Yosemite- Efficient weight coflow scheduling in data center

Han Zhang*‡, Xingang Shi†‡, Xia Yin*‡, Zhiliang Wang†‡,YingYa Guo*‡ *Department of Computer Science and Technology, Tsinghua University
†Institute for Network Sciences and Cyberspace, Tsinghua University
‡Tsinghua National Laboratory for Information Science and Technology (TNLIST)
Beijing, P.R. China
Email:{zhanghan, wzl, yxia,guoyingya}@csnet1.cs.tsinghua.edu.cn, {shixg}@cernet.edu.cn

***Abstract*—Nowadays, applications (map-reduce, distributed storage, etc) in data center need the data center network to provide them with low latency and high throughput. However, network resources in data center network are limited, so efficient network resources schedule methods are needed. Recently, the proposed application level abstraction coflow tries to regard flow set as a whole, so the transfer latency of application will be reduced. This abstraction is better than flow level schedule method because it considers the semantics of the application partly. However, more application semantics should be considered such as the importance of application. This is because different applications in data center may have different level of importance, scheduling only depends on network condition and coflow set are not enough. In this paper, we advocate that when scheduling coflow, we should also take the importance of application into consideration. We incorporate weight to present the importance of application and try to minimize weight completion time of coflow. According to this, we propose Yosemite-a framework that aims to minimize coflow weight completion time. Simulation shows that Yosemite performs about 20%-50% better than Varys to minimize coflow completion time. We also deploy Yosemite system on our open-stack testbed, evaluation shows that for the file backup applications, Yosemite performs about XX better than Varys.**

I. Introduction

Data center is now used as the infrastructure of many applications, including file storage [15], [22], map-reduce [14], etc. These applications need the data center network to provide them low latency and high throughput [20], [21], [25]. However, data center can not satisfy all the demands of every application, due to the limited resources of the network. Many schedule methods are proposed to solve this problem.

Flow level scheduling method such as PDQ [17] and pFabric [8] implement SJF policy to let short flows preempt the bandwidth of large flows. As a result, average flow completion time will reduce and applications latency will reduce. Although, flow level scheduling method can optimize the latency of applications to some extent, it has some deficiencies [12]. On one hand, applications in data center have dozens of flows and flow level optimization can lead to some late-finishing ones lagging the performance of application. One the hand, flow level scheduling is too frequent and this can exert large pressure to scheduler, especially for centralized methods, which need to compute the schedule results as fast as possible.

To overcome the defects of flow level optimization. The recently proposed network abstraction coflow regards job flow set as a whole . Jobs from one or more frameworks create multiple coflows in a shared cluster. The coflow schedule methods regard the flow sets as an entirety. Flows belonging to the same set sharing the same priority. This method is attractive because even the fifo based method [16] can improve much over the state-of-art flow scheduling strategy [8] on the performance of job completion. The centralized method Varys incorporates network bottleneck. It uses SEBF which is known as Smallest-Effective-Bottleneck-First heuristic to schedule coflow and use Minimum-Allocation-for-Desired-Duration (MADD) to allocate rates to flows. Varys is indeed a clairvoyant method whose scheduler should know coflows' width and size of every flows beforehand.

Nowadays, the coflow scheduling methods emphasizes on minimize coflow completion to optimize coflow scheduling. They defines the priority of coflow only according to network. But in reality, the semantic of application should also be considered. In data center, applications have different level of importance. For example, if hadoop and erasure code storage system are running in our data center, they compete for the network resources. Using previous scheduling method, bandwidth scheduling is based on network condition. However, from our application semantic, we think hadoop's flow should more emergency than storage, but we have no explicit deadline. We just want them to transfer as quickly as possible, so in this scenery, we give higher priority to hadoop application than storage application. A good schedule method should consider both network condition and application semantic. We should quantify the importance of application semantic and schedule bandwidth according to the two factors, so minimize the semantic application completion time is important.

In this paper, we incorporate weight into coflow schedule. Coflows whose level of importance are higher will have larger weight. Bandwidth allocation relyes both network condition and coflow weight, as a result, those coflows have higher weight will get more bandwidth under the same network condition. We design Yosemite, a online system to schedule coflow. When coflow arrives, Yosemite firstly computes network load and then range coflows according to weight and network condition. Our scheduling algorithm borrows idea from the 2-approximate offline algorithm. We change our

algorithm into online version and shows that comparing with Varys and Barrat, Yosemite can reduce weight completion time which is also application semantic completion time significantly. Our random generated trace of coflow, shows that Yosemite can improve about 30%-50% of weight completion time over Varys. We use facebook trace of coflow and find that Yosemite improve about 20%-30% over Varys and about 40%-50% over Barrat. We also evaluate Yosemite into our open stack testbed and evaluation shows that Yosemite improves about XX over Varys and Barrat. Thus the contribution we make in this paper:

- A data center weight coflow completion time online algorithm Yosemite to help to reduce coflow weight completion time.
- A Yosemite based Yosemite-simulator to test the performance of Yosemite, our simulation shows Yosemite can perform about 30%-50% better than Varys for weight completion time.
- Yosemite testbed on open stack platform, our real evaluation shows Yosemite performs about XX better than Varys for average weight completion time.

The rest of the paper is organized as follows. In section II we introduce some background and related work. Then at section III, we define the problem WCCO problem and prove the equality of the problem to concurrent open shop problem. At section IV, we first introduce a 2 -approximate algorithm and then proposal an online approximate algorithm to replace the offline algorithm. Section V introduces the design of Yosemite system.

II. BACKGROUND AND RELATED WORK

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

DCTCP [7], D2TCP [31],L2DCT [26] and LPD [34] are rate control method. These methods try to change tcp congestion window to adapt to the bandwidth demand of application. These methods are implicit rate control method [8]. They use ECN marking schema to let the buffer stay at low level, so that the queue delay reduces. These methods are simple but can never precisely estimate the right flow rates, so explicit rate control method which can compute rate accurately is needed.

D3 [32] and PDQ [17] can schedule flows based on some notion of urgency. They compute flow bandwidth according to flow deadline and network condition. A centralized controller which collects the network congestion level is always needed in this kind of method. Explicit methods always performs better to some extent, but only flow level scheduling is not enough. Because jobs of distributed applications like map-reduce and distributed storage system always have many parallel flows. The job transfer finishes only when all flows of the job finish transferring and only scheduling the single flow may make some large flows in the same job falls behind the other flows. The faster ones may wait a long time until the slow flows finish, so scheduling the flows as a whole set is more meaningful.

Having recognized the above limitations, application level schedule schemas occurs. Barrat [16] is FIFO-LM based schedule method. In this method, it schedules tasks in a FIFO order but avoids head-of-line blocking by dyamically changing the level of multiplexing in the network. It schedules application flows as a set, so that average completion time of the whole tasks will reduce. FIFO based schema performs well for some short burst tasks. However, for some long and width applications, it can not reach the expection. Varys [13] regards applications' parallel flows as coflow and it uses SEBF to schedule coflows and MADD to perform rate control. Aalo [23] and sunflows [18] bases on Varys. They need to know coflow information beforehand. Then for some applications, coflow width and length are hard to get beforehand. D-CLAS [11] and CODA [33] try to optimize these applications. They use accumulative or machine learning methods to compromise the lack of coflow knowledge. This kind of method can be used more widely, however, the performance is not as good as the Clairvoyant methods such as Varys. We conclude that no matter Clairvoyant or non-Clairvoyant methods try to schedule application transferring according to network condition. Also, in data center work, applications have different level of importance. When scheduling application, the semantics of application should also be considered.

There are many methods to present the semantics of the application. Some flow-level algorithms such as D2TCP [31] and LPD [34] use deadline to present the emergency of applications. Flow missing deadline will be threw way. Deadline can guarantee the application to have low bound of bandwidth and it can describe the semantics of application to some extent. However, in data center network, applications do not need a minimum bandwidth or have a strict finish time constraints. They just have higher level of emergency and hope to finish as quickly as possible, so another kind of method to present the emergency of application according to the semantics is needed. [27] and [28] incorporate weight into consideration. They set larger weight to the more emergency jobs. Instead of minimizing coflow completion time, they try to minimize weight coflow completion time. Comparing with optimizing weight completion time, optimizing weight completion time considers application semantics. They make progress on application bandwidth allocation. However both [27] and [28] have high complexity, which make them hard to deploy in practice. In reality, we need the method to compute quickly to react the burst of flows in data center network.

III. SYSTEM MODEL AND PROBLEM FORMULATION

A. Data center non-blocking model

Recent researches have regardedc the data center network as a big giant non-blocking switch which interconnects all the machines. Just as Fig. 1 shows. Under this assumption, congestion only occurs at the egress and ingress ports. Flows always contend for the ingress and egress ports' network resources. It is practical because of recent advances in full bisection bandwidth topologies distribute flows efficiently.

Also, some previous study [11] [18] [13] [8] use this model to design the algorithms. To simplify the problem, in this paper, we follow the assumption and only compute the ingress and egress ports' congestion.

B. Application importance semantics

Applications in data center share the networks resources. Different applications have different level of importance. The more emergency applications hope to get more bandwidth. Some previous study [31] [32] [34] uses deadline to present the emergency of applications. Deadline indeed emphasizes a minimal bandwidth constraints on applications. In reality, in data center network, applications do not need a real lower bound of bandwidth, they are urgent and just want to transfer as quickly as possible. To present the level of emergency, we incorporate weight to present the urgent level of applications. Applications whose level of emergency is high, will have a larger weight. Instead of trying to minimize coflow completion time, we try to minimize coflow weight completion time.

C. Problem formulation

In this subsection, we consider the offline schedule problem of minimizing weight coflow completion problem. There are m machines connecting to the data center fabric, thus according to the non-blocking model fabric as Fig. 1 shown, there are m ingress ports and m egress ports. Assume there are n coflows to be scheduled. Each coflow involve $m \times m$ parallel flows. The k-th coflow is a collection flows denoted by $F^{(k)} = \{f^k_{i,j} | 1 \leq i \leq m, 1 \leq j \leq m\}$, 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 $f^k_{i,j} = 0$.

Let C_i denote the completion time coflow F_i. Weight of coflow F_i is w_iThe capacity of each ingress and egress ports is 1. Just as [13], we now define the Weight Coflow Completion Optimization (WCCO) problem as follows:

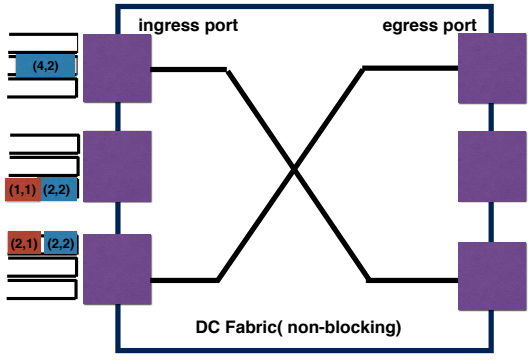


Fig. 1: Data center as non-blocking fabric: in this fabric, congestion only occurs at the ingress ports or egress ports

$$\text{minimize} \quad \sum_{i=1}^{n} w_i C_i \tag{1}$$

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

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

(1) is the optimize goal. The goal is to try to minimize weight completion time. (2) and (3) are ingress and egress ports bandwidth capacity constraints.

D. NP-hard and problem equivalent proof

We give the definition of Concurrent Open Shop problem, then we prove WCCO is equivalent. The problem of concurrent open shop problem can be described as: considering that a job consists of up to m different components to be processed on a specific one of m dedicated machines. Components are independent of each other, such that components of the same job can be processed in parallel on different machines. A job is completed once all of its components are completed [29]. The goal of concurrent open shop problem is to try to find a sequence to minimize weight completion time or to find a sequence to let more jobs finish before deadline [9].

proposition *1:* Minimizing weight completion time of coflows is equivalent to minimizing weight completion time of jobs in concurrent open shop problem.

To prove proposition 1, Firstly we consider the single coflow condition. Let $L(F^{(k)})$ denote the maximum load that the coflow $F^{(k)}$ exerts. The definition of $L(F^{(k)})$ is as follows:

$$L(F^{(k)}) = max\{max_i \sum_{j=1}^{m} f^{(k)}_{i,j}, max_j \sum_{i=1}^{m} f^{(k)}_{i,j}\} \tag{4}$$

(4) gets the larger loads of ingress port and egress port. For the single coflow situation, weight completion time of coflow $L(F^{(k)})$ can be computed as $w_k * L(F^{(k)})$.

Secondly, assume there are two coflows $F^{(k)}$ and $F^{(l)}$, The priority of coflows indicates the schedule sequence of them. Let P(.) denote the priority of coflow and the schedule sequence of $F^{(k)}$ and $F^{(l)}$ can be defined as definition 1.

Definition *1:* For any two coflows $F^{(k)}$ and $F^{(l)}$, $P(F^{(k)}) \prec P(F^{(l)})$ means coflow $P(F^{(k)})$ has lower priority value than $P(F^{(l)})$ and scheduler should schedule $F^{(k)}$ before $F^{(l)}$. $P(F^{(k)}) \succ P(F^{(l)})$ means coflow $P(F^{(k)})$ has higher priority value than $P(F^{(l)})$ and scheduler should schedule $F^{(k)}$ after $F^{(l)}$.

For two coflows, we can define the operation adding of them as follows:

Definition *2:* For any two coflows $F^{(k)}$ and $F^{(l)}$. We can define the single new $m \times m$ coflow $F^{(n)} = F^{(k)} \bigcup F^{(l)}$, whose flow demand $f^n_{i,j} = f^k_{i,j} + f^l_{i,j}$.

Lemma *1:* For the two coflows, say $F^{(k)}$ and $F^{(l)}$, the optimum of $\min\max\{C_l, C_k\}$ is $L(F^{(k)} \bigcup F^{(l)})$

Proof: Firstly, For the single coflow n, its optimal completion time $L(F^{(n)})$. Then we prove that $L(F^{(k)} \bigcup F^{(l)})$

is the low bound of $\min\max\{C_l, C_k\}$. It is obvious that $\max\{C_l, C_k\} \geq L(F^{(k)} \bigcup F^{(l)})$. This is because if $\max\{C_l, C_k\} < L(F^{(k)} \bigcup F^{(l)})$, then both $F^{(k)}$ and $F^{(l)}$ can finish before $L(F^{(k)} \bigcup F^{(l)})$. According to Definition 2, $L(F^{(k)} \bigcup F^{(l)}) = L(F^{(n)})$, so coflow $F^{(n)}$ can finish before $L(F^{(n)})$. This is contradict to the optimal completion time of $F^{(n)}$ is $L(F^{(n)})$. ■

Lemma 1 indicates that for two coflow system, $F^{(k)}$ and $F^{(l)}$, the optimal schedule completion time of the latter one is $L(F^{(k)} \bigcup F^{(l)})$. Then for weight completion time $\min\{w_l * C_l + w_k * C_k\}$, if $P(F^{(k)}) \prec P(F^{(l)})$, the optimal value of $\min\{w_l * C_l + w_k * C_k\}$ is $w_k * L(F^{(k)}) + w_l * L(F^{(k)} \bigcup F^{(l)})$. Else if $P(F^{(k)}) \succ P(F^{(l)})$, the optimal value of $\min\{w_l * C_l + w_k * C_k\}$ is $w_l * L(F^{(l)}) + w_k * L(F^{(k)} \bigcup F^{(l)})$.

Thirdly, we generate this conclusion to coflow set. Assume there are g coflows. Let $\gamma(k)$ denote the priority sequence of coflow k. $WT(\gamma)$ denote the weight completion time under priority permutation γ:

$$WT(\gamma) = \sum_{i=1}^{g} w_i * L(\sum_{j=1}^{i} F^{(\gamma|j)}) \tag{5}$$

So, Minimizing the weight coflow completion for a given set of coflows is equivalent to finding a permutation γ for them (i.e., optimal priority permutation), so that WT is minimized. so far, we have changed the problem of minimizing coflow weight completion time has been changed to finding the optimal order permutation for them, which is quite similar to concurrent open shop problem. They are equivalent indeed. Next, we will describe the equivalence of them.

For each coflow $F^{(k)}$, let $f_i^k = \sum_{j=1}^{m} f_{i,j}^k$, for ingress ports i=1,2...m and $f_{j+m}^k = \sum_{i=1}^{m} f_{i,j}^k$ for egress ports j=1,2...,m. Substitute f_i^k and f_{j+m}^k into (6), then we get the fact that find the permutation of coflow is the same case as finding the optimal sequence for minimizing weight job completion time in concurrent open shop problem. In the problem, there are 2m jobs donated as f_i^k and f_{j+m}^k to be performed on 2m machines. The machine index is denoted as i=1,2,..2m. On the contrary, by letting flows deriving from the same ingress port drain by the same egress port, we can construct concurrent open shop problem's corresponding coflow schedule problem.

Now, we have proved proposition 1 which is the equivalence of minimizing coflow weight completion time and minimizing job weight completion in concurrent open shop problem. According to [29] and [24], finding a sequence of minimizing weight job completion time is NP-hard. As the problem is equivalent to minimizing coflow weight completion time, so finding a permutation of coflow to minimize weight coflow completion time is also NP-hard.

IV. Algorithm design and analysis

Learning from the conclusion of III-D, we can learn that minimizing coflow weight completion time is indeed to find a permutation of γ that make the value of WC minimize. As the hardness of the problem is NP-hard [29], so efficient heuristic is needed. In this section, we first introduce a 2-approximate algorithm, then we conclude the idea of the algorithm and proposal an online algorithm for coflow weight completion optimization problem. At last we compare the online and offline version of the two algorithm.

A. Weight coflow completion time offline algorithm

Currently, the best known result for the problem is a permutation based greedy algorithm, which is proven to be 2-approximation [19], [24], and we can also find a 2-approximate algorithm for minimizing coflow weight completion time according to this. We first change the 2-approximate algorithm from [24] which intends to minimize weight job completion time to minimize coflow weight completion time.

Algorithm 1 2-approximation offline algorithm

Input: n,m,$f_{i,j}$
Output: γ

1: $\gamma : \{1, 2, ...n\} \leftarrow N$
   $UC \leftarrow \{1, 2, 3...n\}$
   $P \leftarrow \{1, 2, 3...m\}$
   $W\{1, 2, ...n\} \leftarrow \{w_1, w_2, w_3...w_n\}$
2: $L_i^{(k)} = \sum_{j=1}^{m} f_{i,j}^{(k)}$ for all k ∈ N and i ≤ m
3: $L_{j+m}^{(k)} = \sum_{i=1}^{m} f_{i,j}^{(k)}$ for all k ∈ N and j ≤ m
4: $L_i = \sum_{k \in N} L_i^k$ for all i ∈ P
5: **for** i ∈ $\{n, n-1, n-2...1\}$ **do**
6: u=arg $\max_{k \in N} L_k$
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 ∈ UC
10: $L_j = L_j - L_j^{\gamma[i]}$ for all j ∈ P
11: $UC = UC \setminus \{\gamma[i]\}$
12: **end for**

Algorithm 1 is the description of minimizing weight coflow completion time. The algorithm 's elementary operations is O(n(m+n)), where n is the number of colfows to be scheduled and m is the number of machines in data center network. The input of the algorithm is m,n and $f_{i,j}$, which indicates that there are n coflows to be scheduled and each coflow will have $d_{i,j}^k$ size of flows to transfer from machine i to machine j, the total number of machines is m. Output of the algorithm is γ, which indicates the scheduling permutation of coflows.

Line1 does some initialism of the operation. UC is the set of coflows to be scheduled, W is the weight set of each coflow and P is the port set. Line2-Line4 computes the load on each port. Load of each ports is the plus of flow size that goes through the port. Line5 - Line12 decides the schedule permutation sequence. Line6 finds the maximum port which owns the heaviest load. Then Line7 finds the corresponding coflow which has the minimal ration of weight and load on that machine. Line8-Line10 is the adjust process. Line11 gets rid of this coflow from the unschedule sets. Algorithm 1 will generate γ which is the permutation of coflow schedule sequence.

Although the 2-approximate algorithm has been proved to work well [24], it has some deficiencies that constraints its usage in practice:

- The algorithm is an offline version, it assumes that coflow arrives simultaneously, indeed, in data center network, coflows wiil arrive dynamically in practice [10], [11]. If we use the offline version algorithm, we have to recompute the schedule sequence of coflows and this will emphasize heavy load on the controller if coflow arrive too frequency.
- The 2-approximate greedy algorithm's elementary operations is O(n(m+n)). If the number of machines in datacenter or the number of coflows to be scheduled is too large. It will take a long time to get the results of γ.

To make the algorithm work practically and efficiently in data center network, we have to make some relaxations.

B. From offline to online

The 2-approximate algorithm decides the schedule position for coflow by the ratio of weight and load on that machine. In reality, today's datacenters generally assign jobs with load balancing [14], as a result, the schedule process does not need to take the load diversity of ports into account. 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 of coflow weight completion schedule.

Algorithm 2 Online approach to minimize weight completion time

Input: Coflow set $\mathcal{C}$; number of hosts m; remaining flow demands $d_{i,j}$ 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: $L_i^{(k)} = \sum_{j=1}^{m} f_{i,j}^{(k)}$ for all k $\in \mathcal{C}$ and i $\leq$ m
3: $L_{j+m}^{(k)} = \sum_{i=1}^{m} f_{i,j}^{(k)}$ for all k $\in \mathcal{C}$and j $\leq$ m
4: $g^{(k)}$= $\max_{1 \leq i \leq 2m} L_i^{(k)}$ for all k $\in \mathcal{C}$
5: $\alpha^{(k)}$=W[k]/$g^{(k)}$ 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

Algorithm 2 is the online version of the 2-approximate. Different from the 2-approximate offline algorithm, the online algorithm makes the relaxation that ignoring the load difference of each port. Line2-Line4 computes the load of every coflow. Similar to [13], the load of each coflow is:

$$g^{(k)} = \max(\max_i \frac{\sum_{j=1}^{m} f_{i,j}^{(k)}}{Rem(P_i^{(in)})}, \max_j \frac{\sum_{i=1}^{m} f_{i,j}^{(k)}}{Rem(P_j^{(out)})}) \quad (6)$$

$Rem(P_i^{(in)})$ denotes the remaining port capacity of ingress port $P_i^{(in)}$. $Rem(P_j^{(out)})$ denotes the remaining port capacity of egress port $P_j^{(out)}$. In our assumption, the capacity of each port is 1. After computing the load of each coflow. We compute the ratio of weight and load at Line5. At last, we sort the ratio of each coflow in non-decreasing order. Comparing with Algorithm 1, instead of having the O(n(m+n)) for-do loop, Algorithm 2 only has the sort operation. Algorithm 2 is called when a new coflow starts. If we use insertion sorting method, the elementary operations is O(logn), where n is the number of coflow to be scheduled.

C. Distance between the online and offline algorithm

The online algorithm makes relaxation, so the scheduling algorithm becomes much easier. However, one may argue, how is the performance of the online algorithm comparing with the 2-approximate algorithm. To compare the performance of the two algorithms, we use the trace of real traffic from facebook [13]. For the weight of coflows, we random set a integer from 1 to 5. Using our simulator [6] that improves from Varys [13], Fig. 2 shows the result.

We can see from Fig. 2 that for 80% of coflows, performance of the two algorithm has little difference. This means comparing with the 2-approximate offline algorithm, the online-algorithm has little performance loss($< 20\%$). It is practical to use the online-algorithm in reality. More details comparison between the two algorithms are shown at Table I

TABLE I: Comparison between the two algorithms

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

We conclude the online algorithm has little performance loss with simple procedure and online mode, so in our system we will introduce below, we prefer to use the online algorithm instead of the 2-approximate offline algorithm.

V. Yosemite System

In this section, we introduce the design of practical schedule system-Yosemite. We have evaluated Yosemite with about 4000 lines of java and scala, we learn some codes from Varys [4] and hadoop [2]. The code of Yosemite can be downloaded at [5]. Similar to Varys [4], we also use Akka [1] for message transferring and kryo [3] for object serialization. In this section, we first introduce the actors of Yosemite and the messages transferring between the actors. Then we introduce the API of Yosemite.

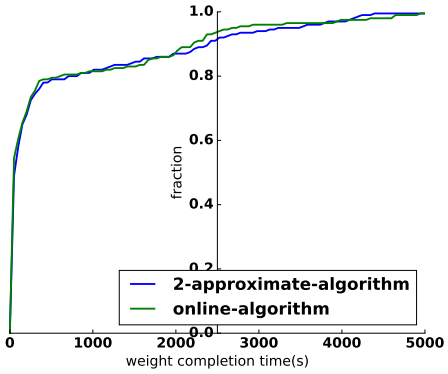


Fig. 2: performance comparison between 2-approximate algorithm and online algorithm

A. Actors and message transferring

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 (6)
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**

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 Slaver

Just as Fig .3 shows. Yosemite has two key actors: master and slaver. Master is the brain of the system. It collects coflow information from slaves, then computes the bandwidth of each coflow and sends the result to slavers at last. Operations detail is shown as algorithm 4. This algorithm is called when a new coflow F arrives. Line1-Line2 tries to find the corresponding position for the coflow. The master uses the online algorithm 2 that tries to minimize weight coflow completion time. Line3 calls algorithm 3 which tries to allocate bandwidth to coflow according to the sequence that is computed at Line2. Algorithm 3 has two key steps. Line2 computes load of the coflow according to (6). After compute the load of coflow, it then computes the bandwidth of coflow. We regard the bottleneck flow bandwidth as the coflow bandwidth. All the flow that belongs to the coflow use the same bandwidth just as Line4 shown. After computing the bandwidth of flow, the master updates the port available bandwidth which as computing as Line5-Line6 shown in algorithm 3. After computing the bandwidth of each flow, the master send the result to slaver. Slavers throttle bandwidth to the value that computed by the master.

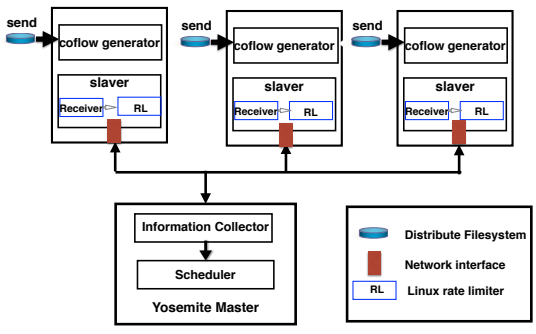


Fig. 3: Yosemite architecture. Computation frameworks interact with Yosemite through a client library

In reality, one of most diffcult things in writing correct distributed, concurrent, fault-tolerant and scalable applications is the communication between the components. Because messages transferring between components needs timely responds, keeping responsive in the face of failure, staying responsive under varying workload. In Yosemite, we build our communication system based on Akka [1]. With the help of Akka, Yosemite has high performance and fault-tolerance communication system.

B. Work-conserving allocation

Just using algorithm 3 to allocate bandwidth to each flows may lead to ports resource idle. To prevent this happening, we allocate the remaining bandwidth fair to the flows that sends from the ports just as Line4 shown at algorithm 4. With work-conserving allocation, resources can used more efficiency, so that less resources will be wasted.

C. Fault-tolerance

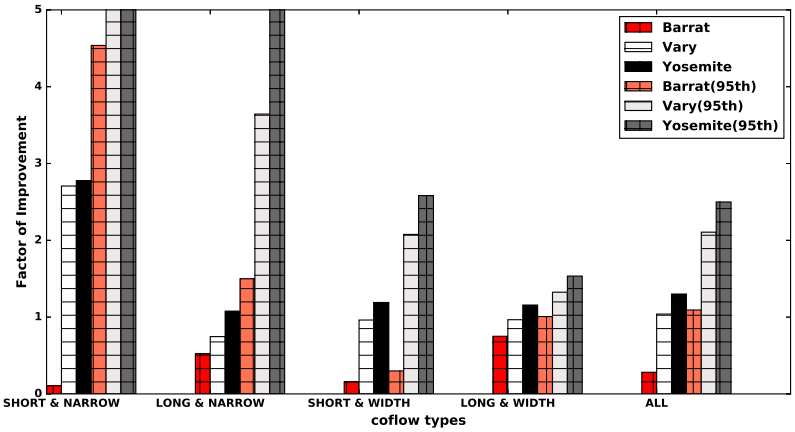
In Yosemite, master plays the role of brain of the system. It collects information from slaver and computes the bandwidth of each flows and sends the result to the slaver. In reality, the master may be crushed. If this happened, the system can not work any more. To prevent this happen, we start a monitor-master which monitors the master. The monitor-master also stores information just as the master. When master collapses, it tries to restart the master and lets it resume the state before crush as fast as possible. Also, the slaver may crash. If the slaver crashes, the master can monitor this and sends restart command to slaver to let it restart. With the monitoring of the actors, Yosemite can make the influence of accidents minimize.

D. API

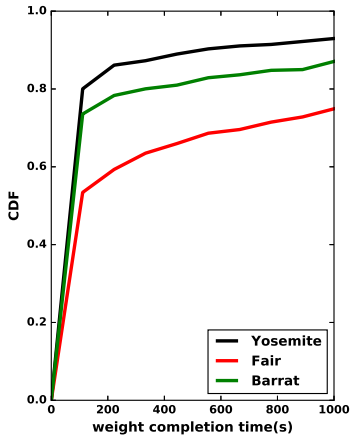
Yosemite provides API like DOT [30] and Varys [13] to abstract the underlying scheduling and communication mechanisms. User jobs do not require any modifications, but should use the API to interact with the master. There are 4 key API that Yosemite provides:

- Register(numFlows, weight), which takes the width, weight of coflows as parameters, and then it returns coflowId.
- Send(coflowId,size,dst), which tasks coflowId, flow size, send destination as parameters.
- Receive(coflowId,src), which tasks coflowId, source ip, as parameters.
- Unregister(coflowId), which tasks coflowId as the parameters.

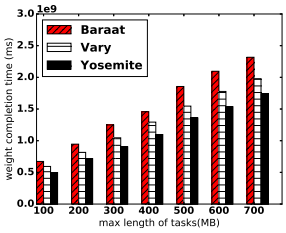
With all the API, users can take advantage of coflow scheduling, thus weight coflow completion time will minimize.



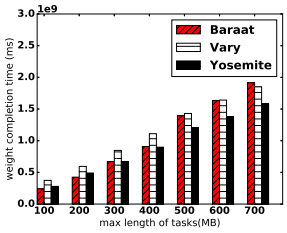
(a) Total weight completion time



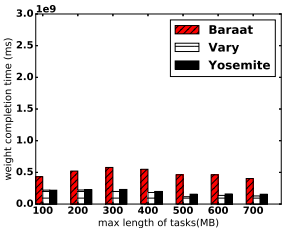
(b) Short coflow weight completion time



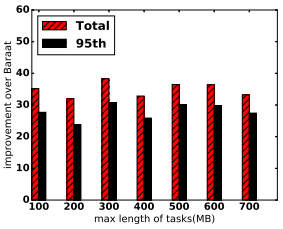
(c) Total weight completion time



(d) Large coflow weight completion time

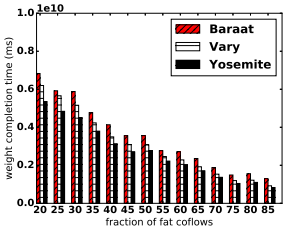


(e) Short coflow weight completion time

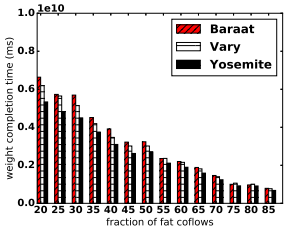


(f) Improvement over Baraat

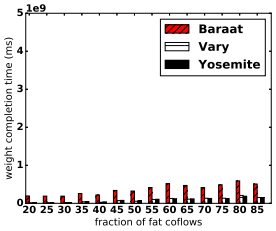
Fig. 4: Yosemite performance comparison with Baraat and Varys when the max length of coflows increasing



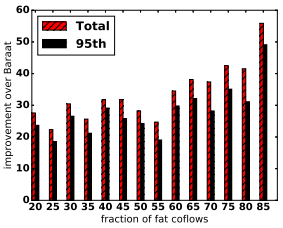
(a) Total weight completion time



(b) Large coflow weight completion time



(c) Short coflow weight completion time



(d) Improvement over Baraat

Fig. 5: Yosemite performance comparison with Baraat and Varys under different percentage of width coflows

REFERENCES

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

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

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

[4] Varys. https://github.com/coflow/varys/.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[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. 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.

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

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

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

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

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

[33] 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.

[34] 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.