Multi-Attributes-Based Coflow Scheduling Without Prior Knowledge

Shuo Wang, Jiao Zhang, Tao Huang, Tian Pan, Jiang Liu, and Yunjie Liu

***Abstract*— In data centers, the coflow abstraction is proposed to better express the requirements and communication semantics of a group of parallel flows generated by the jobs of cluster computing frameworks. Knowing the coflow-level information, such as coflow size, previous coflow scheduling proposals improve the performance over flow-level scheduling schemes. Recently, since some information of coflow is difficult to obtain in cloud environments, designing coflow scheduling mechanisms with partial or even without any information attracts much attention. However, existing information-agnostic mechanisms are generally built on the least attained service heuristic algorithm that schedules coflows only according to the sent bytes of different coflows, and they all ignore other useful coflow-level information like width, length, and communication patterns. In this paper, we investigate that the coflow completion time could be further decreased by jointly leveraging multiple coflow-level attributes. Based on this investigation, we present a Multiple-attributes-based Coflow Scheduling (MCS) mechanism to reduce the coflow completion time. In MCS, at the start of a coflow, a *shortest and narrowest coflow first* algorithm is designed to assign the initial priority based on the coflow width. During the transmission of coflows, based on the sent bytes of coflows, we proposed a *double-threshold* scheme to adjust the priorities of different classes of coflows according to different thresholds. Accordingly, the optimal thresholds are analyzed by using the M/M/1 queuing model. Testbed evaluations and simulations with production workloads show that MCS outperforms the previous information-agnostic scheduler Aalo, and reduces the completion time of small coflows.**

***Index Terms*— Coflow, datacenter networks, scheduling.**

I. Introduction

IN DATA centers, data-parallel cluster computation frameworks (e.g., MapReduce [1], Dryad [2], and Spark [3]) are widely used to analyze big data. In these frameworks, data-parallel applications usually have multiple successive computation stages, and a succeeding computation stage cannot start until all the intermediate data generated by the previous stage are in place [4]. Generally, the intermediate data transfer involves a collection of flows with the same objective between two groups of machines. *To better express the requirements and communication semantics of the group of flows, the coflow abstraction is proposed [5].* For example, a typical coflow is the shuffle between the mappers and reducers in MapReduce. Furthermore, by analyzing the data trace from production data centers, recent work has shown that the intermediate data transmission accounts for more than 50% of applications' completion times [6], and scheduling flows at coflow-level can significantly reduce the completion times of the communication stages [4], [7]–[10]. Thus, managing flows at coflow-level to reduce the Coflow Completion Time (CCT) is commonly known as the coflow scheduling problem.

A large body of coflow optimizations [4], [7]–[14] have emerged to improve the average CCT of all coflows. Most of these proposals assume the coflows' information such as coflow size, flow size within coflows, could be known priori [4], [7]–[9], [11], [12]. However, identifying coflows and knowing all coflows' characteristics are infeasible in cloud data centers. On the one hand, most of the proposed information-aware coflow schedulers have their own coflow API, and it is hard to correctly modify existing data-parallel applications to use the API, making it difficult to identify coflows prior. On the other hand, data is usually transferred as soon as it is generated in data-parallel applications [10], making it hard to know a coflow's size and flow sizes prior.

To overcome the drawbacks of information-aware coflow scheduling mechanisms, recently, information-agnostic coflow scheduling mechanisms have been proposed to schedule coflows with partial or even without the prior knowledge of coflow characteristics [10], [13], [14]. At the heart of these mechanisms is a Least Attained Service (LAS)-based heuristic rate allocation. More specifically, they place coflows into priority queues, and a coflow is gradually demoted from the highest priority queue to lower priority queues when its sent bytes exceeds predefined demotion thresholds. However, through our observation, we find that only relying on sent bytes to allocate rate have some limitations. On the one hand, using sent bytes cannot separate small and large coflows when they arrive in batch, which causes Head-of-line (HOL) blocking and thus increases the completion time of small coflows. On the other hand, since the size of coflow ranges from tens of megabytes to thousands of gigabytes, using one type of demotion threshold for all coflows is too crude to adjust priorities, which may increase the completion time of coflows.

The above shortcomings cause us to conclude that the design space for information-agnostic scheduling should not be limited to the sent-bytes-based rate allocation. Actually, we note that coflows have many other useful attributes, such

Manuscript received June 15, 2017; revised December 6, 2017 and May 31, 2018; accepted July 1, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. Wierman. Date of publication August 7, 2018; date of current version August 16, 2018. This work was supported in part by the National Natural Science Foundation of China under Grant 61502049 and in part by the Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory Program under Grant XX17641X011-09. *(Corresponding author: Jiao Zhang.)*

The authors are with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China, and also with the Beijing Advanced Innovation Center for Future Internet Technology, Beijing 100124, China, and with the Beijing Laboratory of Advanced Information Networks, Beijing 100876, China (e-mail: shuowang@bupt.edu.cn; jiaozhang@bupt.edu.cn; htao@bupt.edu.cn; pan@bupt.edu.cn; liujiang@bupt.edu.cn; liuyj@bupt.edu.cn).

Digital Object Identifier 10.1109/TNET.2018.2858801

as coflow width, coflow length and communication patterns. Unfortunately, these useful attributes are ignored by existing information-agnostic coflow scheduling algorithms, and whether jointly leveraging these attributes can further reduce the completion times of coflows remains unexplored.

In this paper, we first investigate that jointly leveraging multiple coflow-level attributes has the potential to largely decrease the coflow completion time. Then, based on this investigation, we propose MCS, a simple yet effective solution that leverages multiple coflow attributes. First, MCS presents *Shortest and Narrowest Coflow First* (SNCF) to classify newly-arrived coflows based on their width and assign different classes of coflows different initial priorities. This algorithm effectively avoids the HOL blocking problem and reduces the completion time of small coflows. Second, to mimic *Least Attained Service (LAS)* heuristic algorithm [10], [15], the priorities of coflows should be dynamically adjusted during their transmission. During the transmission of coflows, we present a *double-threshold* scheme to adjust the priorities of coflows based on their sent bytes. In the double-threshold scheme, to avoid crudely adjusting priorities, it assigns each type of coflows a series of demotion thresholds and adjusts their priorities accordingly. Furthermore, we model the system through an M/M/1 queue and formulate the problem to drive the optimal demotion thresholds as an optimization problem.

We have implemented MCS and built a small-scale testbed with 9 hosts to evaluate its performance with realistic Facebook workloads [16]. Furthermore, we also implement MCS in a trace-driven simulator [17] to perform large-scale simulations. The testbed experiments show that the performance gap between MCS and Varys is within 20% for $short\&narrow$ coflows. Our simulation results show that compared to the state-of-the-art information-agnostic scheme Aalo [10], MCS reduces the CCT by up to 62% for $short\&narrow$ coflows, and achieves up to 17% lower CCT for $short\&wide$ coflows.

The main contributions of MCS are:

- We investigate that the coflow completion time could be further reduced by leveraging multiple coflow attributes.
- We propose MCS, a simple, effective, information-agnostic coflow scheduler to optimize the average CCT in data center networks.

The rest of paper is organized as follows. Section II briefly overviews the background of our work. Section III shows the motivation. Section IV to Section VI show the architecture of MCS and design the scheduling algorithm. Section VII and Section VIII extensively evaluate our algorithm through testbed experiments and simulations. Finally, the paper is concluded in IX.

II. Background and Related Work

In this section, we first introduce the main attributes of coflows and then summary related work about coflow scheduling.

A. Attributes of Coflows

Unlike the traditional flow abstraction, a coflow consists of multiple parallel flows that have independent input and output. Thus, it has several attributes, and we will define some important coflow attributes. For example, as shown in Figure 1, a coflow has four flows. The number in the figure shows the source or the destination of a flow. The first flow is from host 1 to host 3, the second flow is from host 7 to host 5, the third flow is from host 7 to host 9, and the fourth flow is from host 9 to host 1. We assume for a Coflow k denoted as $C^{(k)}$, it has a set of flows $\{d_{i,j}^{(k)}\}$, where $d_{i,j}^{(k)}$ denotes a flow from source i sends $d_{i,j}^{(k)}$ size data to destination j. Then, the $size$, $length$ and $width$ of a coflow k is defined as $d^{(k)} = \sum_{i,j} d_{i,j}^{(k)}$, $l^{(k)} = \max\{d_{i,j}^{(k)}\}$, and $w^{(k)} = |\{d_{i,j}^{(k)}\}|$, respectively. We can see that

- the $size$ of a coflow is the sum of all its flows' size.
- the $length$ of a coflow is the size of its largest flow.
- the $width$ of a coflow is the number of parallel flows.

Because we cannot know the size and length of a coflow priori in the information-agnostic scenario, we use sent bytes and the maximal sent bytes of a flow within a coflow as the corresponding attributes for a coflow. For simplicity, we use $\widetilde{d_{i,j}^{(k)}}$, $\widetilde{d^{(k)}}$ and $\widetilde{l^{(k)}} = \max\{\widetilde{d_{i,j}^{(k)}}\}$ to represent the sent bytes of a flow, the sent bytes of a coflow and the length of a coflow in the information-agnostic scenario. Furthermore, the coflow width can be estimated through clustering [13] or modifying API, and we will show this in Section 4.

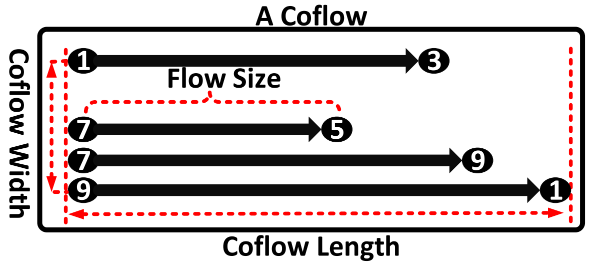


Fig. 1. The attributes of coflows.

B. Related Work

There is a large number of related work about coflow scheduling in data center networks. We summarize them and classify existing coflow scheduling mechanisms into two classes based on the availability of coflow attributes.

Information-Aware Coflow Scheduler: Orchestra [6] and Varys [4] are both centralized coflow schedulers using heuristic algorithms to optimize the average CCT. Orchestra aims at optimizing $shuffle$ and $broadcast$ communication patterns in MapReduce. Varys further improves Orchestra to handle coflows with various structures. Varys proposes the Smallest-Effective-Bottleneck-First (SEBF) heuristic that gives coflows with minimal CCT high priority and schedules these coflows first. Without considering the network topology, Varys is a very good mechanism that can achieve near optimal performance.

Since centralized mechanisms, such as Varys and Orchestra, have a large overhead to handle tiny coflows whose size are smaller megabytes, OPTAS [9] and D-CAS [18] propose distributed scheduling algorithms to improve the performance. OPTAS identifies tiny coflows by observing system calls and schedules tiny coflows according to their start time based on the FIFO manner. D-CAS gives each coflow a priority and schedules coflows based on their priorities. Generally, tiny coflows have the highest priority, and the priorities of other coflows are adjusted according to their waiting time.

Because the above mechanisms all use heuristic algorithms that are suboptimal, RAPIER [7], Chen *et al.* [8], and Qiu *et al.* [19] use optimization theory to solve the coflow scheduling problem. RAPIER formulates the coflow scheduling algorithm as an optimization problem with bandwidth constraints. By approximating the solution of the optimization problem, RAPIER can get the near optimal scheduling policy. Besides using optimization theory, another important contribution of RAPIER is that it is the first mechanism considering the routing of coflows. Instead of minimizing average CCT, Qiu *et al.* [19] design a scheme to minimize the weighted CCT, and Chen *et al.* [8] try to minimize the utility of coflows.

Information-Agnostic Coflow Scheduler: Aalo [10] is the first information-agnostic coflow scheduler and proposes Discretized Coflow-Aware Least-Attained Service (D-CLAS) to divide coflows into multiple priority queues. More specifically, a newly arrived coflow has the highest priority Q_1. Then, a coflow within higher-priority Q_n is demoted to lower-priority Q_{n+1} when its sent bytes $d^{(k)}$ crosses the predefined threshold θ_n. Finally, coflows within each queue are scheduled in the First-In-First-Out (FIFO) order, and flows within each coflow use max-min fairness.

Because Aalo needs to modify data-parallel applications to identify coflows, CODA [13] attempts to automatically identify coflows without any application modifications and proposes error-tolerant coflow scheduling algorithm. In addition, Gao *et al.* [14] theoretically analyze how to choose the thresholds of queues and propose the Down-hill searching (DHS) algorithm.

III. MOTIVATION

In this section, we present two examples to show the main motivations of our work. The first example shows why the performance of short coflows can be further improved. The second example shows why one type of demotion thresholds for all types of coflows is not enough. Through the two motivation examples, we show the key considerations of MCS.

A. Impacts of HOL Blocking

Because using sent bytes cannot differentiate newly-arrived small coflows and large coflows, existing information-agnostic coflow schedulers, such as Aalo [10], CODA [13], simply assign all the newly-arrived coflows with the highest priority. However, this policy may cause HOL blocking in the presence of large coflows. To illustrate this, we consider a simple scenario in Figure 2, where a long and wide coflow (C_1) and a short and narrow coflow (C_2) sharing the same bottleneck link arrive at the same time.

With the Highest Priority: In Figure 2a, we assume that the scheduler assigns C_1 and C_2 with the highest priority and places them in the highest-priority queue Q_1 when they arrive. Then, we can see that C_2 is blocked by the larger coflow C_1 until the sent bytes of C_1 exceed the threshold of Q_1.

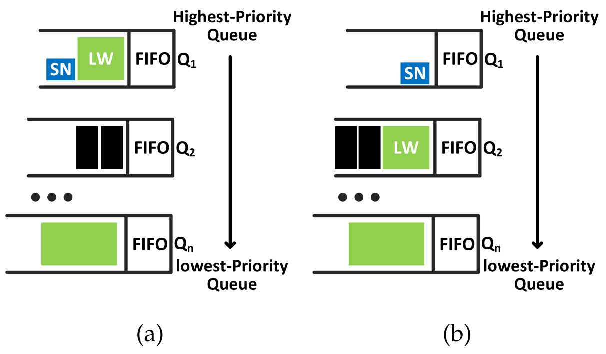


Fig. 2. Impact of HOL blocking. Long-wide coflow C_1 in green is scheduled before short-narrow coflow C_2 (blue). (a) With the highest priority. (b) With different priorities.

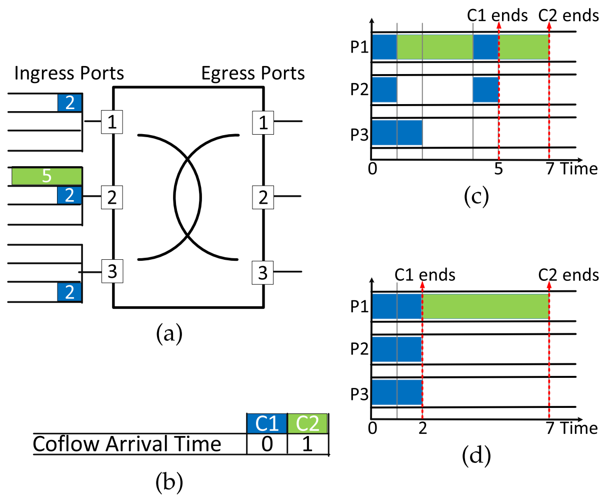


Fig. 3. Impact of only relying on sent bytes. (a) Datacenter fabric. (b) Coflow arrival times. (c) Considering the coflow sent bytes. (d) Considering multiple coflow attributes.

Because the threshold of Q_1 is usually larger than 10MB [10], the CCT of C_2 will greatly increase.

With Different Priorities: In Figure 2b, a well-designed scheduler classifies newly-arrived coflows based on their widths, and then places large coflows into lower-priority queues and places small coflows into higher-priority queues. In this way, the HOL blocking can be avoided. Note that the above example is not a corner case. Because 52% coflows are short and narrow while 17% coflows that create 99% traffic are wide and long [4].

Observation 1: In the presence of large coflows, assigning newly-arrived coflows with different priorities can avoid HOL blocking.

B. Impacts of Only Relying on the Sent Bytes

In existing information-agnostic coflow scheduling algorithms, a coflow's priority is gradually decreased if its sent bytes exceeds the predefined thresholds. However, the problem of this method is that it may increase the average CCT, especially when coflows have very different width.

Consider Figure 3 as an example. As previous work does [4], [10], [12], we abstract the data center network as a non-blocking switch. In Figure 3a, a large coflow C_1 and

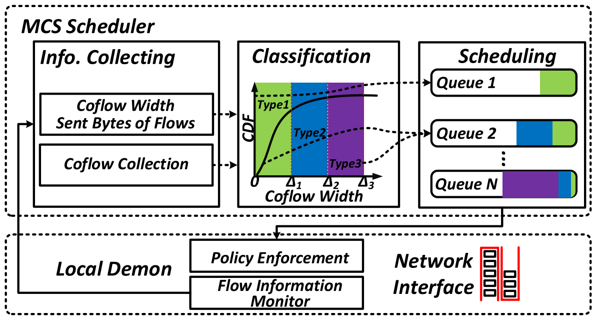


Fig. 4. The architecture of MCS.

a small coflow C_2 are waiting for transmitting in the network. Figure 3b shows their arrival time.

Only Relying on the Sent Bytes: Figure 3c illustrates the scheduling result of Aalo. We assume there are two coflow queues (Q_1 and Q_2), and the threshold of Q_1 is 3 bytes. Therefore, C_1 will demote to Q_2 at $1s$ when it sends 3 bytes data exceeding the threshold of Q_1. Then, C_2 within Q_1 starts transferring and its sent bytes will reach the threshold of Q_1 at $4s$. In this schedule, **the average CCT is** $\frac{5+7}{2} = 6s$**.**

Considering Multiple Coflow Attributes: Unlike Aalo, in Figure 3d, the well-designed scheduler integrally considers *the width, length, and size* of coflows. First, the scheduler classifies C_1 as the wide coflow and C_2 as the narrow coflow based on their widths. Then, the scheduler treats these two types of coflows differently. More specifically, for narrow coflows, the scheduler adjusts the priority based on the coflow size. For wide coflows, the scheduler changes the priority based on the coflow length. If the threshold of Q_1 is also 3 bytes, in this schedule, **the average CCT is** $\frac{2+7}{2} = 4.5s$**.**

The reason for the above problem is that the sent bytes of narrow coflows and wide coflows change in different ways. For narrow coflows, their sent bytes increases slowly, while for wide coflows, their sent bytes increases more rapidly since they contain a large number of flows and their data can be sent in parallel. Besides, we can see from Figure 7b, the size CDF of narrow coflows(Class 1) is very different from the size CDF of wide coflows(Class 2). Therefore, it is too crude to adjust their priorities using the same threshold and method. Note that this example does not conflict with the previous example. In the previous example, we show that the initial priorities of coflow should be different, and we do not indicate that coflows should be scheduled from the widest coflow to the narrowest coflow.

Observation 2: Adjusting coflows' priorities is not only related to the coflow size but also related to other attributes.

IV. MCS Overview

This section outlines the design and architecture of MCS. The goal of MCS is to design an effective and practical information-agnostic coflow scheduler that leverages multiple coflow attributes to avoid HOL blocking and further reduce the average CCT. This idea was first used in our earlier paper [20].

As shown in Figure 4, MCS uses the centralized architecture that contains a centralized scheduler and distributed daemons running on end-hosts. The centralized scheduler receives coflow information from the daemons to determine the global coflow priority and order. The daemons monitor end-hosts and send the locally-observed coflow information to the scheduler every $10 \sim 100\ ms$. Besides, the daemons enforce the scheduling policies made by the scheduler.

The similar centralized architecture is also leveraged by the previous work Aalo [10] and CODA [13]. The two proposals have shown that the centralized architecture has good scalability and can help MCS scale up to more than 10,000 machines with a very small performance loss.

Coflow Information Collecting: MCS assumes the length and size of a coflow cannot be known prior, and it needs to collect sent bytes of each flow, sent bytes of individual coflow, coflow relationships, and coflow width information. Since MCS aims at further improving scheduling algorithms, not focusing on information collecting, MCS leverages the two methods proposed by Chowdhury and Stoica [10] and Zhang *et al.* [13] to collect the information.

- If the clustering computing frameworks are deployed in the private environment, we can directly add API to those frameworks to gather information [10]. Then, the needed information can be easily queried by the API. For example, in MapReduce, we can know the number of map tasks m and the number of reduce tasks r. We can simply use $w = m \times r$ as the coflow width.
- If the clustering computing frameworks is deployed in shared environment, we can monitor sent bytes, IP/port information of all flows at the network stack at end-hosts, and then use the coflow identifier [13] to identify coflow relationships and learn coflow width.

Through the above two methods, we can know the flow width of most of the coflows. However, in some scenarios, the width of coflows can dynamically change over time. Especially, the multi-stage job may consist of multiple waves. For multiple stage job, we can treat the waves in a single state as a single coflow, and not treat all the flows in all waves as a coflow. This abstraction is also used in previous work, such as Varys and Aalo. Fortunately, CODA [13] shows that the flows within in a coflow usually start in a small interval, thus we can dynamically adjust the coflow width without harmful effects(shown in Section VIII-E).

Coflow Classification: Inspired by the observation 1, MCS scheduler will classify the coflows into different types based on their width and then assign each type of coflows a different initial priority. Specifically, to reduce the CCT of small coflows, the type of coflows with smaller width are placed into the queue with higher priority. Thus, small coflows are not blocked by large coflows. Moreover, in order to achieve good performance, we analyze the optimal classification thresholds and proposed an algorithm to automatically determine the classification thresholds from the distributions of coflow width and size.

Multi-Thresholds Coflow Scheduling: Inspired by the observation 2, MCS adjusts the priorities of coflows using double-thresholds to further improve the CCT. Specifically, since the coflow attributes have great variations, it is difficult uses one type of demotion thresholds for all coflows. Thus, MCS splits coflows into narrow and wide coflows and associates each of them with a series of thresholds.

V. Coflow Classification

MCS aims to determine which coflows are small coflows based on the coflow width and assign them proper priorities. To achieves these goals, MCS has to solve the following questions:

1) **Can coflow width indicate the size of coflows?** Since the size and length of coflows cannot be known prior, coflow width is the only attribute that we can know priori and leverage to classify coflows. Thus, MCS have to figure out whether the duration of coflows has a potential connection with coflow width. If not, it is impossible to classify coflows. Fortunately, by leveraging Pearson correlation coefficient (PCC) [21], we show that the coflow width has a strong correlation with coflow size.
2) **How to split coflows?** The optimal classification is that each type of coflows has similar size. However, the coflow size varies dramatically that some coflows with small width can generate a large size of data. MCS needs to choose the proper split thresholds to minimize the number of large coflows that exists in the type of small coflows.
3) **How to assign initial priority for each type of coflows?** The initial priority is the key to avoid HOL blocking. It is easy to know that small coflows should be assigned the highest priority. However, it is challenging to assign the priority of middle and large coflows, since the range of their size have overlap. If we give large coflows higher priority, the CCT of middle and small coflows cannot be effectively improved. If we give them too lower priority, large coflows may be starved by the middle coflows.

A. The Correlation of Attributes

Data Analysis: To evaluate the correlation between coflow width and other attributes, we choose the Pearson correlation coefficient (PCC) as our evaluation metric. The PCC is defined as $\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$, where cov, σ are the covariance and standard deviation of the variables. In statistic, the PCC $\rho_{X,Y}$ is a good measure of the linear correlation between two variables X and Y. The value of PCC is between $+1$ and -1, and a large ρ indicates strong correlation.

We analyze the realistic workload from Facebook production cluster [16] and show the Pearson correlation coefficient between minimal CCT in bytes, length, size of coflows and coflow width in Table I. The minimal CCT in bytes is defined as $\Gamma = \max(\max_i \sum_i d_{i,j}^{(k)}, \max_j \sum_j d_{i,j}^{(k)})$, which indicates the minimal time to transfer all the data of a coflow. Besides, we also draw the coflow attributes along coflow width in Figure 5 to more clearly show their interdependencies.

From Table I, we can find that the size and minimal CCT of coflows have a strong correlation with coflow width, while coflow length has a weak correlation with other attributes. Especially, when the values of coflow attributes are converted into the logarithmic scale, they show stronger correlations than before. From Figure 5, we can further find that *the size and minimal CCT are generally linear with the coflow width in logarithmic scale*, while the range of the coflow length is same for all coflows.

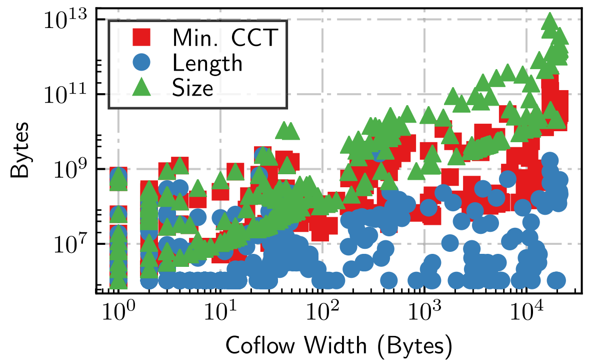


Fig. 5. The minimal CCT in bytes, length and size of coflows along the coflow width.

TABLE I

The Pearson Correlation Coefficient Between Different Attributes. The Values in Each Box Are $\rho_{X,Y}$, $\rho_{\log_{10} X}$, $\log_{10} Y$ and (the Confidence Interval)

Attributes	Coflow Width	Coflow Size	Minimal CCT	Coflow Length
Coflow Width	1/1	0.54/0.91 (0.89,0.92)	0.5/0.81 (0.77,0.83)	0.22/0.37 (0.29,0.44)
Coflow Size	0.54/0.91 (0.89,0.92)	1/1	0.97/0.96 (0.96,0.97)	0.46/0.71 (0.67,0.75)
Minimal CCT	0.5/0.81 (0.77,0.83)	0.97/0.96 (0.96,0.97)	1/1	0.51/0.82 (0.78,0.84)
Coflow Length	0.22/0.37 (0.29,0.44)	0.46/0.71 (0.67,0.75)	0.51/0.82 (0.78,0.84)	1/1

TABLE II

The Pearson Correlation Coefficient Between Coflow Width and Input Data Size, Coflow Size for TPC-H and TPC-DS Benchmarks

Benchmarks	Scale	Input Size	Coflow Size
TPC-H	5	0.69	0.65
TPC-H	10	0.76	0.73
TPC-H	15	0.76	0.75
TPC-DS	5	0.81	0.58
TPC-DS	10	0.77	0.53
TPC-DS	15	0.81	0.54

To show the correlation generally exists in other workloads, we chose two standard benchmark applications (TPC-H [22] and TPC-DS [23]) to generate coflows. TPC-H and TPC-DS are recent benchmarks for testing the performance of big data systems. We run the two benchmarks to generate the input datasets with scale factors of 5, 10, 15. Then, for each dataset, we import the dataset to Hive data warehouse [24] and run 22 SQL quires or 62 SQL queries for TPC-H and TPC-DS respectively. Since we choose the Hadoop [25] as the default Hive execution engine, we analyze the logs of Hadoop to get the coflow information. From the results shown in TABLE II, we can find that the coflow width has a strong correlation with the input data size for both benchmarks, and the coflow width shows more strong correlation with coflow size in TPC-H than

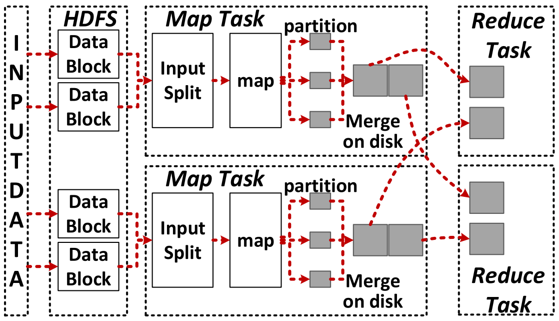


Fig. 6. The data process of MapReudce. Red dash arrows are data flows.

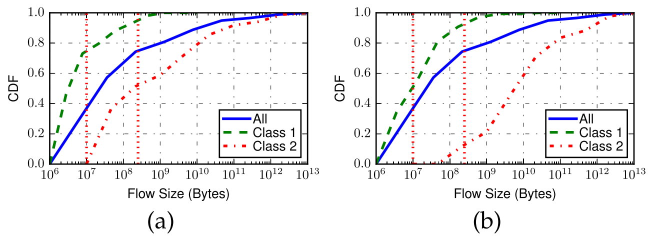


Fig. 7. Splitting coflows into two classes based on different Δ. The left vertical dashed line is 10MB, and the right vertical dashed line is 250MB. (a) $\Delta = 10$. (b) $\Delta = 50$.

TPC-DS. However, note that when $\rho \geq 0.5$ we can say that the correlation is strong, thus we can conclude that coflow width can indicate coflow size in both workloads.

Correlations Deep Dive: Why coflow size has strong correlations with coflow width? We demonstrate this through analyzing the processes of one of the famous data clustering frameworks: MapReduce.

First, we show that the number of map tasks m is related to the input data size. As shown in Figure 6, the input data is split into equal sized *data blocks* (128 MB for Hadoop) stored in HDFS, thus jobs with a large of flows will have more data blocks. According to the Hadoop documents [26], the number of map tasks is determined by the number of data blocks. More specifically, each mapper task only handles one split of data. The size of one split is $Max(minSize, Min(maxSize, block\ size))$, where $minSize = 1$ Byte, $maxSize = 2^{63}$ Bytes in the default settings of Hadoop. Thus, we can conclude that a large input data will have more map tasks. Actually, we verify this by running the WordCount application. We found that the number of map tasks is $\frac{data\ size}{block\ size}$.

Second, we show that the optimal number of reduce tasks r is also related to the input data size. If the application doesn't set the number of reduce tasks through Hadoop API, the default number is one. If users want to achieve the best performance, they should set the number of map tasks to *1) a multiple of the block size* or *2) a task time between 5 and 15 minutes* [26]. Hence, to obtain the best performance, users will set the number of reduce tasks according to the data size.

Then, if we assume the width of a coflow w is $m \times r$, we can know the w is also related to the input data size since m and r is related to the input data size. In Hadoop, each reduce task fetches data from one map task at most once, thus the $m \times r$ is the upper bound of w. Actually, in the TPC-DS and TPC-H benchmarks, for simplicity, we use the $m \times r$ as the coflow width. Finally, it is intuitive that a large amount of data will also generate a large amount of shuffle data, thus the coflow size is highly likely related to the coflow with. Besides TPC-DS and TPC-H, we also analyze the SWIM workload [27] from Facebook data centers, and find the PCC between input data size and coflow size is about 0.79. Note that, SWIM workload lacks the coflow width information, we cannot use it to analyze the PCC between coflow size and coflow width.

B. The Classification of Coflows

In this subsection, we first use a simple example to show how to split coflows into two classes. Next, the problem is generalized to how to split coflows into n classes. We formulate the problem to derive the optimal classification threshold to minimize the classification error. Finally, we show the problem is NP-hard and solve the problem by proposing a heuristic algorithm.

Classification Example: In this simple example, we still use the Facebook workload [16]. Varys classifies coflows whose length is smaller than 5 MB as the short coflow and classifies coflows whose width is smaller than 50 as the narrow coflow. Using this standard, the $short\&wide$ coflows can transfer at most 5 MB $*50 = 250$ MB data, and thus we can classify coflows whose size is smaller than 250 MB as the first class and the other coflow as the second class. Now, given the 250 MB thresh T, our goal is to find a new proper threshold Δ to classify coflows based on their width.

Actually, if $T = 250$ MB, a wide range of Δ can be used to split coflows. For example, Figure 7 shows the distribution of the coflow size for $\Delta = 10$ and $\Delta = 50$. We can find that more than 90% of coflows in class 1 is less than 250 MB for both settings. In other words, if we know the width of a coflow is smaller than Δ, its size is highly likely to be smaller than 250 MB. Thus, $\Delta = 10$ or $\Delta = 50$ is a good solution to split coflows. In contrast, if given $T = 10$ MB, only about 50% of coflows in class 1 is smaller than 10 MB for $\Delta = 50$ while the percentage increases to about 80% for $\Delta = 10$. Hence, for $T = 10$ MB, $\Delta = 10$ is better than $\Delta = 50$ to improve the classification accuracy. The notation is summarized in Table III.

Problem Formulation: Consider a set of coflows $\mathbf{C}$, each coflow k with $d^{(k)}$ size of data has $w^{(k)}$ numbers of flows. We assume there are n classes $P_i (1 \leq i \leq n)$, and we should classify coflow into n classes based on the coflow size to achieve the best performance. If we classify coflows based on coflow size, for each class i, it contains coflows that $T_{i-1} \leq d^{(k)} < T_i$. However, since we don't know the coflow size prior, we have to classify coflows based on coflow width, and for each class i, it contains coflows that $\Delta_{i-1} \leq w^{(k)} < \Delta_i$. As a result, when thresholds $\{T_i\}$ are given, we need to choose $\{\Delta_i\}$ to reduce the classification error.

We denote the joint probability density function for coflow size and coflow width as $f_{\mathbf{S},\mathbf{W}}$, and the joint cumulative distribution function is $F_{\mathbf{S},\mathbf{W}}$. Let $\alpha_i = \int_0^\infty \int_{\Delta_{i-1}}^{\Delta_i} f_{\mathbf{S},\mathbf{W}}(x,y)dxdy$, the percentage of coflows whose width in $[\Delta_{i-1}, \Delta_i)$.

TABLE III
SUMMARY OF NOTATION

P_i	The i th class
T_i	The threshold of class i based on coflow size
Δ_i	The threshold of class i based on coflow width
$d^{(k)}$	The size of a coflow k
$w^{(k)}$	The width of a coflow k
$f_{\mathbf{S},\mathbf{W}}$	The joint probability density function for coflow size and coflow width
$F_{\mathbf{S},\mathbf{W}}$	The joint cumulative distribution function for coflow size and coflow width
α_i	The percentage of coflows that $\Delta_{i-1} \le w^{(k)} < \Delta_i$
β_i	The percentage of coflows that $\Delta_{i-1} \le w^{(k)} < \Delta_i$ and $T_{i-1} \le d^{(k)} < T_i$
$A(T_i, \Delta_i)$	The classification accuracy for priority i
$E(T_i, \Delta_i)$	The classification Error for priority i

Let $\beta_i = \int_{T_{i-1}}^{T_i} \int_{\Delta_{i-1}}^{\Delta_i} f_{\mathbf{S},\mathbf{W}}(x,y)dxdy$, the percentage of coflows whose width in $[\Delta_{i-1}, \Delta_i)$ and size in $[T_{i-1}, T_i)$. Denote the classification accuracy $A(T_i, \Delta_i)$ as the percentage of coflows with size in $[T_{i-1}, T_i)$ in the class P_i according to coflow width based classification. Then, according to Bayesian probability theory [28], $A(T_i, \Delta_i) = \frac{\beta_i}{\alpha_i}$. The classification error E_i for P_i is evaluated by $1 - A(T_i, \Delta_i)$.

Therefore, given the number of classes and the threshold T_i of each class, our goal is to choose an optimal set of thresholds Δ_i to minimize the classification error:

$$\min \ E = \sum_{i=1}^{n} 1 - A(T_i, \Delta_i) \quad (1)$$

$$\text{subject to } \Delta_0 = 0, \Delta_n = \infty \quad (2)$$

$$\Delta_{i-1} \le \Delta_i \quad (3)$$

Classification Algorithm: The above problem is a Sum-of-Linear-Ratios (SoLR) problem and is NP-hard. Thus, we propose the Shortest and Narrowest Coflows First (SNCF) heuristic algorithm to solve the problem. The algorithm mainly has two functions. First, we talk about the classification function. The key idea of this algorithm is trying to search variables in Equation 1 to find the optimal solution. However, since there are a lot of variables, even using the heuristic algorithm, it is still very difficult to find the optimal variables to optimize the original problem. On the other hand, the coflow width ranges from 1 to 10^5, making the problem is more challenging. Thus, we simplify the original problem by limiting the search space of the variables.

Specifically, as we have shown in the first subsection (§V-A), we notice that the coflow size is generally linear with the coflow width *in logarithmic scale*. By leveraging this finding, we add one constraint that $\Delta_i = \Delta_{i-1} \times Step$, where $i \ge 1$, $Step = 10$. As a result, in logarithmic scale, $\lg \Delta_i = \lg \Delta_0 + i \lg Step$, which approximately increases in a linear fashion. Algorithm 1 gives the details for searching the proper thresholds. In each iteration, the algorithm tries to maximize the $A(T_i, \Delta_i)$ greedily. Then, the Δ_i is increased until the calculated percentage stop increasing.

Algorithm 1 The SNCF Algorithm

```
1: function ClassificationThresh(Coflows $\mathbf{C}$,$\{T_i\}$)
2:     $\Delta_0 = min(w^{(k)}), k \in \mathbf{C}$
3:     for $i \in 1...n-1$ do
4:         $\Delta_i = \Delta_{i-1} * Step$
5:         while $A(T_i, \Delta_i) \le A(T_i, \Delta_i * Step)$ do
6:             $\Delta_i = \Delta_i * Step$
7:         end while
8:     end for
9: end function
10: function AssignPriority(Coflow Class $\mathbf{P}$)
11:     for $i \in 1...n-1$ do
12:         for Coflow $k \in \mathbf{P_i}$ do
13:             $minSize = Min(minSize, d^{(k)})$
14:         end for
15:         find $j$ that $T_{j-1} \le minSize < T_j$
16:         $p_i = j$
17:     end for
18: end function
```

C. Assign the Initial Priorities

In this subsection, we present the second function of SNCF to assign the initial priorities of classes. After obtaining a series of thresholds $\{\Delta_i\}$ to classify coflows, we need to assign each class of coflows P_i a priority p_i. This priority is the initial priority for a newly arrived coflow.

On the one hand, since the initial priority is the key to avoid HOL blocking, we hope the class P_i has higher priority than the class P_{i+1}. For instance, if the size of class P_i ranges from 5 MB to 10 MB and the size of class P_{i+1} ranges from 100 MB to 1000 MB, then the class P_i should have a higher priority than the class P_{i+1}. On the other hand, if the range of size of class P_i and class P_{i+1} have a large overlap, they should have close or even the same level of priority to avoid starving. For example, if the size of class P_i ranges from 5 MB to 20 MB and the size of class P_{i+1} ranges from 5 MB to 10 MB, assigning P_i a higher priority will starve P_{i+1}. Hence, the two class should have the same priority.

Recall that the pre-defined thresholds $\{T_i\}$ determine how to classify coflows and the range of size of class P_i, thus we should choose the priority p_i according to $\{T_i\}$. Ideally, for each P_i, we hope to find a interval $[T_{j-1}, T_j]$ that $d^k \in [T_{j-1}, T_j], k \in P_i$. Then, the j is a proper priority for the class since all coflows are only in one interval. However, the range of size of P_i may cover multiple intervals. In this situation, as the AssignPriority function shown in the Algorithm 1, we find the *minimal interval* $[T_{j-1}, T_j]$ that the minimal coflow size of each class belongs to, and use j as the initial priority. By using the *minimal interval*, we can reduce the CCT of small coflows. This is because if small coflows are assigned a higher priority, it is impossible to upgrade their priorities later to avoid starving by large coflows. However, if large coflows are assigned a lower priority, we can downgrade their priorities to avoid blocking small coflows.

Finally, the classification result is plotted in boxplot shown in Figure 8. Considering the size of coflows linearly increases in logarithmic scale, we use $T_i = 10\ T_{i-1}$ [10], thus we can

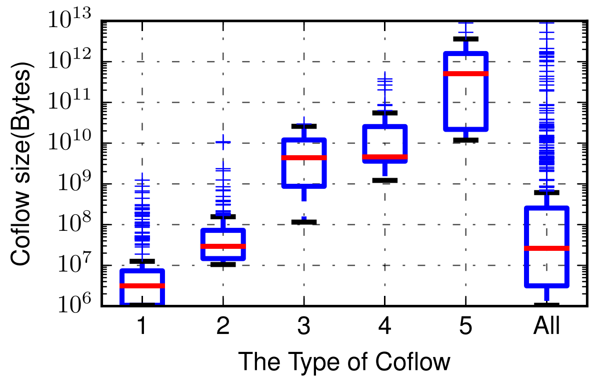


Fig. 8. The proposed SNCF algorithm splits coflows into 5 classes when $T_i = 10^i MB$.

get $\Delta_i = 2^{i-1}$ according to Algorithm 1. Then, by using AssignPriority, we can get $p_i = i$. The result shows that the overlap of two classes of coflows is small, which indicates that our algorithm has a very good result.

VI. Scheduling Algorithm

In this section, we first analyze how to adjust the threshold for different classes of coflows and abstract our system as an M/M/1 queue to formulate the threshold adjustment problem. Then, we introduce the Double-Thresholds algorithm to adjust the threshold for coflows. Finally, we derive the scheduling algorithm of MCS.

A. Threshold Adjustment Problem

Since the size of one type of coflows varies drastically, the initial priority isn't the optimal priority for all coflows within P_i. For example, as shown in Figure 8, the initial priority for class P_5 is 5 that is optimal for 10^4 MB coflows, but this priority is worse for 10^6 MB larger coflows. Because if we don't change the priority of 10^6 MB larger coflows, some 10^4 MB coflows will be blocked by these larger coflows. Therefore, the priority of each coflow needs to be adaptively adjusted to further optimize the average CCT of all coflows.

However, how to adjust the priority of coflows is very challenging. Through observation 2, we can find that the priorities of coflows are influenced by coflow length and coflow width, and thus simple using one type of demotion thresholds is too crude to adjust the priority. Therefore, the threshold should be adjusted by using multi-types of demotion threshold. As we have classified coflows according to their width, we can assign each type of coflows a series of demotion thresholds.

Problem Formulation: We formulate the system by an M/M/1 queue. In multi-thresholds adjustment problem, we adjust the priority of a coflow $k \in P_i$ according to a threshold-vector denoted as $\mathbf{\Theta}^i = [\theta_1^i, \theta_2^i, ..., \theta_n^i]$, where θ_j^i is the threshold of the j th queue for a class P_i. Denote S_j^i as the mean sojourn time for a type i coflows in the j th queue. Denote $F^i(\sim) = F_{\mathbf{S},\mathbf{W}}(\sim, \Delta_i) - F_{\mathbf{S},\mathbf{W}}(\sim, \Delta_{i-1})$ as the coflow distribution of the class P_i. PIAS [15] formulates a similar problem that there is only one type of flows and one type of threshold-vector. Compared to PIAS, our problem has multiple type of coflows and each type of coflows has a threshold-vector. Using the conclusion in PIAS, we can get the upper bound of S_j^i is

$$S_j^i = \frac{\rho(F^i(\theta_j^i) - F^i(\theta_{j-1}^i))}{1 - \rho\sum_{i=1}^{n} F^i(\theta_{j-1}^i)}, \tag{4}$$

where ρ is the average load of all coflows. Note that $F^i(\theta_j^i) = 0$ when $j < p_i$, which means a type i coflow do not generate traffic in the j th queue when its initial priority is larger than j. Then for a type i coflow with size in $[T_{j-1}, T_j)$, it experiences delays from p_i to j, and its upper-bound CCT is $\sum_{l=p_i}^{j} S_l^i$. Denote $S_j^i = 0$ when $j < p_i$, then we can get the average CCT is

$$min \sum_{i=1}^{n}\{\sum_{j=1}^{n}[(F^i(\theta_j^i) - F^i(\theta_{j-1}^i))\sum_{l=0}^{j} S_l^i]\} \tag{5}$$

$$\text{subject to } \theta_0^i = 0, \theta_j^i = \infty \tag{6}$$

$$\theta_{j-1}^i \le \theta_j^i \tag{7}$$

The above problem is also a SoLR problem that is NP-hard. Solving this problem remains an open problem. Besides, note that the above formulation only considers the coflow size. Actually, as we have investigated, the other things, such as the topology of the network and the background flows, also affects the CCT, which makes the problem more complex. Thus, we simplify the problem and propose a double-thresholds algorithm to get the demotion thresholds.

B. Double-Thresholds

In this subsection, we present a double-thresholds scheme to adjust the priority. In the original problem, we classify coflows into n types based on their width and need to assign each type of coflows a threshold-vector $\mathbf{\Theta}^i$. The original problem is difficult to solve, making us think whether it is necessary to use different threshold-vector for each type of coflows. The answer is not, and we think two type of threshold-vector may be enough. Thus, in the double-thresholds scheme, when we need to adjust the coflow priority, we merge n classes of coflows and re-split them into two classes. On the one hand, only using two types of thresholds can reduce the complexity of the problem making solves the problem faster. On the other hand, we think coflows have similar sensitivity to demotion thresholds, merging them has little influence on CCT. This is because the duration of narrow coflows is mainly affected by their size, and the duration of wide coflows are mainly affected by their length since their flows can be transferred simultaneously. For example, if both two classes of coflows spend about 1 second to send 100 MB data, they have similar sensitivity, and we can use the same thresholds for the two classes of coflows. However, if one class spend about 1 second to transfer 100 MB while another coflow spend 0.5 seconds to send 100 MB, they have very different sensitivity.

Therefore, we re-split coflows into wide and narrow coflows according to $\underset{\sim}{\Delta}_w$. We adjust wide coflows' priorities based on their length $\overset{\frown}{l^{(k)}}$ and adjust narrow coflows' priorities based on their sent bytes $\overset{\frown}{d^{(k)}}$. Then, the demotion-thresholds-vector for wide coflows and narrow coflows are denoted as $\mathbf{\Theta}^w = [\theta_1^w, \theta_2^w, ..., \theta_n^w]$ and $\mathbf{\Theta}^n = [\theta_1^n, \theta_2^n, ..., \theta_n^n]$. The setting of each threshold in our algorithm can use the previous standards, such as uniformly-spaced scheme [4], exponentially-spaced scheme [4], DHS scheme [14]. For instance, we can use the uniformly-spaced scheme to choose $\mathbf{\Theta}^w$ while use the exponentially-spaced scheme to choose $\mathbf{\Theta}^n$.

C. MCS Scheduler

We extend the general framework of a non-clairvoyant coflow scheduler described in Aalo [10] and present MCS. MCS mainly has two stages to schedule coflows as shown in Algorithm 2.

Algorithm 2 The Main MCS Algorithm

```
1: function RESCHEDULE(Queues $Q$)
2:     for $i \in 1...n-1$ do
3:         for Coflow $k \in Q_i$ do
4:             if $w^{(k)} \le \Delta_w$ and $\widetilde{d^{(k)}} > \theta_i^n$ then
5:                 Remove $k$ from $Q_i$ and put $k$ into $Q_{i+1}$
6:             end if
7:             if $w^{(k)} > \Delta_w$ and $\widetilde{l^{(k)}} > \theta_i^w$ then
8:                 Remove from $Q_i$ and put to $Q_{i+1}$
9:             end if
10:        end for
11:    end for
12: end function
13: function NEWCOFLOW(Queues $Q$, Coflow $k$)
14:    for $i \in 1...n-1$ do
15:        if $\Delta_{i-1} \le w^{(k)} < \Delta_i$ then
16:            put $k$ into $Q_{p_i}$
17:            Break
18:        end if
19:    end for
20: end function
```

Priority Queues: In our scheduler, we directly adopt the priority queues of Aalo [10], which uses three scheduling disciplines:

1) Across queues: coflows from higher-priority queues are scheduled first.
2) Within each queue: coflows are scheduled based on FIFO.
3) Flows within each coflow: Max-Min principle is used to allocate flow rates.

Assigning Initial Priority: When new coflows arrive, the function NEWCOFLOW is called to classify coflows according to their width. The classification thresholds and priorities of each type of coflows are calculated prior by using the SNCF algorithm. Then, the newly-arrived coflows can be quickly assigned initial priorities with low overhead.

Adjusting Priority: When new coflows arrive or old coflows finish, we will use the function RESCHEDULE to adjust the priorities of all the coflows. Coflows are re-split into two types according to their width, and their priorities are adjusted according to $\mathbf{\Theta}^w$ and $\mathbf{\Theta}^n$, respectively.

Complexity and Overheads: Since SNCF algorithm can be calculated offline, it will not add any overhead to the online scheduling system. Besides, the complexity of SNCF algorithm is linear with the range of coflow size in logarithmic scale (usually $\le$ 13), thus the computation overheads of SNCP is very small that can be omitted. Furthermore, MCS leverages the loose-coordination architecture of Aalo to reduce the communication and coordination overheads between the centralized scheduler and distributed daemons. Hence, MCS can have a good scalability.

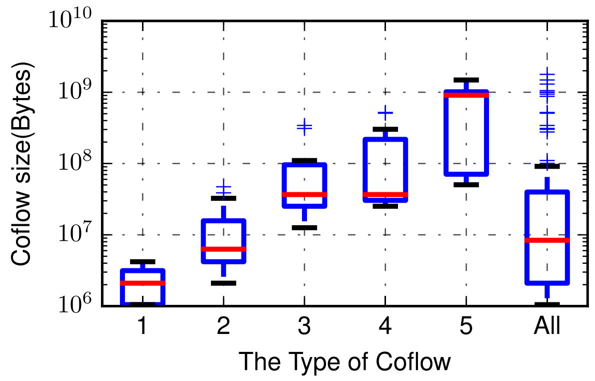


Fig. 9. [Testbed] The proposed SNCF algorithm splits coflows into 5 classes when $T_i = 2^i MB$.

D. Implementation

The MCS prototype implements the master-client architecture shown in Figure 4 in about 3700 lines of C++ code. The coflow schedulers run in the master with each flow-level information reported by the clients. The clients communicate with the master and enforce the scheduling decisions of the master.

MCS client daemons have two main functions: reporting sent bytes of each flow to MCS master and limiting flow rates according to scheduling decisions. The two functions are implemented in the user space. More specifically, the daemon writes 64 KB data each time via $write()$ system call and records the execution time t. Then, the send thread of client sleeps $rates/64$ KB $-t$ for rate limiting. We also set the send socket buffer size of TCP to 128 KB to reduce the buffered data at hosts. MCS daemons resynchronize every 10 milliseconds as Aalo does. Each daemon will report the sent bytes of each flow and flow rates during the period to SkipL master.

After receiving all updated flow information, MCS master can update the sent bytes counter of each coflow and make a decision based on the current knowledge of coflows using the proposed algorithms.

Implementation Overhead of MCS: To measure the CPU overheads of MCS scheduler and daemons, we generated more 25 coflows with more than 250 flows. The MCS scheduler runs on a Dell PowerEdge R730 server with 64GB of memory and a 8-core E5-2637 3.5GHz CPU. The extra CPU overhead of the MCS scheduler is less than 5%, and the extra CPU overhead of the MCS daemon is less than 3%.

VII. EVALUATION

In this section, we evaluate the performance of MCS on our small-scale testbed through three experiments. First, we evaluate how the performance of MCS is influenced by the

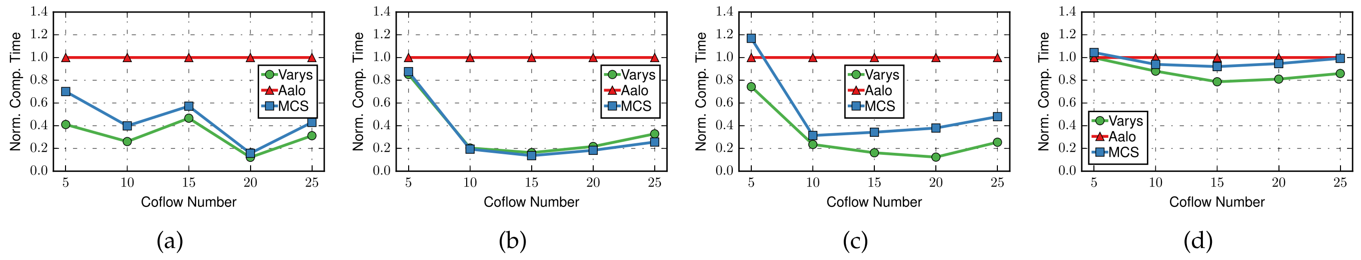


Fig. 10. [Testbed] The impact of coflow number. (a) Short and narrow. (b) Long and narrow. (c) Short and wide. (d) Long and wide.

number of parallel coflows. The experimental results show that MCS achieves a stable performance as the number of parallel coflows increasing, reducing the CCT for $short\&narrow$ coflows by more than 60% compared to Aalo. Second, we evaluate the performance of MCS with realistic workload with increasing load. The experimental results show that MCS achieves a similar performance with Varys for $short\&narrow$ coflows. Third, we consider batch-arrived coflows and show that batch-arrived coflows are more challenging for Aalo.

A. Evaluation Settings

Testbed: We built a small-scale testbed that consists of 9 hosts connected to a DELL n4032 24-port Gigabit Ethernet switch. Eight hosts running MCS daemons are Dell OPTIPLEX with a 2-core Intel I3-3220 3.3GHz CPU, 4G memory, and a Broadcom BCM5719 NIC. One host running MCS scheduler is a DELL R730 with two 8-core Intel E5-2637 3.5GHz CPUs, 64G memory, and 8 Broadcom BCM5719 NICs. The default OS is Ubuntu 16.04 64 bit version with Linux 4.4.0-21 kernel. The base round-trip-time in our testbed is around 200 μs, and we set the coordination interval Δ to 10 ms as Aalo does.

Scheme Compared: We evaluate the following schemes:

- **Varys:** We implement Varys and compare our proposed mechanism with it. Through comparing MCS with Varys, we can inspect how information-agnostic affects the performance of coflow scheduler. Besides, we can see Varys as the optimal results and it shows the room for improvement.
- **Aalo:** We also implement Aalo and use the default parameters in [10]: $\Delta = 20$ ms, $E = K = 10$, and $Q_1^{hi} = 10$ MB.
- **MCS:** MCS is implemented according to the design described above. Because our testbed only have 8 hosts, we scale down the parameters, and the classification result is plotted in boxplot shown in Figure 9. Thus we use $\Delta_i = 2^i$, $p_i = i$, $\theta_i^n = 10 \times 2^{i-1}$ MB and $\theta_i^w = 2^{i-1}$ MB in testbed experiments.

Workload: We use a realistic workload from Facebook production cluster [16] with 150 racks. This trace contains over 500 coflows and the coflow size distribution of the workload are shown in Figure 7. As shown in Table IV, coflows are classified into four types based on their width and length. Basically, a coflow is $short$ if its longest flow is less than 5 MB and $narrow$ if it has at most 50 flows [4]. Note that, similar to the settings of Varys, coflows arrive in batch and the default batch interval is 10 s. Since our testbed only has eight hosts smaller than 150 racks, we scale downs the coflow width accordingly. Specifically, we scale down narrow coflow threshold to 5, and the short coflow threshold is still 5 MB. Then, we randomly generate the coflows according to Table IV. For example, we first generate an SN coflow at 0.52 possibility. Then we randomly chose it width in $[1, 5)$, and randomly chose it length in $[1, 5)$ MB.

TABLE IV

COFLOWS BINNED BY THEIR LENGTH (SHORT AND LONG) AND THEIR WIDTH (NARROW AND WIDE)

Coflow Bin	SN	LN	SW	LW
% of Coflows	52%	16%	15%	17%
% of Bytes	0.01%	0.67%	0.22%	99.10%

Metric: We choose average coflow completion time among coflows as our main performance metric. To clearly show the improvements of different mechanisms, the average CCT is normalized to Aalo:

$$The\ normalize\ CCT = \frac{Compared\ CCT}{Aalo's\ CCT} \tag{8}$$

If the normalized CCT is small, MCS has a better performance than Aalo. To clearly show how MCS influences the CCT of small coflows, we classify coflows into four groups and consider the average CCT across the four groups respectively.

B. Experimental Results

Impact of Coflow Number: We first look at how the number of parrel coflows influences the performance of coflows. In each round of experiments, we generated different numbers of coflows arriving at the same time, and their width and length are randomly selected according to the Facebook workload. Figure 10 shows the experimental results, and each point is the average results by repeating the same setting five times. Due to space limitation, we omit the results for the overall coflows whose performance is quite similar to that of $long\&wide$ coflows.

Figure 10a shows that MCS reduces the average CCT of $short\&narrow$ coflows by up to 90% compared to Aalo when there are 15 parallel coflows. The performance gap between MCS and Varys is within 20%. As expected, the results show that MCS greatly reduces the CCT of small coflows. Figure 10b shows that MCS has very similar performance with Varys and reduces the average CCT by up to 81% compared

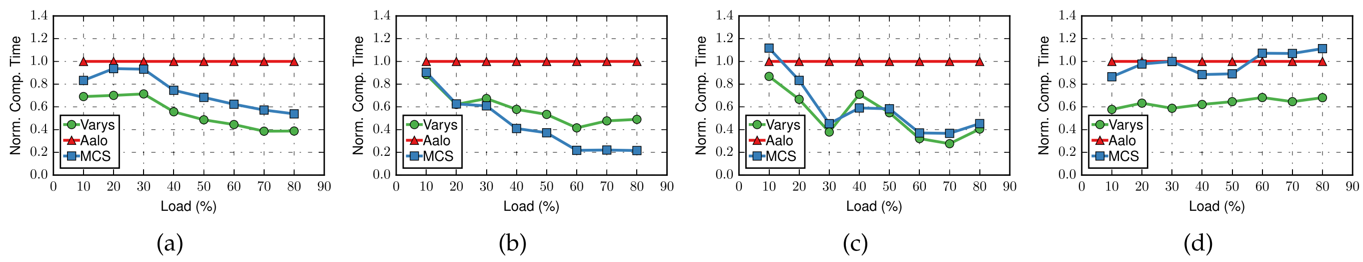


Fig. 11. [Testbed] The impact of coflow load. (a) Short and narrow. (b) Long and narrow. (c) Short and wide. (d) Long and wide.

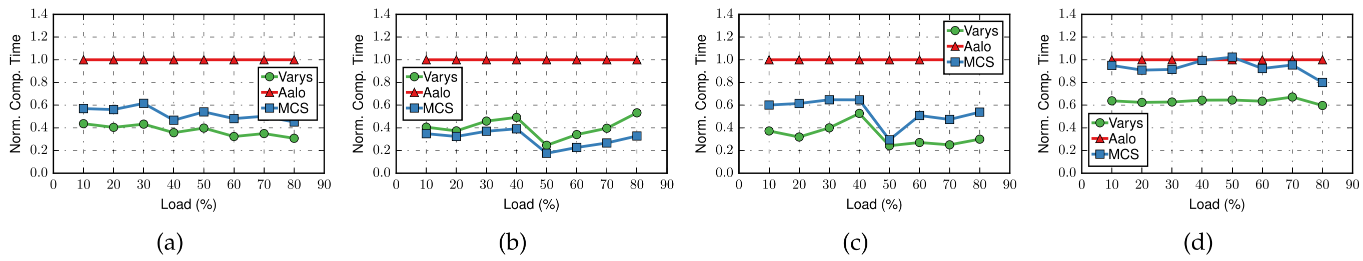


Fig. 12. [Testbed] The impact of batch arrival. (a) Short and narrow. (b) Long and narrow. (c) Short and wide. (d) Long and wide.

to Aalo. Figure 11c and Figure 11d show that corresponding improvements of $short\&wide$ coflows and $long\&wide$ coflows are up to 70% and 10%, respectively.

Across all the Sub-Figures in Figure 10, the improvements in the average CCT are not obvious when there are 5 coflows. This is because when the number of coflows is small, large coflows may not exist in the arrived coflows, which reduces the blocking probability. When the number of coflows is large, the number of coflows also increases, which increases the blocking probability. Therefore, when the number of coflows is larger than 5, the improvements in CCT are obvious.

Impact of Load: In this evaluation, we randomly generate 100 coflows between eight hosts according to a Poisson process. The coflow sizes and width are sampled from the Facebook workload (scaled-down) and the coflow arrival interval is chosen depending on the desired network load.

Figure 11a shows that MCS reduces the average CCT of $short\&narrow$ coflows by up to 42%. Figure 11b and Figure 11c show that corresponding improvements of $long\&narrow$ coflows and $short\&wide$ coflows are up to 82% and 70%, respectively. As expected, as the load increases, MCS can significantly outperform Aalo. When the load is low, it is less likely that multiple coflows are arriving concurrently, thus MCS, Varys and Aalo have the similar performance. When the load becomes larger, the number of coflows waiting for transferring increases. Thus, MCS can leverage the proposed algorithm to better differentiate coflows and achieve better performance by reducing the CCT of small coflows.

For $long\&wide$ coflows show in Figure 11d, MCS provides similar performance as Aalo while Varys provide the best performance. Compared to Aalo, MCS reduces the average CCT by up to 10%. Compared to Varys, the performance gap is within 40%. This indicates that optimizing large coflows is very challenging for both Aalo and MCS.

Impact of Batch Arrival: In this evaluation, we use the same settings and workload of the previous evaluation.

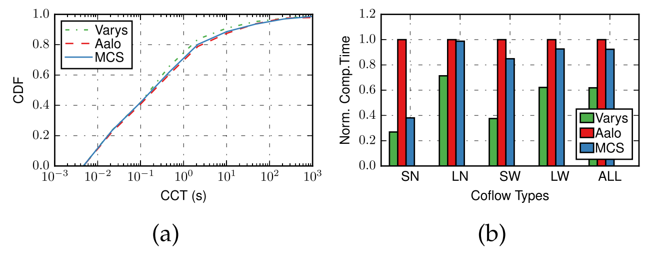


Fig. 13. CCT for Varys, Aalo, and MCS. (a) CCT distributions. (b) The Average improvements in CCTs normalized to Aalo.

In the previous evaluation, coflows arrive randomly without batch. To study the impact of batch arrival, instead of generating a coflow once a time, we randomly generate a batch of coflows with 10 coflows, and the batched-coflows arrive according to the desired load. Therefore, coflows within a batch has the same arrival time.

Figure 12b shows that MCS still achieves better performance than Aalo, and reduces the average CCT of $short\&narrow$ coflows by up to 60%. Figure 11b and Figure 11c show that corresponding improvements of $long\&narrow$ coflows and $short\&wide$ coflows are up to 77% and 70%, respectively. For $long\&wide$ coflows show in Figure 11d, MCS provides similar performance as Aalo while Varys provide the best performance. Compared to Aalo, MCS reduces the average CCT by up to 20%. Compared to Varys, the performance gap is still within 40%.

Compared to the Figure 11 without batch, we can find that the improvement radio increases. For example, for $short\&narrow$ at 10% load, the improvement radio changes from 20% in Figure 11a to 40% in Figure 12d. This indicates that batched-coflows are more challenging for Aalo. Because Aalo is unable to identify small coflows when a batch of coflows arrives at the same time with zero sent bytes. Besides, we can also find that the performances of MCS and Varys

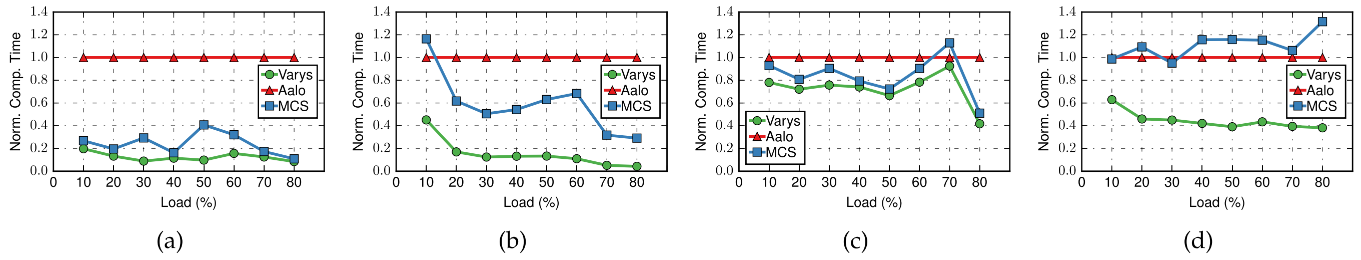


Fig. 14. CCT when batch interval is 10 s. (a) Short and narrow. (b) Long and narrow. (c) Short and wide. (d) Long and wide.

doesn't increase with the load, and the performance is similar to the performance in Figure 10 when the number of coflows is 10. This indicates when coflows arrive in batch, the performance is mainly influenced by the number of parallel coflows.

VIII. Simulation

In this section, we perform large-scale simulations to extensively test the performance of MCS. First, we show MCS has better performance than Aalo by using the same settings as Aalo. Second, we change the load and coflow arrive interval to show MCS works well in more challenging scenarios. Third, we show the impacts of coflow width. Finally, we show how parameters influence the performance of MCS.

A. Simulation Settings

Simulator: We implement MCS in a trace-driven flow-level simulator used by Aalo [17]. This simulator performs a detailed replay of the coflow traces.

Workload: We still use the Facebook workload in our simulations [16]. In simulations, a coflow is $short$ if its longest flow is less than 5 MB and $narrow$ if it has at most 50 flows [4]. Note that, similar to the settings of Varys, coflows arrive in batch and the default batch interval is 10s.

Metrics: We measure the coflow completion time (CCT), and use the average CCT normalized to Aalo as the main performance metric.

Parameters of MCS: In simulations, we use Facebook workload without scale-down. Thus, we set $\Delta_i = 10^i$, $p_i = i$, and use exponentially-spaced scheme [10] to set the thresholds: $\theta_i^n = 10^i$ MB and $\theta_i^w = 2^{i-1}$ MB.

B. Results Under Normal Workloads

We first investigate the performance of MCS under the Facebook workload. Figure 13b shows the performance of different scheduling algorithms in terms of the average CCT normalized to Aalo. It implies that MCS performs as well as information-aware coflow scheduler Varys for $short$ and $narrow$ coflows, and significantly outperforms information-agnostic coflow scheduler Aalo. For example, MCS reduces the average CCT for $short\&narrow$ coflows by up to $\sim 62\%$ and reduces the average CCT for $short\&wide$ coflows by up to $\sim$ 17%. In addition, it is not a surprise that Varys performs the best since it knows coflow information prior. Figure 13a shows the CDFs of CCT under different scheduling algorithms. We can see that MCS is between Varys and Aalo.

C. Impact of Load

To study the impact of network load, we change the coflow arrival time in the Facebook trace according to the desired load. Basically, coflows arrive according to a Poisson process, and the arrival rate is set to $\lambda = \frac{load \times NetBandwidth}{MeanCoflowSize}$. The simulation results by varying the network load from 10% to 80% are shown in Figure 14.

Figure 14a shows that MCS achieves very good performance for $short\&narrow$ coflows. Compared to Aalo, MCS reduces the average CCT for $short\&narrow$ coflows by up to $\sim 90\%$. This is due to the fact that the SNCF mechanism of MCS helps to identify small newly-arrived coflows and only gives them the highest-priority to avoid blocking while Aalo gives all newly-arrived coflows the highest priority. In addition, MCS also achieves good performance for $long\&narrow$ coflows. With the increase of the load, the performance gap between MCS and Aalo becomes large, and MCS achieves 70% lower CCT at 80% load.

Figure 14c shows that MCS has very similar performance with Varys for $short\&wide$ coflows. Compared to Aalo, MCS reduces the average CCT for $short\&wide$ coflows by up to 50%. Note that the performance gap between MCS and Varys is small. On the contrary, MCS achieves worst performance for $long\&wide$ coflows at high loads. This problem is mainly caused by the large batch interval. When large coflows and small coflows arrive in batch, because MCS gives higher-priority to small coflows, the CCT of large coflows will certainly increase. As shown in Figure 15d, if we use small batch interval, the performance gap is very small at low loads.

D. Impact of Batch Arrival

The above subsection has studied the performance of MCS under different load. In this experiment, to study the impact of batch arrival, we change the batch interval to 1 s and repeat the previous simulation.

Figure 15 shows the simulation results. Compared to Figure 14, we observe on obvious degradation in the performance gap for $short\&narrow$ coflows. For example, the performance gap between MCS and Aalo decreases from 90% to 30% at 80% load. This is because when the batch interval is small, the number of newly-arrived coflows in a batch becomes small, making small newly-arrived coflows are less likely blocked by large coflows. For the other types coflows, the performance of MCS and other coflow scheduling algorithms are similar to the result shown in Figure 14.

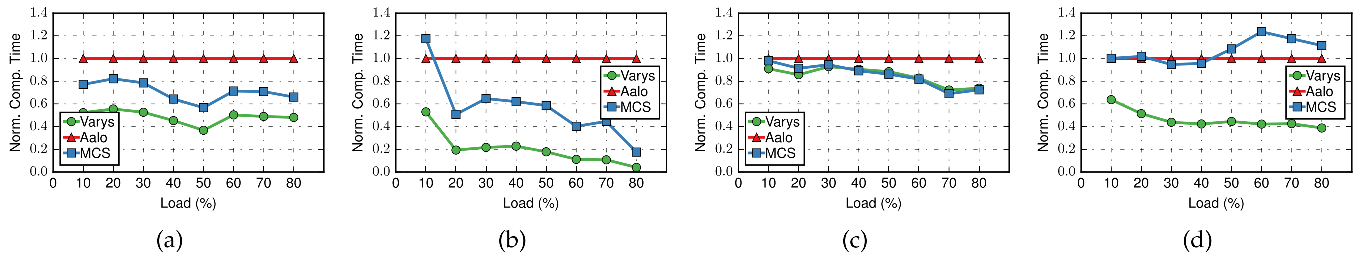


Fig. 15. CCT when batch interval is 1 s. (a) Short and narrow. (b) Long and narrow. (c) Short and wide. (d) Long and wide.

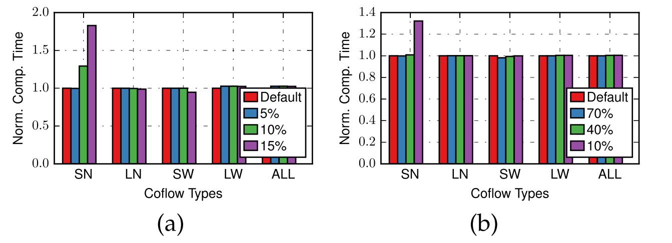


Fig. 16. Impact of the coflow width. (a) Missing information. (b) Inaccurate information.

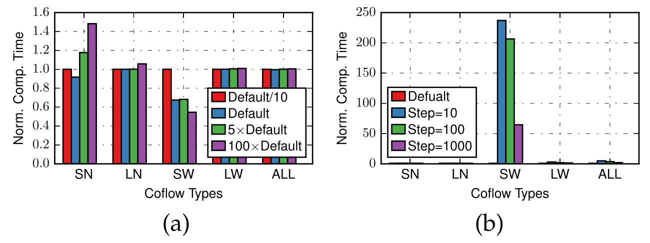


Fig. 17. Impact of Δ. (a) Default: $\Delta_i = 10^i$. (b) $\Delta_i = step \times i$.

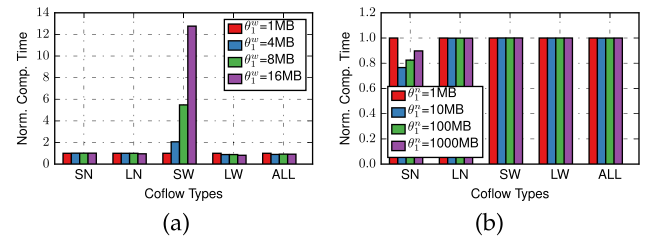


Fig. 18. Impact of $\mathbf{\Theta}^w$ and $\mathbf{\Theta}^n$. (a) $\theta_i^w = \theta_1 \times 2^{i-1}$. (b) $\theta_i^n = \theta_1 \times 10^{i-1}$.

E. Impact of Coflow Width

We use two simulations to show how the performance is influenced if the coflow width information is inaccurate.

In the first simulation, we randomly pick up 5%, 10%, 15% coflows and assume their width are completely unknown, thus, these coflows are always put in the highest priority. The simulation result is shown in Figure 16a. We can see that missing 5% information has a very little worse effect. Missing 15% information will greatly reduce the performance.

In the second simulation, we assume all the coflows' width information are inaccurate that if the real coflow width is w, the scheduler can only know the width is αw, and the α is varied from 70% to 10%. The simulation result is shown in Figure 16a. We can find that if we only know the number of some flows within a coflow, our algorithm still has a good performance. Compared to completely missing the width information, the inaccurate information caused by coflow width change has less worse effect. This is because the thresholds to classify coflows are in logarithmic scale, if $\alpha \geq 0.1, -\log \alpha \leq 1$, the priority of a coflow can change at most one level. As a result, the priority level of a large coflows may not change even some of their flows will start later.

F. Sensitivity to Parameters

We now examine MCS's sensitivity to parameters. Recall that MCS mainly has three parameters to configure: Δ_i, $\mathbf{\Theta}^w$ and $\mathbf{\Theta}^n$.

The first parameter Δ_i is used to classify coflows. As we increase Δ_i, more coflows will be classified as small coflows and given higher-priority. In Figure 17a, we can find that when Δ_i is close to the default settings ($\Delta_i = 5\times$Default), MCS achieves a good performance, while too large and small Δ decrease the performance. Then, instead of using Δ_i calculated by Algorithm 1, we use a uniformly-space scheme to set Δ_i. In the uniformly-space scheme, Δ_i is set to $step \times i$. The simulation results in Figure 17b show that uniformly-space scheme achieves very bad performance compared to the default values calculated by Algorithm 1. This is because that when the step is small, a large number of coflows will be assigned the lowest-priority.

The other two parameters are used to adjust the priorities of coflows during their transmission. First, we focus on how $\mathbf{\Theta}^w$ impact on MCS's performance. As we increase θ_1^w from 1MB to 16MB, more *wide* coflows will be scheduled in the same queue. Figure 18a shows that increasing θ_i will greatly decrease the performance of *short&wide* coflows. Then, We change θ_1^n from 1MB to 1000 MB. As we increase θ_i^n, more *short* coflows will be scheduled in the same queue. As shown in Figure 18b, $\mathbf{\Theta}^n$ has fewer impacts on the performance of *short&narrow* coflows since they have the highest-priority.

IX. Conclusion

This article showed that the design space for information-agnostic coflow scheduling should not be limited to the sent-bytes-based rate allocation. Based on this, we have presented MCS that leveraging multiple coflow attributes to further reduce the completion time of coflows. Testbed evaluations and numerical simulation results demonstrated that MCS could greatly reduce the completion time of small coflows and outperformed the previous information-agnostic coflow scheduling algorithm Aalo.

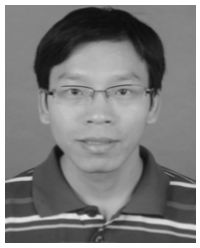
References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. ACM SIGOPS*, 2007, pp. 59–72.
[3] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX NSDI*, 2012, p. 2.
[4] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proc. ACM SIGCOMM*, 2014, pp. 443–454.
[5] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. ACM HotNets*, 2012, pp. 31–36.
[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. ACM SIGCOMM*, 2011, pp. 98–109.
[7] Y. Zhao *et al.*, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *Proc. IEEE INFOCOM*, Apr./May 2015, pp. 424–432.
[8] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.
[9] Z. Li, Y. Zhang, D. Li, K. Chen, and Y. Peng, "OPTAS: Decentralized flow monitoring and scheduling for tiny tasks," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.
[10] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM SIGCOMM*, 2015, pp. 393–406.
[11] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. ACM SIGCOMM*, 2014, pp. 431–442.
[12] Z. Huang *et al.*, "Need for speed: CORA scheduler for optimizing completion-times in the cloud," in *Proc. IEEE INFOCOM*, Apr./May 2015, pp. 891–899.
[13] H. Zhang *et al.*, "CODA: Toward automatically identifying and scheduling coflows in the dark," in *Proc. ACM SIGCOMM*, 2016, pp. 160–173.
[14] Y. Gao, H. Yu, S. Luo, and S. Yu, "Information-agnostic coflow scheduling with optimal demotion thresholds," in *Proc. IEEE ICC*, May 2016, pp. 1–6.
[15] W. Bai *et al.*, "Information-agnostic flow scheduling for commodity data centers," in *Proc. USENIX NSDI*, 2015, pp. 455–468.
[16] *Coflow-Benchmark*. Accessed: May 26, 2018. [Online]. Available: https://github.com/coflow/coflow-benchmark
[17] *CoflowSim*. Accessed: May 26, 2018. [Online]. Available: https://github.com/coflow/coflowsim
[18] S. Luo *et al.*, "Minimizing average coflow completion time with decentralized scheduling," in *Proc. IEEE ICC*, Jun. 2015, pp. 307–312.
[19] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proc. ACM SPAA*, 2015, pp. 294–303.
[20] S. Wang *et al.*, "Leveraging multiple coflow attributes for information-agnostic coflow scheduling," in *Proc. IEEE ICC*, May 2017, pp. 1–6.
[21] N. J. D. Nagelkerke, "A note on a general definition of the coefficient of determination," *Biometrika*, vol. 78, no. 3, pp. 691–692, Sep. 1991.
[22] *T.-H. Benchmark*. Accessed: May 26, 2018. [Online]. Available: http://www.tpc.org/tpch/default.asp
[23] *TPC-DS Benchmark*. Accessed: May 26, 2018. [Online]. Available: http://www.tpc.org/tpcds/default.asp
[24] *Apache Hive*. Accessed: May 26, 2018. [Online]. Available: https://hive.apache.org
[25] *Apache Hadoop*. Accessed: May 26, 2018. [Online]. Available: http://hadoop.apache.org
[26] *How Many Maps and Reduces*. Accessed: May 26, 2018. [Online]. Available: https://wiki.apache.org/hadoop/HowManyMapsAndReduces
[27] *SWIM Workload*. Accessed: May 26, 2018. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki
[28] J. M. Bernardo and A. F. M. Smith, *Bayesian Theory*. Hoboken, NJ, USA: Wiley, 2001.

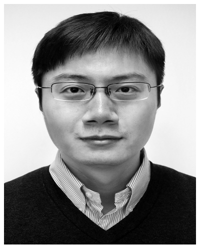
Shuo Wang received the B.S. degree in communication engineering from Zhengzhou University, Zhengzhou, China, in 2013, and the Ph.D. degree in information and communication engineering from the Beijing University of Posts and Telecommunications, Beijing, China, in 2018. He is currently a Post-Doctoral Researcher with the Beijing University of Posts and Telecommunications. His research interests include data center networking and software-defined networking.

Jiao Zhang received the B.S. degree from the School of Computer Science and Technology, Beijing University of Posts and Telecommunications (BUPT), in 2008, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2014. From 2012 to 2013, she was a Visiting Student with the Networking Group of ICSI, UC Berkeley. She is currently an Associate Professor with the School of Information and Communication Engineering, BUPT. Until now, she has coauthored over 30 international journal and conference papers. Her research interests include traffic management in data center networks, software-defined networking, network function virtualization, future Internet architecture, and routing in wireless sensor networks.



Tao Huang received the B.S degree in communication engineering from Nankai University, Tianjin, China, in 2002, the M.S. and Ph.D. degrees in communication and information system from the Beijing University of Posts and Telecommunications, Beijing, China, in 2004 and 2007, respectively. He is currently a Professor with the Beijing University of Posts and Telecommunications. His current research interests include network architecture, routing and forwarding, and network virtualization.

Tian Pan received the B.S. degree from Northwestern Polytechnical University, Xi'an, China, in 2009, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2015. He has been a Post-Doctoral Researcher with the Beijing University of Posts and Telecommunications since 2015. His research interests include router architecture, network processor architecture, network power efficiency, and software-defined networking.



Jiang Liu received the B.S. degree in electronics engineering from the Beijing Institute of Technology, China, in 2005, the M.S. degree in communication and information systems from Zhengzhou University, China, in 2009, and the Ph.D. degree from the Beijing University of Posts and Telecommunications (BUPT) in 2012. He is currently an Associate Professor with BUPT. His current research interests include network architecture, network virtualization, software-defined networking, information-centric networking, and platforms for networking research and teaching.

Yunjie Liu received the B.S. degree in technical physics from Peking University, Beijing, China, in 1968. He is currently the Academician of the China Academy of Engineering, the Chief of the Science and Technology Committee of China Unicom, and the Dean of the School of Information and Communication Engineering, Beijing University of Posts and Telecommunications. His current research interests include next generation network, network architecture, and management.