Yosemite: Efficient Scheduling of Weighted Coflows in Data Centers

Paper ID: #1570386872

***Abstract*—Traditional network resource allocation or scheduling mechanisms are mainly flow or packet based. Recently, coflow has been proposed as a new abstraction to capture the communication patterns in a rich set of data parallel applications in data centers. Coflows effectively model the application-level semantics of network resource usage, so high-level optimization goals, such as reducing the transfer latency of applications, can be better achieved by taking coflows as the basic elements in network resource allocation or scheduling. Although efficient coflow scheduling methods have been studied in this aspect, in this paper, we propose to schedule *weighted coflows* as a further step in this direction, where weights are used to express the emergences or priorities of different coflows or their corresponding applications. We introduce the Weighted Coflow Completion Time (WCCT) minimization problem, and design an information-agnostic online algorithm to dynamically schedule cflows according to their weights and the instantaneous network condition. We implement the algorithm in a scheduling system named Yosemite, and evaluate its performance by trace-driven simulations as well as real deplopyment in a cloud testbed. Our evaluation results show that, compared to the latest information-agnostic coflow scheduling algorithms, Yosemite can reduce more than 40% of the WCCT, and more than 30% of the completion time for coflows with above-the-average level of emergence. It even outperforms the most efficient clairvoyant coflow scheduling method by reducing around 30% WCCT, and 20%∼30% of the completion time for coflows with above-the-average emergence, respectively.**

I. Introduction

Nowadays, low latency [20] [25] and high throughput [21] are required by data center applications, for example, those with map-reduce style data-intensive computing [14], and those using distributed file storage [22] [15], etc. To meet these requirements, the data center networking infrastructure has been specially tailored, with a tremendous amount of efforts to improve the network topology and routing schemes, as well as transport schemes.

Among these efforts, flow level scheduling methods try to schedule arriving packets according to the characteristics of the corresponding flows. For example, PDQ [17] and pFabric [9] approximate the *shortest job first* (SJF) policy to let shorter flows preempt the bandwidth of longer ones. As a result, the average Flow Completion Time (FCT) tends to be reduced, so does applications' transport latency. However, since the communication of an application cannot finish until all its flows have completed their transmission, the overall transport performance of an application may not be effectively improved, or may even be impaired, by scheduling individual flows without considering their inter-relationship.

Recently, coflow has been proposed as a new abstraction to capture the communication patterns in a rich set of data parallel applications. According to [11], a coflow is *a collection of flows between two groups of machines with associated semantics and a collective objective.* The collection of flows share a common performance goal, e.g., minimizing the completion time of the latest flow in the collection, or ensuring that flows in the collection meet a common deadline. Coflows effectively model the application-level semantics of network resource usage, so high-level optimization goals, such as reducing the transfer latency of applications, can be better achieved by taking coflows as the basic elements in network resource allocation or scheduling. Efficient coflow scheduling methods targeting minimizing Coflow Completion Time (CCT) have been studied. Varys [13] proposes the *Smallest-Effective-Bottleneck-First* (SEBF) heuristic to determine the scheduling order of coflows and uses *Minimum-Allocation-for-Desired-Duration* (MADD) to compute the bandwidth to be allocated. However, Varys is a clairvoyant method in the sense that it depends on flow information, such as flow size or flow arrival time, to determine how to scheduling coflows. This may limit its application to scenarios where such information cannot be provided a priori. To solve this problem, non-clairvoyant methods that are flow information agnostic, such as Aalo [12], Barrat [16], sunflows [18] and CODA [31], have also been studied.

In this paper, we propose to schedule *weighted coflows* as a further step in this direction, where weights are assigned to coflows to express the emergences or priorities of different coflows or their corresponding applications. For example, supporting the internal communication requirement of a search engine with intense internal computation should be more emergent than supporting the file backup on a distributed storage system. When scheduling coflows from these two applications, we would like to take this difference into consideration. To accomplish this, we assign different weights to them, i.e., coflows of search engine have a higher weight, while coflows of file backup have a lower weight, and try to minimize the Weighted Coflow Completion Time (WCCT), which is the weighted summation of the completion time of all coflows.

We formulate an Idealized Weighted Coflow Completion Time Minimization (IWCCTM) problem, and prove that one of its special cases is equivalent to minimizing the weighted job completion time in the concurrent open shop problem [24], which is known to be NP-hard. Following the idea presented in [24], we derive for IWCCTM a non-preemptive scheduling algorithm that is 2-approximate optimal. To make it more practical, we then convert it to an information-agnostic algorithm, which introduces only minor performance loss, but can schedule coflows online without knowing their size

or arrival time a priori. We further design and implement a scheduling system named Yosemite using the scala language. Yosemite relies on the Akka [2] and kryo [3] libraries, and can readily be deployed in a productional cloud environment like openstack. We have extensively tested the performance of Yosemite by both trace-driven simulations and testbed evaluations, and compare it against state-of-the-art coflow scheduling algorithms with best known performance, such as Varys [13] , Aalo [12] and Barrat [16]. The corresponding results show that Yosemite performs quite effectively in reducing the Weighted Coflow Completion Time.

Our contributions in this paper are summarized as follows:

- We are the first to propose the scheduling of *weighted coflows* as an essential network resource management scheme in data centers hosting multiple kinds of applications. We collect real application traffic from a medium-sized data center and make an in-depth analysis to reveal the importance of weighted coflow scheduling.
- We formulate an Idealized Weighted Coflow Completion Time Minimization (IWCCTM) problem, study its hardness, and derive a non-preemptive scheduling algorithm that is 2-approximate optimal.
- We further design a practical information-agnostic algorithm and build the Yosemite system to minimize WCCT in real environment. Yosemite can readily be deployed in a cloud environment like openstack.
- Our evaluation results show that, compared to the latest information-agnostic coflow scheduling algorithms, Yosemite can reduce more than 40% of the WCCT, and more than 30% of the completion time for coflows with above-the-average level of emergence. It even outperforms the most efficient clairvoyant coflow scheduling method by reducing around 30% WCCT, and 20%$\sim$30% of the completion time for coflows with above-the-average emergence, respectively.

The rest of the paper is organized as follows. Section II introduces the related work and our motivation. Section III formulates the WCCT minimization problem and discusses its hardness. Section IV presents our information-agnostic coflow scheduling algorithm and the practical Yosemite system in detail. Section V evaluates Yosemite against several coflow scheduling methods, and Section VI concludes.

II. Background and Motivation

Data center is now becoming an important facility for hosting a large number of services and applications. To meet their demands of high throughput and low latency, a tremendous amount of research effort has been devoted to network resource allocation. For example, DCTCP [8], D^{2}TCP [29], L^{2}DCT [26] , LPD [32] ,D3 [30] and PDQ [17] are flow level rate control or scheduling schemes aiming to minimize the Flow Completion Time (FCT) or to meet strict flow deadlines. The concept of coflow has recently been proposed [11] to meet the application performance requirement at a higher level, where a coflow is a collection of flows between two groups of machines with associated semantics and a collective objective. Algorithms are also devised to schedule coflows to minimize the Coflow Completion Time (CCT), and are of the most relevance to our problem. In particular, Varys computes network bottleneck based on flow size and applies its *Effective-Bottleneck-First* heuristic, while D-CLAS [12], Aalo [23], sunflows [18] and CODA [31] try to "guess" various characterics of coflows in different ways to help scheduling. The former is often called a clairvoyant method in the sense that it explicitly depends on flow information like size and arrival time, while the latter four are flow information agnostic.

Although these coflow level scheduling methods indeed effectively improve the communication or data transfer performance, they treat different coflows or applications as equally important. As a result, the difference of performance improvement between different coflows or applications depends on the specific scheduling heuristics. For example, one scheduling scheme may favor short flows, while another may favor coflows with larger width, i.e., consisting of a large number of flows, sometimes even unpredictable or inconsistent. In this sense, we argue that coflows (or applicaitons) often have priorities or emergency levels, and should be explicitly taken into consideration when they are scheduled. For example, supporting the internal communication requirement of a search engine with intense internal computation should be more emergent than supporting the file backup on a distributed storage system. Although in this example the more emergent coflows mainly consist of short flows and tend to have a large width, this is not always the case, and in general emergency is orthogonal to other physical characteristics. Thus, the scheduling methods introduced above lack the capability to improve the performance of coflows with the strongest demand.

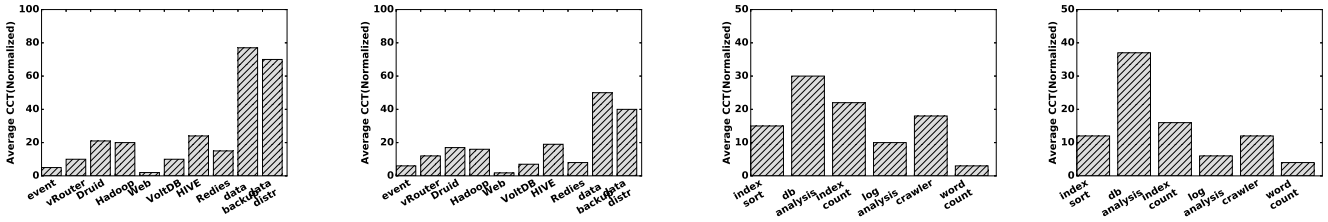
TABLE I: Applications and their levels of emergence

App Name	Type	width	length (MB)	Emergence Level
Event	communication	20	5	significant
vRouter	communication	8	3	significant
Druid	interactive	6	18	important
Hadoop	computation	5	42	normal
Web	interactive	3	5	normal
VoltDB	background	4	21	normal
Hive	background	7	32	unimportant
Redies	background	2	30	unimportant
data-backup	background	3	124	lax
data-dist	background	5	93	lax

TABLE II: Coflows in Hardoop and their levels of emergence

Coflow Type	width	length (MB)	Emergence Level
index-sort	10	3	significant
db-analysis	3	12	important
index-count	6	20	normal
log-analysis	6	31	unimportant
crawler	4	12	unimportant
word-count	3	11	unimportant

To make this argument more convincing, we now use real network traffic from a medium-sized data center, where more than 100 applications run on about 3000 servers simultaneously. After talking with the network operators and software development engineers about the emergence of different applications, we make an in-depth analysis of the traffic



(a) Average CCT (b) Average CCT with Varys (c) Hadoop CCT (d) Hadoop CCT with Varys

Fig. 1: Normalized Coflow Completion Time, grouped by the level of emergence

collected from 720 servers depolyed in 60 racks. The traffic lasts about one month, and the basic statistics of the top ten network-demanding applications are summarized in Table I. There are typically five emergence levels, i.e., significant, important, normal, unimportant and lax. For example, the event application and the vRoute application are responsible for communication of critical components, and are of the highest level of emergence, while data-backup and data-dist are applications running in the background and are of the lowest level of emergence. We can see applications consisting of a larger number of flows or smaller flows are not necessarily more emergence. For example, the average number of flows in Hive (i.e., 7) is larger than that of Web (i.e., 3), but Hive is less emergent than Web. Another example is that, the average flow size of Web (i.e. 5) is smaller than that of Druid (i.e., 18), but Web is of less emergence than Druid. This supports our former statement that in general, emergency is orthogonal to other physical characteristics of coflows. Table.II further shows the details of different coflows in the hadoop application, from which we can make similar conclusions.

To illustrace the ineffectiveness of existing coflow scheduling methods, we plot the average Coflow Completion Time (CCT) of different coflows, grouped by their levels of emergence, in Fig.1. Fig.1(a) shows the direct measurement result of all applications, while Fig.1(b) shows the simulation result when Varys is used to schedule them. Comparing the two plots, we can see that Varys clearly reduces the completion time of less emergent coflows, such as Redies, data-backup and data-dist, at the expenses of the same of even longer completion time of more emergent coflows, such as Event and vRouter. A similar effect can be observed in Fig.1(c) and Fig.1(d), where the measured CCTs of Hadoop coflows are compared to the scheduling results of Varys. This motivates us to optimize the Weighted Coflow Completion Time (WCCT), where weight denotes the emergence of coflows, and the more emergent the coflow, the higher the weight. Coflow scheduling algorithms should take coflow weight into consideration, and optimizing CCT becomes just a special case of optimizing WCCT.

III. Model and problem formulation

In this section, we first introduce the non-blocking fabric model to reasonably simplify the coflow scheduling problem in data center networks.Then we formulate an Idealized Weighted Coflow Completion Time Minimization (IWCCTM) problem and discuss its hardness.

A. Non-blocking Fabric Model for Data Center Networks

Recent researches [12] [18] [13] [9] regard a data center network as a big giant non-blocking switch which interconnects all machines. In this mode, all machines' ports have the same normalized unit capacity, and flows compete bandwidth only at these ports, i.e., the ingress and egress ports of the big fabric. Such an abstraction is reasonable and matches with the full bisection bandwidth topologies widely used in productional data centers, e.g., the non-blocking Clos topology . In this paper, we use this model and assume congestions occur only at the ingress and egress ports.

B. Problem Formulation and Hardness Proof

We consider the scheduling of n coflows in a data center network modeled as a non-blocking fabric with m hosts, i.e., there are m ingresses and m egresses. A coflow is a collection of flows sharing a common performance goal, e.g., minimizing the completion time of the latest flow or ensuring that flows meet a common deadline, and is denoted by $F^{(k)} = \{f^k_{i,j} | 1 \leq i \leq m, 1 \leq j \leq m\}$, where k is the sequence number assigned to the coflow, and $f^k_{i,j}$ is the normalized size of the flow (in this coflow $F^{(k)}$) sent from host i to host j. Since every port has a unit capacity, the transfer time of the flow (i.e., the time the flow actually occupies the port) $f^k_{i,j}$ is also $f^k_{i,j}$. If there is no flow from host i to host j, then $f^k_{i,j} = 0$. In the present, we simply assume all flows arrive at the same time, but we will remove this assumption when designing the online scheduling algorithm in the next section. We use w_k to denote the weight, i.e., the emergence, of each coflow $F^{(k)}$, so that its completion time C_k can be optimized with regard to its weight. Then the Idealized Weighted Coflow Completion Time Minimization (IWCCTM) problem is defined as follows:

$$\text{minimize} \quad \sum_{k=1}^{n} w_k C_k \tag{1}$$

$$\text{s.t.} \quad \forall k, j : \sum_{\forall l: C_l \leq C_k} \sum_{i=1}^{m} f^{(l)}_{i,j} \leq C_k \tag{2}$$

$$\forall k, i : \sum_{\forall l: C_l \leq C_k} \sum_{j=1}^{m} f^{(l)}_{i,j} \leq C_k \tag{3}$$

Our goal is to minimize the sum of the weighted coflow completion times. The constraints (2) and (3) are due to the competition on the capacity of each port. For a coflow $F^{(k)}$ with completion time C_k, consider the set of coflows that finish

before it, i.e., $F^{(l)}$: $C_l \leq C_k$. For any ingress port i (or egress port j), the total flow transfer time of this coflow on this port is at least $\sum_{j=1}^{m} f_{i,j}$ (or $\sum_{i=1}^{m} f_{i,j}$, correspondingly), which must be no greater than C_k.

Consider a simplified version of this problem, where we want to minimize the total completion time of flows competing a single link. It is well known that *shortest job first* is the optimal non-preemptive scheduling policy, while *short remaining time first* is the optimal preemptive one. However, our problem for scheduling weighted coflows over multiple links is much harder, as indicated by the following proposition

proposition *1:* If only non-preemptive scheduling is allowed, the IWCCTM problem is equivalent to the problem of minimizing the sum of weighted completion times in a concurrent open shop [24] and it is NP-hard.

Proof: The concurrent open shop problem [24] can be described as a set of machines $M = \{1, ..., m\}$, with each machine capable of processing one operation type. A set of jobs $N = \{1, ..., n\}$, with each job requiring specific quantities of processing for each of its m operation types. Each job $j \in N$ has a weight $w_j \in R > 0$, and the processing time of job j's operation on machine i is $p_{ij} \in R \geq 0$. A job is completed when all its operations are completed. In particular, operations from the same job can be processed in parallel.

For our $m \times m$ network fabric, a coflow $F^{(k)}$ has $2m$ transfer tasks (m tasks on the ingress ports and m on the egeress ports). Consider the concurrent open shop scheduling problem with $2m$ machines of the same capacity and n jobs, where each job has $2m$ types of operations on the $2m$ machines. There is a one-to-one correspondence between a coflow and a job, and between one type of operation on a machine and one transfer task on a port. It has been proved that minimizing the average weighted job completion time is NP-hard [24] [10] [28] [27], so the IWCCTM problem is also NP-hard. ■

IV. Algorithms and System Design

In this section, we first propose for the IWCCTM problem a non-preemptive scheduling algorithm that is 2-approximate optimal, following the idea presented in [24]. Then we remove the assumption that all coflows arrive at the same time and flow sizes are known a priori, and derive an information-agnostic preemptive scheduling algorithm that can work in an online fashion. Based on this latter algorithm, we further introduce a scheduling system named Yosemite. Yosemite relies on the Akka [2] and kryo [3] libraries, and can readily be deployed in a productional cloud environment like openstack.

A. Non-preemptive Scheduling Algorithm for IWCCTM

For minimizing the average weighted job completion time in the concurrent open shop problem, the best known theoretical result is a 2-approximation greedy algorithm [19] [24]. Since our IWCCTM problem has a close relationship with it, we can derive a similar 2-approximate non-preemptive scheduler for IWCCTM, as shown in Algorithm 1.

Algorithm 1 takes a list $\mathcal{F}$ of n coflows and a list $\mathcal{W}$ of n weights as its input, where the k-th coflow in $\mathcal{F}$ is defined as $F^{(k)} = \{f_{i,j}^k | 1 \leq i \leq m, 1 \leq j \leq m\}$, whose weight is w_k, the k-th weight in $\mathcal{W}$, as already explained in Sec III. It outputs γ, a permutation of $\{1, \ldots, n\}$ that indicates the scheduling order of the n coflows.

Algorithm 1 2-approximate Non-preemptive Scheduling Algorithm for IWCCTM

Input: Coflow list $\mathcal{F} = F^{(k)}$, weight list $\mathcal{W} = w_k$;
Output: a permutation γ of $\{1, \ldots, n\}$;
1: $\mathcal{P} \leftarrow \{1, \ldots, 2m\}$, $\mathcal{R} \leftarrow \{1, \ldots, n\}$
2: $L_i^{(k)} = \sum_{j=1}^{m} f_{i,j}^{(k)}$ for $1 \leq k \leq n$ and $i \leq m$
3: $L_{j+m}^{(k)} = \sum_{i=1}^{m} f_{i,j}^{(k)}$ for $1 \leq k \leq n$ and $j \leq m$
4: $L_p = \sum_{1 \leq k \leq n} L_p^{(k)}$ for each $p \in \mathcal{P}$
5: **for** i from n to 1 **do**
6: $\quad p^* = \arg\max_{p \in \mathcal{P}} L_p$
7: $\quad \gamma[i] = r^* = \arg\min_{r \in \mathcal{R}} w_r / L_{p^*}^{(r)}$
8: $\quad \theta = w_{r^*} / L_{p^*}^{(r^*)}$
9: $\quad \mathcal{R} = \mathcal{R} \setminus \{r^*\}$
10: $\quad w_r = w_r - \theta \times L_{p^*}^{(r)}$ for all $r \in \mathcal{R}$
11: $\quad L_p = L_p - L_p^{(r^*)}$ for all $p \in \mathcal{P}$
12: **end for**

The Algorithm first composes a port list $\mathcal{P} = \{1, \ldots, 2m\}$, corresponding to m ingress and m egress ports, and computes the total load on each port (line $1 \sim 4$). Then its iteratively finds the coflow to be scheduled in the i-th round, nonetheless in a reverse order. In each iteration, it finds the port (p^*) with the heaviest load, chooses the coflow who has the minimal ratio of coflow weight to its load on that port, and saves its index (r^*) in $\gamma[i]$ (line $6 \sim 7$). It then updates the set of coflows remaining to be scheduled as well as their weights, and the port load (line $8 \sim 11$), before it goes to the next iteration.

Algorithm 1 generates the permutation γ as the scheduling order within $O(n(m+n))$ operations and is 2-approximate optimal. However, since IWCCTM makes some idealized assumptions, the algorithm cannot be straightforwardly used in real environment.

B. Information-agnostic Algorithm for minimizing WCCT

IWCCTM assumes that flow sizes are known by the scheduler, and flows arrive a the same time. These idealized assumptions are unrealistic, so special care should be taken when we design a practical scheduler to be deployed in a real environment.

The key idea of Algorithm 1 lines in line 7 therein, where it tries to prioritize coflows with large weight but small load on the most loaded port p^*. In practice, ports in date center networks are expected to be load balanced by utilizing certain load-balancing techniques [14]. We introduce a simplification that ignores the load difference on different ports, and just prioritize coflows with large weight but small load. By replacing the prioritizing rule in Algorithm 1 with this heuristic, we get a simple offline scheduling algorithm. This intermediate stage algorithm is for the sake of performance comparison and analysis only, so we omit its details due to space limitations. The computation of priorities in Algorithm 1 depends on

the total load of each coflow, and we replace that with the cumulative load which can be measured in realtime, so that the scheduler becomes flow information agnostic. This is particularly useful for scenarios where dynamic-sized flows are generated, for example, in hadoop or some other map-reduce style applications. After making all these changes, we process coflow arrivals, and derive Algorithm 2. It is an information-agnostic online algorithm that dynamically schedule cflows according to their weights and the instantaneous network condition.

Algorithm 2 Information-agnostic Online scheduling to minimize WCCT

Input: Coflow $F^{(k)}$; $s_{i,j}^{(k)}$ that has been sent from port i to port j for Coflow $F^{(k)}$, where $1 \leqslant i \leqslant m$,$1 \leqslant j \leqslant m$

Output: γ

1: $L_i^{(k)} = \sum_{j=1}^{m} s_{i,j}^{(k)}$
2: $L_{j+m}^{(k)} = \sum_{i=1}^{m} s_{i,j}^{(k)}$
3: $l^{(k)} = \min_{1 \leq i \leq 2m} L_i^{(k)}$
4: $\alpha^{(k)} = \frac{l^{(k)}}{w[k]}$
5: sort $[\alpha^{(1)}, \alpha^{(2)}, \alpha^{(3)}...]$ in nondecreasing order for all active coflows, and set the result to γ.

Algorithm 2 is called when a new coflow $F^{(k)}$ arrives. Line 3 computes the data that the coflow has sent. Line 4 computes the priority factor of coflow $F^{(k)}$. Algorithm 2 is simple and it is a non-clairvoyant method. However, it has some performance loss, details this we will discuss at section V.

C. Yosemite System

In this part, we introduce the design of the real system Yosemite that aims to schedule weighted coflows. Yosemite includes about 6000 lines of scala code and follows the actor model [1]. It uses Akka [2] for message transferring and kryo [3] for object serialization.

Algorithm 3 Procedure of the scheduler

Input: Ordered coflow set γ, Remaining capacity for ingress: Rem(P_i^{in}), Remaining capacity for egress: Rem(P_j^{out}), where $1 \leqslant i \leqslant m$,$1 \leqslant j \leqslant m$

Output: bandwidth of each flow

1: **for** F $\in \gamma$ **do**
2: rate = $\min(\min_i Rem(P_i^{in}), \min_j Rem(P_i^{out}))$,$\forall$ F goes through port i and port j, $1 \leqslant i \leqslant m$,$1 \leqslant j \leqslant m$
3: f.rate = rate, $\forall f \in F$ and f does not finish
4: $Rem(P_i^{in})$-=rate, $1 \leqslant i \leqslant m$
5: $Rem(P_i^{out}))$-=rate, $1 \leqslant j \leqslant m$
6: **end for**
7: Distribute unused bandwidth to F $\in \gamma$
8: Send result to corresponding Slave

Fig. 2 shows the architecture of Yosemite. There are two type of nodes: master and worker. Each worker node includes two main components: Client and Slave. Client contains API Daemon and Rate Limiter. API Daemon is used to interact with applications and Rate Limiter is responsible for throttling flow rate. Slave has two long running server: data server and comm server. Comm server communicates with master and data server is used to send data. Master node is the brain of the system. It collects coflow information from workers and sends bandwidth information to workers. It also has two components: slave and scheduler. Yosemite Scheduler performs coflow priority and bandwidth computation. Algorithm 3 shows the detail of scheduler.

Algorithm 3 is called when a flow starts or ends. Line2 computes gets the minimal value of port capacity that coflow F goes through and uses the rate as the coflow rate. Line4 $\sim$ Line5 updates the capacity for ingress and egress. Line 7 allocates the remaining bandwidth to coflows to avoid bandwidth wastes. At last, send the result back to senders.

D. Some details

Line7 of Algorithm 3 aims to provide work conserving allocation for resources. Extra network resources may be wasted if just using the computation result. To prevent this happening, we allocate the remaining bandwidth fair to the corresponding flows that transfer from the port. With work-conserving allocation, resources can be used more efficiency, so that less resources will be wasted.

Master is the "brain" of Yosemite. It collects information from slave and computes the bandwidth of each flow. However, for some extreme cases, master may crash. If this happened, the system can not work any more. Although Akka provides fault-tolerant strategy which will restart scheduler when it crush down, scheduling information will be lost. To prevent this happen, we start a monitor which stores the real-time state of master. When master collapses, the monitor tries to restart the master and lets it resume the state.

In practice, Algorithm 3 should be fast enough to get the bandwidth result. According to our measurement in our data center, coflow completion time is second level. Algorithm 3

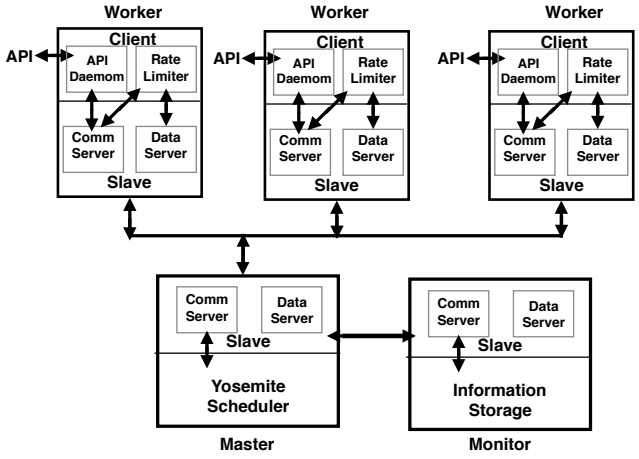


Fig. 2: Yosemite architecture. Computation frameworks interact with Yosemite through a client library

can get the result at millisecond level. We believe this is enough. We discuss more of this at SectionV.

V. EVALUATION

In this section, we test the performance of Yosemite in both simulation and testbed. We evaluate Yosemite at our openstack [4] platform, which can run at most 80 virtual machines (2 Cores, 4GB Memory) simultaneously. For larger scale experiments, we use trace-driven simulation. Main results of our evaluation are as follows:

- For the trace of facebook [13] with random weight, Yosemite improves about 30%, 40%, 50% over Varys [13], Aalo [12] and Barrat [16] on reducing average weighted coflow completion time. For coflows with above-the-average level of emergence, Yosemite performs about 20% ,30%, 40% better than Varys, Aalo, Barrat.
- For the trace of our data center, Yosemite performs about 30% better than Varys, which is the best performing one among existing coflow scheduling algorithms on reducing emergency coflows' average completion time.
- Comparing with the 2-approximation greedy algorithm, the non-clairvoyant online algorithm has less than 30% performance loss.
- Real testbed shows, Yosemite can improve about 25%-30% on reducing the emergency coflows' average completion time. The computation time of Yosemite is short, with ∼23ms on average, which occupies about 0.1% of the total transfer time.

A. methlogy

Experiment settings and Metrics In our simulations, we use two real world traffic trace. The first one is from our data center, in which 100 applications running simultaneously among about 720 servers deployed in 60 racks. We have 5 level of emergence for applications' coflows: Significant, Important, Normal, Unimportant, Lax. In our algorithm, urgent coflows have higher weight and less important ones have lower weight. In our experiments, default corresponding weight value for Significant, Important, Normal, Unimportant, Lax are 5, 4, 3, 2, 1. Traffic of Facebook that was collected from a 3000-machine cluster with 150 racks [13]. As the original trace does not contain information of weight settings, we randomly select weight within 1, 2 ,3, 4, 5 for coflows to present their emergence.

In our experiment, we use two metrics to estimate different methods. The first metrics is average coflow completion time (CCT) for different weight coflows. Coflows are categorized to be five types in terms of importance: Significant, Important, Normal, Unimportant, Lax. In some cases, TCP is chosen as the baseline and we define the factor of improvement=$\frac{baseline\ avg\ CCT}{current\ avg\ CCT}$. The second metrics is average weighted coflow completion time (WCCT). Coflows are categorized to be four types in terms of their length (the size of the largest flow in bytes for a coflow) and width (the number of parallel flows in a coflow): Narrow&Short(N-S), Narrow&Long(N-L), Wide&Short(W-S), and Wide&Long(W-L), where a coflow is considered to be short if its length is less than 100 MB, and narrow if it involves at most 20 flows. In some cases, TCP is chosen as the baseline and we define factor of improvement for WCCT=$\frac{baseline\ avg\ WCCT}{current\ avg\ WCCT}$.

Open source. To make our experiment repeatable, we publish main codes of Yosemite as well as Yosemite-Sim. Sources of Yosemite can be downloaded at [6]. Main codes of the Yosemite-Sim can be downloaded at [7].

B. Trace-driven simulation

In this section, we use facebook trace [13] to test the performance of Yosemite. As the original trace doesn't include weight information, we set random weight within 1, 2, 3, 4, 5 to present the emergence of coflows. Repeat the process 100 times and Fig. 3 shows the performance of Yosemite, Barrat, Varys, Aalo and TCP. Note TCP is selected as the baseline in this group. We repeat the experiment 100 times and error bar indicates maximum, minimum, average value. To eliminate the influence of extreme value, we remove the top and last 2.5% result of each methods and recompute the factor of improvement, so the 95th cases are also shown in the picture.

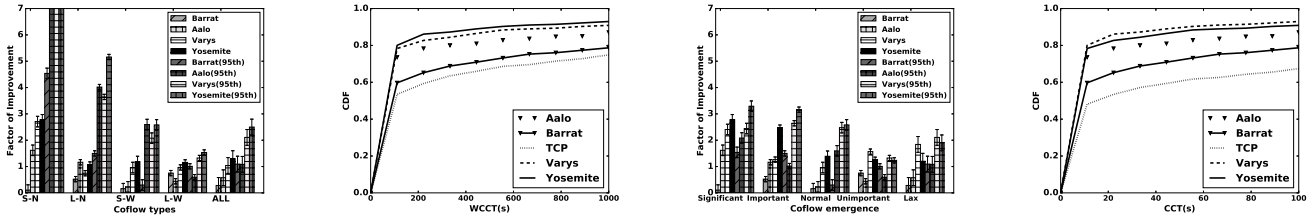
Fig. 3(a) implies that Yosemite greatly reduces the average WCCT across all coflow types. The factors of improvement for Yosemite are 3.5 (Narrow&Short), 1.6 (Narrow&Long), 1.7 (Wide&Short), 1.4 (Wide&Long) and 1.5 (ALL), while Varys are 2.4 (Narrow&Short), 0.9 (Narrow&Long), 1.2 (Wide&Short), 1.3 (Wide&Long) and 1.2 (ALL). For the 95th case, the improvement factors are (Narrow&Short),20 (Narrow&Long), 2.5 (Wide&Short), 1.5 (Wide&Long), and 2.5 (ALL), however Varys is 22 (Narrow&Short),14 (Narrow&Long), 2.1 (Wide&Short), 1.2 (Wide&Long) and 2.1 (ALL) and other method such as Aalo and Barrat performs even worse than Varys. Fig. 3(b) indicates the distribution of WCCT. We can see that more than 80% WCCT with Yosemite is within 200s, while that for Varys, Aalo, Barrat are 70%, 65%, 60%. Yosemite performs about 15%, 25%, 30% better than Varys, Aalo and Barrat.

Fig. 3(c) shows the average CCT for coflows with different level of emergence. We can see the factors of improvement for Yosemite are 2.5(Significant), 1.5(Important), 1.2(Normal), 1.4(Unimportant), 1.5(Lax). Yosemite performs about 20% better than Varys for the emergent coflows. But for the less important ones, Varys perform about 20% better than Yosemite. Fig. 3(d) shows the CCT distribution for different methods. We can see more than 80% CCT under Varys is within 20s, while that for Yosemite, Aalo, Barrat are 75%,65%,60%. Varys performs about 10% better than Yosemite on minimizing average CCT for all coflows.

We can see that Varys performs better than Aalo, Barrat on minimizing average CCT and average WCCT. In the following experiment, we mainly use Varys as the reference method to show the performance of Yosemite.

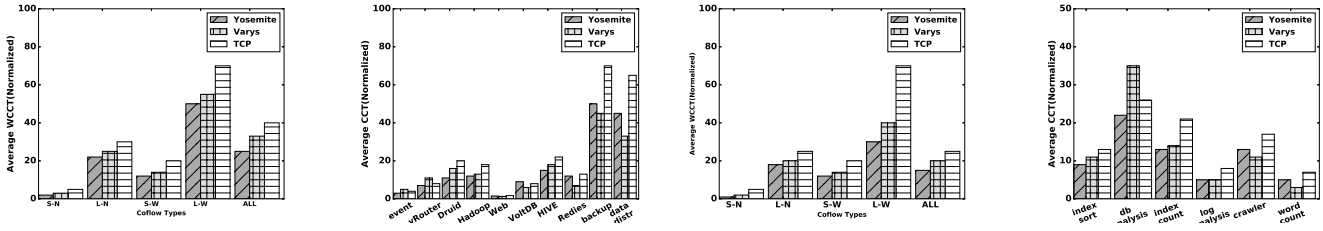
C. Motivation Resolved

In this section, we first solve the problem that is proposed at Section II. Applications have 5 level of emergence in our



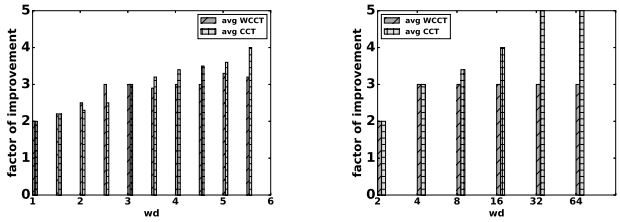
(a) Average WCCT (b) WCCT distribution (c) Average CCT (d) CCT distribution

Fig. 3: [Simulation] Average WCCT and average CCT comparison for facebook trace, TCP is selected as the baseline.



(a) Average WCCT (b) Average CCT (c) Hadoop WCCT (d) Hadoop CCT

Fig. 4: [Simulation] Average CCT (Normalized) and Average WCCT for all applications and Hadoop. Varys ignores the importance of applications ,while Yosemite considers both importance of applications and network condition.



(a) Small wd (b) Large wd

Fig. 5: [Simulation] Average WCCT and average CCT under different weight settings, TCP is selected as the baseline.

data center and we use 5, 4, 3 ,2, 1 to present the Significant, Important, Normal, Unimportant and Lax level of coflows. For the trace in our simulator and Fig. 4 shows the result.

Fig. 4(a) shows average WCCT for applications. We can see the Normalized WCCT of Varys are 8 (Narrow&Short), 20(Narrow&Long), 15 (Wide&Short), 40 (Wide&Long) and 30(ALL), while that for Yosemite are 5 (Narrow&Short), 15(Narrow&Long), 10 (Wide&Short), 30 (Wide&Long) and 20(ALL). Yosemite performs about 20% better than Varys on minimizing average WCCT. From Fig. 4(b) we can see that for vRouter and event (Significant), Varys performs almost the same as TCP, while Yosemite performs about 20% better than TCP. For Hadoop and Druid (Important), Yosemite performs about 30% better than Varys. While other unimportant applications Varys perform about 10%∼20% better than Yosemite. Fig.4(c) shows Hadoop average WCCT and Fig.4(d) shows Hadoop CCT. For index sort and db analysis which are important in our data center, Varys performs even worse than TCP sometimes. However, using Yosemite, it performs about 30%∼ 40% better. But for the unimportant and lax coflows, Varys performs better. For average WCCT, Yosemite performs about 20% better than Varys.

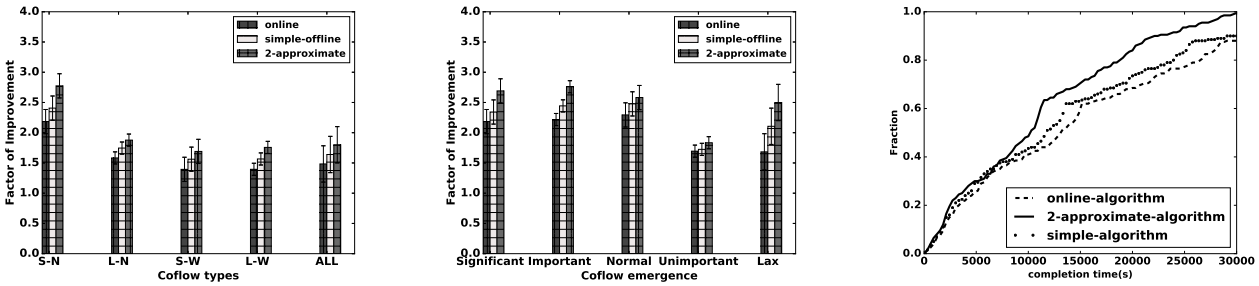
D. How to set weight in practice

Weight plays an important role in Yosemite scheduling system. Adjacent weight difference is 1 in our prior experiments. In this group, we use the facebook trace and set weight within 1,1+wd,1+2*wd,1+3*wd,1+4*wd to study the influence of weight to the performance of Yosemite. Then we change wd from 1 to 10 and Fig.5(a) shows the average WCCT, important coflows' ($weight > 1 + 2 * wd$) average CCT. We can see that with the increasing of weight difference, improvement of average WCCT increases, but average CCT for important coflows almost remains the same. The reason for this is that Yosemite uses products of weight and sent size as the rank factor. With a larger weight difference, weight plays an more important role, so that the important coflows will rank front. Fig.5(c) shows the result of larger wd difference, where wd ranges from 2 to 64. We can see when wd > 8, improvement of cct for important coflows becomes small and this is similar to Fig.5(a).

In practice, we can set weight by the demands of network administrator. When we regard the emergence of coflows as the key schedule factor, we can use large wd. When we think both the emergence level of coflows and network condition are the key factors, we can use small wd.

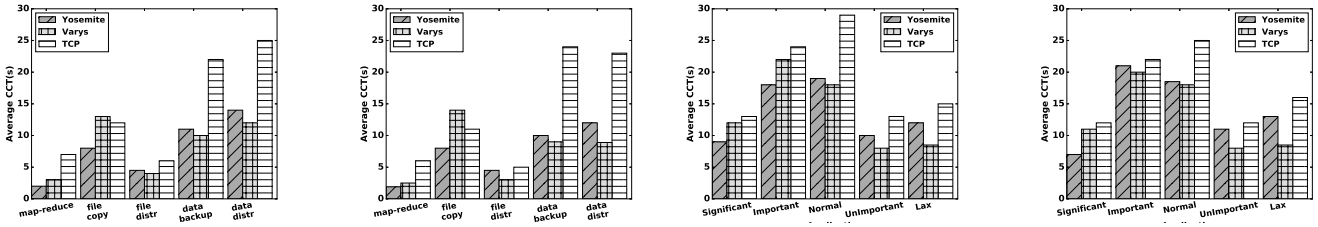
E. Distance between online and offline

Two relaxations are adopted in Yosemite. Simple-offline algorithm assumes data center generally assigns jobs with load balancing and ignores load difference of each port. The online algorithm further assumes that larger coflows that last long time and should have low priority. The two relaxations can absolutely loss the performance. To see the performance loss, we consider the schedule of 100 coflows served by a small-scale cluster consisting of 60 hosts in our simulator. Let all the coflow start at t = 0 and set weight random within 1 , 2 , 3 , 4 ,5



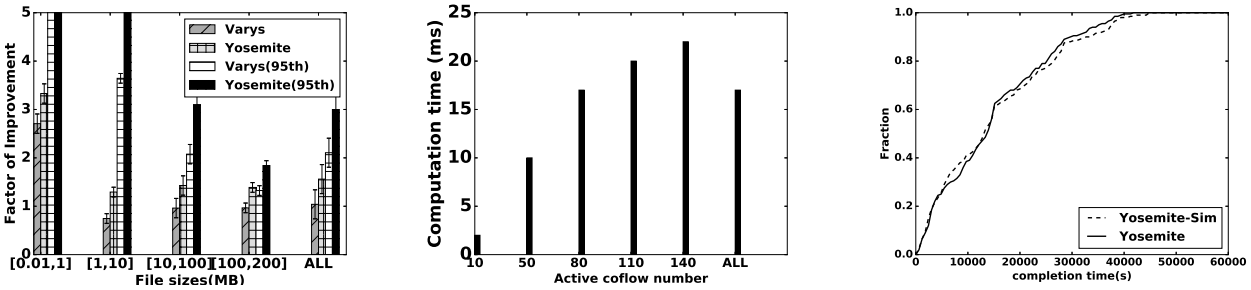
(a) Average WCCT (b) Average CCT (c) CCT distribution

Fig. 6: [Simulation] Comparison between the online and offline algorithms, TCP is selected as the baseline



(a) Average CCT (b) Average CCT (95th) (c) file-dist CCT (d) file-dist CCT (95th)

Fig. 7: [Testbed] Performance under different applications in openstack



(a) Average WCCT comparison (b) Computation time of scheduler (c) Testbed and Simulator comparison

Fig. 8: [Testbed] Some details of Yosemite

then repeat the experiment 100 times. Fig. 6 shows the result. Error bar in the picture denote the maximum and minimum value. From Fig. 6(a), we can see that factor of Improvement for WCCT of the online method is 1.6. While that for simple-offline method is 1.7 and for the 2-approximate method is 1.8. Comparing the 2-approximate algorithm, the online algorithm has about 10% performance loss on minimizing average WCCT. Fig. 6(b) shows the performance for different important level of coflows. We can see that online method factor of improvement for significant, important coflows are 2 and 2.1 on average, while the simple-offline method are 2.2 and 2.3. The 2-approximate methods are about 2.5, 2.6, which indicates that the online algorithm has about 30% performance loss. Fig. 6(c) shows the distribution for CCT. We can see more than 80% CCT for 2-approximate method is within 20000s, while that for simple-offline and online methods are 70% and 60%. TABLE III shows the comparison between the methods. We can see though the online method has some performance loss, it is non-clairvoyant method. As flow size for some applications can not be known beforehand, so that non-clairvoyant method could be more widely used in practice.

TABLE III: Comparison between the three algorithms

#Scheme	Mode	Procedure	Performance	information
2-approximate	offline	Complex	High	clairvoyant
simple-offline	offline	simple	High	clairvoyant
Yosemite	online	simple	High	non-clairvoyant

F. Real testbed evaluation

We deploy Yosemite at our private openstack platform which can start at most 80 virtual machines (2 Cores, 4GB Mem) simultaneously. Operation system of each virtual machine is Ubuntu16.04. With the help of Traffic Control module [5], we constrain the maximum bandwidth of each VM's NIC to 1GB/s.

Firstly, we deploy 5 applications in our data center: map-reduce, file-copy, file-distribute, data-backup, data-distribute. Importance of the 5 applications are: Significant, Important, Normal, Unimportant, Lax. Fig. 7(a) shows the total result and

Fig. 7(b) shows the 95th percentile. We can see for the important applications map-reduce, file-copy, Yosemite performs about 20% $\sim$ 30% better than Varys, but for the unimportant applications, Yosemite performs about 20% worse than Varys. Fig. 7(c) shows the performance for file-distribution. Set five important level to files, we can see that Yosemite performs about 20% better than Varys. The 95th percentile as Fig. 7(d) shown is almost similar.

Then for the file distribution application, we let each virtual node constantly construct files whose size ranging from 1KB to 200MB. Destinations are nodes which are randomly chosen from the total nodes set. TCP-fair is chosen as the baseline and Fig. 8 shows performance comparison between Yosemite and Varys. We repeat the process 10 times and the error bar indicates the max, min and average factor of improvement. From Fig. 8(a), we can see in total, improvement factor of Yosemite is 3.1 , while Varys is 2.1. Yosemite improves 30% better than Varys. The 95th result shows Yosemite performs 31% better than Varys.

In reality, scheduler should be fast enough to compute out the scheduler order of coflows. Fig. 8(b) shows the computation rate of Yosemite scheduler. We can see for peak load, when active coflows is 140, computation time of the scheduler is $\sim$23ms and average computation time is less than 17ms. In our system, more than 90% file transfers' time are more than 10s. We think the average computation time is fast enough to get the schedule result.

Fig. 8(c) shows the result comparison for the simulator and real testbed. In this experiment, we use the same parameters just as the file distribution. We can see that the total performance gap between the two methods are about 20%. This result shows the credibility of our simulation.

VI. Conclusion

In this paper, we use weight to quantify the emergence of applications. Coflows belonging to emergent applications own large weight. We design and implement Yosemite-a coflow scheduling system that aims to minimize average weighted coflow completion time. We test the performance of it by trace-driven simulation and test-bed deployment. Experiments show Yosemite can reduce average weighted coflow completion time significantly. Methods of Yosemite is clairvoyant, which means that flow size do not need to know beforehand. This makes the method widely used in practice. In our experiment, we use 5 level of importance, however, in practice, thin granularity of emergence may be needed. Also, we use 1, 2, 3, 4, 5 as the default weight, indeed, more accuracy weight setting can be set to improve the performance of Yosemite. In the future, we will discuss how to set weight from theory.

References

[1] Actor model. https://en.wikipedia.org/wiki/Actor_model/.

[2] Akka. http://akka.io/.

[3] kryo. https://code.google.com/p/kryo/.

[4] openstack. https://www.openstack.org/.

[5] Traffic control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html/.

[6] Yosemite. https://github.com/YosemiteCode/Yosemite.

[7] Yosemitesim. https://github.com/YosemiteCode/YosemiteSim.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). SIGCOMM '10, pages 63–74, 2010.

[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. SIGCOMM '13, pages 435–446, 2013.

[10] Z.-L. Chen and N. G. Hall. Supply chain scheduling: Assembly systems. Technical report, Working Paper, Department of Systems Engineering, University of Pennsylvania, 2000.

[11] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[12] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 393–406. ACM, 2015.

[13] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 443–454. ACM, 2014.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran. Decentralized erasure codes for distributed networked storage. *IEEE/ACM Transactions on Networking (TON)*, 14(SI):2809–2816, 2006.

[16] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 431–442. ACM, 2014.

[17] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[18] X. S. Huang, X. S. Sun, and T. E. Ng. Sunflow: Efficient optical circuit scheduling for coflows. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 297–311. ACM, 2016.

[19] A. Kumar, R. Manokaran, M. Tulsiani, and N. K. Vishnoi. On lp-based approximability for strict csps. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 1560–1573. Society for Industrial and Applied Mathematics, 2011.

[20] Latency. Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, 2009.

[21] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 467–478. ACM, 2014.

[22] H.-Y. Lin and W.-G. Tzeng. A secure decentralized erasure code for distributed networked storage. *IEEE transactions on Parallel and Distributed Systems*, 21(11):1586–1594, 2010.

[23] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li. Towards practical and near-optimal coflow scheduling for data center networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3366–3380, 2016.

[24] M. Mastrolilli, M. Queyranne, A. S. Schulz, O. Svensson, and N. A. Uhan. Minimizing the sum of weighted completion times in a concurrent open shop. *Operations Research Letters*, 38(5):390–395, 2010.

[25] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not foes: synthesizing existing transport strategies for data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 491–502. ACM, 2014.

[26] A. Munir, I. Qazi, Z. Uzmi, A. Mushtaq, S. Ismail, M. Iqbal, and B. Khan. Minimizing flow completion times in data centers. In *INFOCOM, 2013 Proceedings IEEE*, pages 2157–2165, April 2013.

[27] Z. Qiu, C. Stein, and Y. Zhong. Experimental analysis of algorithms for coflow scheduling. *arXiv preprint arXiv:1603.07981*, 2016.

[28] T. A. Roemer. A note on the complexity of the concurrent open shop problem. *Journal of scheduling*, 9(4):389–396, 2006.

[29] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). SIGCOMM '12, pages 115–126, 2012.

[30] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. SIGCOMM '11, pages 50–61, 2011.

[31] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 160–173. ACM, 2016.

[32] H. Zhang, X. Shi, X. Yin, F. Ren, and Z. Wang. More load, more differentiationa design principle for deadline-aware congestion control. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 127–135. IEEE, 2015.