Yosemite: Efficient Scheduling of Weighted Coflows in Data Centers

Paper ID: #117

***Abstract*—Nowadays, low latency and high throughput are required by data center applications. Coflow has been proposed as a new abstraction to capture the communication patterns in a rich set of data parallel applications, so that their application-level semantics can be effectively modeled. By taking coflows, instead of flows or packets, as the basic elements in network resource allocation or scheduling, application-level optimization goals such as reducing the coflow completion time (CCT) or application transfer latency can be better achieved. Although efficient coflow scheduling methods have been studied in this aspect, in this paper, we propose *weighted coflows* as a further step in this direction, where weights are used to express the importance or priorities of different coflows or applications, and weighted coflow completion time (WCCT) is set as the optimization target. We design and implement Yosemite, a system that aims to minimize WCCT by dynamically scheduling coflows according to their weights and the instantaneous network condition. We test Yosemite's performance against several flow or coflow scheduling methods, by trace-driven simulations as well as real delopyment in a cloud testbed. Our trace-driven simulations show that, Yosemite's online scheduling algorithm performs about 20%-50% and 30%-60% better than that of Varys and Aalo, two state-of-the-art coflow scheduling systems, respectively. And in our testbed evaluations, Yosemite achieves 30% average improvement against Varys.**

I. Introduction

Nowadays, low latency [22] [27] and high throughput [23] are required by data center applications, for example, those with map-reduce style data-intensive computing [16], and those using distributed file storgage [24] [17], 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, as well as the transportation.

Among these efforts, flow level scheduling methods try to schedule arriving packets according to the characteristics of the corrsponding flows. For example, PDQ [19] and pFabric [10] 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 lantency. However, flow level scheduling methods still have several deficiencies [14]. On the one hand, flow level scheduling has to be carried out quite frequently, such that centralized scheduler may face huge pressure on its own computation and communication, while decentralized scheduler may face worse scheduling performance. On the other hand, some tardy flows may lag behind and affect the overall performance of an application, because the communication of the application cannot finish until all its flows have completed their transmission.

Recently, Coflow has been proposed as a new abstraction to capture the communication patterns in a rich set of data parallel applications, so that their application-level semantics can be effectively modeled. According to [12], 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 or ensuring that flows meet a common deadline, capturing the application-level requirements in a way better than individual flows can. Efficient coflow scheduling methods targeting minimizing coflow completion time (CCT) have been studied. Varys [15] takes a coflow's network bottleneck as the most important factor that determines the coflow completion time. It proposes the *Smallest-Effective-Bottleneck-First* (SEBF) heuristic, which schedules coflows according to their bottleneck completion time. It also uses *Minimum-Allocation-for-Desired-Duration* (MADD) to compute the bandwidth that coflows deserve to get. Non-clairvoyant methods are also studied, including Aalo [13], sunflows [20] and CODA [35], which do not assume the priori knowledge about a coflow's characteristics such as the number, the length, or the arrival time of its constituting flows. Naturally, they tend to have relatively worse performance than Varys.

In this paper, we propose *weighted coflows* as a further step in this direction, where weights are used to express the importance or priorities of different coflows or applications in meeting a short completion time, and weighted coflow completion time (WCCT) is set as the optimization target. This is natural since there exist both delay sensitive and insensitive applications, and some applications may gain (or lose) more revenue than others in the same amount of time. For example, suuporting 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 and assign different weights to them, i.e., coflows of search engine have a higher weight, while coflows of file back have a lower weight.

We formulate the weighted coflow completion time optimization (WCCO) problem, and present a 2-approximate offline algorithm that borrows ideas from the concurrent open shop problem [26]. We then modify the algorithm to an online version, which introduces only minor performance loss compared to the offline one. We further design and implement Yosemite, a coflow scheduling system which is consisted of a central master node, a central monitor node, and other

distributed worker nodes, and can be readily deployed in a production cloud environment like openstack. We test the performance of Yosemite by both trace-driven simulations and testbed evaluations, where Yosemite performs quite well in reducing the weighted coflow completion time. Our trace-driven simulations show that, Yosemite's online scheduling algorithm performs about 20%∼50% and 30%∼60% better than that of Varys and Aalo, respectively. And in our testbed evaluations, Yosemite achieves 30% average improvement against Varys.

Our contributions in this paper are summarized as follows:

- We propose the weighted coflow completion time optimization (WCCO) problem and prove its equivalence to the job weighted completion time problem in the concurrent open shop setting.
- We propose for the WCCO problem an online algorithm that performs nearly as well as a two-approximation offline algorithm.
- We design and implement a weighted coflow scheduling system named Yosemite that is ready to be deployed in a cloud environment like openstack.
- We thoroughtly evaluate our scheduling algorithm and system both by simulations and testbed deployment, and show their significent performance improvement against several state-of-the-art coflow scheduling algorithms such as Varys, Aalo and Barrat.

The rest of the paper is organized as follows. In Section II we introduce some background and the related work. We formulate the WCCO problem and examine its hardness in Section III, and propose an efficient online scheduling algorithm in Section IV. We present the details of our Yosemite schduling system in Section V, and evalute it against several coflow schedule methods in Section VI. At last, we conclude in Section VII.

II. BACKGROUND AND RELATED WORK

Data center is now becoming an important infrastructure in data center network. Application needs data center to provide them with high throughput and low latency. However, resources in data center are limited, so efficiently scheduling in data center is necessary. Coflow scheduling methods can date back to flow scheduling.

DCTCP [9], D2TCP [33],L2DCT [28] and LPD [36] emphasize on flow level rate control. They propose to mark packets on a switch when instantaneous queue length exceeds a certain threshold k. The endpoints then estimate the extent of congestion by the marked packets, and throttle flow rates in proportion to that extent. With the help of ECN marking mechanism, they reduce the queue length on switches, and can reduce the queueing delay. TCP-based methods are simple, however they are implicit rate control method [10] and can never precisely estimate the right flow rates to use so as to schedule flows to minimize FCT while ensuring that the network is fully utilized. Then explicit methods are proposed.

D3 [34] and PDQ [19] can schedule flows based on some notion of urgency. They compute flow bandwidth according to flow deadline as well as network condition. A centralized controller which collects the network condition is needed. Although these methods can assign rates according to flow deadlines or flows' estimated completion time, they can not satisfy the demands of applications well. This is because distributed applications always have many parallel flows and often an application's communication stage cannot finish until all its flows have completed. If we only focus on single flow scheduling, some tardy flows may finish late and it doesn't make any sense to allocate much bandwidth to the application. As a result, scheduling applications' flows as a whole is meaningful.

Until this, both the implicit and explicit methods are all flow-level optimization methods. Having recognized limitation of flow-level methods, application level schedule methods occur. Barrat [18] schedules tasks in FIFO order but avoids head-of-line blocking by dynamically changing the level of multiplexing in the network. Using Barrat, each task has a globally unique priority-all flows within the task use this priority, irrespective of when these flows start or which part of the network they traverse. Barrat is simple, but for some large coflow, it can down many smaller ones. Varys [15] regards applications' parallel flows as coflow and it uses SEBF to schedule coflows and MADD to perform rate control, as a result, other coexisting coflows can make progress and the average CCT decreases. Varys is a Clairvoyant schedule method, which means some coflow information such as coflow length and coflow width should be known beforehand. Unfortunately, in many cases, coflow characteristics are unknown a priori and this will limit the its utilization. To overcome this, some non-Clairvoyant methods like Aalo [25] ,sunflows [20], D-CLAS [13] and CODA [35] do not need to know coflow information beforehand. They use information accumulation or machine learning to "guess" the characters of coflow. Non-Clairvoyant methods can be used more widely, however, their performance is not as good as the Clairvoyant ones such as Varys. Although coflow level scheduling methods such as Varys and Aalo perform well on reducing application transfer latency, they ignore different applications always have different level of importance and just scheduling depends on network condition and characters of coflows such as coflow length as well as width is not enough. In reality, the importance of applications should also be taken into consideration.

Some theory paper such as [29] sets weight to coflow to minimize average coflow weight completion time. We think this is good direction, however, methods such as [29] should solve complex equations and it is too complex to use in practice.

III. SYSTEM MODEL AND PROBLEM FORMULATION

In this part, we define Weight Coflow Completion Optimization (WCCO) problem and try to prove its equivalent to the concurrent open shop problem [31].

A. Data center non-blocking model

Recent researches [13] [20] [15] [10] regard the data center network as a big giant non-blocking switch which interconnects all the machines. In this model all ports have the same

normalized unit capacity, and bandwidth compete bandwidth at ingresses or egresses. Such an abstraction is reasonable and matches with recent full bisection bandwidth topologies widely used in current production data centers- To provide uniform high capacity as well as other targets like equidistant endpoints with network core, unlimited workload mobility, etc., today's production data center networks widely adopt non-blocking Clos topologies by design [25].

B. Why we choose weight

Applications in data center have to contend with each others for the limited network resources. The state-of-art schedule methods regard applications' flow set as a whole to reduce the transfer latency of applications. However, applications in data center have different level of importance, scheduling just depends on network condition as well as the physical property of coflows is not enough. We have to differentiate applications to satisfy their demand of bandwidth. As a result, how to distinguish the demands of applications is becoming an important problem.

Some previous study [33] [34] [36] use deadline to present the emergency of applications, applications have different deadlines, thus bandwidth. Deadline-aware methods aim to let applications finish before a particular time point. However, in reality, applications just need to transfer as quickly as possible rather than finishing before the time constraint. For the example of hadoop and erasure code storage system running in one data center to compete for network resources If we assume hadoop is more important than storage, we can give computation application higher priority to guarantee it to transfer as quickly as possible. In this case, setting deadline to applications is not a good choice, as applications do not have the expectation of finishing moment. We use weight to present the importance of applications, different applications have different level of importance. As a result, instead of minimizing average coflow completion time, we try to minimize average weight completion time.

C. Problem formulation

We consider the offline scheduling of n coflows on a non-blocking DCN with m hosts, i.e., there are m ingresses and m egresses, and their capacity is 1. All coflows arrive at time 0. The k-th coflow indicated by $F^{(k)}$ is a collection of parallel flows. $F^{(k)} = \{f^k_{i,j} | 1 \leq i \leq m, 1 \leq j \leq m\}$, where $f^k_{i,j}$ is the flow size sends from host i to host j. In reality, there are no flows sending from host i to host j. Under this condition $f^k_{i,j} = 0$. C_k denotes the completion time coflow $F^{(k)}$. Weight of coflow $F^{(k)}$ is w_k. We now define the Weight Coflow Completion Optimization (WCCO) problem as follows:

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

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

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

For a given set of coflows, labeled 1...n, minimizing their average weight completion time is equivalent to minimizing their sum of weight completion time, just as (1) shown. (2) is the constraint on sender side and (3) is the constraint on receiver side.

D. NP-hard proof

In this section, we prove that the target of minimizing average coflow weight completion time is equivalent to the problem that minimizing average job weight completion time in concurrent open shop problem.

proposition *1:* Weight Coflow Completion Optimization (WCCO) problem is equivalent to the problem of minimizing the sum of job weight completion times in a concurrent open shop.

Proof: For the $m \times m$ network fabric, we mark ingress ports as $1..m$ and egress ports as $m+1...2m$. We consider the transferring time of coflow $F^{(k)}$ through port p:

$$T^{(k)}_p = \begin{cases} \sum_{j=1}^{m} f^{(k)}_{i,j} & 1 \leq p \leq \text{m} \\ \sum_{i=1}^{m} f^{(k)}_{i,j} & \text{m} < p \leq 2\text{m} \end{cases} \tag{4}$$

As a result, for the given coflow $F^{(k)}$, there are 2m transfer subflows and each transfer time is computed as (4). Now, we consider the concurrent open shop scheduling problem with 2m identical machines (hence the same capacity). n jobs arrive at time 0 and each job has 2m types of operations on the 2m identical machines. Then each coflow can be regarded as a job and each port can be regarded as one machine. Coflow k transfers through port p is the same as job j operates on machine m. The two problems are equivalent. ■

It has been proved that minimizing average job weight completion time is a NP-hard problem [26] [11] [31] [30], so even the offline problem WCCO is also NP-hard.

IV. Algorithm design and analysis

Minimizing average coflow weight completion is indeed to find a permutation of coflows. It is NP-hard just as the proof of last section, so efficient heuristic is needed. To solve this, we first change a 2-approximate algorithm that aims to minimize average job weight completion time to coflow scheduling. Then we proposal an on-line algorithm based on this. Last of the section, we test the performance loss of the online algorithm.

A. Weight coflow completion time offline algorithm

Currently, the best known results for minimizing average job weight completion time is a 2-approximation greedy algorithm [21] [26]. The algorithm intends to minimize average job weight completion time and we can't put the original algorithm into coflow scheduling directly. According to the correspondence between WCCO and minimizing average job weight completion time problem that is introduced at section III-D, we get the 2-approximate algorithm for WCCO as Algorithm 1 shown.

Algorithm 1 2-approximation offline algorithm

Input: Coflow set $\mathcal{C}$; flow size $f_{i,j}$ from port i to port j, where $1 \leqslant i \leqslant m$,$1 \leqslant j \leqslant m$

Output: γ

1: $\gamma : \{1, 2, ...n\} \leftarrow \mathcal{C}$
   $UC \leftarrow \{1, 2, 3...n\}$
   $P \leftarrow \{1, 2, 3...2m\}$
   $W\{1, 2, ...n\} \leftarrow \{w_1, w_2, w_3...w_n\}$
2: $L_i^{(k)} = \sum_{j=1}^{m} f_{i,j}^{(k)}$
3: $L_{j+m}^{(k)} = \sum_{i=1}^{m} f_{i,j}^{(k)}$ for all k $\in \mathcal{C}$ and j $\leq$ m
4: $L_i = \sum_{k \in \mathcal{C}} L_i^k$ for all i $\in$ P
5: **for** i $\in \{n, n-1, n-2...1\}$ **do**
6: u=arg $\max_{k \in \mathcal{C}} L_k$
7: $\gamma[i]$=arg $\min_{F \in UC} W[F]/L_u^{(F)}$
8: $\theta = W[\gamma[i]]/L_u^{\gamma[i]}$
9: W[j]=W[j]-θ*$L_u^{(j)}$ for all j $\in UC$
10: $L_j = L_j - L_j^{\gamma[i]}$ for all j $\in$ P
11: $UC = UC \setminus \{\gamma[i]\}$
12: **end for**

Input of Algorithm 1 is $\mathcal{C}$ and $f_{i,j}$, where $\mathcal{C}$ is the coflow set and $f_{i,j}^k$ is the size of data from port i to j. Output of the Algorithm 1 is γ, which indicates scheduling permutation of coflows. For the detail of algorithm. Line1 does some initialism operations, where UC is the set of coflows to be scheduled,$W$is the weight set and P is the port set. Line2$\sim$Line4 computes the load on each port. Line5 $\sim$ Line12 decides the coflow scheduling sequence. Firstly, Line6 finds the port which owns the heaviest load. Then Line7 finds the corresponding coflow with the minimal ration of weight and load on that port. Line8$\sim$Line10 is adjust process. Line11 gets rid of this coflow from the unschedule sets. After the loop of Line5$\sim$Line12, Algorithm 1 will generate the schedule sequence of coflow set.

Within O(n(m+n)) elementary operations, Algorithm 1 will generate the schedule sequence γ, in which Coflow ranks front will be scheduled prior. Although Algorithm 1 has been proved to work well [26] for the offline situation, it has the constraint that assuming coflows arrive at time 0. In practice, coflows arrive dynamically, so that directly using the algorithm is impossible. Some changes should be made to let it work well for the online situation.

B. From offline to online

Algorithm 1 decides coflow's position by the ratio of coflow weight and port load. As today's data centers generally assign jobs with load balancing [16], so that we can make the relaxation of ignoring load difference of each port. When a new coflow comes, we can set the lowest priority to the coflow which has maximum per-port load. Then we get the online approach which is shown as Algorithm 2.

Algorithm 2 is called when a new coflow starts. Different from Algorithm 1, Input of Algorithm 2 is remaining flow size from each port pair. Line2$\sim$Line4 computes the load on each port. Instead of considering load diversity on each port,

Algorithm 2 Online approach to minimize average coflow weight completion time

Input: Coflow set $\mathcal{C}$; active flow *remaining* size $r_{i,j}^{(k)}$ from port i to port j, where $1 \leqslant i \leqslant m$,$1 \leqslant j \leqslant m$

Output: γ

1: $W\{1, 2, ...n\} \leftarrow \{w_1, w_2, w_3...w_n\}$
2: $L_i^{(k)} = \sum_{j=1}^{m} r_{i,j}^{(k)}$ for all k $\in \mathcal{C}$ and i $\leq$ m
3: $L_{j+m}^{(k)} = \sum_{i=1}^{m} r_{i,j}^{(k)}$ for all k $\in \mathcal{C}$ and j $\leq$ m
4: $l^{(k)}$= $\max_{1 \leq i \leq 2m} L_i^{(k)}$ for all k $\in \mathcal{C}$
5: $\alpha^{(k)}$=W[k]/$l^{(k)}$ for all k $\in \mathcal{C}$
6: sort[$\alpha^{(1)}$,$\alpha^{(2)}$,$\alpha^{(3)}$...$\alpha^{(n)}$] in non-decreasing order and set the result to γ

we directly compute the ratio of weight and load at Line5. At last, we sort the ratio of each coflow in non-decreasing order. Comparing with the offline algorithm which has the O(n(m+n)) for-do loop, Algorithm 2 only has sort operation at last. If we use insertion sorting, main elementary operations is only O(m+nlogn) and this much faster and easier than Algorithm 1.

C. Distance between the online and offline algorithm

Comparing with the offline 2-approximate algorithm, algorithm 2 ignores load diversity and this may lead to performance loss. In this section we study the performance gap between the online and the 2-approximate offline algorithm. Firstly, we consider the schedule of 500 coflows served by a small-scale cluster consisting of 60 hosts. These hosts are connected with a non-blocking network fabric, where each port (both ingresses and egresses) has the capacity of 1 Gbps. Coflow traces are synthesized with the same real-world parameters detailed in Section VI. Weight of each coflow ranges within 1 to 5. Fig. 1 shows the result.

From Fig. 1,we can see some details of performance gap between the two methods. In total, the online algorithm performs close to the 2-approximate algorithms and it only prolongs the transfer of large coflows. As coflows are generated randomly, we repeat the numerical simulation 200 times. Table I shows the proportions of $\frac{avgcwct\ of\ online}{avgcwct\ of\ 2-appr} - 1$, where $avgcwct$ means average coflow weight completion time. We can see that more than half of the performance loss is less than 0.025. More than 91% cases have the less than 0.1 performance gap and no cases exceed 0.15.

TABLE I: Performance loss result

$\leq 0\%$	≤ 0.025	≤ 0.05	≤ 0.1	≤ 0.15
0	64%	72%	91%	100%

TABLE II: Comparison between the two algorithms

#Scheme	Sched-Mode	Procedure	Performance
2-appr	offline	Complex	High
Yosemite	online	simple	High

We conclude comparing with the offline algorithm, online algorithm has little performance. Indeed, it is much more simple and can maintain relative high performance just as Table. II shown, so in reality, using the online algorithm to schedule coflow is a good choice.

V. YOSEMITE SYSTEM

In this section, we introduce the design of Yosemite which is a system aims to minimize average coflow weight completion time. Yosemite includes about 4000 lines of java as well as scala code and it follows the actor model like Alluxio [2], hadoop [3],etc. Yosemite uses Akka [1] for message transferring and kryo [4] for object serialization. Code of Yosemite can be downloaded at [7]. In this section, we first introduce actors of Yosemite and the communication between them. Then we introduce the usage of Yosemite.

A. Actors and communication

Algorithm 3 Procedure of bandwidth allocate

Input: Sorted coflow set γ, Remaining capacity for ports set $Rem(.)$
Output: bandwidth of each coflow, Remaing capacity for ports set $Rem(.)$

1: **for** k $\in \gamma$ **do**
2: Compute Load $g^{(k)}$ according to (5)
3: **for** $f_{i,j} \in k$ **do**
4: $b_{i,j} = f_{i,j}/g^{(k)}$
5: $Rem(P_i^{(in)}) -= b_{i,j}$
6: $Rem(P_j^{(out)}) -= b_{i,j}$
7: **end for**
8: **end for**

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 according to the computation result sent from master.

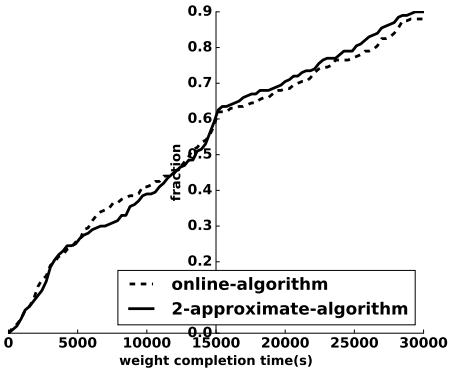


Fig. 1: Performance comparison between 2-approximate algorithm and online algorithm

Algorithm 4 Procedure of the master

Input: Coflow F
Output: bandwidth of each flow

1: $\mathcal{C} = \mathcal{C} \cup F$
2: Sort $\mathcal{C}$ according to Algorithm 2
3: Allocate bandwidth to $\mathcal{C}$ according to Algorithm 3
4: Distribute unused bandwidth to $\mathcal{C}$ for all ports
5: Send $b_{i,j}$ to corresponding Slave

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 worker and sends bandwidth information to worker. It also has two components: slave and scheduler. The function of slave is same as that as worker node. Yosemite Scheduler performs coflow sorting and bandwidth computation. Algorithm 4 shows the operation detail of scheduler.

Algorithm 4 is called when a new coflow arrives. Line1∼Line2 tries to sort coflow. It calls the online coflow scheduling Algorithm 2. Line3 tries to allocate bandwidth to coflow on the basis of sequence that is computed at Line2. Algorithm 3 has two key steps. Firstly, it computes the bottleneck time of each coflow. The computation method of it is:

$$g^{(k)} = \max(\max_i \frac{\sum_{j=1}^{m} f_{i,j}^{(k)}}{Rem(P_i^{(in)})}, \max_j \frac{\sum_{i=1}^{m} f_{i,j}^{(k)}}{Rem(P_j^{(out)})}) \quad (5)$$

Where Rem(.) denotes the remaining bandwidth of an ingress or egress port estimated by the scheduler. (5) gets the slowest time of each coflow according to the bottleneck link. After this, on the base of bottleneck time, it computes $b_{i,j}$ for each flow that transfers from port i to port j Then master updates port available bandwidth as Line5∼Line6 shown. After the process of Algorithm.3, scheduler calculates out bandwidth for each flow. At last, scheduler sends the result to each worker, then worker sends flows according to the computation result.

In particular, communication between components is important for a correct distributed, concurrent, fault-tolerant

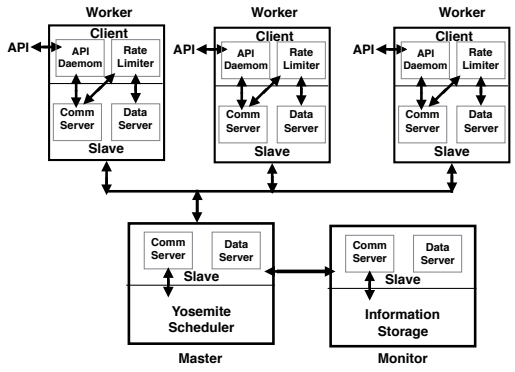


Fig. 2: Yosemite architecture. Computation frameworks interact with Yosemite through a client library

and scalable application. As messages transferring between components needs timely responds, keeping responsive in the face of failure, staying responsive under varying workload. In Yosemite, Comm Server is responsible for this. Comm Server extends Akka [1]. Using the library provided by Akka, Comm Server can provide stable communication for Yosemite components.

B. Work-conserving allocation

Line4 of Algorithm 4 aims to provide work conserving allocation for resources. Just using the computation result of Algorithm 3, extra network resources may be wasted. 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.

C. Fault-tolerance

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 be crushed. 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 just the same as no-crushing down. Also, work node may also crash. If this happens, master will try to restart the worker node daemon to let it work normally.

D. API

Yosemite provides API like DOT [32] and Varys [15] to abstract the underlying scheduling and communication mechanisms. User jobs do not require any modifications, but should create Yosemite client to interact with the master. The client of Yosemite provides 4 key API just as Table. III shown.

TABLE III: Yosemite API

Name	parameters	Return	Description
RegisterCoflow	coflowdesc,weight	coflowId	Regisger a coflow
Send	coflowId,size,dst	bool	Send size data to dst
Receive	coflowId	bool	Receive a flow
ReleaseCoflow	coflowId,	bool	Release coflow

Register takes coflow description and weight of the coflow as parameters. After sending the register information to master, coflow id will be returned to worker node. Send tasks coflowId and IP address of destination node as parameters. This method sends a flow that belongs to coflowId. Receive tasks coflowId as the parameter. This method is used to receive data belongs to the corresponding coflow. ReleaseCoflow takes coflowId as the parameter and it is used to release the coflow when finishing transferring.

VI. EVALUATION

We evaluate Yosemite at our openstack [5] platform. Our private cloud can run at most 80 virtual machines (2 Cores, 4GB Mem) simultaneously. To evaluate a slightly larger scale experiments, some VMs will start at most 3 dockers. For larger scale experiments, we use trace-driven simulation. Main results of our evaluation are as follows:

- Running file distribute application at our private cloud platform, we find Yosemite improves 30% over Varys on minimizing average weight coflow completion time.
- Using facebook trace [15] with randomly generating weight, Yosemite improves about 30%, 40%, 50% over Varys [15], Aalo [13] and Barrat [18] on reducing average coflow weight completion time respectively. However, comparing with Varys, it only has 5% performance gap on reducing average coflow completion time.
- Under different length, width, weight and concurrent number of coflows, we find Yosemite improves more under larger length variance, width variance, weight variance and concurrency.
- Comparing with the LP-based algorithm [29] , Yosemite has only less than 10% performance loss.

A. methlogy

Experiment settings. In some of our experiment we use the traffic of Facebook that was collected from a 3000-machine cluster with 150 racks [15]. As the original trace does not contain information of weight settings, we randomly generate weight within 1 to 10. The original cluster had a 10 : 1 core-to-rack oversubscription ratio with a total bisection bandwidth of 300 Gbps. Due to hardware limits, in our testbed experiment, we limit the bandwidth of ingress and egress port capacity to 1Gbps and maintain the characters of the original traffic. In our experiment, 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.

Metrics. In our experiment, metrics for comparison is the improvement in average weight coflow completion time of jobs

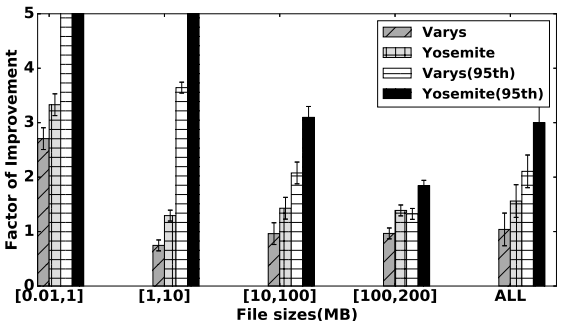


Fig. 3: [cloud] File distribute application performance comparison

(when its last task finished) in the workload. In this paper, we use factor of improvement as our metrics. The definition of factor of improvement of current method is :

$$Factor\ of\ improvement = \frac{baseline\ avg\ WCCT}{current\ avg\ WCCT} \quad (6)$$

Note, in this paper, we use TCP-fair sharing as our baseline method.

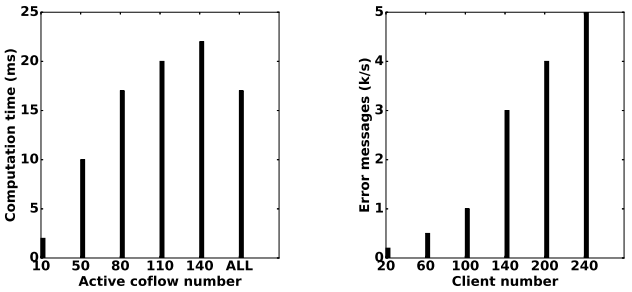
Open source. To make our experiment repeatable, we publish main codes of Yosemite as well as Yosemite-Sim. Sources of Yosemite can be downloaded at [7]. For larger scale of experiment in this paper, we use trace-driven simulation. Main codes of the Yosemite-Sim including trace generator can be downloaded at [8].

B. Cloud platform 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 [6], we constrain the maximum bandwidth of each VM's NIC to 1GB/s.

Firstly, we run file distribute application to test the performance of Yosemite. Each virtual node constantly construct files whose size ranging from 1KB to 200MB. Weight of each job is randomly generated ranging from 1 to 10. Destinations are K nodes which are randomly chosen from the total nodes set. Job can be regarded as finished until all the K nodes receive the file. TCP-fair is chosen as the baseline and Fig. 3 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. To decrease the influence of extreme values, we remove the top and last 2.5% of each methods and recompute the factor of improvement, then we get the 95th factor of improvement. From Fig. 3, 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, which is similar to Varys.

Work conserving aims to allocate the spare network resources to flows, as a result, network resources will be used more efficiently. Efficient network utilization will accelerate the transfer process. In Yosemite, scheduler allocates the spare bandwidth to flows equally to guarantee the utilization of network resources. From Fig. 4 , we can see that with work conserving, factor of improvement of Yosemite is 2.1 in total, while that drops to 1.8 without work conserving.



(a) Computation time of scheduler (b) Error messages frequency

Fig. 5: [cloud] Overheads of Yosemite

In reality, scheduler should be fast enough to compute out the scheduler order of coflows. Fig. 5(a) 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. Messages transferring between comm server is the main overheads of Yosemite system. Error messages occurs when component crashes and server resources are insufficient. As 80 VMs in our platform will reach full load, we start $1 \sim 3$ dockers at each VM to get a slightly larger scale experiment. Fig. 5(b) shows the error messages frequency. We can see that when the concurrent worker number is over 200, peak error messages is over 4k/s.

Write down the trace of distribute file application and let the simulator run the trace under the same parameter. Fig. 6 shows the performance gap between Yosemite and trace-driven simulator. We can see performance gap between cloud platform and Yosemite is narrow($< 15\%$). In the following

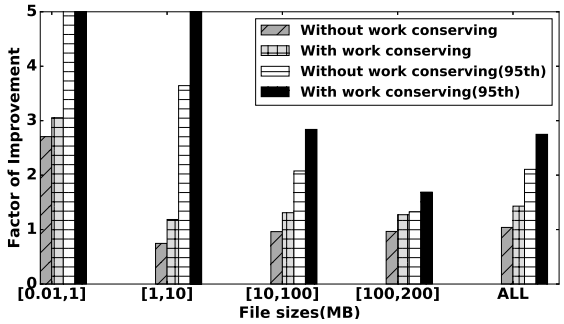


Fig. 4: [cloud] With and without work conserving comparison

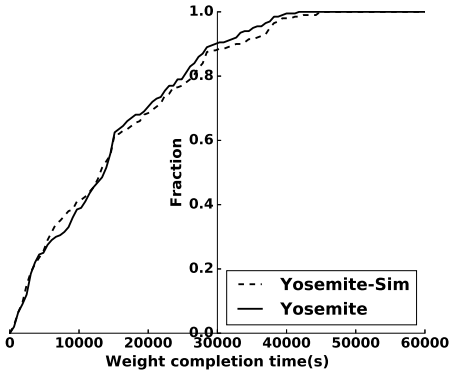
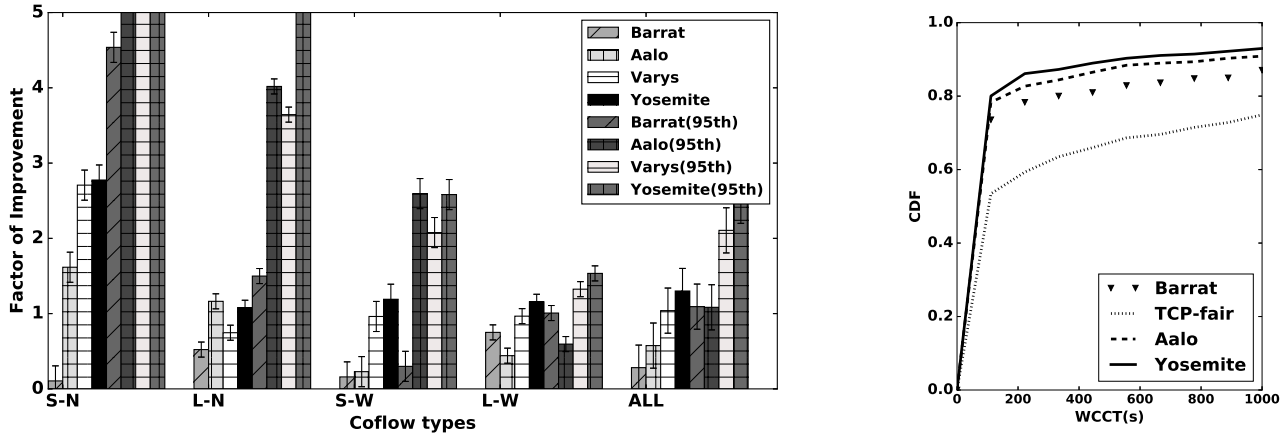
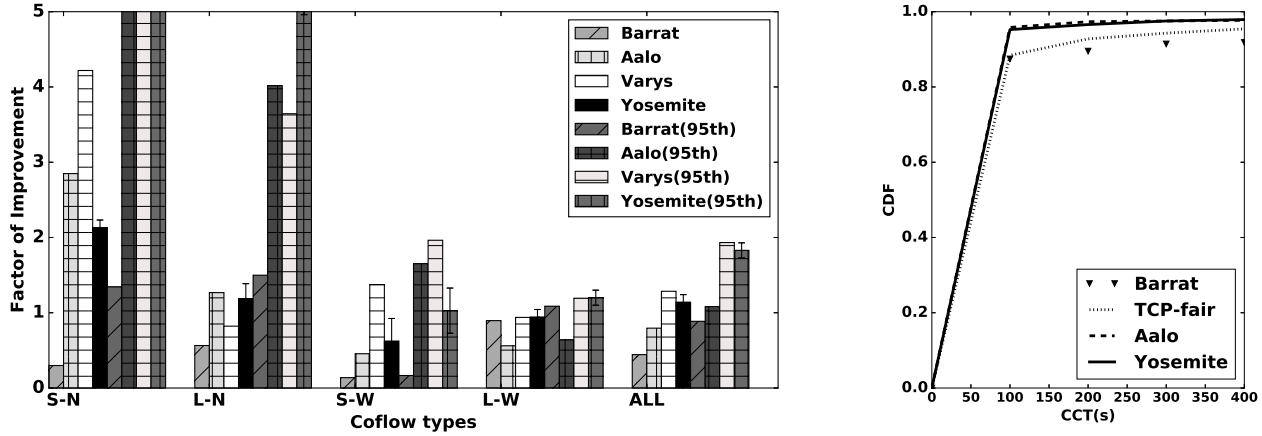


Fig. 6: [cloud] Performance comparison between Yosemite and Yosemite-sim under the same setting



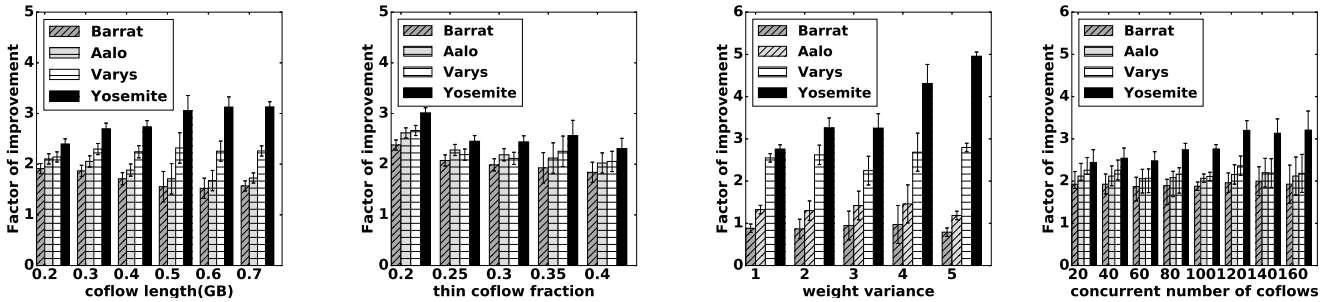
(a) Improvement of wcct (b) Distribution of wcct(s)

Fig. 7: [Simulation] Detail of wcct for Yosemite, Barrat, Varys, Aalo, TCP fair-sharing under real data center traffic trace,TCP fair-sharing is selected as the baseline.



(a) Improvement of cct (b) Distribution of cct(s)

Fig. 8: [Simulation] Detail of cct for Yosemite, Barrat, Varys, Aalo, TCP fair-sharing under real data center traffic trace, TCP fair-sharing is selected as the baseline.



(a) Impact of coflow length (b) Impact of coflow width (c) Impact of coflow weight (d) Impact of coflow concurrency

Fig. 9: [Simulation] Yosemite performance comparison with Baraat and Varys, Aalo under different settings, TCP fair-sharing is selected as the baseline.

section, we use our trace-driven simulator to do some larger experiments.

C. Trace-driven simulation

In this section, we use trace of facebook [15] to test the performance of Yosemite. As the original trace doesn't include

weight information, we randomly generate weight ranging from 1 to 10. Repeat the process 100 times and Fig .7 shows the performance comparison between Yosemite, Barrat, Varys, Aalo and TCP fair-sharing. Note TCP fair-sharing is selected as the baseline in this group.

Fig. 7(a) implies that Yosemite greatly reduces the average wcct across all coflow types. The improvement factors of Yosemite are 3.5 (Narrow&Short), 1.6 (Narrow&Long), 1.7 (Wide&Short), 1.4 (Wide&Long) and 1.5 (ALL), while that of Varys are 2.4 (Narrow&Short), 0.9 (Narrow&Long), 1.2 (Wide&Short), 1.3 (Wide&Long) and 1.2 (ALL). We repeat the experiment 100 times and error bar indicates maximum, minimum and average value. To eliminate the influence of extreme value, we we remove the top and last result 2.5% of each methods and recompute the factor of improvement, then we get the 95th factor of improvement. For the 95th case, we can find that Yosemite has the improvement factors on average weight coflow completion time of 30 (Narrow&Short),20 (Narrow&Long), 2.5 (Wide&Short), 1.5 (Wide&Long), and 2.5 (ALL), however Varys has 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. 7(a) also indicates that FIFO_LM based solution such as Barrat only accelerates the transmission of long tasks (including Narrow&Long and Wide&Long), however, for the short ones, it has poor performance. This is because that FIFO_LM based solution can lead to head-of-line blocking problem especially for the heterogenous coflows which has various flow length, width and arrival time.

Fig. 7(b) is the distribution of wcct. Compare with TCP-fair sharing method, through Yosemite can reduce average wcct, it prolongs the tail value. That is to say, for applications which involves lot long flows, weight completion time will have long duration using Yosemite and this is trivial for private cloud.

Fig. 7 indicates, Yosemite can decrease wcct significantly. One may be interested in the coflow completion time (cct) performance of Yosemite. Fig. 8 shows cct performance of Yosemite, Barrat, Varys, Aalo, TCP-fair sharing with the same settings, TCP-fair sharing is selected as the baseline. Fig. 8(a) shows that the improvement factors of Yosemite on cct are 2 (Narrow&Short),1.1 (Narrow&Long), 0.8 (Wide&Short), 0.9 (Wide&Long) and 1.3 (ALL) , while Varys performs about $\sim 5\%$ better than Yosemite. Fig. 8(a) also shows the maximum value and minimum value of Yosemite under different weight settings. For Barrat, Aalo and Varys, as their cct have no relationships with weight, so that average cct will not change under different weight settings. Fig. 8(b) shows the result of distribution of coflow completion time which is similar to Fig. 7(b).

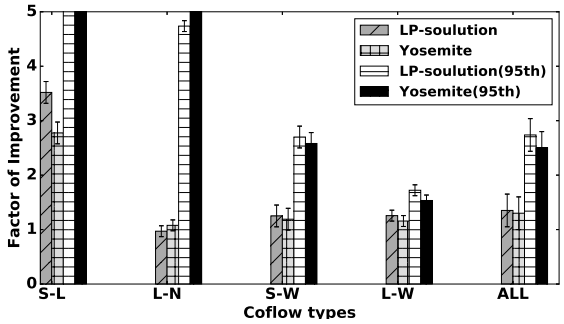


Fig. 10: [Simulation] Performance comparison between the LP-based solution and Yosemite

D. Performance under different settings

In this section, we explore the performance of Yosemite under different settings. From Algorithm 2, we can see performance of Yosemite is mainly influenced by coflow length, coflow width, coflow weight and coflow concurrency. In this part, we explore Yosemite's performance from the four aspects.

TABLE IV: Default parameters in each group of experiment

Number	Length	Width	Weight	Arriva	Capacity
512	[100KB,10GB]	60	[1,10]	[0s,1000s]	1GB/s

In each group of experiment, we fix the value of other parameters and only change the factor we want to study. Table IV shows the default settings of parameters. We start 512 coflows in each group of experiment and length of each coflow are within 100KB to 10GB. Width of each coflows are at most 60 and weight of each coflows are within 1 to 10. coflow's arrival time ranges 0s to 1000s. Ingress and egress port capacity in our experiment are 1GB. We repeat each group of experiments 100 times and draw maximum, minimum as well as average value in the picture.

1) Impact of coflow length: In this group of experiment, we test the influence of coflow length. Coflow width, weight, arrival time are set as Table IV shown. In each group of experiment, we only change length of coflow. Coflow length in our experiment are set within [100KB,K]. Varying K from 200MB to 700MB. Fig. 9 (a) shows the result. From Fig. 9 (a) we can see that with the increasing of K, improvement of Yosemite becomes larger. For example, when K is 200MB, improvement of Yosemite is 2 and when K is 700MB, improvement of Yosemite is 3. Indeed, when K becomes larger, length differentiation between coflows becomes larger, scheduling result is better.

2) Impact of coflow width: Fig. 9(b) shows the influence of coflow width. In this group of experiment, we vary the fraction of thin coflows (width$<$ 30) from 20% to 40%. Coflow length, width, as well as coflow arrival time are set as Table IV shown. From Fig. 9(b), we can see that with the increasing fraction of thin coflows, improvement of Yosemite becomes smaller. The reason for this is that, when fraction of thin coflows is small, conflict probability between coflows is large, so it is necessary to decide a suitable scheduling sequence of coflows.

3) Impact of coflow weight: Weight indicates the importance of the application in reality. Weight setting has huge influence on the performance of weight coflow completion time. In this section, we study the influence of weight. Coflow length, width and arrival time are set according to Table IV. Weight are generated within $[8-\sigma, 8+\sigma]$, varying σ from 1

to 5 and Fig. 9(c) shows the result. From Fig. 9(c), we can see that when weight variance σ becomes larger, improvement of Yosemite becomes larger. This means Yosemite works better for larger weight variance cases.

4) Impact of coflow concurrency: In real word, coflow can start at any time. Arrival time of coflows can impact the traffic load in data center network. In this part, we explore the impact of coflow concurrency. Changing the number of coflows that start at time 0. Fig. 9(d) shows the result. We can see that with the concurrent number of coflows increasing, improvement of Yosemite becomes larger. The reason for this is that when workload becomes heavier, the optimization space for preemptive scheduling become larger.

E. Distance to Optimiztion

It is hard to find the optimal schedule is infeasible, we tried to find an LP-based solution [29]. We run the experiment with facebook trace and generate weight within [1,5]. Repeat the experiment 20 times and experiment is shown as Fig. 10. We can see that in total Yosemite has less than 10% performance loss.

VII. Conclusion

In this paper, we use weight to quantify the importance of applications. Coflows belonging to important applications own large weight. We design and implement Yosemite-a coflow scheduling system that aims to minimize average weight coflow completion time. We test the performance of it by trace-driven simulation and test-bed deployment. Experiments show Yosemite can reduce average weight coflow completion time significantly.

Although Yosemite works well on minimizing average weight coflow completion time, It is Clairvoyant method and this constraints its usage. Indeed in data center network, applications such as file backup, file distributes can know these information. For some computation applications, they are hard to get. As a result, in the future, we will make advance of Yosemite to let it work in Non-Clairvoyant way.

References

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

[2] alluxio. http://www.alluxio.org/.

[3] Hadoop. http://hadoop.apache.org/.

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

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

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

[7] Yosemite. https://github.com/zhanghan1990/Yosemite/.

[8] Yosemitesim. https://github.com/zhanghan1990/YosemiteSim/.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 98–109. ACM, 2011.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Z. Qiu, C. Stein, and Y. Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 294–303. ACM, 2015.

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

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

[32] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *NSDI*, 2006.

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

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

[35] 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.

[36] 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.