Journal of Parallel and Distributed Computing 152 (2021) 45–56

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Django: Bilateral coflow scheduling with predictive concurrent connections

Jiaqi Zheng *, Liulan Qin, Kexin Liu, Bingchuan Tian, Chen Tian, Bo Li, Guihai Chen

State Key Laboratory for Novel Software Technology, Nanjing University, China

ARTICLE INFO

Article history:
Received 14 January 2020
Received in revised form 1 August 2020
Accepted 23 January 2021
Available online 17 February 2021

Keywords:
Coflow scheduling
Data center
Prediction

ABSTRACT

For data-parallel frameworks, their communication is highly structured. Coflow is a networking abstraction proposed for their *all-or-nothing* job-specific semantics. Minimizing coflow completion time (CCT) decreases the completion time of corresponding jobs. However, state-of-the-art coflow scheduling approaches suffer from several drawbacks. On the one hand, both sender-driven and receiver-driven scheduling approaches fail to achieve optimal especially when the bandwidth bottleneck exists. On the other hand, they fail to optimize the number of concurrent connections since the CCT can be prolonged due to too many or too few concurrent connections.

In this paper, we propose Django, a bilateral coflow scheduling framework. We first use Support Vector Machine (SVM) as the machine learning model to automatically identify the optimal number of concurrent connections, *i.e.*, the queue limitation in the switch. Based on the predicted results, we further develop a set of distributed coflow scheduling algorithms in a scalable manner. Testbed experiments and trace-driven simulations show that Django can estimate the number of concurrent connections with an accuracy of 98%, reduce the average CCT and 95th percentile CCT by 15% and 40%, respectively.

© 2021 Elsevier Inc. All rights reserved.

1. Introduction

Motivation: For mainstream data-parallel frameworks such as Hadoop [12,23] and Spark [28], network communication is highly structured. They usually implement a data-parallel computing model, where each group of data flows is required to experience a successive communication stage before producing the final results. At each communication stage, the parallel flows require to exchange data among a set of hosts. Usually such a communication stage cannot complete until all its flows have finished [10]. Hence, the network-level optimization like minimizing the individual flow completion time or improving the fairness among flows cannot respect the application-level semantics and significantly degrade the performance, which misses user demand in modern data centers [20].

Coflow [8] is a networking abstraction proposed for this all-or-nothing job-specific application-level semantics. A coflow is a set of correlated flows in a communication stage. Its completion requires that even the last one of all these flows has finished. This abstraction shortens the gap between the application-level semantics and network-level optimization.

Existing work, especially Varys [11] and Aalo [9] propose to minimize coflow completion time (CCT) thus decrease the completion time of corresponding jobs [10,13]. Varys presents a centralized optimization to schedule all coflows. Its calculation procedure involves solving all flow variables with scalability challenges and introduces high overhead especially for small coflows. Besides, its assumption that all flows start at the same time is not practical due to the ignorance of the multi-wave flow arrival pattern [9]. Aalo distributes the centralized scheduling to each host and develops a sender-driven approach, where the sender hosts can enforce the prioritization among coflows by differentiating their flows into multiple prioritized or weighted shared sending queues. Accordingly, each coflow is assigned with a priority that is decreasing in the total number of its sent bytes and scheduled following the least attained service (LAS) discipline. These two approaches suffer from several drawbacks, however.

On the one hand, they fail to achieve the optimal scheduling. For example, Aalo's sender-driven coflow scheduling can be sub-optimal when the bandwidth at the host ingress links is the bottleneck. Correspondingly, moving to receiver-driven scheduling is not a right choice as well. It can also be sub-optimal when the bandwidth at the host egress links is the bottleneck. In a real network, these two cases usually coexist together due to imbalanced task distribution. An efficient scheduling approach can be able to handle these two cases simultaneously (Section 2.2). On the other hand, existing work fail to optimize the number of concurrent connections at each host. It has been proven that the good throughput has a strong correlation with the number of

* Corresponding author.
E-mail address: jzheng@nju.edu.cn (J. Zheng).

https://doi.org/10.1016/j.jpdc.2021.01.006
0743-7315/© 2021 Elsevier Inc. All rights reserved.

concurrent connections. Of the corresponding switch egress port for each host ingress link, the queue length increases together with the number of concurrent connections even when DCTCP [3] is used. When this number exceeds the limitation of queue buffer, the switch starts to drop packets. As a result, the CCT can be prolonged due to re-transmission and even re-connection. We cannot arbitrarily minimize the number of concurrent connections to avoid packet loss, since this would shrink the feasible solution space of the first problem. In summary, the CCT can be prolonged due to too many or too few concurrent connections (Section 2.3).

Our contributions: In this paper, we propose Django, a distributed bilateral coflow scheduling framework. The optimal number of concurrent connections is predicted in advance, which avoids the cases that the congestion happens resulting from too many connections and the bandwidth wastage resulting from too few connections. Meanwhile, Django can combine the advantages of both sender-driven and receiver-driven scheduling approaches and achieve near optimal.

Our first contribution is that we can automatically estimate the optimal number of concurrent connections during the flow transmission, which will be used for jointly optimizing the coflow scheduling. Considering that the packet loss rate is easy to measure and has a strong correlation with the buffer size, we use packet loss rate as a key feature to predict the optimal number of concurrent connections. We choose Support Vector Machine (SVM) as the machine learning model and use *C*-SVC model as the multi-class classifier [14,16]. By measuring the performance of different setting, we establish a prediction model to determine the optimal number of the concurrent connections.

Our second contribution is that we develop a distributed bilateral coflow scheduling framework, where the sender and receiver hosts can interact with each other independently and asynchronously. Taking the optimal number of concurrent connections as an input, a receiver host has the authority to open or close the connections to the senders. Each receiver host allocates permitted number of connections among coflows according to their priority order. Once a receiver host detects that its ingress link becomes bottleneck, it moves the connections from low-priority coflows to high-priority ones. When this link is not the bottleneck, it moves connections in the reverse direction. In each sender host, the coflows are prioritized as usual sender-driven approaches. Since it cannot control connection status, it throttles low-priority coflows when its egress link becomes the bottleneck.

Our third contribution is a concrete implementation and evaluation of Django. We evaluated our algorithms by replaying production traces from Facebook, on a small-scale testbed with 8 Dell servers and a commodity Mellanox Spectrum Ethernet switch. To complement our small-scale testbed experiments, we further conducted large-scale trace-driven simulations on NS-3. The results show that, Django can estimate the queue limitation with an accuracy of 98%. We can reduce the average CCT and tail CCT 15% and 40%, respectively.

2. Background and motivation

2.1. Background

Related work: We briefly review prior art on different coflow scheduling approaches. Orchestra [10] first introduces the concept of coflow and shows that even a simple FIFO discipline can significantly improve the application performance. Later, Barrat [13] multiplexes multiple transfer jobs to prevent head-of-line blocking. Varys [11] develops a set of heuristic algorithms like *smallest-effective-bottleneck-first* and *minimum-allocation-for-desired-duration* to schedule coflows. To reduce the high overhead of centralized computation and improve the scalability, Aalo [9] follows the classic least attained service scheduling discipline and develops a distributed sender-driven approach, which uses the total number of bytes that a coflow has already sent to approximate the coflow's total size [9].

Another line of this work focuses on designing approximation algorithms to minimize the total weighted CCT. Qiu et al. [21] first develop a $\frac{67}{3}$-approximation algorithm by linear relaxation techniques. Khuller et al. [18] and Sharfiee et al. [22] improve the approximation ratio to 12 and 5, respectively. Besides, Li et al. [19] focus on combining coflow routing and scheduling, while Tian et al. [25] aim at scheduling coflows with dependencies. However, their calculation requires involving all the flows in the whole network, which is slow and does not scale well. Besides, most of these are offline algorithms and require the detail information of all upcoming coflows in advance, which cannot be easily implemented in a real system.

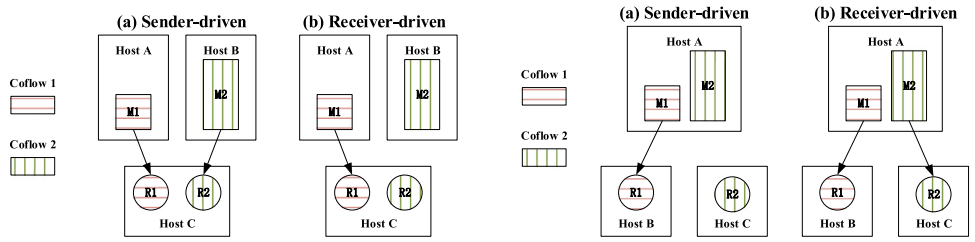
The novelty of our work lies in designing a machine learning based scheduling framework that can predict the optimal number of concurrent connections and coordinate the network information from both sender side and receiver side, which to our knowledge has not been done before.

Network topology and protocol: We consider the entire data center fabric as one non-blocking switch in our paper. This assumption is the same as that in previous work [9,11,21,22,25] and can be established in practice [15], where the fabric is congestion-free and the ingress and egress links (i.e, switch egress and ingress links) for each host are always the bottleneck. In addition, DCTCP is adopted as the congestion control protocol, which is a default setting for most modern data center networks [17].

2.2. Sender-driven and receiver-driven scheduling approaches can be sub-optimal

Sender-driven coflow scheduling, such as Aalo, can be sub-optimal in performance. A toy example is shown in Fig. 1(a). Assume that there are two coflows each with a single mapper and a single reducer respectively. Coflow 1 has a mapper on host *A* and a reducer on host *C*, and the shuffle size is 1 unit. Coflow 2 has a mapper on host *B* and a reducer on host *C*, and the shuffle size is 2 unit. We assume that each egress and ingress port has 1 unit bandwidth. In this case, the receiver host's ingress link is the network bottleneck. For a sender-driven scheduling algorithm, each sender host (*i.e.*, *A* or *B*) observes only one coflow. Thus both coflows 1 and 2 have the highest local priority in its sender host and will transmit simultaneously. They finish at the time of 2 and 3 respectively and the average CCT is $\frac{2+3}{2} = 2.5$ time units. However, for a reducer-end scheduling algorithm, reducer host knows all two coflows. Specifically, the prioritized coflow 1 finishes at the time of 1, while the other finishes at the time of 3, thus the average CCT is only $\frac{1+3}{2} = 2$ time units.

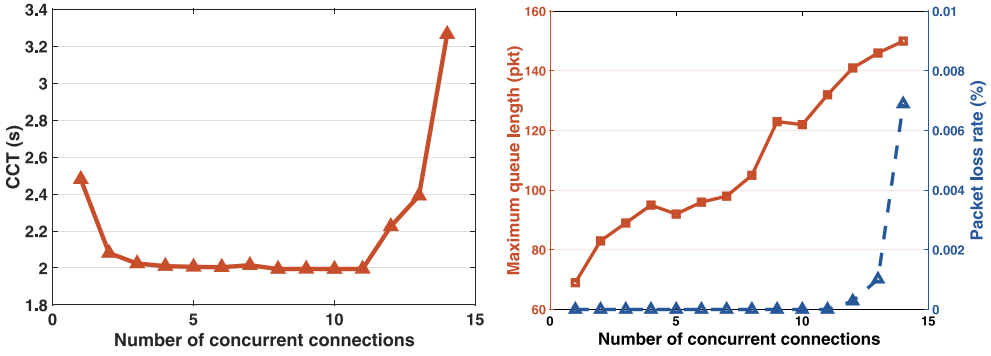
Receiver-driven scheduling can also be sub-optimal when host-egress links are the bandwidth bottleneck. A toy example is shown in Fig. 1(b). The only change is the topology. Coflow 1 has a mapper on host *A* and a reducer on host *B*. Coflow 2 has a mapper on host *A* and a reducer on host *C*. For a sender-driven scheduling algorithm, sender host *A* would prioritize coflow 1 over coflow 2. The average CCT is $\frac{1+3}{2} = 2$ time units. On the contrary, for a receiver-driven scheduling algorithm, each receiver host (*i.e.*, *B* or *C*) observes only one coflow. Thus both coflows 1 and 2 have the highest local priority and will start simultaneously and the average CCT is $\frac{2+3}{2} = 2.5$ time units.



(a) Sender-driven scheduling can be sub-optimal and receiver-driven scheduling can be optimal.

(b) Receiver-driven scheduling can be sub-optimal and sender-driven scheduling can be optimal.

Fig. 1. A motivating example.



(a) The CCT variation

(b) The maximum queue length and packet loss rate variation

Fig. 2. The optimal number of concurrent connections.

2.3. CCT can be prolonged due to too many or too few concurrent connections

To show that the connection number of each host can be neither too large nor too small, we conduct extensive experiments to thoroughly evaluate the CCT and each data point is an average of more than 30 runs. At each run, we generate a coflow with 64 mappers and 64 reducers uniformly distributed on 16 hosts, which is connected by a 1 Gbps switch. As the same setting in Hadoop, the total connection resources at each host is uniformly distributed to each reducer. Each reducer will randomly choose some hosts to connect and pull the output data of mappers. Fig. 2 shows the CCT, maximum queue length and packet loss rate variations with different number of concurrent connections. In Fig. 2(a), we can observe that the CCT can be effectively reduced when the number of connections is 11. The CCT is prolonged if the number of connections is less than or larger than 11. The reason is that too few connections cannot fully utilize the bandwidth resource to optimize the flow transmission, leading to the uplink bandwidth wastage and longer CCT. When the number of concurrent connections becomes too many, the switch buffer can be filled up instantaneously, where large number of packets can be dropped due to serious flash congestion and even reconnection behavior can frequently occur, which will also harm the CCT. Looking more closely into Fig. 2(b), we can observe that the maximum queue length and packet loss rate increases with the number of concurrent connections when this number is larger than 11. For example, when the number of concurrent connections is 13, the CCT, maximum queue length and packet loss rate is about 2.4, 146 and 10^{-4}, respectively, which indicates re-transmission happens.

Hence the optimal number of concurrent connections needs to be determined in advance to affiliate the coflow scheduling. We develop a learning model to automatically identify the optimal number of concurrent connections.

3. Django overview

On a high level, Django is a loosely-coordinated coflow scheduling framework that does the following at each interval: (1) predicts the optimal number of concurrent connections at each host, (2) collects the mapper and the reducer information and generates a global candidate list on the tracker, (3) determines the number of concurrent connections for each reducer on a local scheduler according to a dynamic priority list and (4) re-adjusts the connection states for each reducer on a micro scheduler.

The architecture of Django is shown in Fig. 3. The local scheduler maintains a dynamic priority list for each reducer at this host. When a new reducer arrived, the local scheduler will update the priority list, then reallocate the number of connections to each reducer and notify the new connection information to the micro scheduler. Once a micro scheduler received this notification, it will request a candidate list from the global scheduler, and open new connections or close old connections accordingly to the resulting connection state from the local scheduler. At the same time, the micro scheduler will report its current connection state to the global scheduler, which is used for information synchronization.

Now using the running example as shown in Fig. 1, we illustrate how Django works. The local scheduler first allocates a priority for each reducer on the same host according to its coflow size. As shown in Fig. 1(a), the reducer R_1 is prioritized as its coflow size is less than that of the reducer R_2. Then the reducer R_1 searches the candidate list on the tracker to determine the

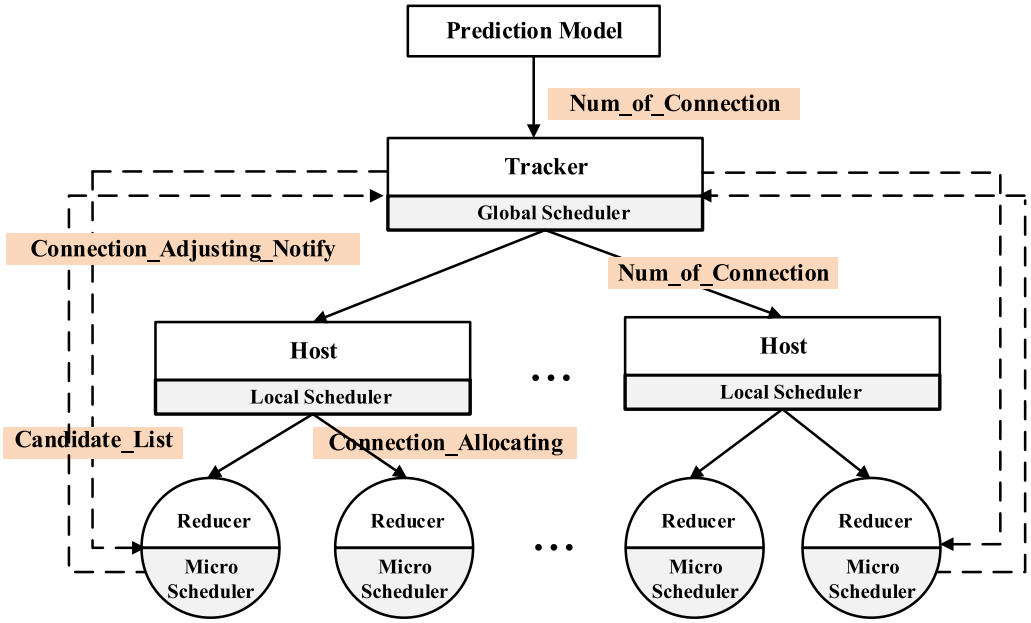


Fig. 3. The architecture of Django.

mapper M_1. Next the connection between the mapper M_1 and the reducer R_1 can be established and the corresponding information is sent to the tracker. Since the link bandwidth at the receiver side is fully consumed by the reducer R_2, the reducer R_2 has to wait until the transmission of R_1 is finished. The average CCT is 2 time units. As shown in Fig. 1(b), the reducers R_1 and R_2 are both prioritized as they are located on different hosts. Meantime, the tracker generates a candidate list according to the remaining coflow size at sender side. Here the remaining size of the mapper M_1 is less than that of the mapper M_2 and thus the connection between the mapper M_1 and the reducer R_1 can be established. Since the link bandwidth at the sender side is fully consumed by the reducer R_1, the reducer R_2 has to wait until the transmission of R_1 is finished. The average CCT is 2 time units as well.

The central challenge in designing Django is how to predict the optimal number of concurrent connections and design a set of distributed algorithms that can provide a reasonable trade off between scalability and performance, which is our focus in the following sections.

4. A scheduling framework

We introduce our scheduling framework with predicted number of concurrent connections in Django. As discussed, we first obtain the optimal number of concurrent connections using a machine learning model. Based on the prediction results, we design a set of distributed algorithms to perform coflow scheduling in a scalable manner.

4.1. Prediction

Before building our prediction model, we first illustrate an important observation. For DCTCP, once the converged queue length exceeds the switch buffer size, the packet loss rate of a group of flows can indicate the switch buffer size. The smaller the buffer size is, the more packets are dropped. Meanwhile, the switch buffer size is a linear function of ECN marking threshold and the number of concurrent connections as shown in Theorem 4.1, which is captured by both theoretical analysis [3] and system validation [6].

Theorem 4.1 ([3]). *There exists a linear function mathematically formulated by the following equation,*

$$q_{max} = k + n \tag{1}$$

where q_{max} is the buffer size, k is the ECN marking threshold and n is the number of concurrent connections.

Considering that the packet loss rate is easy to measure and has a strong correlation with the buffer size, we advocate to use packet loss rate as a key feature to predict the optimal number of concurrent connections. However, the mathematical relationship between the packet loss rate and switch buffer size is hard to capture and seems nonlinear even when all other conditions are fixed. We focus on machine learning method and choose Support Vector Machine (SVM) as our learning model for its simplicity and efficiency. Specifically, we use LibSVM library [5] to build a machine learning system.

Algorithm 1: Training the prediction model

Input: The number of flows ξ; the measured packet loss rate r; the ECN marking threshold k; the set Q of buffer size; the number of iterations t.
Output: The number of concurrent connections n

1 **for** $i = 1$ *to* t **do**
2 Generate an incast traffic with ξ flows
3 Choose a candidate y_i randomly from the set Q of buffer size
4 Measure the packet loss rate x_i and chosen buffer size y_i as a data record.
5 Obtain the data set with t records where each data point x_i is scaled into the $[-1, 1]$ interval
6 Partition the data set into training set and testing set
7 Apply C-SVC model [4] as the multi-class classifiers to establish a prediction function $f(\cdot)$
8 Apply Theorem 4.1 and obtain the number of concurrent connections $n = f(x) - k$
9 **return** n;

The prediction algorithm is shown in Algorithm 1. First, we generate an incast traffic with ξ concurrent small flows for t times randomly (line 2). For the ith run, the buffer size is randomly chosen from the set Q, then we record the packet loss rate

x_i and chosen buffer size y_i as a data record (lines 3–4). Totally, we obtain a data set with t records where each data point x_i is scaled into the $[-1, 1]$ interval (line 5). We partition the data set into training set and testing set and use C-SVC model [4] as multi-class classifiers (lines 6–7). Finally, we apply Theorem 4.1 to obtain the number of concurrent connections (line 8). Note that C-SVC model is composed by $\frac{|Q|\times(|Q|-1)}{2}$ two-class classifiers, where $|Q|$ is the number of classes. For our problem, in training phase, each two-class classifier will try to find a hyperplane to divide certain two classes by solving a convex quadratic programming. In testing phase, for a given input x, all of the two-class classifiers will vote to determine the class number y, which indicates the estimated buffer size.

4.2. Local scheduler

The local scheduler runs as Algorithm 2 on each host, which is responsible for the connection reallocation between the sender and the receiver side according to a local priority list. We now explain the high level working of Algorithm 2. At the beginning, to apply the shortest remaining time first (SRTF) principle, the reducers are sorted according to the remaining size of their coflows and the remaining size of reducers (line 1). Here we say a reducer with highest local priority as a prioritized reducer, others are called non-prioritized reducers. We only schedule non-prioritized reducers if the prioritized reducer keeps the same in two successive schedules (line 3). At the same time, we check the vacant downlink bandwidth and the remaining number of connections (line 4). If the vacant bandwidth exceeds $\alpha \cdot b$ and the remaining number of connections is larger than zero, we allocate one more connection to the first non-prioritized reducer. The allocated number of connections for this reducer cannot exceed a half of the number of free connections (line 5). When these two conditions cannot be established simultaneously (line 4), we stop allocating connection to non-prioritized reducers, preventing from interfering with the prioritized reducer.

When the prioritized reducer changes, *i.e.*, a reducer with the smaller size comes (line 8), we will close certain connections instantly to preserve a portion of $\beta \cdot m$ connections for current prioritized reducer (line 10), which can be regarded as a pre-emption. If the remaining number of connections is larger than or equal to $\beta \cdot m$ (*i.e.*, the free connections are large enough to satisfy the requirements of the new prioritized reducer), the allocated connections for previous prioritized reducer will be taken back until the flow transmission is finished (lines 13–14). For a non-prioritized reducer, the number of its connections is approximately exponentially decreased with its local priority. Note that we never allocate all of the connections to one reducer, since a prioritized coflow in the receiver side may not has a matching priority in the sender side. This mismatch can also harm CCT. To address this problem, the priority between the sender and receiver side is dynamically balanced using Algorithm 4, which will be discussed soon. Finally, we apply Algorithm 3 to adjust the number of connections and compete the connection allocation.

4.3. Micro scheduler

A micro scheduler runs Algorithm 3 on each reducer, which reacts to the connection reallocation from the corresponding local scheduler and adjusts the number of connections for each reducer. When the number of current connections exceeds the allocated number of connections, the micro scheduler will close certain connections with the highest flow transmission rate one by one until the number of current connections satisfies the requirements (lines 1–4). Note that the release of allocated connections usually indicates that a preemption happens, which is used to leave more bandwidth resource to the prioritized reducer as fast as possible. It is necessary to close the connections with the highest flow transmission rate firstly.

Algorithm 2: Allocating connections at each host

Input: The reducer list R on the host; the predicted number of connections m; the total number of current connections n; the bottleneck link capacity b; the parameters $\alpha, \beta, \eta \in [0, 1]$.

1 Sort reducer list R according to the remaining coflow size (the first keyword) and the remaining reducer size (the second keyword) in ascending order;
2 $rem = m - n$;
3 **if** $R[0]$ *keeps the same in recent two successive schedules* **then**
4 **if** *the vacant downlink capacity exceeds* $\alpha \cdot b$ ***and*** $rem > 0$ **then**
5 Allocate one more connection to the first non-prioritized reducer in R whose connection number does not exceed $\eta * rem$;
6 $rem = rem - 1$;
7 **else**
8 Set $R[0]$ as the only prioritized reducer in this host;
9 **if** $rem < \beta \cdot m$ **then**
10 Close the connections from previous prioritized reducer instantly to guarantee that there are $\beta \cdot m$ free connections;
11 $rem = \max\{rem, rem - \beta \cdot m\}$
12 **else**
13 Close the connections once the transmission of previous prioritized reducer finished;
14 Update rem;
15 Allocate at most $\beta \cdot m$ connections to current prioritized reducer;
16 Notify all reducers whose connection number changed (then these reducers will apply Algorithm 3 to adjust the number of connections) ;

Algorithm 3: Adjusting the number of connections for each reducer

Input: The reducer id r; the number of current connections n for reducer r; the number of new connections n^* for reducer r.

1 **while** $n > n^*$ **do**
2 Close the connection with the highest flow transmission rate in reducer r to release more bandwidth for the newly prioritized reducer;
3 Notify the tracker that a connection is closed;
4 $n \leftarrow n - 1$;
5 **while** $n < n^*$ **do**
6 Request the candidate list CL from the tracker;
7 **if** *reducer* r *is prioritized* **then**
8 Connect to the mapper in CL which has the largest vacant uplink capacity;
9 **else**
10 Connect to the mapper in CL which has the largest remaining size to transmit;
11 Notify the tracker that a new connection is created;
12 $n \leftarrow n + 1$;

When the allocated number of connections exceeds the number of current connections, the micro scheduler will request a candidate list from the tracker scheduler to open more connections. The candidate list is a subset of sender hosts for this coflow maintained by the tracker, and we can only choose the sender from this list to connect. The procedure of generating this candidate list will be discussed soon in Algorithm 4. More specifically, if this reducer is prioritized, we will choose the host that has the largest vacant uplink capacity to connect, which could prevent from bandwidth wastage (line 8). Otherwise, we will choose the host that has largest remaining size to transmit, which could balance the non-uniformly distributed mappers (line 10).

4.4. Global scheduler

The global scheduler runs Algorithm 4 on the tracker that can monitor network state of each host and dynamically generate a candidate list once a request from the micro scheduler is received. When a micro scheduler requests for a candidate list, the global scheduler will first query the pending sender hosts for this reducer. Here a sender host is said to be pending if it has not transmitted any packets to the reducer, or the transmission procedure has already been terminated due to the preemption.

Algorithm 4: Generating the candidate list on the tracker

Input: The reducer id r; the maximum size of candidate list m; the balance ratio $\lambda \in [0, 1]$; the vacant uplink capacity u on each host i; the link capacity b; the parameter θ.
Output: The candidate list CL

1 Define array L as the pending sender host list corresponding to the reducer r;
2 $l = |L|$;
3 $CL \leftarrow \emptyset$;
4 **if** *reducer r is prioritized* **then**
5 **foreach** *sender host i* **do**
6 Define s_i as a prioritized mapper whose remaining coflow size is the smallest in its sender host list;
7 **if** *r and s_i belong to the same coflow* **then**
8 $u_i \leftarrow u_i + \lambda \cdot b$;
9 Sort L according to their balanced bandwidth u such that $u_{L[0]} \leq \cdots \leq u_{L[l]}$;
10 Add the first $\min(m, l)$ elements in L to CL;
11 **else**
12 Define bw as the vacant downlink capacity of the host that the reducer r belongs to;
13 **if** $bw > \theta \cdot b$ **then**
14 Sort L according to their vacant uplink capacity u such that $u_{L[0]} \leq \cdots \leq u_{L[l]}$;
15 $p \leftarrow \arg\min_i |bw - u_i|$;
16 Add $L[\max(p - m + 1, 0)], \cdots, L[p]$ to CL;
17 **return** CL;

For a prioritized reducer, we choose the sender host according to its vacant uplink bandwidth, i.e, its balanced vacant uplink bandwidth. We have stressed that a prioritized reducer may not have a matched priority on each sender host, thus we have to take the priority of sender host into consideration to jointly optimize the CCT. To trade off the complexity and performance, we introduce a concept in Algorithm 4 called balance ratio, which is defined as a real number that ranged from 0 to 1. Now we show that how balance ratio affect the coflow scheduling. When we sort the set of sender hosts in the candidate list, a constant will be added to the free uplink bandwidth if the coflow that this reducer belongs to is prioritized in the sender host as well. This constant is defined as the product of balance ratio λ and the link capacity b corresponding to the sender host. Therefore, the priority-matched sender host will be chosen into the candidate list with high probability (lines 4–10).

For non-prioritized reducer, what is essential is bandwidth matching between the sender and receiver side. That is to say, the global scheduler should guarantee that the difference between the downlink bandwidth at reducer host and the uplink bandwidth at the possible sender host cannot be too large, since the flow completion time is determined by the bottleneck bandwidth (lines 12–16). Besides, the maximum size of candidate list (line 16) is usually much smaller than that for prioritized reducer (line 10). Otherwise, either the bandwidth is wasted, or the prioritized reducer is interfered (lines 12–16).

Before analyzing the performance of Django, we introduce the following definition.

Definition 4.1 (*Deadlock*)**.** A deadlock indicates that the network throughput is zero, where all the reducers are waiting for each other and stop transmitting any data.

Theorem 4.2. *Django is deadlock-free.*

Proof. Assume there is a deadlock, thus all reducers cannot receive any data at the time. For an arbitrary prioritized reducer (*e.g.*, reducer R), according to Algorithm 4, it must have been connected to at least one mapper host (*e.g.*, host H). However, only when all of the uplink bandwidth of host H is occupied by other reducers, can network throughput of reducer R be zero, which is against the definition of deadlock. □

5. Implementation

5.1. Framework design

We develop a prototype of our scheduling framework, which is mainly composed of the prediction modules, the tracker modules and worker modules. The tracker maintains the host information corresponding to all the mappers and reducers and dynamically schedules the jobs. At the same time, the workers conduct the data transmission function. As we know, Akka [2] has the master and worker modules and is in line with our framework design. Hence we use Akka for the communications and define two categories of new Akka messages for tracker–worker and worker–worker communication. We use the master modules to act as our tracker and the worker module to act as our hosts.

We now explain the designed messages in our implementation. For the communication between the tracker and the worker, we defined two types of messages `REQ_CANDIDATE` and `REPLY_CHOSEN`. When a reducer is launched, it first sends `REQ_CANDIDATE` message to the tracker for searching candidate hosts. The tracker then uses Algorithm 4 to generate the candidate list and reply to the reducer. The message `REPLY_CHOSEN` is used when reducer tells the tracker which sender-host it will choose using Algorithm 3. For the communication among workers, we define three types of messages `SEND_START`, `SEND_PAUSE` and `SEND_RECOVERY`. The `SEND_START` message is used for requesting data from the mapper. Once a reducer plans to obtain data from the mapper, it will send `SEND_START` message to the corresponding host. The worker periodically runs Algorithm 3 and the reducer on this worker adjusts the number of their active connections. When the number of current connections is larger than the expected number, the reducer sends `SEND_PAUSE` message to make the mapper stop sending data. Otherwise, the reducer sends `SEND_RECOVERY` message to increase the number

of connections. The messages designed in our implementation are shown in Table 1. Note that our system design is compatible with the famous system YARN and can be seamlessly integrated into it.

5.2. Testbed setup

Our testbed consists of 8 servers connected to a 32-port Mellanox switch, where one of them acts as the tracker, the rest act as the normal servers. Each server is a DELL PowerEdge R730, equipped with a 12-core Intel Xeon E5-2650V4 CPU, 128 GB RAM and a Mellanox CX5 dual-port Ethernet NIC. Each server runs Ubuntu Server 14.04. We use DCTCP as the transport layer protocol to ensure low latency and for comparison. We replay production traces from Facebook on our testbed.

Table 1
Messages designed in our implementation.

Message	Parameter	Description
REQ_CANDIDATE	–	Search the candidate hosts
REPLY_CHOSEN	–	Choose the sender host
SEND_START	HOST_ID	Request the data from the mapper
SEND_PAUSE	HOST_ID	Make the mapper stop sending data
SEND_RECOVERY	HOST_ID	Increase the number of connections

Table 2
Testbed results.

CCT	Django	DCTCP
Average	15.7	18.1
95th percentile	10.9	30.1
99th percentile	133.6	1037.9

6. Experimental evaluation

6.1. Methodology

We evaluate our scheduling algorithm with both small-scale testbed and large-scale simulation. The trace used in our evaluation is collected from Facebook log [11] that contains 526 coflows and over 30 TB transmission data, which has been widely accepted as a benchmark to evaluate coflow scheduling systems [9,11,19,21,22]. We use both the average CCT and tail CCT (*e.g.*, the 95th/99th percentile) as metrics by default, and each data point is averaged from multiple runs.

We compare the following two algorithms with our framework. **(1) DCTCP** is a widely-used congestion control protocol in data centers. We set the ECN marking threshold to 30 kB (about 20 MTU) for 1 Gbps link, and 100 kB (about 65 MTU) for 10 Gbps link, as recommended in [3]. **(2) Aalo+** is a modified version of state-of-the-art coflow scheduling approach Aalo [9]. Here we assume all the size information are known a prior to perform an apples-to-apples comparison.

Unless stated otherwise, we configure the bottleneck link capacity b to be 1 Gbps, and α, β, λ, m to be 0.1, 0.7, 0.2 and 0.5, respectively in Algorithms 2 and 4. To make the trace match our topology, the size and (or) arrival time of coflows is scaled down accordingly.

By using the appropriate connection number and the scheduling algorithms to manage connection, Django improves the average CCT and tail CCT (95th and 99th percentiles) significantly both in simulation and testbed. Specifically, Django reduces the average CCT to 85% and 65% compared with Aalo+ and DCTCP. At the same time, it reduces the 95th percentile CCT to 60% and 23% compared with Aalo+ and DCTCP. And we investigate the CCT performance of Django under various settings. Django can perform good under small buffer size due to the appropriate connection number setting. Besides, the results show that Django can scale very well nevertheless the machine number scales, the network load scales or the number of reducers per machine increases.

6.2. Queue buffer estimation

We generate an incast traffic with about 200 concurrent small flows for 400 times randomly. We enumerate the number of classes from 2 to 5 and set the number of iterations to be 400. There are around 300 records for training and 100 records for testing. To investigate how many records are needed for training, we gradually increase the size of training set from 50 to 300. The results are shown in Fig. 4, and we make several observations.

First, our model performs well when the number of classes is less than or equal to 3. When the number of classes is equal to 4 and 5, the estimation accuracy is 87.7% and 73.6% respectively, which is a great improvement compared with naive lower bound. Second, one hundred records for the training are large enough, which suffer only 3% accuracy loss. Third, C-SVC model runs quite fast in our scenarios, which costs less than 20 ms in total in our 5-classification mission, and each estimation costs only 75 μs on average.

One may wonder whether our system is robust to such non-negligible estimation error. Recalling the CCT variation in Fig. 2(a), a near flat interval can be observed, which indicates that our system can tolerate estimation error in a relevant large range.

6.3. Testbed results

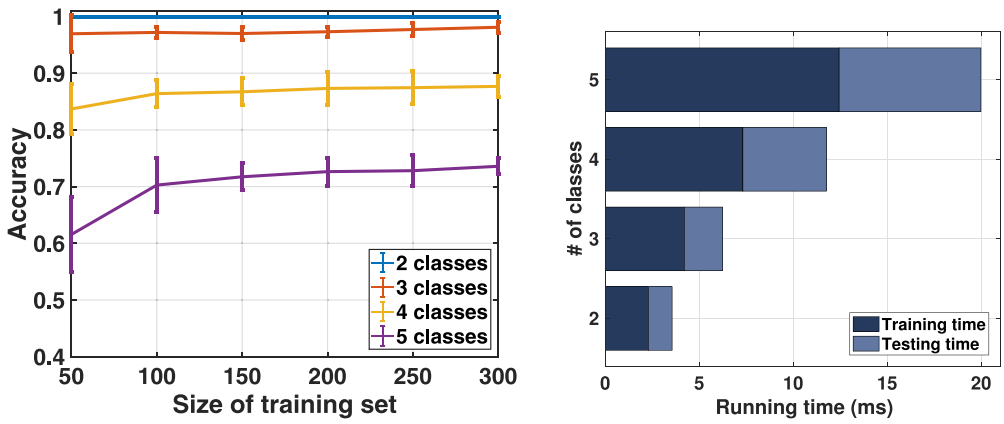
We perform testbed experiments to evaluate our algorithm in real servers, and compare our algorithm with DCTCP. Results are shown in Table 2. Each data is an average of at least 10 experiments. We can see that Django decreases both the average and tail CCT and is around 1.3x faster than DCTCP. The reason is that DCTCP guarantees the max–min fairness among flows instead of scheduling coflows, which will cause bandwidth competition and congestion in egress port, thus resulting in prolonged CCT. We observe the packet loss rate is almost zero when using Django, and this is benefit from the appropriate connection number and the scheduling algorithm on our master–worker system.

6.4. Simulation results

We also conduct extensive simulations to thoroughly evaluate our algorithm at scale.

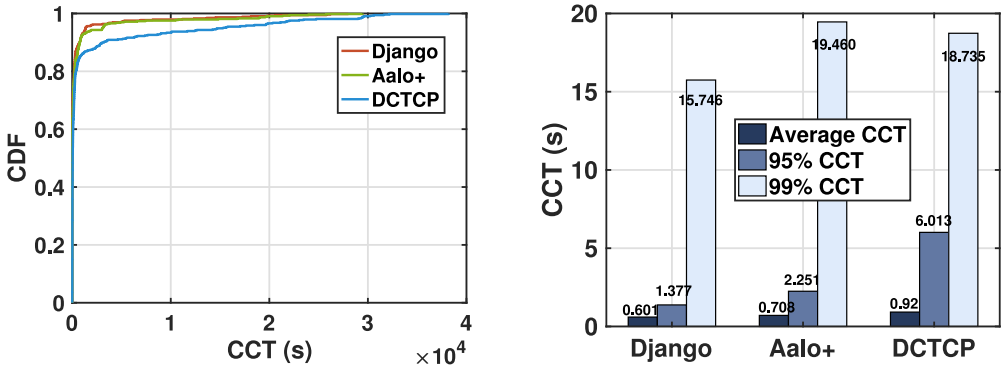
Experiment results: We first investigate the CCT variations for Django, Aalo+ and DCTCP. Fig. 5(a) shows the CDF of CCT for different schemes. Looking closely into this figure, we can observe that Django performs well and in general decreases the tail CCT by 40% and 77% compared to Aalo+ and DCTCP. More detailed information – the average CCT, 95th percentile CCT and 99th percentile CCT – can be shown in Fig. 5(b). This demonstrates that the joint optimization on the number of concurrent connections and coflow scheduling in Django can lead to perfect performance on reducing both the average and tail CCT.

As discussed Django can improve the performance by predicting the number of concurrent connections. We measure the 95th percentile CCT and average CCT under different buffer size setting as shown in Figs. 6(a) and 6(b), respectively. We can see that Django can always perform perfect under all buffer size settings. The reason is that the prediction model in Django can produce an optimal number of concurrent connections and the hosts can adjust this number accordingly when the buffer size changes. However, Aalo+ and DCTCP are more rigid to the buffer size: they perform poorly especially when the buffer size is less than 75



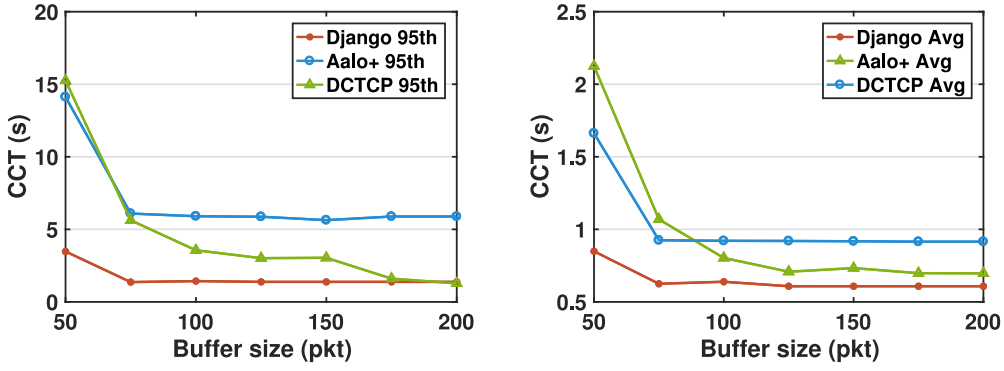
(a) Accuracy (b) Running time

Fig. 4. The prediction results.



(a) CDF of CCT (b) Average and tail CCT

Fig. 5. Basic performance.



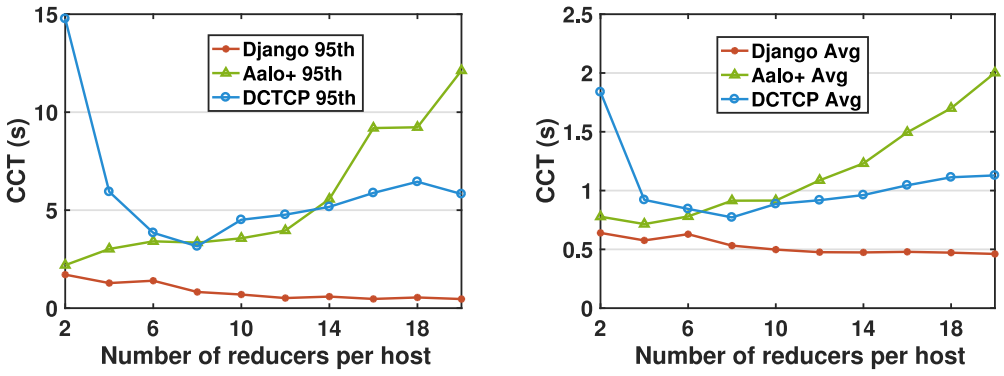
(a) 95-th percentile (b) Average

Fig. 6. The CCT variation with different buffer size.

packets (around 110 kB). Specifically, as shown in Fig. 6(a), we can see that Django performs 2.3x and 1.9x faster compared to Aalo+ and DCTCP when the buffer size is 50 packets (around 73 kB), and performs 1.2x and 1.5x faster compared to them when the buffer size is larger than 100 packets (around 146 kB).

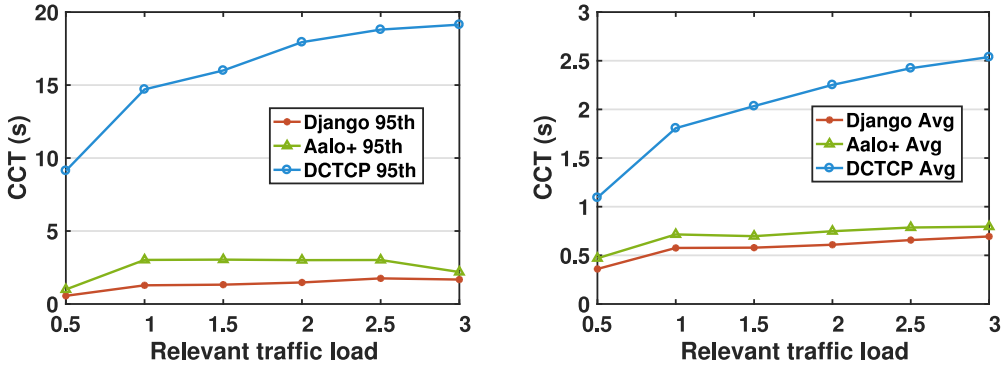
Fig. 7 shows the CCT variation with different numbers of reducers per host. The number of reducers per host represents the maximum concurrent jobs that a host can accommodate. We can observe that Django performs very stable even the number of reducers becomes large, which indicates that Django can fully take advantage of more resources in the host to optimize the reducers scheduling. On the contrary, Aalo+ and DCTCP perform poorly especially when the number of reducers increases. Specifically, when the number of reducers exceeds 10, the CCT of Aalo+ and DCTCP increase significantly, which will degrade the application performance. The reason is that Aalo+ does not respect the link bandwidth in the receiver side, always making the prioritized mapper transmit the data firstly, which is inevitably not optimal.

To evaluate the performance with different workloads, we can scale up or scale down the coflow arrival intervals to increase



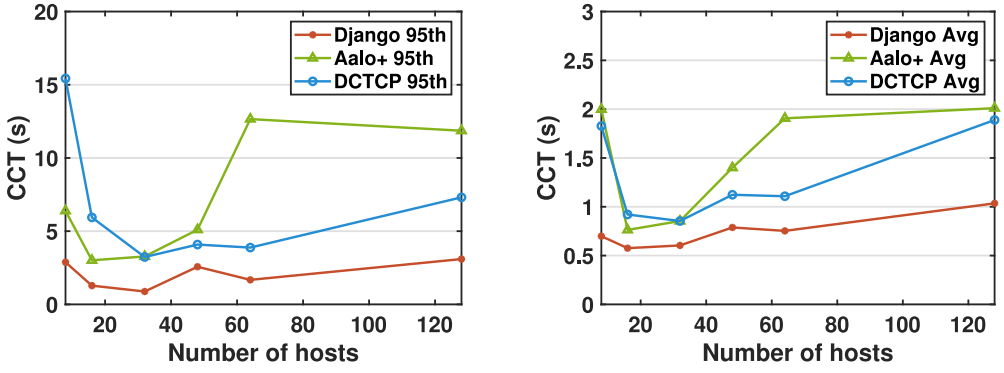
(a) 95-th percentile (b) Average

Fig. 7. The CCT variation with different numbers of reducers.



(a) 95-th percentile (b) Average

Fig. 8. The CCT variation with different workloads.



(a) 95-th percentile (b) Average

Fig. 9. The CCT variation with different numbers of hosts.

or decrease the workloads in the network. Here we use the load ratio in our simulations to capture different coflow arrival intervals. The standard load ratio is 1.0 when the coflow arrival ratio is 1 s. We change the load ratio ranging from 0.5 to 3 at the increment of 0.5. Fig. 8 shows that the CCT variation with the different workloads. When the load ratio increases, the CCT of Django increases slower compared to Aalo+ and DCTCP. The reason is that the reducer in Django takes the bandwidth at the sender side into consideration when the connections between them will be established. For 95th percentile CCT, we can see that Django performs 1.3x and 2.2x faster compared to Aalo+ and DCTCP, respectively when the load ratio is 3. For average CCT, Django performs 1.2x and 1.38x faster compared to Aalo+ and DCTCP.

We measure the performance of Django at large-scale topology to investigate its scalability. As shown in Fig. 9, we increase the number of hosts ranging from 8 to 128 and observe the CCT variations for the different schemes. The CCT for Django changes slightly when the number of host becomes large, which indicates that it works well for both small-scale and large-scale networks. However, for Aalo+ and DCTCP, the CCT decreases firstly when the number of hosts becomes larger, and when this number exceeds

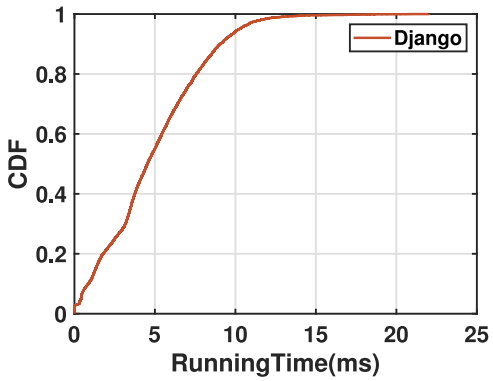


Fig. 10. The running time of Django.

32, the CCT increases significantly. The performance of Aalo+ and DCTCP cannot scale well when the number of hosts is large. Here how to optimally scheduling the connections is a key factor and leads to significant difference. We can see that the 95th percentile CCT of Django is 2x and 2.1x faster than that of Aalo+ and DCTCP. The average CCT of Django is 1.4x faster than Aalo+. When the number of hosts increases, the gap between them is even more larger. And the average CCT of Django is 1.75x faster than that of DCTCP. This indicates that our scheduling algorithms can be applied to the large-scale networks.

As shown in Fig. 10, we investigate the running time of algorithms in Django. Looking closely into this figure, we can observe that 94.1% running time is less than 10 ms and the average running time is 4.8 ms.

As shown in Fig. 11, we did extensive experiments to show the influence on the performance when using different parameters in our algorithm. A smaller α value tends to allocate the connections to the non-prioritized reducers, which can improve the bandwidth utilization and CCT as shown in Fig. 11(a). Specifically, we can select the non-prioritized reducer if the number of its connections does not exceed the product of η and the number of remaining connections. Regarding η, Fig. 11(c) shows that our algorithm performs better when its value is set to be around 0.6. Furthermore, β is the maximum ratio between the number of connections for the prioritized reducer and the number of total connections. As shown in Fig. 11(b), the CCT can be largely improved when β lies in the interval [0.7 0.9]. It means that the number of connections for the prioritized and non-prioritized reducers should be balanced by adjusting the value of β to optimize the performance.

Insights: These experiment results make two key conclusions. First, the number of concurrent connections significantly influences the CCT. Hence, it is necessary to choose an optimal number of connections to jointly optimize the coflow scheduling. Second, we reduce both the average and tail CCT using Django since we allocate the number of connections taking both bandwidth and remaining size of coflow into consideration. And the results above also show that Django can perform well even in large-scale topology including large number of hosts.

7. Related work

Sincronia [1] uses the sizes of individual flows to order coflows and determine the network bottlenecks, which achieves the average coflow completion time (CCT) within 4x of the optimal and can be implemented on top of any transport layer supporting the priority-based scheduling. Utopia [26] considers the tradeoff between minimizing the average coflow completion time and providing optimal service isolation guarantee between contending coflows. Instead of enforcing strict guarantee to minimize the completion time of "small" coflows, Utopia advocates long-term isolation guarantee: as long as a coflow completes no later than an isolation-optimal scheduler, its isolation is guaranteed in a long run. The bandwidth allocation of subflows in one coflow with higher priority helps determine the bandwidth allocation of subflows in another coflow with low priority. In [27], the flaws of coflow have also been discussed recently. It addresses the problem of scheduling weighted coflows, where weights are used to indicate the importance of different coflows. Swallow [29] introduces a coflow compression mechanism to minimize completion time in data-intensive applications. A heuristic algorithm for solving NP-hardness problem called Fastest-Volume-Disposal-First (FVDF) is proposed, which minimizes coflow completion time (CCT) while ensuring resource conservation and starvation freedom. Furthermore, a randomized 2-approximation algorithm was proposed in [7]. It studies a different model of coflow scheduling over general graphs.

OMCoflow [19] proposes a theoretical performance assurance algorithm which considers both routing and scheduling for multi-coflows. Extensive simulations using a real-world data trace show that OMCoflow performs well. Stream [24] is a decentralized and easily implementable solution for coflow scheduling. It tries to be readily deployed to data centers by using a decentralized protocol without modifying hardware.

8. Conclusions

In this paper, we developed a bilateral coflow scheduling framework that can automatically identify the optimal number of concurrent connections and combine the advantages of both sender-driven and receiver-driven scheduling approaches. Experimental and simulation results show that our algorithms can reduce the average CCT and tail CCT, respectively.

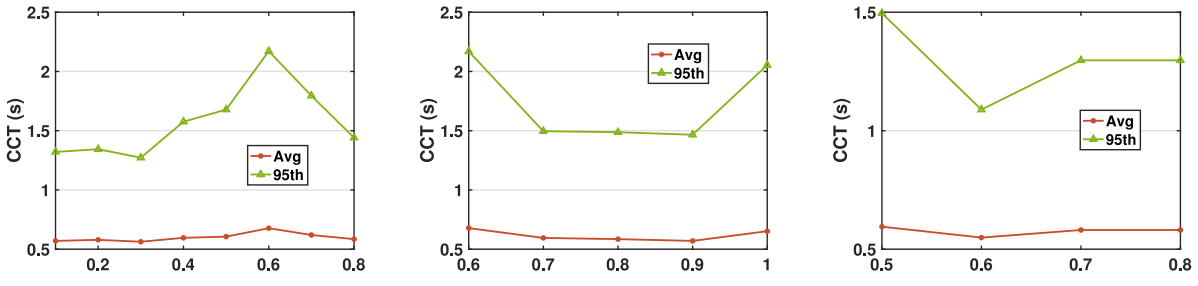


Fig. 11. The parameter selection for α, β and η.

CRediT authorship contribution statement

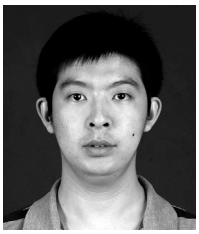
Jiaqi Zheng: Conceptualization, Funding acquisition, Formal analysis, Interpretation of data, Writing - review & editing, Accuracy or integrity. **Liulan Qin:** Conceptualization, Funding acquisition, Formal analysis, Interpretation of data, Writing - review & editing, Accuracy or integrity. **Kexin Liu:** Conceptualization, Funding acquisition, Formal analysis, Interpretation of data, Writing - review & editing, Accuracy or integrity. **Bingchuan Tian:** Conceptualization, Funding acquisition, Formal analysis, Interpretation of data, Writing - review & editing, Accuracy or integrity. **Chen Tian:** Conceptualization, Funding acquisition, Formal analysis, Interpretation of data, Writing - review & editing, Accuracy or integrity. **Bo Li:** Conceptualization, Funding acquisition, Formal analysis, Interpretation of data, Writing - review & editing, Accuracy or integrity. **Guihai Chen:** Conceptualization, Funding acquisition, Formal analysis, Interpretation of data, Writing - review & editing, Accuracy or integrity.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

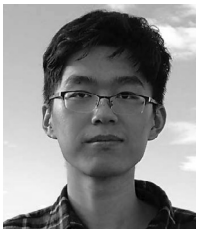
[1] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, A. Vahdat, Sincronia: Near-optimal network design for coflows, in: SIGCOMM, 2018, pp. 16–29.
[2] Akka., http://akka.io.
[3] M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Data center tcp (dctcp), in: SIGCOMM, 2011, pp. 63–74.
[4] B.E. Boser, I.M. Guyon, V.N. Vapnik, A training algorithm for optimal margin classifiers, in: COLT, 1992, pp. 144–152.
[5] C.-C. Chang, C.-J. Lin, Libsvm: a library for support vector machines, ACM Trans. Intell. Syst. Technol. 2 (3) (2011) 1–27.
[6] I. Cho, K. Jang, D. Han, Credit-scheduled delay-bounded congestion control for datacenters, in: SIGCOMM, 2017, pp. 239–252.
[7] M. Chowdhury, S. Khuller, M. Purohit, S. Yang, J. You, Near optimal coflow scheduling in networks, in: SPAA, 2019, pp. 123–134.
[8] M. Chowdhury, I. Stoica, Coflow: a networking abstraction for cluster applications, in: HotNets, 2012, pp. 31–36.
[9] M. Chowdhury, I. Stoica, Efficient coflow scheduling without prior knowledge, in: SIGCOMM, 2015, pp. 393–406.
[10] M. Chowdhury, M. Zaharia, J. Ma, M.I. Jordan, I. Stoica, Managing data transfers in computer clusters with orchestra, in: SIGCOMM, 2011, pp. 98–109.
[11] M. Chowdhury, Y. Zhong, I. Stoica, Efficient coflow scheduling with varys, in: SIGCOMM, 2014, pp. 443–454.
[12] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[13] F.R. Dogar, T. Karagiannis, H. Ballani, A. Rowstron, Decentralized task-aware scheduling for data center networks, in: SIGCOMM, 2014, pp. 431–442.
[14] K.-B. Duan, S.S. Keerthi, Which is the best multiclass svm method? An empirical study, in: International Workshop on Multiple Classifier Systems, 2005, pp. 278–285.
[15] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, S. Sengupta, Vl2: a scalable and flexible data center network, in: SIGCOMM, 2009, pp. 51–62.
[16] C.-W. Hsu, C.-J. Lin, A comparison of methods for multiclass support vector machines, IEEE Trans. Neural Netw. 13 (2) (2002) 415–425.
[17] G. Judd, Attaining the promise and avoiding the pitfalls of tcp in the datacenter, in: NSDI, 2015, pp. 145–157.
[18] S. Khuller, M. Purohit, Brief announcement: Improved approximation algorithms for scheduling co-flows, in: SPAA, 2016, pp. 239–240.
[19] Y. Li, S.H.-C. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou, F. Lau, Efficient online coflow routing and scheduling, in: MobiHoc, 2016, pp. 161–170.
[20] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica, Sparrow: distributed, low latency scheduling, in: SOSP, 2013, pp. 69–84.
[21] Z. Qiu, C. Stein, Y. Zhong, Minimizing the total weighted completion time of coflows in datacenter networks, in: SPAA, 2015, pp. 294–303.
[22] M. Shafiee, J. Ghaderi, Brief announcement: A new improved bound for coflow scheduling, in: SPAA, 2017, pp. 91–93.
[23] A. Shinnar, D. Cunningham, V. Saraswat, B. Herta, M3r: increased performance for in-memory hadoop jobs, VLDB Endow. 5 (12) (2012) 1736–1747.
[24] H. Susanto, H. Jin, K. Chen, Stream: Decentralized opportunistic inter-coflow scheduling for datacenter networks, in: ICNP, 2016, pp. 1–10.
[25] B. Tian, C. Tian, H. Dai, B. Wang, Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time, in: INFOCOM, 2018, pp. 1–9.
[26] L. Wang, W. Wang, B. Li, Utopia: Near-optimal coflow scheduling with isolation guarantee, in: INFOCOM, 2018, pp. 891–899.
[27] Z. Wang, H. Zhang, X. Shi, X. Yin, Y. Li, H. Geng, Q. Wu, J. Liu, Efficient scheduling of weighted coflows in data centers, IEEE Trans. Parallel Distrib. Syst. 30 (9) (2019) 2003–2017.
[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: NSDI, 2012, pp. 15–28.
[29] Q. Zhou, K. Wang, P. Li, D. Zeng, S. Guo, B. Ye, M. Guo, Fast coflow scheduling via traffic compression and stage pipelining in datacenter networks, IEEE Trans. Comput. 68 (12) (2019) 1755–1771.



Jiaqi Zheng is currently a Research Assistant Professor from Department of Computer Science and Technology, Nanjing University, China. His research area is computer networking, particularly data center networks, SDN/NFV, machine learning system and online optimization. He was a Research Assistant at the City University of Hong Kong in 2015 and collaborated with Huawei Noah's Ark Lab. He visited CIS center at Temple University in 2016. He received the Best Paper Award from IEEE ICNP 2015, Doctorial Dissertation Award from ACM SIGCOMM China 2018 and the First Prize of Jiangsu Science and Technology Award in 2018. He is a member of ACM and IEEE.

Liulan Qin received the B.S. degree from the Department of Computer Science and Engineering at the Nanjing University of Science and Technology, China, in 2018. She is a 2nd-year M.S. student in the Department of Computer Science and Technology at Nanjing University, China. Her research interests include datacenter networks.

Kexin Liu received the B.S. degree from the Department of Software Engineering at Sun Yat-sen University, China, in 2017. She is working toward the Ph.D. degree in the Department of Computer Science and Technology at Nanjing University, China. Her research interests include datacenter networks and network architecture.



Bingchuan Tian received the B.S. degree from the Department of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics, China, in 2016. Now he is a 4th-year Ph.D. student in Department of Computer Science and Technology in Nanjing University, China. His research interests include intent-based networking, congestion control, and network scheduling.

Chen Tian is an associate professor at State Key Laboratory for Novel Software Technology, Nanjing University, China. He was previously an associate professor at School of Electronics Information and Communications, Huazhong University of Science and Technology, China. Dr. Tian received the B.S. (2000), M.S. (2003) and Ph.D. (2008) degrees from Department of Electronics and Information Engineering at Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research

interests include data center networks, network function virtualization, distributed systems, Internet streaming and urban computing.

Bo Li received the B.S. degree from the Department of Computer Science and Engineering at the Nanjing University of Science and Technology, China, in 2016. He is a 2nd year M.S. student in Nanjing University, China. His research interests include distributed networks and systems.

Guihai Chen is a distinguished professor of Nanjing University. He earned B.S. degree in computer software from Nanjing University in 1984, M.E. degree in computer applications from Southeast University in 1987, and Ph.D. degree in Computer Science from the University of Hong Kong in 1997. He had been invited as a visiting professor by Kyushu Institute of Technology in Japan, University of Queensland in Australia and Wayne State University in USA. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture and data engineering. He has published more than 350 peer-reviewed papers, and more than 200 of them are in well-archived international journals such as IEEE TPDS, IEEE TC, IEEE TKDE, ACM/IEEE TON and ACM TOSN, and also in well-known conference proceedings such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNext and AAAI. He has won 9 paper awards including ICNP 2015 best paper award and DASFAA 2017 best paper award.