang, Z. Han, and F. C. M. Lau, "Joint Online Coflow Routing and Scheduling in Data Center Networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1771–1786, 2019.

[60] T. Zhang, R. Shu, Z. Shan, and F. Ren, "Distributed Bottleneck-Aware Coflow Scheduling in Data Centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. pp, no. 99, pp. 1–1, 2018.

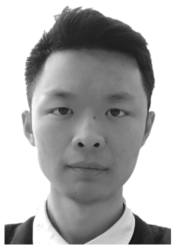
[61] C. Zhang, H. Tan, C. Xu, X. Li, S. Tang, and Y. Li, "Reco: Efficient Regularization-Based Coflow Scheduling in Optical Circuit Switches," in *Proc. IEEE ICDCS*, 2019.

Libin Liu received his Ph.D. degree from the Department of Computer Science, City University of Hong Kong and his B.E. degree in software engineering from Shandong University. He is currently an Assistant Professor with the Shandong Computer Science Center (National Supercomputing Center in Ji'nan), Qilu University of Technology (Shandong Academy of Sciences). His current research interests include data analytics systems and machine learning for networking. From 2020 to 2021, he was a researcher in Theory Lab, Huawei Hong Kong Research Center, Hong Kong. Before that, he was a senior engineer in Department of Networking Platform of Tencent, China. He is a member of ACM and IEEE.

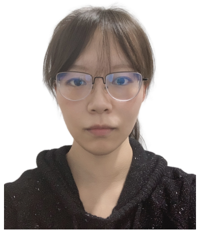
Chengxi Gao is an assistant professor at Shenzhen Institutes of Advanced Technology (SIAT), Chinese Academy of Sciences (CAS). Before joining CAS, he was a research associate in City University of Hong Kong. He received his PhD degree from the Department of Computer Science, City University of Hong Kong, and his B.S. and M.S. degrees from the Department of Computer Science, Northeastern University, China. His research interests include data center networking and networking systems.



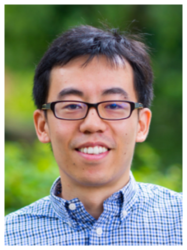
Peng Wang received his PhD degree from the Department of Computer Science, City University of Hong Kong and the B.S. degree in information engineering from Xidian University, Xian, China. He is currently a researcher in Theory Lab, Huawei Hong Kong Research Center, Hong Kong. His research interests include data center networking and cloud computing. He received the best paper award from ACM CoNEXT Student Workshop 2014.



Hongming Huang is currently a Ph.D. student in the Department of Computer Science, City University of Hong Kong. He received his B.E. degree from School of Computer Science and Engineering, Beihang University. His current research interests include machine learning systems and machine learning for networking. He is a student member of IEEE and ACM.



Jiamin Li is currently a Ph.D. student in the Department of Computer Science, City University of Hong Kong. She received the B.S. degree from City University of Hong Kong in 2019. Her research interests include distributed machine learning, machine learning systems, and resource scheduling. She is a student member of ACM and IEEE.



Hong Xu is an Associate Professor in Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research area is computer networking and systems, particularly big data systems and data center networks. From 2013 to 2020 he was with City University of Hong Kong. He received his B.Eng. from The Chinese University of Hong Kong in 2007, and his M.A.Sc. and Ph.D. from University of Toronto in 2009 and 2013, respectively. He was the recipient of an Early Career Scheme Grant from the Hong Kong Research Grants Council in 2014. He received three best paper awards, including the IEEE ICNP 2015 best paper award. He is a senior member of IEEE and member of ACM.

Wei Zhang received the B.E. degree from Zhejiang University in 2004, the M.S. degree from Liaoning University in 2008, and the Ph.D. degree from Shandong University of Science and Technology in 2018. He is currently a Professor with the Shandong Computer Science Center (National Supercomputer Center in Ji'nan), Qilu University of Technology (Shandong Academy of Sciences). His research interests include future generation network architectures, edge computing and edge intelligence.