Utopia: Near-optimal Coflow Scheduling with Isolation Guarantee

Luping Wang, Wei Wang, Bo Li
Hong Kong University of Science and Technology
{lwangbm, weiwa, bli}@cse.ust.hk

***Abstract*—Performance and service isolation come as two top objectives for coflow scheduling. However, the common wisdom is that these two objectives are often *conflicting* with each other and cannot be achieved simultaneously. Existing coflow scheduling frameworks either focus only on minimizing the average coflow completion time (CCT) (e.g., Varys), or providing optimal isolation between contending coflows by means of fair network sharing (e.g., HUG). In this paper, we make an attempt to achieve the best of both worlds through a novel coflow scheduler, Utopia, to attain *near-optimal* performance with *provable* isolation guarantee. This is particularly challenging given the correlation of bandwidth demands across multiple links from coflows. We show that Utopia is capable of reducing the average CCT dramatically, while still guaranteeing that no coflow will ever be delayed beyond a constant time than its CCT in a fair scheme. Both trace-driven simulation and EC2 deployment confirm that Utopia outperforms the fair sharing policy by $1.8\times$ in terms of average CCT, while producing no completion time delay for a single coflow. Even compared with performance-optimal Varys, Utopia speeds up average coflow completion by 9%.**

I. Introduction

Communication in data-parallel applications widely follows the bulk-synchronous parallel (BSP) model [1]–[3], in which a collection of parallel flows, termed *coflow* [4], transfer intermediate data between a group of machines. Not until all parallel flows have completed will a coflow complete. In a shared cloud, diverse coexisting coflows from many users and applications contend for the limited network bandwidth. Consequently, today's network scheduler needs to schedule contending coflows to attain *optimal performance* towards *fast completion* while providing *service isolation* between coflows by means of fair network sharing.

However, *simultaneously* achieving optimal performance and service isolation is challenging. Many network schedulers settle for minimizing the *average coflow completion time* (CCT) as the primary objective, e.g., [5]–[8], among which Varys [5] is arguably of the best performance. Varys generalizes the Shortest-Job-First (SJF) heuristic to the context of coflow scheduling. The algorithm preferentially prioritizes "small" coflows over "large" ones, hence minimizing the average CCT. However, Varys and its refinements [6]–[8] are *incapable* of providing service isolation: because "large" coflows are less favored than "small" ones, the former end up with less bandwidth and are likely slowed down by the latter [9].

On the other hand, many solutions turn to *fair network sharing* to provide optimal isolation between coflows, e.g., [9]–[12]. These schedulers allocate each coflow a fair share of the cloud network bandwidth, isolating the completion of each coflow with predictable CCT. However, fair schedulers do not explicitly optimize performance and fall short of minimizing average CCT [9]. In fact, performance (i.e., minimizing average CCT) and fairness (i.e., service isolation) have long been considered as *conflicting* objectives that cannot be achieved simultaneously, which motivates the recent work [13] to navigate the two-way tradeoff space.

In this paper, we challenge the status quo with a bold question: *is it possible to provide isolation guarantee between contending coflows while still minimizing the average CCT to ensure near-optimal performance?* We provide an affirmative answer to this question. Our key insight is to preferentially serve coflows following their *completion order* in a fair scheduling scheme, i.e., DRF [10] or HUG [9]. Intuitively, this simple approach is capable of achieving the best of both worlds in two aspects. First, because coflows are served at the same *progress* in a fair scheme [9], "small" coflows likely complete earlier than "large" ones. Therefore, tracking the coflow completion order in a fair scheme largely imitates Varys, in which "small" coflows are prioritized over "large" ones to minimize the average CCT. Second, service isolation can only be observed by users when their coflows *complete*. By tracking the completion order in a fair scheme, the scheduler likely completes a coflow earlier than it would have had in a fair scheme, hence providing an equivalent *long-term isolation* between users and applications.

However, directly implementing this promising insight using existing approaches (e.g., [5]) results in *priority inversion*. A coflow has *correlated* bandwidth demand across *multiple* links. In case that one link is preempted by another coflow with a higher priority, the entire coflow gets *delayed* even if it can still transfer on other links. Existing approaches [5] then suspend the transmission of that coflow on *all* links, and offer the revoked bandwidth to others with *lower* priorities. The coflow hence yields bandwidth to low-priority contenders. As we shall show in Sec. III-B, priority inversion unnecessarily delays coflow completion.

We address this challenge through a novel algorithm called Utopia. Utopia sequentially aggregates coflows into a *super-coflow*, where the k^{th} super-coflow is the *cumulative aggregation* of the first k coflows to complete in a fair scheme. Utopia sequentially allocates bandwidth to each super-coflow towards the maximum progress. Utopia provides *provable* isolation guarantee: compared to a fair scheme, each coflow completes with *no more than a constant delay* in the offline scenario.

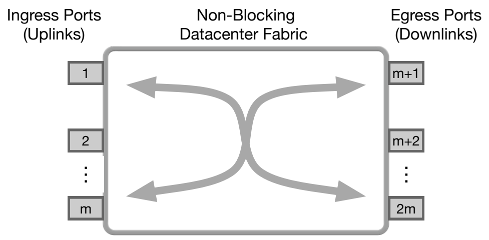


Fig. 1: An $m \times m$ non-blocking datacenter fabric with ingress/egress ports connecting to m machines.

We evaluated Utopia against performance-optimal Varys [5] and isolation-optimal HUG [9] through both trace-driven simulations and EC2 deployment on a 60-machine cluster. Evaluation results show that compared with HUG, Utopia reduces the average CCT by $1.8\times$, *without* delaying a single coflow. More impressively, even compared with performance-optimal Varys, Utopia slightly outperforms by 9%. To our knowledge, Utopia is the first coflow scheduler that simultaneously optimizes performance while providing provable isolation guarantee close to fair sharing.

II. Model and Objective

In this section, we describe our model for datacenter fabric and coflow scheduling. We clarify two objectives, i.e., minimizing the average CCT (performance) and optimizing isolation guarantee (fairness), followed by a brief review of prior works.

A. Model

Datacenter Fabric. Given that full-bisection bandwidth network is now available in production datacenters (DCs) [14], we model the DC fabric as a *non-blocking* switch [4], [5], [9], where the edges are the only place of congestions. As shown in Fig. 1, we assume an $m \times m$ DC fabric connecting m machines. Each machine i has a full-duplex link shown as an uplink connecting ingress port i and a downlink connecting egress port $m + i$. Without loss of generality, we assume that all links are of equal capacity normalized to one.

Coflow. The coflow abstraction captures the communication requirement between two computation stages in the BSP (bulk-synchronous parallel) model. A coflow consists of a collection of parallel flows across a group of machines. Not until all constituent flows have completed will a coflow complete.

In many applications such as MapReduce and machine learning, the amount of data transferred in the communication stage can be known *a priori* [5], [9], [15], e.g., shuffle between map and reduce. Specifically, we characterize coflow-k by a *demand vector* $\mathbf{d}_k = \langle d_k^1, \ldots, d_k^{2m} \rangle$, where d_k^i denotes the amount of data transferred on link-i. Among all links of coflow-k, we identify the most heavily loaded link b_i as the *bottleneck link*, i.e., $b_k = \arg\max_i d_k^i$, and let $\bar{d}_k = \max_i d_k^i$ be the *bottleneck demand*. To better characterize the demand correlation across links, we define *correlation vector* $\mathbf{c}_k = \langle c_k^1, \ldots, c_k^{2m} \rangle$, where c_k^i is the total amount of data transferred on link-i normalized by the bottleneck demand, i.e., $c_k^i = d_k^i / \bar{d}_k$. Meaning, for every bit coflow-k transfers on the bottleneck link, *at least* c_k^i bits should be transferred on link-i.

Given demand and correlation vectors (i.e., $\mathbf{d}_k$ and $\mathbf{c}_k$), the network scheduler determines, for each coflow-k, the bandwidth allocation $\mathbf{a}_k = \langle a_k^1, \ldots, a_k^{2m} \rangle$, where a_k^i is the share of bandwidth on link-i. Once the allocation has been given, the coflow transmission progress is bottlenecked on the *slowest* link. Formally, we measure the *progress* of a coflow as the minimum demand-normalized allocation across links, i.e.,

$$P_k = \min_{i: c_k^i > 0} a_k^i / c_k^i. \tag{1}$$

Intuitively, a coflow progress captures the attainable transmission rate on the slowest link, which critically affects the CCT.

B. Objectives

The general consensus [4] is that the network scheduler should minimize average CCT for optimal performance while providing isolation between coflows in a shared network.

Minimizing Average CCT. For data-intensive applications [4], [16], communication is frequently a bottleneck. Consequently, a network scheduler should finish *as many coflows as possible*, each in its fastest possible way. We therefore settle for minimizing the average CCT as the first and foremost objective for coflow scheduling.

Isolation Guarantee (Fairness): In multi-tenant environments such as public cloud, coexisting coflows from different users and applications contend for communication bandwidth in a shared network. To ensure performance isolation between users and applications, a network scheduler should provide a *minimum progress guarantee*. Following Chowdhury et al. [9], given an allocation, we measure *isolation guarantee* as the *minimum progress* across coflows, i.e., $\min_k P_k$. A direct approach to optimize isolation guarantee is to seek an allocation that maximizes the minimum progress, i.e.,

$$\text{maximize} \quad \min_k P_k. \tag{2}$$

C. Performance-optimal vs. Isolation-optimal

Despite the rich literature on coflow scheduling, a large body of work settle on one goal—either performance or isolation—as the primary objective, while treating the other as a *secondary* concern, often without an explicit focus. Among these works, Varys [5] and HUG [9] are the two representative schedulers that respectively optimizes performance and isolation.

Performance-optimal Varys. In a nutshell, Varys [5] minimizes average CCT through the *smallest-effective-bottleneck-first* (SEBF) heuristic. In our model, the *effective bottleneck* of a coflow is equivalent to its bottleneck demand $\bar{d}_k$, and the SEBF heuristic greedily schedules coflows in an ascending order of their bottleneck demand. While optimal in performance, Varys provides no isolation guarantee, as large coflows would give way bandwidth to small contenders. In fact, Chowdhury et al. [9] showed that Varys delays the maximum shuffle completion time by 77% in production MapReduce traces.

Isolation-optimal HUG. As opposed to Varys, HUG [9] targets to provide optimal isolation guarantee by means of fair sharing. Specifically, HUG employs a two-stage allocation algorithm. In the first stage, it seeks a Dominant Resource Fairness (DRF) [10] allocation to optimize isolation guarantee. To this end, it *equally* increases the progress of each coflow to the *maximum level*, i.e.,

$$P^* = \frac{1}{\max_i \sum_k c_k^i}. \tag{3}$$

In the second stage, it allocates spare bandwidth to speed up coflow completion on a *best-effort* basis. Compared with Varys, HUG results in longer average CCT and delays shuffle completion time by 45% on average in production traces [9].

Coflex as a Middle Ground. The recently proposed Coflex [13] comes as a middle ground between Varys and HUG. In particular, Coflex navigates the tradeoff space between performance and isolation through a tunable *fairness knob* α in the range of 0 and 1. Coflex allocates bandwidth in two stages. In the first stage, it equally increases the progress of each coflow to an α fraction of the optimum, providing isolation guarantee αP^*. In the second stage, it turns to minimizing the average CCT by allocating the spare bandwidth using the SEBF heuristic similar to Varys.

To our knowledge, Coflex is the only work that explicitly considers the two objectives. However, the choice of fairness knob critically depends on the underlying workload and cannot be easily determined in real-world systems. In fact, even if the knob has been carefully tuned for a particular workload, Coflex still cannot achieve the best of both worlds.

D. Achieving the Best of Both Worlds in a Long Run

We argue that the tradeoff between performance and isolation is not a fundamental limit for coflow scheduling, but a consequence of an *artificial restriction* that optimal isolation guarantee must be provided by means of *instantaneous* fair allocation. That is, at all time instant, the progress of each coflow must be maintained at the maximum level P^*.

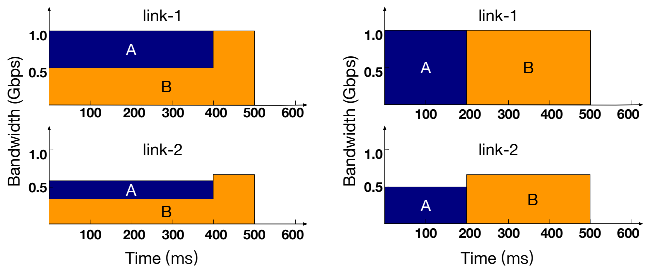
However, from an application's perspective, the effect of isolation guarantee can only be observed in the *long run* when the coflow *completes*. Intuitively, if we use fair scheduling (e.g., DRF [10] or HUG [9]) as a *baseline* algorithm, from an application's view, as long as its coflow completes *no later* than it would have had in the baseline scheme, its isolation is guaranteed in the long run. This motivates us to turn to *long-term* isolation guarantee as follows.

Definition 1 (Long-term isolation guarantee): For coflow-k, let F_k be its CCT with scheduler S, and let F_k^{F} be the CCT in a fair scheme that enforces *instantaneous* fair allocation with the optimal isolation guarantee P^*. We say scheduler S provides *long-term isolation guarantee* if it completes each coflow-k within a *constant* delay D beyond F_k^{F}, i.e.,

$$F_k \leq F_k^{\mathrm{F}} + D. \tag{4}$$

We shall show in the next section that long-term isolation guarantee is not necessarily in conflict with optimal performance. Our objective in this paper is to achieve the best of both worlds in a long run—minimizing the average CCT while providing long-term isolation guarantee, i.e.,

$$\begin{aligned} \text{minimize} \quad & \sum_k F_k, \\ \text{s.t.} \quad & F_k \leq F_k^{\mathrm{F}} + D, \quad \text{for all coflow-}k. \end{aligned} \tag{5}$$



(a) DRF (b) Prioritize coflow-A over coflow-B

Fig. 2: Illustration of the inefficiency of fair scheduling. Two coflows contend on two 1-Gbps links: coflow-A with demand vector $\mathbf{d}_A = \langle 200, 100 \rangle$ and coflow-B with $\mathbf{d}_B = \langle 300, 200 \rangle$. (a) Bandwidth share over time using DRF. (b) Bandwidth share over time where coflow-A is prioritized over coflow-B.

III. Key Insight and Challenges

In this section, we present our key insight of achieving both optimal performance and isolation guarantee in the long run. We also discuss the challenges to implement this insight.

A. Key Insight

We begin our discussion by analyzing why providing isolation by means of fair sharing poisons the performance.

Inefficiency of Fair Sharing. Fair schedulers, notably HUG [9] and DRF [10], seek to enforce the same progress P^* across all coflows. However, blindly enforcing an equal progress irrespective of coflow demand results in a long average CCT. Intuitively, those small coflows that could have completed faster if receiving more bandwidth than fair share are now forced to stay at the same progress with others.

To illustrate this point, we refer to a simple example in Fig. 2, where two coflows contend on two 1-Gbps links. Coflow-A transfers 200 Mb on link-1 and 100 Mb on link-2, and has demand $\mathbf{d}_A = \langle 200, 100 \rangle$; coflow-$B$ has $\mathbf{d}_B = \langle 300, 200 \rangle$. Because coflow-$A$ transfers less amount of data on the bottleneck link than coflow-B, the former is considered smaller than the latter. As shown in Fig. 2a, with DRF allocation, both coflows equally share the *bottleneck* link-1 with the same progress of 0.5 Gbps.[1] However, this is inefficient and unnecessary as we are only concerned with the coflow completion time. In fact, if we give up on fair sharing but instead *prioritize* the small coflow over the large one, we could speed up the former by $2\times$ *without delaying* the latter, as illustrated in Fig. 2b.

Key Insight. We learn from the simple example above that maintaining instantaneous fairness (i.e., equal progress) at all

[1]Link-2 is not fully utilized as DRF enforces the same completion time across all flows within a coflow.

time is too restrictive and *not necessary* to retain *long-term* isolation guarantee; instead, it can stall coflow completion. In fact, long-term isolation guarantee imposes a rather loose requirement that each coflow should not be delayed long beyond its completion under fair scheduling. Therefore, to minimize the average CCT, we should prioritize *as many small coflows as possible*, while scheduling the other large coflows only when needed, i.e., close to their completion in a fair scheme.

Following this observation, our key insight is to *preferentially schedule coflows in ascending order of their completion time under fair scheduling, i.e., DRF [10] or HUG [9].* We argue that this simple approach can potentially achieve the best of both worlds in two aspects.

- *Near-optimal performance:* First, by serving coflows following their completion order in fair scheduling, we expect that the resultant scheduling scheme resembles the performance-optimal SEBF (smallest-effective-bottleneck-first) heuristic in Varys [5]. Because fair scheduling enforces an equal progress across all coflows, those with small bottleneck demand (i.e., effective bottleneck) likely complete early in it. These coflows are therefore scheduled at higher priority than the others with larger bottleneck demand, which aligns with the behaviors of SEBF.
- *Long-term isolation guarantee:* Second, because coflows are prioritized based on their completion order in fair scheduling, a coflow can only be delayed by its *predecessors* who complete earlier in the fair scheme, but not its successors. We therefore expect the coflow to achieve a shorter CCT than it would have had in the fair scheme, hence providing long-term isolation guarantee.

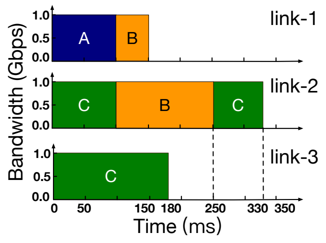
B. Challenges

However, directly implementing the insight above is challenging. It is well known that *greedily* offering available bandwidth to coflows following their priorities is usually not optimal [5], [8], mainly because coflows have *correlated* bandwidth demands across *multiple* links. To address this problem, a common approach [5], [16] is to use the Minimum-Allocation-for-Desired-Duration (MADD) algorithm to *sequentially* allocate bandwidth for each coflow, from the highest priority to the lowest. However, we show that this approach suffers from *priority inversion* that harms isolation guarantee.

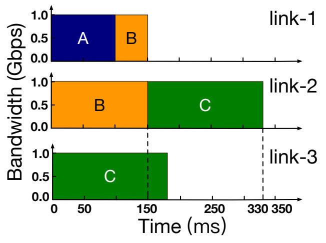
Minimum-Allocation-for-Desired-Duration. In a nutshell, the MADD algorithm [5], [16] allocates a coflow the *least amount* of bandwidth to attain the *maximum possible progress*. Specifically, suppose that there remains R_i amount of spare bandwidth on link-i, which is used to serve coflow-k with demand correlation vector $\mathbf{c}_k = \langle c_k^1, \ldots, c_k^{2m} \rangle$. By Eq. (1), the maximum attainable progress of the coflow is

$$\bar{P}_k = \min_{i: c_k^i > 0} R_i / c_k^i,$$

To achieve this maximum progress, the amount of bandwidth allocated on link-i is at least $\bar{P}_k c_k^i$. MADD produces exactly the minimum allocation $\bar{P}_k \mathbf{c}_k$.



(a) MADD scheme



(b) Optimal scheme

Fig. 3: MADD results in priority inversion. Three coflows contend on three 1-Gbps links, each having demand $\mathbf{d}_A = \langle 100, 0, 0 \rangle$, $\mathbf{d}_B = \langle 50, 150, 0 \rangle$, and $\mathbf{d}_C = \langle 0, 180, 180 \rangle$. Coflows are scheduled with priority $A > B > C$. (a) With MADD, coflow-B is preempted by low-priority coflow-C. (b) Optimal schedule.

Given the scheduling order of coflows, existing approaches use MADD algorithm to sequentially allocate bandwidth for each coflow, one after another. Bandwidth left unallocated by MADD, if any, is backfilled to coflows in the end. Intuitively, MADD speeds up a coflow using the least amount of bandwidth, so as to save *as much spare bandwidth as possible* for the next coflow. MADD has been successfully applied to implement the SEBF heuristic in Varys [5].

Priority Inversion. However, using MADD to track the completion order under fair scheduling is problematic: a coflow can be *preempted* by another with *lower priority* on the bottleneck link, causing unnecessary delay. We consider a toy example where three coflows contend on three 1-Gbps links. coflow-A transfers 100 Mb on link-1 and has demand $\mathbf{d}_A = \langle 100, 0, 0 \rangle$; coflow-$B$ has demand $\mathbf{d}_B = \langle 50, 150, 0 \rangle$; coflow-$C$ has demand $\mathbf{d}_C = \langle 0, 180, 180 \rangle$. Given that coflow-$A$ has the lowest bottleneck demand of 100 Mb, it completes the earliest under fair scheduling, followed by B and C. Following our insight described in the previous subsection, we should schedule the three coflows with priority $A > B > C$.

Fig. 3a depicts the allocation scheme produced by MADD. Because coflow-A is of the highest priority, it monopolizes the entire link-1. The other two idle links are then offered to coflow-B. However, without link-1, coflow-B cannot gain any progress as it has demand on it. Consequently, MADD allocates no bandwidth to coflow-B but offers links 2 and 3 to coflow-C. Coflow-B hence gets preempted by low-priority coflow-C on its bottleneck link-2, which delays its completion as opposed to the optimal scheme depicted in Fig. 3b.

We attribute the root cause of priority inversion to MADD *using attainable progress to justify bandwidth allocation*, which has led to a *misbelief* that transferring on a link without making progress only wastes the bandwidth allocation, and should be ruled out in the first place. This is a mistake. We refer back to the previous example in Fig. 3b and focus on coflow-B. While getting blocked on link-1, coflow-B should still transfer on link-2. Even though this gains no progress for coflow-B, it reduces the backlog on the bottleneck link, which in turn speeds up the coflow completion.

IV. Utopia

In this section, we present a new network scheduler, which we call *Utopia*, in that it achieves the best of both worlds for coflow scheduling. We start to focus on offline scheduling with a novel algorithm that sequentially aggregates coflows into *super-coflows*. We show that this algorithm provides long-term isolation guarantee. We then generalize the offline solution to an efficient *online* scheduler for dynamic coflow arrivals.

A. *Offline Scheduling*

We consider an offline problem of scheduling N coflows $C_1, \ldots, C_N$ that arrived at time 0. Without loss of generality, let coflows be indexed based on their completion order assuming fair scheduler DRF,[2] where coflow C_1 has the least amount of bottleneck demand and completes first, followed by C_2, and so on. A coflow that completes earlier in DRF is assigned a higher priority than its DRF successor. We therefore obtain a prioritized scheduling scheme $\mathbb{C} = (C_1, \ldots, C_N)$.

Super-coflow. Given scheduling scheme $\mathbb{C}$, we aggregate coflows into *super-coflows* $\mathbb{S} = (S_1, \ldots, S_N)$, where S_k is a *sequential aggregation* of the first k coflows $C_1, \ldots, C_k$. Specifically, let $\mathbf{D}_k = \langle D_k^1, \ldots, D_k^{2m} \rangle$ be the demand vector of S_k, where D_k^i is the amount of data the super-coflow transfers on link-i. We have $\mathbf{D}_k = \sum_{l=1}^{k} \mathbf{d}_l$. In other words, the demand of a super-coflow is simply the cumulative demands of all its coflows. Super-coflow S_1 trivially reduces to coflow C_1 as a special case.

Unlike prior works which focus on individual coflows [4], [5], [7], [9], [13], we seek to speed up the completion of super-coflows, each in its fastest possible way. In particular, our algorithm *sequentially* allocates bandwidth to each super-coflow atop the previous allocation, with an objective of achieving the *maximum progress*. Such a sequential allocation for super-coflows $\mathbb{S}$ aligns with scheduling scheme $\mathbb{C}$. Recall that within super-coflow S_k, coflow C_k is scheduled at the lowest priority and is likely the last one to complete. Therefore, speeding up the entire super-coflow also speeds up C_k.

Compared with prior works, allocating bandwidth at the granularity of super-coflows provides two benefits:

1) *It enables the scheduler to have a global view of the cumulative demands, based on which bandwidth allocation can be well planned.* For example, with the knowledge of the bottleneck link of super-coflow S_k, the scheduler can better justify the bandwidth allocation for coflow C_k. If the coflow has demands on the bottleneck link of S_k, its completion is likely bounded by the cumulative demands on that link. Therefore, allocating it more bandwidth on the other links will not reduce its CCT, and is not justified. Such a justification is not possible if we only focus on individual coflows.
2) *It avoids priority inversion as much as possible.* In fact, an allocation to coflow C_k is justified as long as it helps to speed up the entire super-coflow S_k, even if the allocation translates into no progress for C_k. This in stark contrast to Varys [5], where a coflow easily yields bandwidth to low-priority contenders if it gains no progress from the allocation (cf. Fig. 3).

[2]DRF and HUG share the same coflow completion order, as they enforce an equal progress across all coflows.

Bandwidth Allocation. Given scheduling scheme $\mathbb{S} = (S_1, \ldots, S_N)$, our algorithm allocates bandwidth in *rounds*. In round k, it allocates super-coflow S_k the *least amount* of bandwidth to complete it in *minimum possible* time, *without overriding* the allocations for $S_1, \ldots, S_{k-1}$ settled in previous rounds. Taking the difference between allocations for S_k and S_{k-1}, we obtain the bandwidth allocation for coflow C_k. Formally, let f_k^{ij} be a flow in C_k transferring data from uplink-i (ingress port) to downlink-j (egress port) in the fabric. We next explain how our algorithm computes the amount of bandwidth r_k^{ij} allocated for each flow $f_k^{ij} \in C_k$ in round k.

We start to compute the minimum attainable CCT of super-coflow S_k if it was running *alone* in the fabric, which is simply the *bottleneck demand* assuming unit link capacity, i.e.,

$$\bar{D}_k = \max_i D_k^i.$$

Let D_k^{ij} be the amount of data transferred from uplink-i to downlink-j in S_k. To achieve the minimum CCT, we need to allocate *at least* $D_k^{ij}/\bar{D}_k$ bandwidth for traffics between the two links. Note that among these traffics, those belonging to coflows $C_1, \ldots, C_{k-1}$ have already been allocated bandwidth in the previous rounds along with super-coflows $S_1, \ldots, S_{k-1}$, i.e., flow f_l^{ij} received r_l^{ij} share of bandwidth, where $l < k$. Because these flows have higher priorities than f_k^{ij}, their allocations $\sum_{l<k} r_l^{ij}$ must be guaranteed. Therefore, out of the planned allocation $D_k^{ij}/\bar{D}_k$, we can allocate flow f_k^{ij} *at most*

$$(D_k^{ij}/\bar{D}_k - \textstyle\sum_{l<k} r_l^{ij})^+, \tag{6}$$

where $(x)^+ = \max\{0, x\}$ is a ramp function that takes the *positive part* of x. Also note that the amount of bandwidth allocated to flow f_k^{ij} is bounded by the available bandwidth of the two access links. We finally have

$$r_k^{ij} = \min\{(D_k^{ij}/\bar{D}_k - \textstyle\sum_{l<k} r_l^{ij})^+, R_i, R_j\}, \tag{7}$$

where R_i and R_j are the amount of remaining bandwidth on link-i and link-j, respectively. We refer to `allocBandwidth(`$\mathbb{C}$`)` in Algorithm 1 as a summary of the entire process.

Retaining Work Conservation. To achieve work-conserving allocations, the final stage of our algorithm is to distribute unused bandwidth, if any, to coflows. We adopt the same back-filling strategy as in Varys [5]: for each uplink-i, we allocate its remaining bandwidth R_i to active flows, in proportion to their current allocation (r_k^{ij}) ratio, subject to the capacity constraints in the corresponding downlink-j. We summarize our offline solution as `UtopiaOffline(`$\mathbb{C}$`)` in Algorithm 1.

Example. To better illustrate the algorithmic behaviors of Utopia, we refer back to the previous example in Fig. 3b, where three coflows are scheduled in ascending order of their bottleneck demands, following scheme $\mathbb{C} = (A, B, C)$. Utopia allocates the entire link-1 to coflow-A in recognition of its top

Algorithm 1 Utopia

```
procedure ALLOCBANDWIDTH(Coflows $\mathbb{C}$)
    Initialize remaining b/w $R_i \leftarrow 1$ on all link-$i$
    for $k = 1$ to $|\mathbb{C}|$ do
        Aggregate demands into super-coflow $\mathbf{D}_k = \sum_{l=1}^{k} \mathbf{d}_l$
        $\bar{D}_k \leftarrow \max_i D_k^i$
        for all flow $f_k^{ij} \in C_k$ do
            $r_k^{ij} \leftarrow \min\{(D_k^{ij}/\bar{D}_k - \sum_{l<k} r_l^{ij})^+, R_i, R_j\}$
            $R_i \leftarrow R_i - r_k^{ij}$    ▷ Update remaining b/w on link-$i$
            $R_j \leftarrow R_j - r_k^{ij}$    ▷ Update remaining b/w on link-$j$
procedure UTOPIAOFFLINE(Coflows $\mathbb{C}$)
    Sort $\mathbb{C}$ in ascending order of completion time under DRF
    allocBandwidth($\mathbb{C}$)
    Distribute unused bandwidth to all $C \in \mathbb{C}$
procedure UTOPIAONLINE(Coflow $C$, Bool isArrival)
    if isArrival then
        $\mathbb{C} \leftarrow \mathbb{C} \cup \{C\}$    ▷ Coflow $C$ arrives
    else
        $\mathbb{C} \leftarrow \mathbb{C} \setminus \{C\}$    ▷ Coflow $C$ completes
    Update the remaining demands for all $C \in \mathbb{C}$
    UtopiaOffline($\mathbb{C}$)
```

priority. In the second round, it aggregates A and B into a super-coflow S_2 with demand $D_2 = \langle 150, 150, 0\rangle$. If running alone, super-coflow S_2 would have fully taken both link-1 and link-2. Excluding link-1 which has already been given to coflow-A, we allocate the entire link-2 to coflow-B. In the final round, we aggregate all three coflows into S_3 with demand $\mathbf{D}_3 = \langle 150, 330, 180\rangle$. If running alone, the super-coflow takes 330 ms to complete, with minimum allocation $\langle \frac{5}{11}, 1, \frac{6}{11}\rangle$ on three links. Excluding link-1 and link-2 from consideration as they have already been allocated and cannot be revoked, we shall allocate $\frac{6}{11}$ of link-3 to coflow-C. The unused share $\frac{5}{11}$ is distributed to coflow-C in the final stage. In the end, Utopia settles on the optimal allocation depicted in Fig. 3b.

B. Long-term Isolation Guarantee

Utopia provides long-term isolation guarantee (given by Definition 1) for offline scheduling: each coflow is guaranteed to complete within a small *constant* bound beyond its completion in DRF. Formally, we have the following theorem.

Theorem 1 (Long-term isolation guarantee): Assume that coflows $\mathbb{C}$ arrived at time 0. For coflow $C_k \in \mathbb{C}$, let F_k be its CCT in Algorithm 1, and F_k^{F} its CCT in DRF. We bound the maximum completion delay beyond DRF as follows:

$$F_k - F_k^{\mathrm{F}} \le \bar{d}_{\max}, \tag{8}$$

where $\bar{d}_{\max}$ is the maximum bottleneck demand of a coflow.

We make two remarks on Theorem 1:

1) The delay bound $\bar{d}_{\max}$ is a small constant in production datacenters. Recall that we assume a unit link capacity. The bottleneck demand also measures the *minimum* CCT if the coflow were transferring *alone* in the fabric. Given the high-speed, full-bisection bandwidth fabric in production datacenters, transferring a coflow alone can be very fast.
2) The delay bound is established as a worst case guarantee, but coflows usually complete faster with much shorter average CCT. In fact, as we shall show in Sec. V, no coflow gets delayed in our evaluation.

We next give a proof sketch of Theorem 1. The complete proof is deferred to our technical report [17] due to space constraints.

Proof Sketch. To bound the completion delay of coflow C_k, we turn to super-coflow S_k. We differentiate between the following two cases.

Case-1: Throughout the transmission of S_k, its bottleneck link has been fully utilized. In this case, we show that coflow C_k cannot be delayed, i.e., $F_k \le F_k^{\mathrm{F}}$.

Because the bottleneck link of S_k is kept busy, the CCT of super-coflow S_k is simply the time to finish transferring its bottleneck demand $\bar{D}_k$ at full bandwidth. By that time, coflow C_k must have completed, i.e., $F_k \le \bar{D}_k$. We now turn to DRF. Recall that C_k is the k^{th} coflow to complete in DRF. By the time it finishes in DRF, the previous $k-1$ coflows $C_1, \dots, C_{k-1}$ must have completed, so does super-coflow S_k. Given that $\bar{D}_k$ is the minimum possible CCT of S_k, we have

$$F_k^{\mathrm{F}} \ge \bar{D}_k \ge F_k. \tag{9}$$

Case-2: During the transmission of S_k, the bottleneck link of S_k is not fully utilized at some time. This occurs when the allocation on the bottleneck link is restricted due to the lack of available bandwidth on the coupled links (line 7 in Algorithm 1). Specifically, let B_k be the bottleneck link of S_k. We say link l is *coupled* with B_k if there exists a flow in S_k communicating between l and B_k whose allocation is less than $D_k^{lB_k}/\bar{D}_k$, i.e., its DRF share if S_k is running *alone* in the fabric. Let $L(B_k)$ be the set of links coupled with B_k. When B_k is not fully utilized, link l in $L(B_k)$ must have run out of bandwidth,[3] and is fully used to transfer flows in S_k.

Let t_B be the time when bottleneck link B_k starts to transfer at full bandwidth. The only reason that B_k cannot be saturated before t_B is due to the allocation of the first $k-1$ coflows on its coupled links $L(B_k)$. Such a roadblock will be cleared after those coflows have completed on all coupled links. Formally, let t_l be the time when coflows $C_1, \dots, C_{k-1}$ complete on coupled link $l \in L(B_k)$. We have $t_B \le \max_{l\in L(B_k)} t_l$. We next focus on the worst case where $t_B = \max_{l\in L(B_k)} t_l$ but defer the complete analysis to our technical report [17].

We turn to DRF and show $t_l \le F_k^{\mathrm{F}}$ for all link $l \in L(B_k)$. This is because by the time coflow C_k completes in DRF, all the previous $k-1$ coflows must have completed. Therefore, the CCT of coflow C_k in DRF is at least t_l. Putting it altogether, we have $t_B \le F_k^{\mathrm{F}}$.

Also note that, by time t_B, the first $k-1$ coflows have completed transferring data on bottleneck link B_k. From then on, bottleneck link B_k is dedicated to transferring coflow C_k until its completion. We therefore have

$$F_k \le F_k^{\mathrm{F}} + \bar{d}_k \le F_k^{\mathrm{F}} + \bar{d}_{\max}. \tag{10}$$

[3]Otherwise, Utopia would have used the spare bandwidth to transfer flows between a coupled link and bottleneck B_k.

C. From Offline to Online

Our solution can be easily generalized to *online* scheduling with dynamic coflow arrivals. Specifically, we maintain a list of *active* coflows sorted in ascending order of their CCTs in DRF. Upon an arrival (departure) of a coflow, we insert (remove) it into (from) the list and update the scheduling order. We then allocate bandwidth based on the new order. The `UtopiaOnline` routine in Algorithm 1 summarizes the entire process.

Towards an efficient implementation of Utopia, we need a fast algorithm to track the coflow completion order under DRF. This can be done through the *virtual time* approach widely used in the fair queueing literature [18], [19]. In particular, we define *virtual time* $V(t)$ as a function that increases at the marginal rate equal to the coflow progress under DRF, i.e., $\frac{\mathrm{d}}{\mathrm{d}t}V(t) = P^*$, where P^* follows (3). For each coflow C_k, we associate it with a *virtual finish time* V_k^{F} upon its arrival, i.e.,

$$V_k^{\mathrm{F}} = V(t_k^{\mathrm{A}}) + \bar{d}_k, \tag{11}$$

where t_k^{A} is the arrival time of C_k, and $\bar{d}_k$ is its bottleneck demand. The virtual finish time of a coflow, once computed, requires no update in the future. By maintaining the list of active coflows in ascending order of virtual finish time, we can accurately track the coflow completion order in DRF.

V. Evaluation

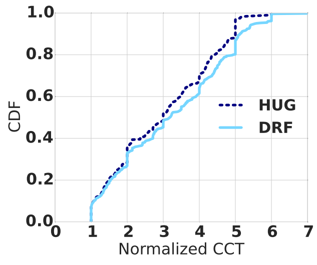
We evaluated Utopia through trace-driven simulations and testbed deployment in a 60-machine Amazon EC2 [20] cluster. The highlights of our evaluations are:

- Utopia offers long-term fairness with guaranteed isolation between coflows. It dominates HUG [9] and DRF [10] in both trace-driven simulations and EC2 deployment, without delaying any single coflow.
- Utopia minimizes the average CCT and outperforms HUG and DRF by $1.8\times$ and $1.9\times$, respectively. In fact, even compared with performance-optimal Varys, Utopia reduces the average CCT by 9%, showing no longer CCT in 97% of coflows.

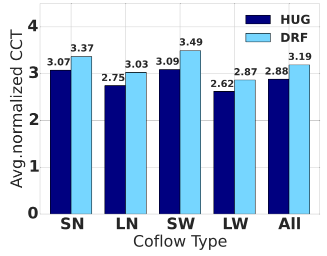
A. Trace-Driven Simulations

Workload. We used CoflowBenchmark [21] as input of our simulations. The benchmark consists of a one-hour snapshot of a production Hive/MapReduce trace collected from a 3000-machine cluster in Facebook [5]. The benchmark consists of 526 coflows scaled down to a 150-port DC fabric, where all mappers (reducers) in the same rack are combined into one rack-level mapper (reducer).

Setup. In our simulations, we modeled the DC fabric as a 150×150 non-blocking switch with 150 ingress/egress ports corresponding to the uplinks/downlinks of 150 racks connected to it. We implemented the scheduling logic of Utopia along with three baseline algorithms: DRF [10], HUG [9], and Varys [5]. Our implementations are based on `CoflowSim` [22], the *de facto* simulator for coflow scheduling. We evaluated isolation guarantee of Utopia against DRF and HUG, while comparing its performance with Varys.



(a) Distribution of normalized CCT



(b) Average normalized CCT

Fig. 4: Utopia dominants DRF and HUG with isolation guarantee. (a) Distribution of normalized CCT. (b) Average normalized CCT in coflow bins.

TABLE I: Coflows binned by their lengths (**S**hort or **L**ong) and widths (**N**arrow or **W**ide) in the Coflow-Benchmark [21].

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

Isolation Guarantee. We have shown in Theorem 1 that Utopia provides isolation guarantee for offline scheduling where coflows arrived at time 0. A natural question is: can this isolation guarantee be similarly provided in a more practical *online* setting where coflows dynamically arrive over time? To answer this question, we compared Utopia against DRF and HUG based on a metric called *normalized CCT*. Normalized CCT is defined, for each coflow, as the CCT under the compared scheduler normalized by that under Utopia, i.e.,

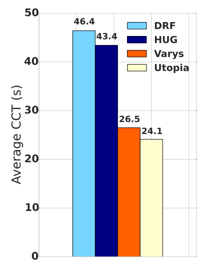
$$\text{Normalized CCT} = \frac{\text{Compared CCT}}{\text{CCT under Utopia}}.$$

Intuitively, if the normalized CCT is greater (smaller) than 1, the coflow finishes faster (slower) under Utopia.

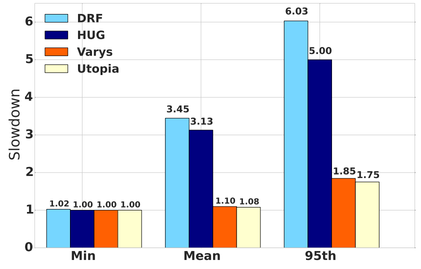
We measured the normalized CCT for each coflow under DRF and HUG, as shown in Fig. 4a. Utopia strongly dominates the two fair schedulers. Across all 526 coflows, the normalized CCT is consistently greater than 1. Meaning, *no* single coflow gets delayed by Utopia. In fact, Utopia speeds up the completion of over 70% of coflows by more than $2\times$. These encouraging results clearly indicate that Utopia provides long-term isolation guarantee for online scheduling, which complements our previous analysis in the offline setting.

To better understand the performance impact on different coflows, we further divide 526 coflows into four bins based on their shuffle types. Following [5], [7], [9], we consider a coflow *small* (*long*) if its largest flow is less (greater) than 5 MB, and *narrow* (*wide*) if it consists of *less* (more) than 50 flows. Table I summarizes the distribution of binned coflows.

We compare the average normalized CCT of the two fair schedulers in four coflow bins and plot the results in Fig. 4b. We observe a general trend that the normalized CCT of a small coflow (i.e., SN and SW) is usually higher than that of a large one, meaning that small coflows likely gain more salient speedup than large ones when switching to Utopia. We attribute this trend to Utopia emulating the "small-coflow-first" heuristic (i.e., SEBF) to a large extent by tracking the completion order under a fair scheduler. Because small coflows



(a) Average CCT



(b) Statistical summary of slowdown

Fig. 5: Illustration of Utopia's efficiency of minimizing CCT. (a) Average CCT under different schemes. (b) Statistical summary of the minimum, average, and the 95th percentile slowdown.

dominate in population (Table I), speeding up their completion critically improves the average performance. Overall, a coflow can expect an average speedup of 2.88$\times$ (3.19$\times$) if switching from HUG (DRF) to Utopia.

Near-optimal Performance. We next show that Utopia also achieves near-optimal performance. Fig. 5a compares the average CCT of Utopia as opposed to the three baseline algorithms. Utopia easily dominates the two fair schedulers, almost halving the average CCT. More impressively, Utopia even outperforms Varys with 9% shorter average CCT. In fact, we found that among 526 coflows, there are 508 coflows having *no longer* CCT, meaning that 97% of coflows are better off using Utopia.

Motivated by the remarkable results above, we are curious to know how far Utopia is from the optimal. To this end, we turn to another metric termed *completion slowdown*. Specifically, completion slowdown is defined, for each coflow, as the CCT in the compared scheme normalized by the *minimum possible* CCT if the coflow were running *alone* in the fabric, i.e.,

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

We measured the slowdown of each coflow under the four schedulers. Fig. 5b gives a statistical summary of the minimum, average, and the 95th percentile slowdown. We see that with Utopia, coflows are only slowed down by 8% on average as compared with running alone—a strong evidence that Utopia has pushed the performance to the limit, without many space for further improvement.

B. EC2 Deployment

Implementation. We have prototyped Utopia in `Python` with a master-slave architecture. The `master` identifies coflows and makes scheduling decisions following the logic of Algorithm 1; a `slave` runs in a cluster machine as a daemon program and enforces the specified flow transmission rate using Linux's `tc` and `htb qdisc` tools. In particular, upon arrival, a coflow registers to the `master` with its transmission demands through a public Utopia API. The `master`, after receiving the registration, computes a new allocation for active coflows and notifies the flow transmission rates to the corresponding `slaves` for local enforcement. A `slave` periodically sends heartbeat messages to the `master` to update its status. Once a coflow departure has been detected, the `master` quickly responds with a new allocation. We have developed a dedicated plugin based on `Ansible` [23] to automate the deployment and management of Utopia in a large cluster.

TABLE II: Summary of coflow information in experiment.

	Commun. Pattern	# of Flows	Arrival Time
coflow-A	all-to-all	360	0 s
coflow-B	pairwise one-to-one	60	10 s
coflow-C	pairwise one-to-one	60	20 s

Cluster deployment. We performed experiments in a 60-node Amazon EC [20] cluster. For each node, we used a `c4.xlarge` EC2 instance with 4 cores and 7.5 GB RAM. For simplicity, we configured 200 Mbps as the bandwidth capacity to each uplink/downlink.

Micro-benchmark. We micro-benchmarked the behavior of Utopia against three coflows of different communication patterns contending in a 60-machine cluster. Table II summarizes the configurations of three coflows. In particular, coflow-A contains 10 groups of all-to-all shuffle, each performing 6×6 communications. In total, coflow-A has 360 flows. coflow-B has 60 flows following a *pairwise one-to-one* communication pattern between machine i and machine $i + 30$, where $1 \leq i \leq 30$. coflow-C also has 60 flows following the same communication pattern, where machine j communicates with machine $j + 15$, for all $1 \leq j \leq 15$ and $30 \leq j \leq 45$. In total, we have 480 flows in three coflows. For each flow, we randomly configured its size between 30 MB and 100 MB. In this setting, coflow-A is considered large, whereas the other two coflows are considered small. Table II summarizes the information of the three coflows.

*1) **Coflow Completion Time:*** Fig. 6 depicts the CCTs of three coflows using three schedulers. We see that Utopia consistently outperforms HUG and Varys, resulting in shorter CCTs of all three coflows. Compared with HUG, Utopia significantly speeds up the two "small" coflows (B and C)—a consequence that Utopia preferentially serves small coflows that complete early under fair scheduling.

*2) **Coflow Progress:*** Fig. 7 depicts the progress of each coflow under different schedulers over time. As expected, HUG enforces the same progress at all time, which unnecessarily delays coflow completion. Both Utopia and Varys prioritize small coflows: large coflow-A is preempted by the two small coflows. We note that even following the same scheduling priority, Utopia outperforms Varys with shorter CCT of coflow-C. We attribute this advantage to the fact that Utopia does not use progress to justify bandwidth allocation, but takes a global view based on super-coflows. In fact, we have observed that coflow-C starts to transfer on some links upon its arrival, even without gaining any progress. Such an early start pays off in the end with faster coflow completion.

VI. Related Work

Many recently proposed coflow schedulers focus on minimizing average CCT. For example, unlike Varys [5] which is

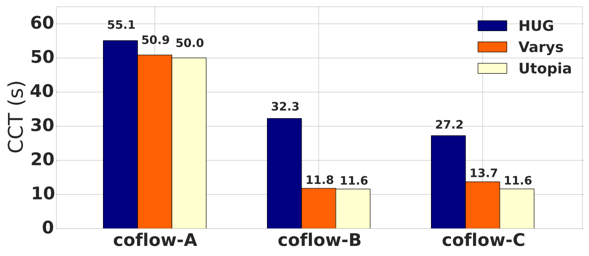


Fig. 6: CCT of three coflows in a 60-machine cluster.

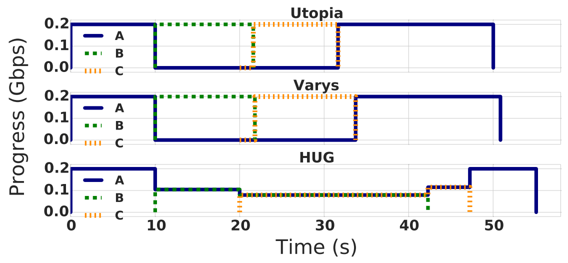


Fig. 7: Coflow progress measured over time.

a centralized scheduler, Baraat [15] employs a decentralized design based on FIFO. Aalo [6] and CODA [7] come as *non-clairvoyant* schedulers that make scheduling decisions without *a priori* knowledge about the flow size and/or coflow identification. Qiu et al. [8] proposed the first approximation algorithm for clairvoyant coflow scheduling assuming full knowledge of future coflow arrivals. These schedulers, while excelling in performance, are incapable of isolating coflows with predictable performance.

To achieve predictable networking performance, fair network schedulers, notably FairCloud [11], HUG [9] and its variant [12], have been proposed to provide optimal isolation guarantee with high network utilization. These scheduling policies, however, is inefficient in minimizing the average CCT. Coflex [13] comes as a middle ground to navigate the tradeoff between fairness and performance. Instead of seeking instantaneous fair allocations, Utopia aims to provide long-term isolation guarantee, which allows it to achieve the best of both worlds.

We note that the idea of giving up instantaneous fairness for higher efficiency has been explored in an early work [24] for web server protocols. Similar insight has recently been applied in the context of job scheduling [25], in which parallel tasks are assigned to compute slots for fast job completion. All these works focus on *single-resource* scheduling. In contrast, coflows have correlated demands across multiple links, and a coflow scheduler must deal with *multi-resource* scheduling with *coupled* constraints on ingress/egress ports [8].

VII. Conclusion

In this paper, we have proposed a new network scheduler, Utopia, to minimize the average CCT without compromising isolation guarantee. Utopia achieves the best of both worlds by preferentially scheduling coflows in ascending order of their CCTs under DRF. To avoid priority inversion, Utopia employs a novel bandwidth allocation algorithm based on super-coflow, which is the sequential aggregation of the first k coflows to complete in DRF. We have shown that with Utopia, each coflow is guaranteed to complete within a small constant bound beyond its completion under DRF. Both trace-driven simulations and EC2 deployment have confirmed that Utopia achieves near-optimal performance with isolation guarantee.

Acknowledgement

This research was support in part by RGC GRF grants under the contracts 16211715 and 16206417, as well as an RGC CRF grant under the contract C7036-15G.

References

[1] 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 *USENIX NSDI*, 2012.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX OSDI*, 2004.

[3] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[4] M. Chowdhury, "Coflow: A networking abstraction for distributed data-parallel applications," Ph.D. dissertation, University of California, Berkeley, 2015.

[5] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM*, 2014.

[6] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM*, 2015.

[7] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "CODA: Toward automatically identifying and scheduling coflows in the dark," in *ACM SIGCOMM*, 2016.

[8] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *ACM SPAA*, 2015.

[9] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *USENIX NSDI*, 2016.

[10] 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.

[11] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: sharing the network in cloud computing," in *ACM SIGCOMM*, 2012.

[12] W. Wang and A.-L. Jin, "Friends or foes: Revisiting strategy-proofness in cloud network sharing," in *IEEE ICNP*, 2016.

[13] W. Wang, S. Ma, B. Li, and B. Li, "Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling," in *IEEE INFOCOM*, 2017.

[14] 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.

[15] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM*, 2014.

[16] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM*, 2011.

[17] L. Wang, W. Wang, and B. Li, "Utopia: Near-optimal coflow scheduling with isolation guarantee," http://www.cse.ust.hk/~weiwa/papers/utopia_tr.pdf, HKUST, Tech. Rep., 2018.

[18] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, no. 3, pp. 344–357, 1993.

[19] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM SIGCOMM*, 1989.

[20] "Amazon ec2," http://aws.amazon.com/ec2.

[21] M. Chowdhury, "Coflow-Benchmark," https://goo.gl/szsBQE.

[22] "Coflowsim," https://github.com/coflow/coflowsim.

[23] "Ansible," https://www.ansible.com/.

[24] E. J. Friedman and S. G. Henderson, "Fairness and efficiency in web server protocols," in *ACM SIGMETRICS*, 2003.

[25] C. Chen, W. Wang, S. Zhang, and B. Li, "Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees," in *IEEE INFOCOM*, 2017.