Coflex: Navigating the Fairness-Efficiency Tradeoff for Coflow Scheduling

Wei Wang[†], Shiyao Ma[†], Bo Li[†], Baochun Li[‡]
[†]Hong Kong University of Science and Technology, [‡]University of Toronto
{weiwa, smaad, bli}@cse.ust.hk, bli@ece.toronto.edu

***Abstract*—Fair and efficient coflow scheduling improves application-level networking performance in today's datacenters. Ideally, a coflow scheduler should provide *isolation guarantees* on the minimum coflow progress to achieve predictable networking performance. Network operators, on the other hand, strive to decrease the average coflow completion time (CCT). Unfortunately, optimal isolation guarantees and minimum average CCT are *conflicting* objectives and cannot be achieved at the same time. Existing coflow schedulers either optimize isolation guarantees at the expense of long CCTs (e.g., HUG [1]), or decrease the average CCT without performance isolation (e.g., Varys and Aalo [2], [3]). The lack of a smooth tradeoff in between poses a dilemma between low efficiency and no performance isolation. To bridge this gap, we develop a new coflow scheduler, *Coflex*, to navigate this tradeoff. Coflex allows network operators to specify the desired level of isolation guarantee using a tunable fairness knob, while at the same time decreasing the average CCT. Both our real-world deployments and trace-driven simulations have shown that Coflex offers a smooth tradeoff between fairness and efficiency. At an appropriate tradeoff level, Coflex outperforms fair schedulers by $2\times$ in minimizing the average CCT.**

I. Introduction

Communications in data-parallel applications typically involve a collection of parallel flows between groups of machines (e.g., shuffle between map and reduce)—known as *coflows* [4]. The coflow abstraction captures the *all-or-nothing* communication requirement of data-parallel jobs: a coflow is not considered complete until *all* its constituent flows have completed. Reducing the coflow completion time (CCT) speeds up the completion of the corresponding job.

Many coflow schedulers have been proposed recently to minimize the average CCT, ranging from FIFO variants [5], [6] to the shortest-first heuristics [2], [4]. These schedulers, while achieving a salient CCT reduction in real systems, do not provide performance isolation to individual jobs. For example, with the shortest-first heuristics, a large coflow can be preempted by a newly arrived small coflow, and may suffer from a long completion time if small coflows keep arriving over time.

Fair schedulers come as a solution to provide predictable *isolation guarantees*. By allocating each coflow a fair share of the datacenter network [1], [7]–[12], fair schedulers ensure a guaranteed minimum progress of coflows, hence isolating the completion of each coflow from another. Among all fair schedulers, the recently proposed HUG [1] is the most efficient. HUG achieves the optimal isolation guarantee with the highest attainable network utilization. However, HUG does not perform well in minimizing the average CCT. It has been shown in [1] that HUG suffers from $1.45\times$ longer average shuffle completion time compared to Varys [2], a performance-optimal scheduler.

Therefore, a tradeoff generally exists between fairness and efficiency for coflow scheduling, which has so far received little attention in the literature. The result is a dilemma facing network operators between little or no isolation guarantee with performance-optimal schedulers (e.g., [2], [3], [5], [6]) and inefficient performance with fair schedulers (e.g., [1], [10]).

In this paper, we aim to design a new coflow scheduler to address this problem by navigating the tradeoff between fairness and efficiency. As a starting point, we show that simply trading off the optimal isolation guarantee of existing fair schedulers does not necessarily improve their efficiency. Worse, it wastes more bandwidth with even lower network utilization. We find that the root cause of this inefficiency is the retaining of *strategy-proofness*, in that a coflow cannot improve its progress by lying about its bandwidth demands across links.

Unlike prior work [1], [10], we give up on strategy-proofness to pursue higher efficiency. In particular, we develop a new coflow scheduler, *Coflex*, which offers a smooth tradeoff between fairness and efficiency. Coflex exposes a tunable *fairness knob* that allows network operators to flexibly trade off isolation guarantees for faster coflow completion. Coflex employs a *two-stage* bandwidth allocation algorithm. In the first stage, Coflex increases the progress of each coflow to a level specified by the fairness knob. In the second stage, Coflex decreases the average CCT using the *smallest-effective-bandwidth-first* (SEBF) heuristic [2] that preferentially schedules a coflow with the smallest bottleneck's completion time, using spare bandwidth unallocated in stage-1. Though Coflex is not strategy-proof, it can still achieve *provable isolation guarantees* in the presence of strategic manipulations.

We evaluated Coflex through real-world deployments on a 60-machine cluster, as well as in simulations over production traces collected from a 3000-machine cluster at Facebook [13]. Our evaluation results show that Coflex strikes a flexible balance between isolation guarantees and high efficiency. In particular, by trading off 50% of fairness, Coflex increases the average coflow progress by $2.2\times$ over HUG, which translates to 50% higher network utilization. In the long run, Coflex decreases the average CCT by over 50% as compared to HUG. The price paid is reflected in fewer than 5% of coflows, as they experience a 13% longer CCT on average. Compared to Varys, Coflex provides predictable service isolation, even

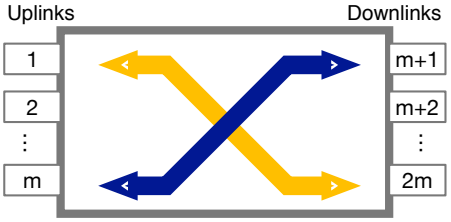


Fig. 1: An $m \times m$ datacenter fabric with m ingress/egress ports connecting to m machines.

though Varys leads to a 24% shorter average CCT. Last but not the least, Coflex is able to scale to large clusters. Even with 10,000 machines, a scheduling decision can be made and enforced in just one second.

II. Model and Motivation

In this section, we present our models and motivate the need to offer a smooth tradeoff between fairness and efficiency for coflow scheduling.

A. Model

Network model. Given the recent advances in datacenter fabrics [14]–[16] and the full bisection bandwidth network in production datacenters [17], we model the datacenter network as one *non-blocking fabric* where the edges are the only sources of contention [1], [4], [11]. Fig. 1 illustrates a non-blocking datacenter fabric connecting m machines through full-duplex links, where link-i and link-$m+i$ correspond to the uplink and downlink of machine-i, respectively. Let $\mathbf{C} = \langle C^1, \ldots, C^{2m} \rangle$ be the capacity vector of the entire datacenter fabric, where C^i is the capacity of link-i.

The Coflow abstraction. A coflow corresponds to the communication stage of a data-parallel job, where a collection of flows transfer data between groups of machines. In many data-parallel jobs such as MapReduce, the amount of data each flow needs to transfer can be known before the flow starts [2], [5], [6]. In particular, let S_k^i be the amount of data coflow-k transfers on link-i. Vector $\mathbf{S}_k = \langle S_k^1, \ldots, S_k^{2m} \rangle$ captures the size of coflow-k. In this paper, we assume $\mathbf{S}_k$ is reported by coflow-k through the Coflow API [2]. Let Γ_k be the *bottleneck's completion time* of coflow-k, which is the minimum CCT if it were running alone in the fabric, i.e.,

$$\Gamma_k = \max_{1 \le i \le 2m} S_k^i / C^i. \tag{1}$$

Unlike individual flows, coflows have *correlated* and *elastic* bandwidth demands across multiple links. Specifically, we characterize the demand of coflow-k by a *correlation vector* $\mathbf{d}_k = \langle d_k^1, \ldots, d_k^{2m} \rangle$, where $d_k^i = \frac{S_k^i}{C^i \Gamma_k}$ is the normalized demand on link-i. We say link-b is the *bottleneck link* of coflow-k if $d_k^b = 1$. Intuitively, for every bit coflow-k transfers on the bottleneck, *at least* d_k^i bits should be sent on link-i.

Given an allocation $\mathbf{a}_k = \langle a_k^1, \ldots, a_k^{2m} \rangle$, where a_k^i is the *bandwidth share* allocated to coflow-k on link-i, its *progress* is defined as the minimum demand-normalized allocation across links, i.e.,

$$P_k = \min_{i: d_k^i > 0} a_k^i / d_k^i. \tag{2}$$

Intuitively, progress P_k captures the attainable transmission rate of coflow-k. In this paper, we assume a *non-cooperative* environment where each coflow strives for the maximum progress. For convenience, Table I summarizes our notations and definitions.

TABLE I: Summary of notations and definitions.

$\mathbf{C} = \langle C^1, \ldots, C^{2m} \rangle$	Link capacity of the datacenter fabric
$\mathbf{S}_k = \langle S_k^1, \ldots, S_k^{2m} \rangle$	Amount of data coflow-k transfers
$\mathbf{d}_k = \langle d_k^1, \ldots, d_k^{2m} \rangle$	Correlation vector of coflow-k
$\mathbf{a}_k = \langle a_k^1, \ldots, a_k^{2m} \rangle$	Bandwidth allocation of coflow-k
$\Gamma_k = \max_i S_k^i / C^i$	Bottleneck's completion time (minimum CCT) of coflow-k
$P_k = \min_i a_k^i / d_k^i$	Progress of coflow-k
$\min_k P_k$	Isolation guarantee

B. Objectives

In this paper, we focus on optimizing two objectives, the average CCT and isolation guarantee.

1) *Average CCT:* To speed up job completion, network operators need to finish as many coflows as possible, each in the fastest possible way. A performance-oriented coflow scheduler should therefore strive to minimize the average CCT.
2) *Isolation guarantee:* In a shared datacenter, coflows expect guarantees on the minimum progress to achieve predictable performance. Specifically, given an allocation, the *isolation guarantee* is defined as the minimum progress across all coflows, i.e., $\min_k P_k$, where P_k is given by (2). A coflow scheduler optimizes the isolation guarantee if the minimum progress is maximized.

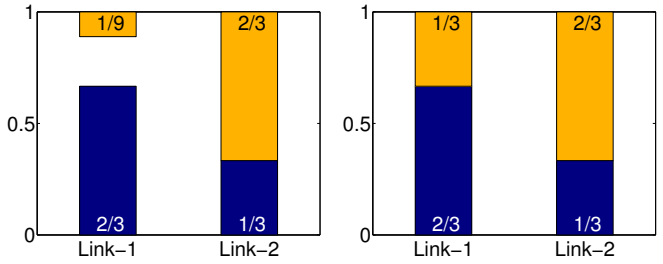
Unfortunately, an optimal isolation guarantee and minimum average CCT are *conflicting* objectives that cannot be achieved at the same time. To see this, recall that coflow scheduling captures traditional flow scheduling on a single link as a special case, where the optimal isolation guarantee is given by max-min fairness. However, max-min fairness is not optimal in minimizing the average flow completion time.

Given the impossibility of achieving both objectives at the same time, navigating their tradeoffs becomes particularly important and relevant in practice. Ideally, network operators should be allowed to specify the desired level of isolation guarantee, denoted by P. The scheduler should then minimize the average CCT subject to the constraint that the isolation guarantee achieves at least a level higher than P, i.e.,

$$\begin{aligned} \text{minimize} \quad & \text{Average CCT}, \\ \text{s.t.} \quad & \min_k P_k \ge P. \end{aligned} \tag{3}$$

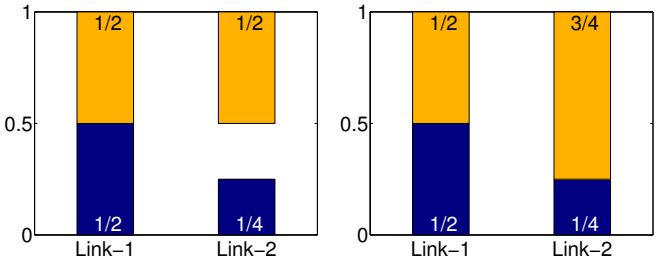
C. The Lack of Tradeoff in Existing Coflow Schedulers

Yet, none of the existing coflow schedulers (e.g., [1]–[3], [5], [11]) consider this tradeoff. They either optimize the isolation



(a) Stage-1: DRF allocation. (b) Stage-2: Spare b/w allocation.

Fig. 2: Illustration of HUG. Coflow-A (blue) demands $\mathbf{d}_A = \langle 1, \frac{1}{2} \rangle$, and coflow-$B$ (orange) demands $\mathbf{d}_B = \langle \frac{1}{6}, 1 \rangle$. (a) Stage-1: use DRF allocation [18] to achieve the optimal isolation guarantee $\frac{2}{3}$. (b) Stage-2: allocate spare bandwidth under a cap equal to the coflow's progress.



(a) Stage-1: DRF allocation. (b) Stage-2: Spare b/w allocation.

Fig. 3: HUG without a cap. Coflow-A (blue) remains truthful, but coflow-B (orange) lies with $\mathbf{d}'_B = \langle 1, 1 \rangle$. (a) Both coflows receive lower progress $\frac{1}{2}$ in stage-1. (b) Coflow-B is assigned spare bandwidth in stage-2, increasing the progress to $\frac{3}{4}$.

guarantee or decrease the average CCT, without considering the other objective. Among these schedulers, Varys [2] and HUG [1] are the two representatives that respectively achieve the best performance in efficiency and fairness.

Varys. Strictly minimizing the average CCT is strongly NP-hard [2]. Given this hardness result, Varys employs an efficient *smallest-effective-bottleneck-first* (SEBF) heuristic that greedily schedules a coflow with the smallest bottleneck's completion time, i.e., Γ_k defined in (1). Varys significantly reduces the average CCT in real systems, and serves as the *de facto* benchmark for coflow scheduling [1], [3].

However, Varys does not provide isolation guarantees, and may sacrifice large coflows for minimizing the average CCT. Compared to HUG, Varys delays the maximum shuffle completion time by 77% in production MapReduce traces [1].

HUG. Unlike Varys, HUG is not designed for CCT reduction, but it optimizes the isolation guarantee with high network utilization. HUG employs a two-stage allocation algorithm [1]. In the first stage, HUG uses Dominant Resource Fairness (DRF) [18] and increases the progress of each coflow to the *maximum* level, computed as

$$P^* = \frac{1}{\max_i \sum_k d_k^i}. \tag{4}$$

In the second stage, HUG evenly allocates unused bandwidth to each coflow under a *cap* equal to its progress. Specifically, for each coflow-k, the amount of bandwidth it receives on a link should not exceed its progress, i.e., $a_k^i \leq P^*$ for all link-i.

Consider an example in Fig. 2. Two coflows compete for two links, where coflow-A has a correlation vector $\mathbf{d}_A = \langle 1, \frac{1}{2} \rangle$, and coflow-$B$ has $\mathbf{d}_B = \langle \frac{1}{6}, 1 \rangle$. In the first stage, HUG applies DRF and raises the progress of both coflows to the maximum level $P^* = \frac{2}{3}$ (Fig. 2a), at which link-1 still has unallocated bandwidth, but link-2 is fully utilized. In the second stage, spare bandwidth on link-1 is evenly distributed to the two coflows. Because coflow-B has already reached the cap $\frac{2}{3}$, all the spare bandwidth goes to coflow-A instead (Fig. 2b).

Enforcing an allocation cap equal to a coflow progress is needed to retain *strategy-proofness*, a *necessary condition* to achieve the optimal isolation guarantee [1]. Without this cap, a coflow may game its demands to increase its progress at the expense of others. Referring back to the previous example, suppose that coflow-A remains truthful, but coflow-B misreports by $\mathbf{d}'_B = \langle 1, 1 \rangle$. Fig. 3 shows the allocation of the two coflows after stage-1. Suppose that no cap is enforced, and the spare bandwidth on link-2 is assigned to coflow-B in stage-2. We see from Fig. 3b that by gaming its demand, Coflow-B successfully increases its progress to $\frac{3}{4}$. The isolation guarantee drops to $\frac{1}{2}$. Enforcing an allocation cap equal to the progress eliminates the incentive of misreporting, retaining strategy-proofness with the optimal isolation guarantee [1].

However, HUG is inefficient in minimizing the average CCT. In production MapReduce traces, HUG delays the average shuffle completion time by 45% compared to Varys [1].

To summarize, Varys and HUG represent the two extremes on the spectrum of coflow scheduling. The lack of a smooth tradeoff between these extremes forces a network operator to face a dilemma between no isolation guarantee and long CCTs.

III. Inefficiency of Naive Tradeoff Algorithm

In this section, we consider a naive tradeoff algorithm that simply balances between HUG and Varys. We show that the naive algorithm is highly inefficient and identify the root cause as the retaining of strategy-proofness.

A. Naive Tradeoff Algorithm

In traditional flow scheduling [19], a common approach is to use a *fairness knob* α to trade off fairness for efficiency. The knob α takes values in the range of $[0, 1]$ and intuitively quantifies the extent to which the scheduler adheres to fair allocations. In particular, the scheduler first allocates each flow-k an α-fraction of the fair share f_k, and then uses the remaining bandwidth $1 - \alpha \sum_k f_k$ for efficiency optimization.

Following this approach, we consider a naive two-stage tradeoff algorithm shown in Algorithm 1. In the first stage, the algorithm assigns each coflow an α-fraction of its DRF allocation, thus ensuring an α-fraction of the optimal isolation guarantee, i.e., $P_k = \alpha P^*$, where P^* is defined in (4) and is achieved using HUG. In the second stage, the algorithm

Algorithm 1 NaiveTradeoff(α)

Stage 1: Increase the progress of each coflow to an α-fraction of the optimal isolation guarantee, i.e., $P_k = \alpha P^*$ for all coflow-k:
for all coflow-k **do**
$\mathbf{a}_k \leftarrow \alpha \bar{\mathbf{a}}_k$ ▷ $\bar{\mathbf{a}}_k$: DRF allocation of coflow-k

Stage 2: Allocate unused bandwidth using SEBF [2], subject to the constraint that the amount of bandwidth each coflow-k receives on a link does not exceed its progress, i.e., $a_k^i \leq \alpha P^*$, for all link-i.

switches to performance-optimal heuristics using SEBF. Similar to Varys, it greedily offers unallocated bandwidth to a coflow with the smallest bottleneck's completion time. Recall that in HUG, to retain strategy-proofness, bandwidth allocation is capped by the corresponding coflow's progress. Algorithm 1 enforces a similar cap, which is αP^*, to restrict the allocation of unused bandwidth. We show in the following theorem that Algorithm 1 retains strategy-proofness, with a tunable isolation guarantee that is an α-fraction of the optimum. The proof is similar to the analysis of HUG (cf. [1, Theorem 3]) and is thus omitted.

Theorem 1: Algorithm 1 is *strategy-proof* with an isolation guarantee of αP^*, where P^* is defined in (4).

B. Inefficiency of the Naive Tradeoff Algorithm

The naive tradeoff algorithm is however problematic. Recall that bandwidth allocation is capped by αP^* to retain strategy-proofness. The smaller the α, the lower the cap. Consequently, trading off isolation guarantee with a small α puts a low cap on allocation, resulting in poor network utilization! Such a tradeoff, therefore, is neither desirable nor justifiable.

It turns out that the root cause of the inefficiency of Algorithm 1 is that strategy-proofness needs to be retained, for which the cap αP^* is needed. One might think that the efficiency problem caused by strategy-proofness occurs only when α is small. We show in the following theorem that even with no isolation tradeoff ($\alpha = 1$), the allocation cap required by strategy-proofness may result in *arbitrarily low* network utilization.

Theorem 2: The network utilization of Algorithm 1 can be arbitrarily close to 0 even if $\alpha = 1$.

Proof: Consider $n + 1$ coflows competing for n links. Each coflow-k, where $k < n$, has demands on two links only, i.e., $d_k^1 = 1$, $d_k^{k+1} = \frac{1}{2}$. Coflow-n has demand on link-1 only, i.e., $\mathbf{d}_n = \langle 1, 0, \ldots, 0 \rangle$. Coflow-$n+1$ has equal demands on all but link-1, i.e., $\mathbf{d}_{n+1} = \langle 0, 1, \ldots, 1 \rangle$. Algorithm 1 allocates all coflows the same progress $P^* = \frac{1}{n}$ in the first stage. In the second stage, the algorithm allocates spare bandwidth under an allocation cap of $\frac{1}{n}$. Specifically, each coflow-k, where $k < n$, receives $a_k^1 = a_k^{k+1} = \frac{1}{n}$; coflow-$n$ receives $\mathbf{a}_n = \langle \frac{1}{n}, 0, \ldots, 0 \rangle$; and coflow-$n+1$ receives $\mathbf{a}_{n+1} = \langle 0, \frac{1}{n}, \ldots, \frac{1}{n} \rangle$. To summarize, Algorithm 1 allocates the entire capacity of link-1 but $\frac{2}{n}$ of that of the other links, and the network utilization is $\frac{1}{n} + \frac{2(n-1)}{n^2}$, and is arbitrarily close to 0 when $n \to \infty$. ■

Prior work shows that enforcing an allocation cap no more than the coflow progress is a *necessary condition* to retain strategy-proofness [1, Lemma 1]. Given this result, we see from Theorem 2 that to achieve high efficiency, we must give up on strategy-proofness. This, in turn, opens the door to strategic manipulations, which may result in a lower isolation guarantee as seen in the example of Fig. 3. But to what extent can the isolation guarantee be harmed? Are we opening up a Pandora's box by giving up strategy-proofness? We answer this question in the next section.

IV. COFLEX

In this section, we analyze the strategic behaviors of coflows. We show that capping the bandwidth allocation does not help mitigate the loss of the isolation guarantee once strategy-proofness is no longer retained. Based on this observation, we present a new coflow scheduler, *Coflex*, to allocate bandwidth without such a restriction. We show that despite untruthful, strategic coflows, Coflex can still provide isolation guarantees comparable to the state-of-the-art fair schedulers. Coflex allows network operators to flexibly navigate the tradeoff between fairness and efficiency.

A. Allocation Cap vs. Isolation Guarantee

We learn from previous discussions that completely eliminating strategic behaviors results in low efficiency; allowing them, on the other hand, harms isolation guarantees. Is there a middle ground in between? That is, can we limit the "damage" on the isolation guarantee due to misreports within an acceptable range, while at the same time improving the efficiency? Recall that in Algorithm 1, we use an allocation cap of αP^* to eliminate manipulations. This prompts us to use the cap as a control knob to tune the loss of isolation guarantees.

In particular, we inflate the allocation cap of Algorithm 1 by a factor of $1 + e$, where $e \geq 0$ is a tunable *inflation factor*. The first stage remains the same, in which each coflow-k receives an α-fraction of its DRF allocation, i.e., $\mathbf{a}_k = \alpha \bar{\mathbf{a}}_k$, where $\bar{\mathbf{a}}_k$ is the DRF allocation of coflow-k. Unallocated bandwidth is distributed in stage-2, under an inflated cap $(1 + e)\alpha P^*$. Intuitively, with a small inflation factor e, we are conservative in allocating more spare bandwidth as it may benefit misreporting; with a larger e, we are more aggressive in pursuing high efficiency. By tuning the inflation factor e, we expect to control the loss of isolation guarantees due to manipulations.

To our surprise, the isolation guarantee is not smoothly traded off as the cap increases. Instead, it suddenly declines to a certain level when the cap does not retain strategy-proofness, i.e., $e > 0$. To see this, we consider no tradeoff with $\alpha = 1$ and an arbitrarily small inflation factor, i.e., $e \to 0$. Referring back to the example in Fig. 2, we might expect the resulting allocation arbitrarily close to that of HUG. Contrary to our expectation, we next show that such a slight increase of cap incurs a "race to the bottom," where the two coflows game their reports and finally reach an equilibrium at which both claim $\mathbf{d}'_A = \mathbf{d}'_B = \langle 1, 1 \rangle$. The isolation guarantee drops to $\frac{1}{2}$ as compared to the optimum, which is $\frac{2}{3}$.

Analysis of the example in Fig. 2. We start with an initial state where both coflows are truthful. We analyze their game theoretic behaviors in rounds. In each round, we fix the report

Algorithm 2 Coflex(α)

▷ Called upon the arrival or departure of a coflow
Stage 1: Assign each coflow an α-fraction of the its DRF allocation:
  for all coflow-k **do**
    $\mathbf{a}_k \leftarrow \alpha \bar{\mathbf{a}}_k$ ▷ $\bar{\mathbf{a}}_k$: DRF allocation of coflow-k
Stage 2: Allocate unused bandwidth using SEBF [2].

of one coflow as its claim in the previous round, and let the other make its best response. The two coflows alternate their best responses in rounds, where coflow-B is the first mover in round-1. We characterize the best responses of the two coflows in the following theorem. The proof is deferred to the technical report [20] due to the limit of space.

Theorem 3: There exists a threshold round T, such that

1) In each round-r, where $r \leq T$, coflow-A makes best response $\mathbf{d}_A^{(r)} = \langle 1, \frac{1}{2} + re \rangle$ if r is even; coflow-B makes best response $\mathbf{d}_B^{(r)} = \langle \frac{1}{2} + re, 1 \rangle$ if r is odd.
2) When round-r runs beyond the threshold T, the best response of each coflow is $\mathbf{d}_A^* = \mathbf{d}_B^* = \langle 1, 1 \rangle$.

Theorem 3 rules out the necessity of enforcing an allocation cap higher than that required to retain strategy-proofness. The cap does not help mitigate the damage caused by manipulations—as long as $e > 0$, the race-to-the-bottom phenomenon incurs. Worse, the cap restricts bandwidth allocation, which inevitably results in low utilization. The cap should therefore be removed if strategy-proofness is given up.

B. Coflex

Coflex is designed to trade off the isolation guarantee for high efficiency. To this end, Coflex does not cap bandwidth allocation, and is shown in Algorithm 2. Coflex uses a fairness knob α to navigate the tradeoff. Similar to HUG, Coflex applies a two-stage bandwidth allocation algorithm upon the arrival or departure of a coflow. In the first stage, Coflex computes the DRF allocation of each coflow based on the reported demands. It then assigns each coflow an α-fraction of its DRF allocation. In the second stage, Coflex decreases the average CCT using the SEBF heuristic that preferentially schedules coflows in an ascending order of their bottleneck's completion time (i.e., Γ_k defined in (1)).

Coflex is not strategy-proof. Nevertheless, we show that coflows can still expect isolation guarantees even in the presence of manipulations and untruthful reports.

Theorem 4 (Isolation Guarantee): Let n_i be the number of coflows having demands on link-i, i.e., those with $d_k^i > 0$. Coflex ensures that, at an equilibrium, the (true) progress of each coflow-k is at least

$$P_k \geq \frac{\alpha}{\max_{1 \leq i \leq 2m} n_i}. \tag{5}$$

Proof: Let $\mathbf{d}_k$ and $\mathbf{d}'_k$ respectively denote the true and reported demands of coflow-k. We show that for any coflow-k, irrespective of the other coflows' claims, it achieves the stated isolation guarantee by reporting

$$d'^i_k = \begin{cases} 1, & d_k^i > 0; \\ 0, & d_k^i = 0. \end{cases} \tag{6}$$

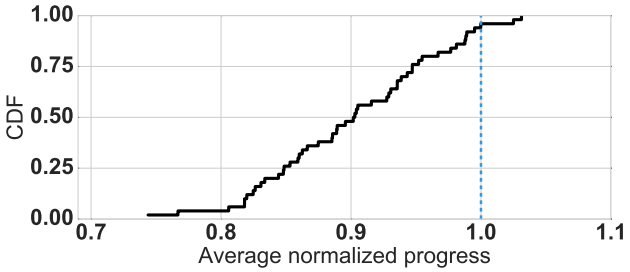


Fig. 4: CDF of average normalized progress by misreporting.

With this report, coflow-k receives

$$a_k^i = \alpha \bar{a}_k^i = \alpha d'^i_k / \max_l \sum_j d'^l_j \tag{7}$$

on link-i in stage-1, where $\bar{a}_k^i$ is the computed DRF allocation. Therefore, in stage-1, coflow-k achieves progress

$$\begin{aligned} P_k &= \min_{i: d_k^i > 0} a_k^i / d_k^i && \text{(by (2))} \\ &\geq \min_{i: d_k^i > 0} a_k^i && (d_k^i \leq 1) \\ &= \alpha / \max_l \textstyle\sum_j d'^l_j && \text{(by (6) and (7))} \\ &\geq \alpha / \max_i n_i. && (d'^i_j \leq 1) \end{aligned}$$

Because coflow-k can only receive more bandwidth in stage-2, its progress never decreases. ■

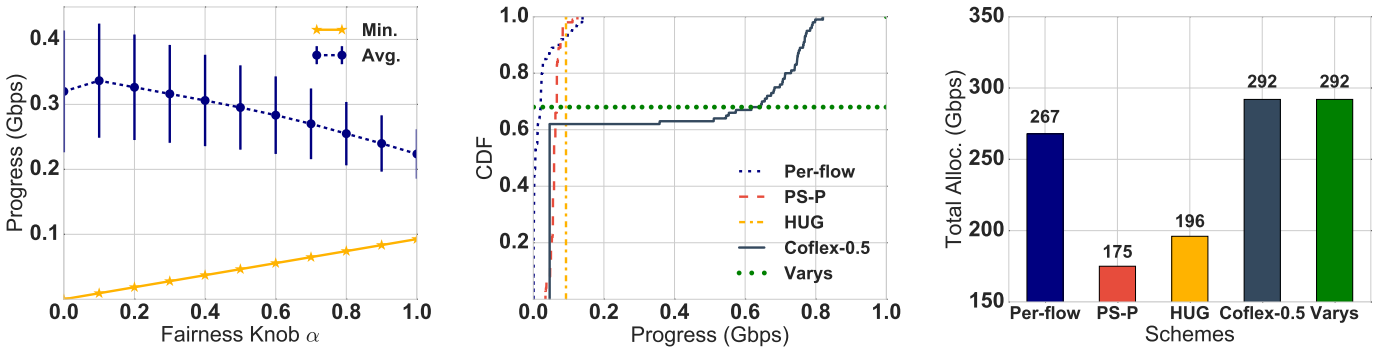
We note that an isolation guarantee of $1/\max_i n_i$ is achieved by the state-of-the-art fair network sharing policies such as PS-P in FairCloud [10] and EyeQ [11]. Theorem 4 states that with Coflex, coflows can expect at least an α-fraction of the isolation guarantees offered by the status quo.

C. Practical Impact of the Lack of Strategy-Proofness

So far, our analysis on isolation guarantees assumes an *idealized* scenario, where each coflow has the *full knowledge* of others, and reacts with its best response. Such a full-knowledge game represents the worst case where strategic manipulations harm isolation guarantees the most: as shown in Theorem 3, alternating best responses incurs the race to the bottom. In this sense, the isolation guarantee given by Theorem 4 is the most pessimistic estimation.

However, in shared, multi-tenant environments like public clouds, often a coflow has no knowledge (or imperfect knowledge at most) of the other coflows belonging to another tenant. Without the information of the others, how and to what extent can a misreport improve a coflow's progress?

To answer this question, we ran simple numerical experiments using Coflex with $\alpha = 1$ and measured the expected gain a coflow can derive from misreporting. In particular, we simulated 100 coflows and let them compete on two links. The coflow demands on the two links are uniformly generated. In each simulation run, we randomly selected a coflow-k and enumerated all the possible reports coflow-k can claim (the step size of enumeration on each link is set to 0.1). For each report enumerated, we computed the *normalized progress*, which is the progress coflow-k achieves with the report normalized



(a) Impact of tradeoff. (b) Distribution of instantaneous progress. (c) Total allocation out of 300 Gbps availability.

Fig. 5: Characteristics of instantaneous allocation with 100 concurrent coflows using different coflow schedulers.

by that with the true demand. After enumerating all possible claims, we computed the *average* normalized progress. If the average normalized progress is greater than 1, the coflow can expect higher progress by misreporting; otherwise, the coflow is better off truthful. We ran 100 experiments and break down the average normalized progress in Fig. 4. It can be observed that only by a very small chance, less than 3%, can misreporting be beneficial in expectation—even so, the expected progress gain is no more than 3%.

We emphasize that the simple scenario considered in our numerical experiments consists of only two links, which can be easily "hacked" by exhaustively searching over the entire strategy space. However, datacenter fabrics typically consist of tens of thousands of links, and a coflow of a data-parallel job can easily span hundreds of machines. At such a large scale, finding a strategy that is consistently beneficial is computationally challenging, if not impossible. Given that beneficial misreporting is rare, computationally difficult and of marginal gains, we believe that the lack of strategy-proofness hardly poses a disadvantage for Coflex in realistic datacenters.

V. Evaluation

We have evaluated Coflex using both trace-driven simulations and a real-world implementation across a 60-machine cluster. Highlights of our evaluations are summarized as follows.

- Coflex offers a smooth tradeoff between fairness and efficiency. With $\alpha = 0.5$, Coflex outperforms HUG by $2.2\times$ in terms of the average coflow progress, which translates to $1.5\times$ higher network utilization. Varys, on the other hand, provides no isolation guarantee, starving 68% of coflows with zero progress (Sec. V-A1).
- Coflex dominates HUG in minimizing the average CCT. With $\alpha = 0.5$, Coflex is on average $2\times$ faster than HUG in terms of the normalized CCT, with fewer than 5% of coflows suffering from longer CCT (Sec. V-A2).
- Coflex is able to scale to large clusters. Even with 10,000 machines, new bandwidth allocations can be computed and enforced in around 1 second (Sec. V-B).

A. Trace-Driven Simulations

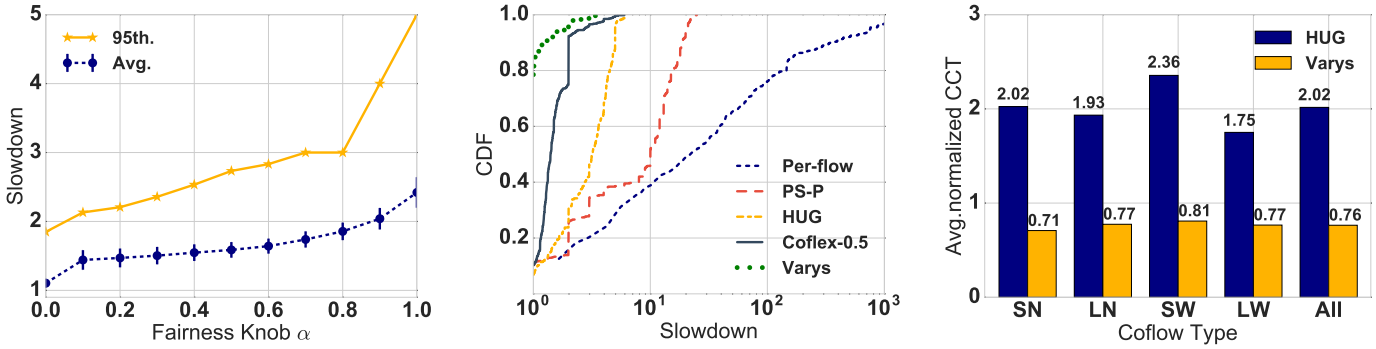
Workload. We use a suite of production traces in the Coflow-Benchmark [13] as input in our simulations. These traces are synthesized based on the one-hour workload collected from a 3000-machine Hive/MapReduce cluster with 150 racks at Facebook [2]. In total, the traces consist of 526 coflows scaled down to a 150-port fabric, where mappers (reducers) in the same rack are combined into one rack-level mapper (reducer). For each coflow, the traces log its arrival time, placements of mappers/reducers, and the amount of data shuffled.

Setup. We abstract out the datacenter fabric as a 150×150 non-blocking switch, where an ingress (egress) port corresponds to a 1 Gbps uplink (downlink) of a rack. We compare Coflex against four scheduling schemes: per-flow fairness (TCP), FairCloud's PS-P policy [10] that seeks per-link fairness, HUG [1], and Varys [2]. We implemented these schemes along with Coflex on top of `CoflowSim` [21] and compared their instantaneous and long-term performance.

*1) **Instantaneous Performance**:* We start to evaluate the instantaneous performance of Coflex. We randomly sampled 100 coflows and ran them *concurrently*. To eliminate sampling bias, we repeated the simulations several times and observed a consistent performance trend. We therefore report results in one simulation run.

Impact of tradeoff. We ran Coflex at different fairness levels, from none ($\alpha = 0$) to the highest ($\alpha = 1$). Fig. 5a shows the minimum and average coflow progress with different α, where the former quantifies achieved isolation guarantees, and the latter captures the instantaneous efficiency. As expected, the minimum progress linearly increases with increasing α. Meanwhile, the average progress declines, also linearly. Intuitively, the higher the isolation guarantee, the less the spare bandwidth available. Recall that spare bandwidth is preferentially offered to small coflows. Having less spare bandwidth lowers their progresses. Given that the population of small coflows dominates (72%), the decrease of their progress drags down the average accordingly.

To summarize, Coflex offers a smooth tradeoff between isolation guarantee and efficiency. Given that the tradeoff is



(a) Impact of tradeoff. (b) Distribution of slowdown. (c) Average normalized CCT in coflow bins.

Fig. 6: Long-term characteristics of Coflex against alternative coflow scheduling schemes.

linear, unless otherwise specified, we settle on $\alpha = 0.5$ to balance fairness and efficiency in the following evaluations.

Coflow progress. We next compare the distribution of coflow progress under different schemes in Fig. 5b. We see that Varys' SEBF heuristic runs into *winner-take-all*, where 68% of coflows end up with no progress, whereas the others are serviced at the maximum progress. Varys therefore provides no isolation guarantee and is not fair. HUG goes to another extreme by enforcing a uniform progress of 93 Mbps across coflows. Compared to these two schedulers, Coflex lands in an attractive middle ground. Coflex offers the isolation guarantee of 46 Mbps with $\alpha = 0.5$, and does not cap bandwidth allocation. The results are 40% of coflows receiving a progress over 500 Mbps. Compared to HUG, Coflex is 2.2× better in terms of the average progress. Coflex also outperforms per-flow fairness and PS-P, by a significant margin.

Network utilization. Fig. 5c compares the total amount of bandwidth allocated using different schedulers. We see that Coflex and Varys are able to achieve the highest utilization, taking 97% of the total 300 Gbps availability. Per-flow fairness follows with 89% utilization. In comparison, HUG is inefficient: the enforcement of allocation cap results in 96 Gbps less bandwidth allocation than that under Coflex, which accounts for 31% of the total availability. PS-P is even worse, with less than 60% utilization. We attribute its low utilization to being agnostic to the correlated demands of coflows. Because bandwidth is assigned separately on each link, often, the allocated ingress bandwidth does not match that on the egress ports.

*2) **Long-Term Performance**:* We next evaluate the long-term performance of different schedulers in minimizing the coflow completion time (CCT). In particular, we replayed 526 coflows in the trace [13] and used two metrics throughout our evaluation: shuffle slowdown and normalized CCT.

- *Shuffle slowdown* is defined, for each coflow, as the CCT under the compared scheduler normalized by the minimum CCT, i.e., its bottleneck's completion time defined in (1):

$$\text{Slowdown} = \frac{\text{Compared CCT}}{\text{Minimum CCT}}.$$

- *Normalize CCT* is defined, for each coflow, as the CCT under the compared scheduler normalized by that under Coflex:

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

If the normalized CCT is greater (smaller) than 1, the coflow finishes faster (slower) using Coflex.

TABLE II: Coflows binned by their lengths (**S**hort or **L**ong) and widths (**N**arrow or **W**ide) in the Coflow-Benchmark [13].

Bin	SN	LN	SW	LW
% of Coflows	60%	16%	12%	12%

In addition, to better understand the performance impact on different coflows, 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 5 MB, and *narrow* (*wide*) if it consists of *less* (more) than 50 flows [1]–[3]. Table II summarizes the distribution of binned coflows.

Impact of tradeoff. We start by characterizing how Coflex trades off the isolation guarantee for faster coflow completion. Fig. 6a shows the shuffle slowdown on average and at the 95$^{\text{th}}$ percentile at different levels of the isolation guarantee. We make two observations. First, at the 95$^{\text{th}}$ percentile, the best tradeoff point is given by $\alpha = 0.8$, beyond which the slowdown sees a sharp increase. On the other hand, in terms of the average performance, the best tradeoff is achieved at $\alpha = 0.5$ — a *saddle point* starting from which the increase of slowdown becomes more salient. Because Coflex focuses on improving the average performance, we settle on $\alpha = 0.5$ in the evaluation.

Slowdown. We compare in Fig. 6b the distribution of shuffle slowdown under different schedulers. We see that per-flow fairness and PS-P do not perform well for their ignorance of coflow's correlated demands. HUG addresses this problem and uniformly outperforms these two schemes. However, HUG is not designed to speed up coflow completion, and its performance is dominated by Varys. Coflex comes as a middle ground. While the tradeoff is taken at the midpoint ($\alpha = 0.5$), Coflex performs more closely to Varys than to HUG. It is worth noticing that Coflex does not cross HUG, and is less likely to delay the coflow completion beyond that of HUG, even if the isolation guarantee has been traded off.

TABLE III: Statistical summary of slowdown.

	Per-flow	PS-P	HUG	Coflex	Varys
Min	1.00	1.00	1.00	**1.00**	1.00
Mean	120.06	9.47	3.13	**1.59**	1.10
95th	757	20.73	5.00	**2.73**	1.85
Std.	248	6.75	1.38	**0.66**	0.35

Table III gives the minimum, average, and the 95th percentile slowdown, as well as the standard deviation measured under different schedulers. To summarize, in terms of the shuffle slowdown, Coflex outperforms HUG by $2\times$ on average, and by $1.8\times$ at the 95th percentile.

Normalized CCT. Fig. 6c shows the average normalized CCT under HUG and Varys in all coflow bins. Here, we exclude per-flow fairness and PS-P from comparison due to their poor performance. We see that Coflex consistently outperforms HUG, with normalized CCT greater than 1 in all the bins. In particular, small coflows have higher normalized CCTs than large ones, implying that they are favored by Coflex with a more significant speedup. Such a bias towards small coflows is attributed to the SEBF heuristic. On average, Coflex outperforms HUG by $2\times$ in terms of the normalized CCT, where only 25 coflows ($< 5\%$) suffer from on average a 13% longer CCT (not shown in the figure). On the other hand, we note that, being performance-optimal, Varys is 24% faster than Coflex, but at the expense of providing no isolation guarantee at all.

B. Testbed Experiments

We next micro-benchmark the performance of Coflex using our real-world implementation.

Implementation. We have implemented Coflex in `Python`, with performance hotspots such as the scheduling algorithm implemented in `C` for further optimization. Our implementation of Coflex adopts a master-slave architecture. Upon arrival, a coflow registers at the Coflex `master` and indicates the amount of data its flows need to transfer, through a public Coflex API. The `master`, after receiving this registration, runs Algorithm 2 and computes a new allocation. The `master` then sends the allocation to its corresponding `slaves` on cluster machines for local enforcement. In our implementation, each `slave` uses Linux's `tc` and `htb qdisc` to shape the flow rates, where packets are filtered by the TCP quintuple and added to the corresponding `htb` classes subject to specified rate limits. Each `slave` also updates its status with the `master` periodically. This allows the `master` to quickly respond to coflow completion events and compute new allocations for running coflows.

Cluster deployment. We performed our experiments on a 60-machine Linode [22] cluster running `Debian` stable with kernel version 4.5.5. Each machine has 12 GB RAM, 6 cores, 1 Gbps uplink, and 40 Gbps downlink.

Micro-benchmark. To micro-benchmark the behavior of Coflex in a more controlled manner, we ran three coflows, each having endpoints on all 60 machines. Coflows have different communication patterns. For coflow-A, we evenly divide its endpoints into 10 groups. Within a group, the communications follow an all-to-all pattern with 36 flows. In total, coflow-A consists of 360 flows. Coflow-B has 60 flows following a pairwise one-to-one communication pattern between machine i and $i + 30$, where $1 \leq i \leq 30$; coflow-C also has 60 flows, with a one-to-one communication pattern between machine j and $j + 15$, where $1 \leq j \leq 15$ or $31 \leq j \leq 45$. All flows are of the same size. In this setting, coflow-A represents a large job, whereas the other two coflows are considered small. Coflow-A, B, and C arrive at 0 s, 3 s, and 4 s, respectively.

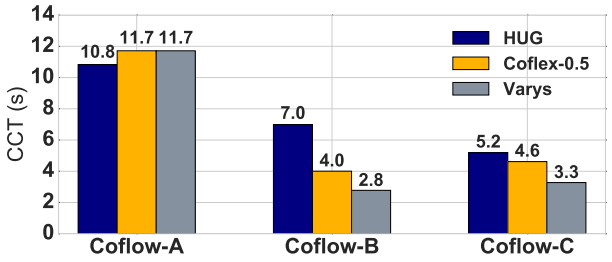


Fig. 7: CCT of three coflows in a 60-machine cluster.

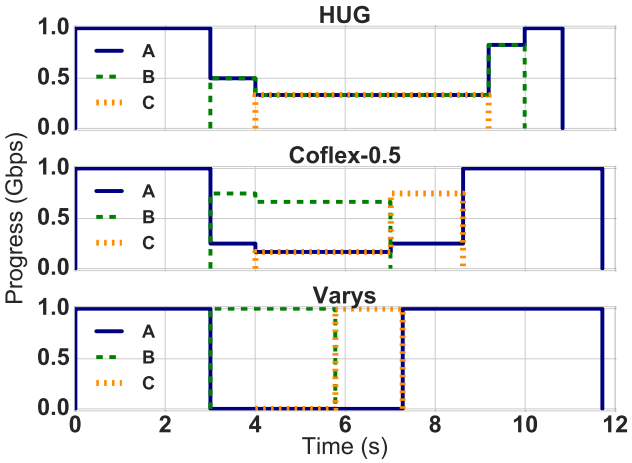


Fig. 8: Coflow progress measured over time in a 60-machine cluster.

We compare in Fig. 7 the completion time of the three coflows using three schemes: Coflex with $\alpha = 0.5$, HUG, and Varys. We see that Coflex outperforms HUG in speeding up the two small coflows, B and C, while slightly delaying the completion of coflow-A by 8%. Varys exhibits a similar performance trend, with more salient speedups to small coflows.

Fig. 8 breaks down the progress of each coflow under different schemes over time. We see that HUG enforces an equal progress to each active coflow at all times, isolating the performance of each coflow from that of another. In contrast, Varys provides no progress guarantees: the arrivals of coflow-B and C preempt the running coflow-A. Coflex avoids this problem with guarantees on the coflow progress, but at a level lower than HUG. Unlike HUG, with Coflex, small coflows are favored and can achieve a much higher progress above the minimum guarantee, which significantly decreases their CCTs.

Scalability. Coflex is able to scale to large clusters. In our 60-machine deployment, the computation of new bandwidth allocations due to coflow arrivals and departures takes less than 30 microseconds on average. We stress tested the computation

time by *emulating* a fabric of 10,000 machines using the same implementation code. Even at such a large scale, the time to compute new allocations is about 26 milliseconds. We also measured the communication time for Coflex `master` to notify *slaves* about new bandwidth allocation results. In our deployment, it takes less than 10 milliseconds on average to communicate to 60 machines. To emulate a large cluster with 10,000 machines, we repeatedly sent the same notification 200 times to each machine. The measured communication time is around 1 second.

VI. Related Work

The coflow abstraction captures the multipoint-to-multipoint communication patterns of data-parallel jobs. Many recently proposed coflow schedulers strive for high efficiency with the minimum average CCT. For example, Orchestra [6] and Baraat [5] use FIFO-based scheduling to decrease the average CCT. Varys [2] uses the *Smallest-Effective-Bottleneck-First* heuristic to prioritize small coflows. Aalo [3] improves Varys as a non-clairvoyant scheduler in that a *priori* knowledge about the flow size is not needed. These schedulers, while efficient in decreasing the average CCT, fall short in isolating coflows and in providing predictable performance. Coflex addresses this problem by offering a tunable isolation guarantee, without losing much on efficiency even compared to Varys.

Isolating coflow performance by means of fair network sharing among tenants has also received considerable attention. Systems like SecondNet [9] and Oktopus [8] allow tenants to express their network requirements and meet them using static, reservation-based bandwidth allocation policies. However, static bandwidth reservation can be inefficient: the reserved bandwidth, even idle, cannot be used by coflows of anther tenant. This problem is avoided by work conserving policies such as FairCloud's PS-P [10] and EyeQ [11], where bandwidth are dynamically allocated based on the communication pattern of underlying coflows. Unfortunately, these policies are agnostic to the correlated demands of coflows, resulting in a low coflow progress. The state-of-the-art is represented by the recently proposed HUG [1] and its variant [12], under which coflows expect the optimal isolation guarantee. The price paid is, however, a significant efficiency loss, both in network utilization and in the average CCT. Coflex trades off fairness for much higher efficiency, at the expense of only a few coflows experiencing a slight delay of completion.

VII. Conclusion

In this paper, we have studied the tradeoff between fairness and efficiency for coflow scheduling. We have shown that to achieve high efficiency, strategy-proofness—the common requirement in shared, multi-tenant environments like cloud network—must be given up. We have quantified the impact of strategic manipulations on coflow scheduling, analytically and experimentally, and highlighted two key observations: (1) isolation guarantees can still be achieved even in the presence of strategic manipulations; (2) misreporting is seldom beneficial with marginal gains in practical settings. Based on these results, we have developed a new coflow scheduler, Coflex, that for the first time strikes a flexible and tunable balance between isolation guarantee and high efficiency. Trace-driven simulations and testbed implementation have demonstrated that Coflex significantly decreases the average CCT at the specified tradeoff level, and is able to scale to large clusters.

Acknowledgement

This work was supported in part by grants from RGC under the contracts 615613, 16211715 and C7036-15G (CRF), as well as a grant from NSF (China) under the contract U1301253.

References

[1] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *USENIX NSDI*, 2016.
[2] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM*, 2014.
[3] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM*, 2015.
[4] M. Chowdhury, "Coflow: A networking abstraction for distributed data-parallel applications," Ph.D. dissertation, University of California, Berkeley, 2015.
[5] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM*, 2014.
[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM*, 2011.
[7] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *USENIX NSDI*, 2011.
[8] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM*, 2011.
[9] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *ACM CoNEXT*, 2010.
[10] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: sharing the network in cloud computing," in *ACM SIGCOMM*, 2012.
[11] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical network performance isolation at the edge," in *USENIX NSDI*, 2013.
[12] W. Wang and A.-L. Jin, "Friends or foes: Revisiting strategy-proofness in cloud network sharing," in *IEEE ICNP*, 2016.
[13] M. Chowdhury, "Coflow-Benchmark," https://goo.gl/szsBQE.
[14] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *ACM SIGCOMM*, 2009.
[15] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *ACM SIGCOMM*, 2009.
[16] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM*, 2014.
[17] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *ACM SIGCOMM*, 2015.
[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *USENIX NSDI*, 2011.
[19] E. Danna, S. Mandal, and A. Singh, "A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering," in *IEEE INFOCOM*, 2012.
[20] W. Wang, S. Ma, B. Li, and B. Li, "Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling," https://www.cse.ust.hk/~weiwa/papers/coflex.pdf, HKUST, Tech. Rep., 2017.
[21] "Coflowsim," https://github.com/coflow/coflowsim.
[22] "Linode," https://www.linode.com.