Joint Reducer Placement and Coflow Bandwidth Scheduling for Computing Clusters

Yangming Zhao, *Member, IEEE*, Chen Tian, *Senior Member, IEEE*, Jingyuan Fan, *Member, IEEE*, Tong Guan, Xiaoning Zhang, and Chunming Qiao, *Fellow, IEEE*

***Abstract*— Reducing Coflow Completion Time (CCT) has a significant impact on application performance in data-parallel frameworks. Most existing works assume that the endpoints of constituent flows in each coflow are predetermined. We argue that CCT can be further optimized by treating flows' destinations as an additional optimization dimension via reducer placement. In this article, we propose and implement RPC, a joint online Reducer Placement and Coflow bandwidth scheduling framework, to minimize the average CCT in cloud clusters. We first develop a 2-approximation algorithm to minimize the CCT of a single coflow, and then schedule all the coflows following the Shortest Remaining Time First (SRTF) principle. We use real testbed experiments and extensive large-scale simulations to demonstrate that RPC can reduce the average CCT by 64.98% compared with the state-of-the-art technologies.**

***Index Terms*— Computing clusters, reducer placement, coflow scheduling.**

I. Introduction

DATA transfer has a significant impact on application performance in data-parallel frameworks such as MapReduce [2], Pregel [3], and Dryad [4]. These computing paradigms all implement a data flow computing model, in which a group of data flows need to pass through a data transfer phase (*e.g.*, shuffle in MapReduce) before generating the final results. For some applications, 50% of the job completion time is spent on transferring data across the network [5]. Although [6] claimed that reducing data transfer time does not significantly improve job performance, it is proven that this happens only when the CPU becomes a system bottleneck [7]. Optimizing network performance can significantly reduce job completion time as long as the CPU is not a system bottleneck [7]. Accordingly, in this article we focus on improving the performance of data-intensive applications by reducing the data transfer time. As in many previous works [8]–[12], we focus on the data transfer phase of each job, but do not consider the computation phase in our optimization framework.

In data-parallel frameworks, a data transfer phase is not considered complete until all its constituent flows have finished. For example, in MapReduce [2], a computation stage cannot complete, or sometimes even start, before it receives all the flows from its previous stage. These flows between two stages are known as a *coflow*. Minimizing the average Coflow Completion Time (CCT) can improve both responsiveness and throughput [11].

Prior works on minimizing data transfer time have focused on either task placement or coflow bandwidth scheduling, but not both. In such task placement schemes [13]–[16], the main idea is to place each task close to its input data in order to increase data locality, which can help reduce the amount of data to be transferred and the associated transfer time. Coflow scheduling schemes [8]–[12] control the priority and/or the sending rate of each flow to minimize the average CCT. They assume that the tasks have already been placed and hence the endpoints of flows are predetermined.

We observe that the average CCT can be further reduced by jointly optimizing reducer placement (*i.e.*, task placement) and coflow bandwidth scheduling. Fig. 1 shows an example. There are two coflows, $C^{(1)}$ and $C^{(2)}$. In $C^{(1)}$, there are three reducers to fetch flows of 1 Gb, 1.5 Gb, and 2 Gb, respectively, while in $C^{(2)}$, there are three reducers to fetch flows of 1.5 Gb, 2 Gb, and 2 Gb, respectively. Three hosts (M_1, M_2, and M_3) can be used to allocate the reducers. The available incoming bandwidths at these three hosts are 2 Gbps, 1 Gbps, and 1 Gbps, respectively. (Note that such a heterogeneous bandwidth scenario can happen if part of the bandwidth has been assigned to other applications.) For simplicity, we assume that the available outgoing bandwidths at the hosts issuing these flows are 10 Gbps; the network connecting all the hosts is nonblocking; and none of the hosts used to allocate the reducers are the same as those issuing these flows. Therefore, the only network bottleneck exists on the incoming links to the reducers' hosts.

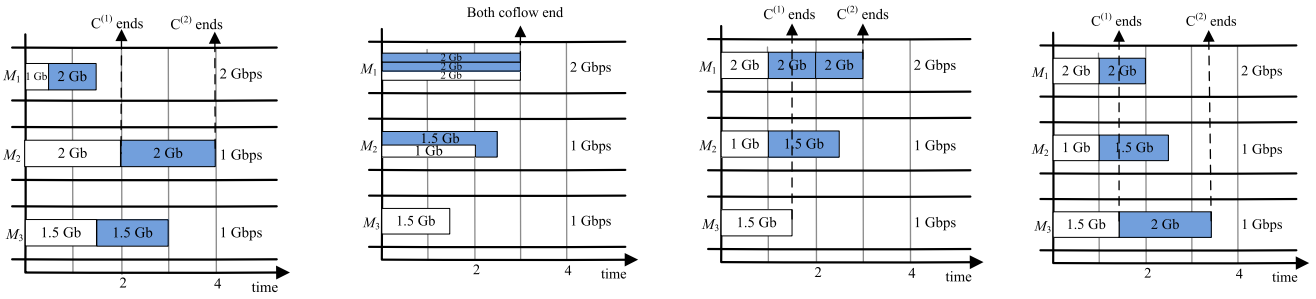
Manuscript received December 19, 2018; revised May 3, 2019, August 31, 2019, November 5, 2019, and March 25, 2020; accepted November 5, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Llorca. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1003505; in part by the NSF under Grant CNS-1737590; in part by the National Natural Science Foundation of China under Grant 61772265, Grant 61802172, Grant 62072228, Grant 61671130, and Grant 61671124; in part by the Fundamental Research Funds for the Central Universities under Grant 14380073; in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization; and in part by the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program. The preliminary version of this paper [1] was presented at IEEE ICNP 2018, London, U.K. *(Corresponding author: Chen Tian.)*

Yangming Zhao, Jingyuan Fan, Tong Guan, and Chunming Qiao are with the Department of Computer Science and Engineering, University at Buffalo, The State University of New York, Buffalo, NY 14260 USA.

Chen Tian is with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: tianchen@nju.edu.cn).

Xiaoning Zhang is with the School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China.

Digital Object Identifier 10.1109/TNET.2020.3037064



(a) Optimal bandwidth scheduling with suboptimal reducer placement. (b) Optimal reducer placement with suboptimal bandwidth scheduling. (c) Optimal reducer placement and bandwidth scheduling. (d) Why work conservation should not be pursued.

Fig. 1. Motivation example. All the flows belonging to $C^{(1)}$ are drawn in white, and all flows in $C^{(2)}$ are drawn in blue.

Fig. 1(a) shows the case that we schedule bandwidth to these two coflows optimally, but with suboptimal reducer placement. In this case, the reducers for the largest flow in both coflows are placed on a host with only 1 Gbps incoming bandwidth. By scheduling coflows following the Shortest Remaining Time First (SRTF) principle to minimize the average CCT [8], the completion times of these two coflows are 2 s and 4 s, respectively. The average CCT is 3 s. For comparison, in Fig. 1(b), we place the reducers in an optimal way but let all flows share bandwidth equally. Both coflows finish in 3 s. Finally, we can optimize both reducer placement and coflow bandwidth scheduling using the optimal solution shown in Fig. 1(c). Here, the reducers to process the largest flows in each coflow are placed on the host with the largest incoming bandwidth, and we schedule these two coflows following the SRTF principle. Now, $C^{(1)}$ completes in 1.5 s and $C^{(2)}$ completes in 3 s. The average CCT is 2.25 s.

In this article, we propose and implement RPC, a joint online Reducer Placement and Coflow bandwidth scheduling framework, to minimize the average CCT. The key idea of RPC is to first optimize the completion time of each single coflow through reducer placement and flow transmission rate control. Then, the coflow with the shortest remaining time has the highest priority to occupy the bandwidth. In other words, we schedule all coflows following the SRTF principle (Section III).

Though it is NP-hard to minimize the CCT of a single coflow via reducer placement and flow transmission rate control, we develop a 2-approximation algorithm (Section IV). We use experiments on a real testbed and extensive large-scale simulations to show that RPC can reduce the average CCT by 64.98% compared with the state-of-the-art technologies (Section V and Section VI).

II. Background and System Model

A. Previous Works

There are a number of related works on task placement and coflow scheduling. We review those most closely related to our work.

Coflow Scheduling: Orchestra [5] is the first work to take the coflow concept into consideration when optimizing flow transfers in data centers. Varys [8] and Baraat [12] also apply the coflow concept in their network optimization. D-CAS [10] proposes a distributed coflow scheduling scheme, and Aalo [9] extends the coflow scheduling to scenarios where flow sizes are not known in advance. All these works assume that task placement has already been determined.

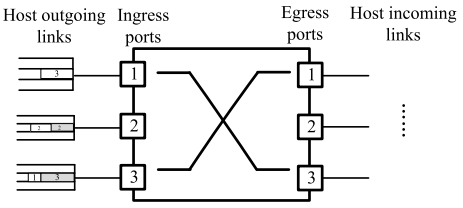


Fig. 2. Data center fabric with three ingress/egress ports.

Task Placement: Most task placement approaches try to maximize the data locality. Delay Scheduling [13] and Quincy [14] try to place tasks on the hosts or racks where most of their input data is located. ShuffleWatcher [17] attempts to localize map tasks of a job to one or a few racks, and to reduce cross-rack shuffling. But none of them take bandwidth scheduling into consideration.

Given a bandwidth scheduling algorithm, NEAT [18] chooses the best task placement for new requests. Without joint optimization, its performance is suboptimal, as we will show later in our evaluations. 2D-Placement [19] also leverages task placement to balance network load for future scheduling. It makes an impractical assumption that for a given job, the amount of traffic in the shuffle phase is known before the start of the map phase. Even if the amount of traffic in the shuffle phase is known in advance, we argue that the mapper placement should take into consideration the traffic from Distributed File System (DFS), *e.g.*, HDFS in Hadoop, to mappers. Our evaluations demonstrate that, even if we know the flow sizes in advance, without considering the traffic from DFS to mappers, the performance of optimizing both mapper and reducer placement can be even worse than that of NEAT, which only considers reducer placement.

B. System Model

Network Model: Given the recent progress in data center fabrics [20]–[22], we abstract the network as a giant non-blocking switch that interconnects all physical hosts (as shown

in Fig. 2). Such a nonblocking connection means that the inner links of the clusters would never experience congestion, and hence, as in many previous works [8], [9], [23], we assume network bottlenecks only exist on the last hop (incoming links to reducer hosts) or the first hop (outgoing links from mapper hosts). Therefore, we focus on how to schedule outgoing and incoming bandwidths, without paying attention to the flow routing or bandwidth scheduling inside the clusters. It should be noted that, since physical hosts in a cluster can be heterogeneous, or some of the bandwidth can be occupied by other applications, the available bandwidth on each link can be different.

Placement Model: Different from [19], which optimizes both mapper and reducer placement, we consider only reducer placement in RPC. There are at least three reasons for this setting. Firstly, the amount of traffic to be sent during a shuffle phase is hard to estimate before the completion of its corresponding map phase [24], [25]. Without this information, it is difficult to optimize reducer placement. Secondly, after the map phase (before the start of the shuffle phase), traffic information already exists in the log and meta-data files [26], [27], which provides sufficient information for optimizing reducer placement and bandwidth scheduling. Last but not least, even if we know the traffic information before the map phase starts, considering mapper placement would significantly increase the problem complexity. Mapper placement couples the sources of shuffle traffic with the destinations of the traffic from DFS. In addition, enumerating all the mapper placement possibilities would be computationally expensive for an online system. For example, when there are 10 mappers to be placed onto 10 hosts, there are up to 10^{10} possibilities to enumerate. Accordingly, taking both mapper and reducer placement into consideration is not suitable for an online system such as RPC.

Coflow Abstraction: After the map phase of a specific job, we can extract the corresponding coflow information from the log and meta-data files [26], [27], before its shuffle phase starts. Accordingly, we use vector $C^{(i)} =< v_1^{(i)}, v_2^{(i)}, \ldots, v_{K_i}^{(i)} >$ to denote the traffic requirement of coflow i, where $v_k^{(i)}$ is the amount of data that should be transferred by the k^{th} flow of coflow i and K_i is the number of flows belonging to coflow i. For simplicity, we also use $v_k^{(i)}$ to denote the k^{th} flow of coflow i. In addition, we use $D_n^{(i)}$ to denote the set of flows in coflow i that need to be fetched by the n^{th} reducer, and $k \in D_n^{(i)}$ means that $v_k^{(i)}$ should be fetched by $D_n^{(i)}$.

III. Design Overview

Given each coflow with information about its constituent flows, such as flow sizes and sources, RPC determines *where* to place the reducers, *when* to start and *at which rate* to serve each individual flow. Inspired by [5], [8], RPC works in a centralized, cooperative manner. This is also consistent with many recent centralized data center designs such as those in [2], [20], [21], [28], [29], etc.

At a high level, to achieve high scalability, RPC mainly orchestrates large coflows of data-intensive applications. It treats latency-sensitive individual flows and small coflows as background traffic, randomly places the reducers for the background traffic and sends out these flows with a high priority. A site broker periodically predicts the usage of background traffic on the incoming and outgoing links, and derives the available bandwidth for coflow scheduling.

Algorithm 1 The RPC Framework

Input: Uncompleted coflows Ω; available bandwidth B

1: Sort all the coflows in Ω non-increasingly according to their waiting time
2: **while** $\Omega \neq \Phi$ **do**
3: $T_{min} \leftarrow \infty$, $C_{min} \leftarrow \Phi$;
4: **for** $C \in \Omega$ **do**
5: Compute the minimum completion time for coflow C, T_C, reducer placement and bandwidth scheduling scheme to achieve T_C
6: **if** $C.waitTime() > \delta$ **then**
7: $T_{min} \leftarrow T_C$, $C_{min} \leftarrow C$;
8: break;
9: **end if**
10: **if** $T_C < T_{min}$ **then**
11: $T_{min} \leftarrow T_C$, $C_{min} \leftarrow C$;
12: **end if**
13: **end for**
14: $\Omega \leftarrow \Omega \setminus C_{min}$;
15: Assign all the flows in coflow C_{min} using reducer placement and bandwidth scheduling scheme derived in Line 5, and then update B;
16: **end while**

We describe the optimization framework of RPC with Algorithm 1, which is invoked whenever a new coflow comes or an existing flow finishes. More specifically, when a new coflow arrives (*e.g.*, when the map phase of a job completes and the flows for shuffling are ready), since the source of each flow is determined by the location of the mapper that generates it, RPC is invoked to compute its reducer placement and the transmission rate for its constituent flows. When an existing flow finishes, RPC is invoked to determine which flows should take up the released bandwidth. The underlying scheduling policy taken by RPC is SRTF [8], [11], [30].

The inputs of Algorithm 1 are all uncompleted coflows Ω and available bandwidth B. Even if a coflow is occupying the bandwidth in the network, it may be preempted if a "smaller" coflow arrives. In addition, if a coflow is partially served, its remaining volume should be updated when we recompute the bandwidth scheduling scheme. *It should be noted that when an individual flow starts, the location of the reducer to fetch this flow can no longer be changed.* To prevent starvation, RPC first schedules the coflows that have been waiting for a long time (Lines 6 – 9). Otherwise, it turns to the coflow with the shortest remaining completion time (Lines 10 – 12). When the coflow to be scheduled is selected, RPC sets up corresponding reducers, assigns bandwidth to its constituent flows and updates bandwidth utilization (Line 15). At this point, one of the uncompleted coflows has been scheduled and RPC continues to schedule the next one.

In most flow scheduling systems, work conservation is an important property to pursue. However, this is not the case when reducer placement is introduced as an optimization ingredient. When a host with a small incoming bandwidth is released for reducer placement, and a waiting reducer needs to fetch a large volume of data, placing such a waiting reducer on this host would bottleneck the entire coflow. It would be better to wait for another host with a larger incoming bandwidth to place the waiting reducer. Recall the example discussed in Fig. 1(c). At the time 1.5 s, the reducer on host M_3 completes and there is still one reducer that has not started. To pursue work conservation, we will place the reducer that has not started onto host M_3 to fully utilize network resources. As shown in Fig. 1(d), though the completion time of $C^{(1)}$ is still 1.5 s, pursuing work conservation would delay the completion time of $C^{(2)}$ to 3.5 s, which increases the average CCT. Accordingly, *work conservation is not an objective to pursue when we joint the reducer placement and coflow bandwidth scheduling to minimize the average CCT.*

The key algorithm in RPC is to calculate the minimum completion time for each coflow (Line 5). In the next section, we will discuss this algorithm in detail.

IV. Algorithm Details

In Section IV-A, we discuss how to minimize the completion time for a single coflow by jointly optimizing the reducer placement and bandwidth scheduling. After that, we analyze the approximation ratios of our proposed algorithms in Section IV-B. Since the available bandwidth is dynamically changing in practice, we discuss how to adjust the CCT for more efficient scheduling in Section IV-C. Finally, we discuss a few practical issues in RPC in Section IV-D.

A. Minimize Single Coflow Completion Time

In this section, we discuss the algorithms to minimize the completion time of a single coflow, which are the key elements in Algorithm 1. Since the algorithms we propose are based on a series of optimization models, we first present the relationship among these models, which can help understanding of how these algorithms work and their efficiency.

Firstly, we formulate the problem of minimizing the CCT of a single coflow in the most general way in (1), in which not only the incoming and outgoing bandwidths of each host can be varying, but every individual flow can also be sent with a time-varying rate. To address the problem of varying incoming and outgoing bandwidths, RPC considers the CCT under two extreme cases: 1) a coflow first waits for the completion of all previous ones, and then uses all the available bandwidth; and 2) a coflow only uses the remaining bandwidth at the time when it arrives to determine the reducer placement, and then bandwidth adjustment is introduced to calculate the minimum CCT. When considering these extreme cases, the incoming and outgoing bandwidths can be treated as constant in (2) – (6). After deep analysis, we find that even if every individual flow is transmitted with a time-invariant rate, the CCT can be minimized without any performance degradation. Accordingly, (1) is reformulated as (2). Since (2) is an Integer Linear Programming (ILP) which cannot be solved in a timely manner, we relax it as (3) to get an upper bound of the flow transmission rate.

The solution derived by (3) may not be feasible for our problem. Therefore, (4) is formulated to derive a feasible solution based on the flow transmission upper bound provided by (3). If we can get the optimal solution of (4), the optimal solution of (1) is also derived. Unfortunately, (4) is equivalent to (5), a classic unrelated parallel machine scheduling problem, which is known to be NP-hard. Hereby, we propose a 2-approximation algorithm (Algorithm 2) based on relaxation and rounding to solve it.

Based on all these optimization models, the algorithm to solve (1) when each host has constant incoming and outgoing bandwidths is summarized in Algorithm 3. Theorem 5 shows that the approximation ratio of Algorithm 3 is 2, while Theorem 6 ensures this approximation ratio is tight.

In the following, we present in detail the algorithms designed to minimize the completion time of a single coflow. Given the information of coflow i, we formulate the problem of minimizing its completion time as follows:

$$\text{minimize } T^{(i)} \tag{1}$$

$$\text{subject to } \sum_k r_{kj}^{(i)}(t) \le b_j^{in}(t), \quad \forall j,t \tag{1a}$$

$$\sum_j \sum_{k:s(i,k)=u} r_{kj}^{(i)}(t) \le b_u^{out}(t), \quad \forall u,t \tag{1b}$$

$$\sum_j \int_0^{T^{(i)}} r_{kj}^{(i)}(t)dt = v_k^{(i)}, \quad \forall k \tag{1c}$$

$$r_{kj}^{(i)}(t) \le x_{nj}^{(i)} b_j^{in}(t), \quad \forall j,k,t,n : k \in D_n^{(i)} \tag{1d}$$

$$\sum_j x_{nj}^{(i)} = 1, \quad \forall n,j \tag{1e}$$

$$x_{nj}^{(i)} \in \{0,1\}, \quad \forall j,n. \tag{1f}$$

The objective is to minimize the CCT of coflow i, which is denoted as $T^{(i)}$. With $r_{kj}^{(i)}(t)$ denoting the rate of sending the k^{th} flow in coflow i to host j at time t, and $b_j^{in}(t)$ denoting the remaining incoming bandwidth of host j at time t, the first constraint states that the entire rate of all the individual flows sent to host j cannot exceed the remaining incoming bandwidth of host j at any time. $(1b)$ is used to limit the rate of all the flows sent by host u, where $s(i,k)$ is the source host of the k^{th} flow in coflow i and $b_u^{out}(t)$ is the outgoing bandwidth limitation of host u at time t. With $v_k^{(i)}$ denoting the volume of the k^{th} flow in coflow i (also denoting this flow), $(1c)$ means that all the data of $v_k^{(i)}$ should be sent out before the coflow completes. In $(1d)$, $x_{nj}^{(i)}$ is a binary variable to denote if reducer $D_n^{(i)}$ is placed onto host j or not. This constraint states that a flow can only be sent to the host where its reducer is placed. $(1e)$–$(1f)$ are used to indicate that every reducer should be placed onto one and only one host.

Problem (1) is NP-hard even if all the flows are issued by the same host and the incoming and outgoing bandwidths of each host are constant [31]. It is difficult to solve for the following three reasons: 1) the remaining bandwidths (both incoming and

outgoing) on each host are time varying; 2) the upper bound of the integration in constraint (1c) is a variable; and 3) $x_{nj}^{(i)}$ is a binary variable. Hereafter, we present how to address these issues.

To address the first issue, we consider two extreme cases: 1) the coflow only uses the remaining bandwidth left by the coflows already scheduled, and can start transmission at once; and 2) the coflow can use all the available incoming and outgoing bandwidth, but it should wait for the completion of previous flows. In each of these two cases, we calculate the reducer placement and bandwidth scheduling scheme. Then, we adjust the bandwidth scheduling to derive a better minimum CCT estimation (see details in Section IV-C).

As to the second issue, we propose the following theorem:

Theorem 1: When the incoming and outgoing bandwidths of each host j are constant (denoted as b_j^{in} and b_j^{out}), suppose $\hat{r}_{kj}^{(i)}$ and $\hat{x}_{nj}^{(i)}$ are the optimal solutions, and $\hat{f}^{(i)}$ is the objective value of the following optimization problem:

$$\begin{aligned}
&\textit{maximize } f^{(i)} && (2)\\
&\textit{subject to } \sum_k r_{kj}^{(i)} \le b_j^{in}, \quad \forall j && (2a)\\
&\sum_j \sum_{k:s(i,k)=u} r_{kj}^{(i)} \le b_u^{out}, \quad \forall u && (2b)\\
&\sum_j r_{kj}^{(i)} = v_k^{(i)} f^{(i)}, \quad \forall k && (2c)\\
&r_{kj}^{(i)} \le x_{nj}^{(i)} b_j^{in}, \quad \forall j,k,n : k \in D_n^{(i)} && (2d)\\
&\sum_j x_{nj}^{(i)} = 1, \quad \forall n && (2e)\\
&x_{nj}^{(i)} \in \{0,1\}, \quad \forall n,j. && (2f)
\end{aligned}$$

Then, $T^{(i)} = \frac{1}{\hat{f}^{(i)}}$ is the optimal objective value of (1)*. $r_{kj}^{(i)}(t) = \begin{cases} \hat{r}_{kj}^{(i)} & for\ t \in (0, \frac{1}{\hat{f}^{(i)}}) \\ 0 & for\ t \in (\hat{f}^{(i)}, \infty) \end{cases}$ and $x_{nj}^{(i)} = \hat{x}_{nj}^{(i)}$ are the solutions to achieve the optimal objective value.*

Proof: Suppose T_{opt} is the optimal objective of (1), and $r_{kj}^{opt}(t)$ is the corresponding solution. By setting

$$r_{kj}^{(i)} = \frac{\int_0^{T_{opt}} r_{kj}^{opt}(t)dt}{T_{opt}},$$

we have

$$\sum_k \int_0^{T_{opt}} r_{kj}^{opt}(t)dt = \sum_k r_{kj}^{(i)} T_{opt}.$$

Since

$$\begin{aligned}
\sum_k \int_0^{T_{opt}} r_{kj}^{opt}(t)dt &= \int_0^{T_{opt}} \sum_k r_{kj}^{opt}(t)dt \\
&\le \int_0^{T_{opt}} b_j^{in} dt = T_{opt} b_j^{in},
\end{aligned}$$

we know $\sum_k r_{kj}^{(i)} \le b_j^{in}$. In the same way, we can verify that $r_{kj}^{(i)}$ also satisfies constraints (2b) and (2d).

For constraint (2c), we can see that

$$\sum_j \int_0^{T_{opt}} r_{kj}^{opt}(t)dt = \sum_j r_{kj}^{(i)} T_{opt} = v_k^{(i)}.$$

Let $f^{(i)} = \frac{1}{T_{opt}}$, and we get

$$\sum_j r_{kj}^{i} = \frac{v_k^{(i)}}{T_{opt}} = v_k^{(i)} f^{(i)}.$$

The discussion shows that $r_{kj}^{(i)} = \frac{\int_0^{T_{opt}} r_{kj}^{opt}(t)dt}{T_{opt}}$ and $f^{(i)} = \frac{1}{T_{opt}}$ are feasible solutions of (2). Therefore, we have

$$\hat{f}^{(i)} \ge \frac{1}{T_{opt}}.$$

In addition, we can easily verify that the variable settings claimed in Theorem 1 are feasible solutions of (1). Therefore, we have

$$T_{opt} \le \frac{1}{\hat{f}^{(i)}}.$$

Accordingly, $T_{opt} = \frac{1}{\hat{f}^{(i)}}$. ■

As $f^{(i)}$ is the inverse of the minimum CCT, we refer to it as the coflow transmission *frequency*. Theorem 1 shows that when the incoming and outgoing bandwidths of each host are constant, we can solve problem (2) instead of (1) to calculate the reducer placement and bandwidth scheduling for each individual flow. With (2), we can eliminate the variable in the upper bound of the integration in (1c)

However, there is still a binary variable in problem (2), *i.e.*, the third issue discussed above, which makes the problem intractable in large size systems. To address this issue, we first combine all the reducer hosts to be a single "big" host. Then, reducers are to be placed onto the unique "big" host, and hence the binary variable $x_{nj}^{(i)}$ is eliminated:

$$\begin{aligned}
&\text{maximize } f^{(i)} && (3)\\
&\text{subject to } \sum_k r_k^{(i)} \le \sum_j b_j^{in} && (3a)\\
&\sum_{k:s(i,k)=u} r_k^{(i)} \le b_u^{out}, \quad \forall u && (3b)\\
&r_k^{(i)} = v_k^{(i)} f^{(i)}, \quad \forall i. && (3c)
\end{aligned}$$

In this formulation, $r_k^{(i)}$ is the transmission rate of $v_k^{(i)}$. Now, problem (3) becomes a Linear Programming (LP) problem which is easy to solve. After solving problem (3), $r_k^{(i)}$ is the *upper bound* of the transmission rate of the k^{th} flow in coflow i, since it is a relaxation of the original problem by combining all the hosts that can be used for reducer placement as a single "big" host. When we place reducers onto different hosts, it is inevitable to scale down the flow transmission rate. Say the *scale-down ratio* is α; then, $v_k^{(i)}$ would be transmitted at the rate $r_k^{(i)}/\alpha$ in the final solution. Hence, we should

minimize such a scale-down ratio to reduce the CCT:

$$\text{minimize } \alpha \tag{4}$$

$$\text{subject to } \sum_{n}\Big(\sum_{k\in D_n^{(i)}} r_k^{(i)}\Big)x_{nj}^{(i)} \le \alpha b_j^{in}, \quad \forall j \tag{4a}$$

$$\sum_{j} x_{nj}^{(i)} = 1, \quad \forall n \tag{4b}$$

$$x_{nj}^{(i)} \in \{0,1\}, \quad \forall n, j. \tag{4c}$$

It should be noted that, in (4), $r_k^{(i)}$ is a constant parameter, which is derived by solving (3). By defining $e_{nj}^{(i)} = \sum_{k\in D_n^{(i)}} r_k^{(i)}/b_j^{(in)}$, (4) can be modified as

$$\text{minimize } \alpha \tag{5}$$

$$\text{subject to } \sum_{n} e_{nj}^{(i)} x_{nj}^{(i)} \le \alpha, \quad \forall j$$

$$(4b), (4c). \tag{5a}$$

This is a classic unrelated parallel machine scheduling problem, which is NP-hard. To derive a high-performance solution efficiently, we try to solve it based on a relaxation and rounding method [32]. We first relax the binary variable constraint on $x_{kj}^{(i)}$ and get

$$\text{minimize } \alpha \tag{6}$$

$$\sum_{n\in E_j(\alpha)} e_{nj}^{(i)} x_{nj}^{(i)} \le \alpha, \quad \forall j \tag{6a}$$

$$\sum_{j\in H_n(\alpha)} x_{nj}^{(i)} = 1, \quad \forall n \tag{6b}$$

$$x_{nj}^{(i)} \ge 0, \quad \forall n, j, \tag{6c}$$

where $E_j(\alpha)$ is the set of reducers $\{n|e_{nj}^{(i)} \le \alpha\}$, and $H_n(\alpha)$ is the set of hosts $\{j|e_{nj}^{(i)} \le \alpha\}$. For a fixed α, (6) is an LP model. Thus it can be solved via binary search with logarithmic iterations. Each iteration can be completed with polynomial time complexity.

By solving (6), we may get a fractional reducer placement solution, which is infeasible to the original problem. Accordingly, we should round the fractional solution to derive a feasible reducer placement scheme. To this end, we first propose the following lemma:

Lemma 2: Suppose there are N reducers and M hosts for reducer placement, then at most $(N + M)$ variables will be non-zero in the optimal solution of (6)*.*

Proof: The objective α is the minimum value that makes (6) feasible. When setting α to its optimal value, the feasible region is a single point determined by v linearly independent rows of the constraint matrix such that all of these constraints hold with the equality, where v is the number of variables in (6) for the given α.

Consider that there are $v+M+N$ constraints in (6), but only M constraints in (6a) and N constraints in (6b). Accordingly, there are at least $v-N-M$ constraints in (6c) that hold with the equality. Therefore, at most $N + M$ constraints in (6c) do not hold with equality, which means that at most $N + M$ variables have non-zero values. ■

From Lemma 2, we can get the following corollary.

Corollary 3: We construct a bigraph $G(x) = \{U, V, E\}$ according to the solution of (6)*, x, where $U = \{u_1, u_2, \ldots, u_M\}$ is the set of nodes denoting hosts, called host nodes, while $V = \{v_1, v_2, \ldots, v_N\}$ is the set of nodes denoting reducers, called reducer nodes. There is an edge between v_n and u_j, iff $x_{nj}^{(i)} > 0$. In this case, any connected component, P, in $G(x)$ can be modified to a pseudo tree (a tree or a tree plus one edge) without increasing the scale-down ratio.*

For brevity, hereafter, we refer to "reducer/host v" instead of "the reducer/host associated with node v". *Proof:* If we solve (6) by only using the reducers and hosts associated with P, say the solution is x', it is obvious that the scale-down ratio is smaller than or equal to that derived by using all the reducers and hosts. According to Lemma 2, the number of non-zero variables in the solution is at most the number of nodes in P. Therefore, P can be modified to a pseudo tree by changing the edges according to x'. ■

Based on Corollary 3, Algorithm 2 is designed for reducer placement. Line 2 handles the reducers with only one host to place according to the solution of (6). After that, each reducer node in P has at least two node degrees. For each connected component $P \in BG$, if $|N(P)| = |L(P)|$, where $N(P)$ is the set of nodes in P and $L(P)$ is the set of edges in P, there must be a cycle in P. In this case, we first find out this cycle with Depth-First Search (DFS), and determine the reducer placement on this cycle (Line 6).

By removing this cycle from P, there must remain a forest of trees, each of which contains at most one reducer leaf node. If there is a reducer leaf node in the resulting tree, we can root the tree at this reducer leaf node, and place the reducer onto its child host that serves the largest fraction of this reducer. Otherwise, we root the tree at an arbitrary reducer node and place each reducer onto the child host that serves the largest fraction of this reducer (Lines 8–11). In this way, *each host serves at most one reducer that is fractionally placed onto multiple hosts according to the solution of* (6).

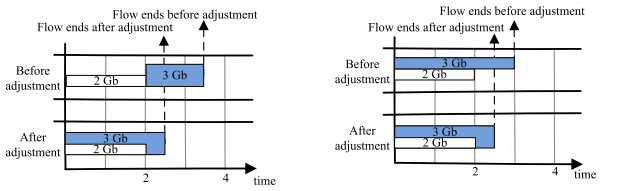
With Algorithm 2, we can find the reducer placement that can minimize the CCT. However, it does not determine the transmission rate of each individual flow. Algorithm 3 leverages Algorithm 2 to determine the reducer placement and bandwidth scheduling. In this algorithm, we first calculate the maximum coflow transmission frequency by combining all the hosts into a "big" one (Line 1). Then, we place reducers onto hosts according to Algorithm 2 (Line 2). Based on the placement, we calculate the maximum transmission rate of each individual flow in Lines 3–8. It should be noted that in Lines 5–8, we only scale down the flow transmission rate if the scale-down ratio is larger than 1. For a flow with a scale-down ratio smaller than 1, we need to increase its transmission rate to fully utilize the bandwidth at the reducer host side. However, we should note that the bandwidth at the mapper host side has been fully utilized before we increase the flow transmission rate, and hence RPC cannot increase the flow transmission rate when the scale-down ratio is smaller than 1.

Algorithm 2 Reducer Placement

Input: The solution of problem (6), $\{x_{nj}^{(i)}\}$
Output: Reducer placement
1: Construct a bigraph BG according to $\{x_{nj}^{(i)}\}$ as in Corollary 3
2: Remove all the reducer nodes with only one node degree and place these reducers onto the connecting host
3: **for** all connected components $P \in BG$ **do**
4: **if** $|N(P)| = |L(P)|$ **then**
5: Find the unique cycle in P with Depth-First Search
6: Arbitrarily orient the cycle in one direction and place each reducer onto the host succeeding it on the cycle
7: Remove this cycle from P, and what remains overall is a forest of trees, each of which contains at most one reducer leaf node
8: **for** all the remaining trees **do**
9: Rooting at the unique reducer leaf node (if there is), or arbitrary reducer node
10: Place each reducer onto its child host that serves the largest fraction of this reducer
11: **end for**
12: **else**
13: Treat an arbitrary reducer as the root to form a tree and place each reducer onto its child host that services most of this reducer
14: **end if**
15: **end for**

Algorithm 3 Minimize CCT Through Reducer Placement and Coflow Bandwidth Scheduling

Input: The size of individual flows $v_k^{(i)}$, incoming/outgoing bandwidth of each host $\{b_j^{in}\}$ and $\{b_j^{out}\}$
Output: Reducer placement $x_{nj}^{(i)}$ and flow transmission rate $\{r_{kj}^{(i)}\}$
1: Formulate and solve (3) and get the maximum transmission rate $r_k^{(i)}$
2: Based on the solution of (3), formulate model (4) and solve it with Algorithm 2, say the solution is $x_{nj}^{(i)}$
3: **for** all host j **do**
4: $r_{kj}^{(i)} \leftarrow r_k^{(i)} x_{nj}^{(i)}|_{n:k\in D_n^{(i)}}$, $\alpha_j \leftarrow \frac{\sum_k r_{kj}^{(i)}}{b_j^{in}}$
5: **if** $\alpha_j > 1$ **then**
6: $r_{kj}^{(i)} \leftarrow \frac{r_{kj}^{(i)}}{\alpha_j}$
7: **end if**
8: **end for**
9: **return** $x_{kj}^{(i)}$ and $\{r_{kj}^{(i)}\}$



(a) Calculate CCT based on full bandwidth (b) Calculate CCT based on remaining bandwidth

Fig. 3. Flow completion time adjustment.

B. Approximation Bound Analysis

In this section, we analyze the approximation ratios of the algorithms we proposed to optimize the CCT of a single coflow. However, before we present the approximation ratio of Algorithm 3, we first analyze the approximation ratio of Algorithm 2, which is an important component to calculate the reducer placement.

Theorem 4: The approximation ratio of Algorithm 2 is 2.

Proof: Let α_{min} be the optimal objective value of problem (6), which is a lower bound of the optimal objective value of problem (4). In the Line 2 of Algorithm 2, we place all the unsplit reducers according to the solution of (6). All these unsplit reducers contribute to the scale-down ratio by less than α_{min}. In Lines 3–15, Algorithm 2 ensures that every host serves at most one split reducer, which leads to a scale-down ratio increase of at most α_{min}. Accordingly, the scale-down ratio derived by Algorithm 2 is at most $2\alpha_{min}$. ■

Theorem 5: The approximation ratio of Algorithm 3 is 2.

Proof: There are two cascading bottlenecks in our system, namely, the outgoing links from mapper hosts and the incoming links to reducer hosts. When we solve problem (3), we can get the optimal solution which solves the bottleneck at the mapper side. Then, Algorithm 3 scales down the flow transmission rate to solve the bottleneck at the reducer side, which results in an approximation factor loss of 2. Accordingly, the approximation ratio of Algorithm 3 is 2. ■

Theorem 6: The approximation bounds for both Algorithm 2 and 3 are tight.

Proof: For Algorithm 2, suppose there are $M(M-1)+1$ reducers (reducer 0 to reducer $M(M-1)$) and M hosts (host 0 to host $M-1$) that can be used to place the reducers. Suppose for the reducers $0 \le n \le M(M-1)-1$, we have $e_{nj}^{(i)} = T$ for all j and for the one remaining reducer, we have $e_{M(M-1),j}^{(i)} = MT$ for all j. The optimal solutions to problem (5) are $x_{M(M-1),M-1}^{(i)} = 1$ and $x_{n,n \mod (M-1)}^{(i)} = 1$; otherwise $x_{nj}^{(i)} = 0$, with the objective value of MT. However, the optimal solutions of (6) are $x_{M(M-1),j}^{(i)} = 1/M$ and $x_{n,n \mod M}^{(i)} = 1$; otherwise $x_{nj}^{(i)} = 0$. After the rounding procedure of Algorithm 2, the objective value is $(2M-1)T$. The approximation ratio is $\frac{2M-1}{M}$. When M approaches infinity, the approximation ratio approaches 2. Accordingly, the approximation bound for Algorithm 2 is tight.

For Algorithm 3, we can see that the only step that introduces an approximation is leveraging Algorithm 2 to determine reducer placement. Accordingly, we can conclude that the approximation bound for Algorithm 3 is also tight. ■

C. Coflow Completion Time Adjustment

In the last subsection, we calculated the CCT of a coflow under the assumption that remaining incoming and outgoing bandwidths are constant. However, this is not the case in

practice. To solve this problem, we try two extreme cases with Algorithm 3. The first one is that we assume every coflow waits for the completion of previous ones, and transmits with all the available bandwidth; the second one is that the coflow starts at once, but each flow only uses the remaining bandwidth when it arrives.

However, neither method fully utilizes the bandwidth. For the first case, we can first transmit the flow at a smaller transmission rate. As shown in Fig. 3(a), say the incoming bandwidth is 2 Gbps and there has already been a flow scheduled with 1 Gbps from 0 s to 2 s. If the later flow waits for the completion of the previous one, it will end in 3.5 s. Actually, we can transmit the later flow with the 1 Gbps remaining bandwidth from 0 s to 2 s, and transmit it at 2 Gbps after the first flow ends, the later flow will end in 2.5 s.

If we calculate the CCT only with the remaining bandwidth, the flow scheduling may be as shown in Fig. 3(b). Actually, we can increase the flow transmission rate when previous flows end. After adjusting the transmission rate of all the individual flows in a coflow, we set the completion time of the latest flow as the CCT. The bandwidth adjustment can be done with the classic water-filling algorithm [33]. After the bandwidth adjustment, the smaller CCT will be used to determine which coflow has the highest priority to be scheduled.

D. Discussions

Started Reducer: Since RPC updates the scheduling scheme when a coflow arrives or a flow completes, some of the reducers may have already started on a certain host. If reducer n has already started on host j, we add a constraint $x_{nj}^{(i)} = 1$ into the problem (4). None of the algorithms to calculate the CCT of coflow i nor the approximation ratios will change.

Reducer Number Constraint: Sometimes, a host can support only a few reducers in practice. In most cases, this would not be a problem since we distribute the reducers among as many hosts as we can to reduce the CCT, and hence not too many reducers would be placed onto the same host. Even if we need to limit the number of reducers per host, we can monitor the number of reducers that have already been placed onto each host. When the number of reducers placed on host j meets the capacity constraint, we fix all the reducers whose placement has already been determined, and then invoke Algorithm 2 once more. In order to prevent more reducers being placed onto host j, we set $x_{nj}^{(i)} = 0$ in all the associate optimization models for all yet to be placed reducer n.

Local Reducer: When the host issuing flows can also be used to allocate reducers, we first place reducers onto the hosts that contain most of their required data. If no such hosts exist, we still use Algorithm 2 to calculate the reducer placement.

Multi-Wave Reducers: When there are not enough hosts to place reducers for a single coflow, the reducers will be executed in a multi-wave fashion [25]. Because the CCT is determined by the completion of the shuffle traffic associated with the last wave of reducers, we have at least two solutions to deal with this problem. First, we can place all but the last wave of reducers and transmit the corresponding flows by pursuing work conservation, and only optimize the reducer placement and bandwidth scheduling associated with the last wave of reducers. Second, we can treat the flows belonging to one wave of reducers as a coflow, and leverage our proposed algorithms to minimize the average CCT.

System Overhead: There are five types of potential overheads introduced by RPC: 1) waiting for all the mappers to finish; 2) collecting flow size information; 3) solving the optimization problem; 4) placing and removing reducers; and 5) controlling flow rates. In current multi-job systems, such as Spark and Hadoop 2.x, the shuffle phase of each job will not start until all its associated mappers complete, and hence RPC would not introduce overhead to wait for the completion of the map phase. To collect flow size information, the overhead should be several RTTs in a cluster, which is typically hundreds of micro-seconds [34]. As we will see in Section VI, the running time of Algorithm 1 is hundreds of micro-seconds in a system with 500 mappers/reducers. With container technology, the time to set up or remove a reducer is several micro-seconds [35]. With Linux Traffic Control (TC), the overhead to shape the flow rate is also in the micro-second range [36]. In summary, the total overhead introduced by RPC would be at most several milliseconds. Since RPC only operates on large coflows, whose completion time should be at least hundreds of milliseconds, the overhead can be considered negligible.

Cost of Pursuing Minimum Average CCT: To pursue the minimum average CCT in the system, we hold some of the coflows that should be sent earlier in other scheduling schemes such as Varys [8], NEAT [18], and 2D-Placement [19]. This inevitably prolongs the completion time of some coflows. However, we can imagine that this would not be a serious problem. Firstly, the small coflows that are delivered first will complete very soon, and hence it would not have a significant negative effect on the coflows that are held. Secondly, when multiple coflows are sharing the same incoming/outgoing link, transmitting coflows sequentially could improve the performance of the small ones without degrading that of the large ones. Last but not least, a better reducer placement scheme could discount the offside effect of holding the large coflows. Actually, in Section VI-B, we will see that only a small fraction of coflows (usually $< 10\%$) suffer the CCT increase in RPC.

V. Implementation

We implement RPC on a testbed with seven hosts. One of them works as a job scheduler to execute the algorithms in RPC, while the remaining six are used to transfer coflows. All of these hosts are connected by a switch, and the NIC bandwidth on each host is 1 Gbps.

Our scheduling algorithms are implemented on the scheduler with CPLEX 12.3 as an LP solver. Whenever a coflow is issued, the corresponding hosts will notify the scheduler through the coflow API [5]. The scheduler derives a reducer placement and bandwidth scheduling scheme, and responds to the hosts. Whenever a host receives the signal from the scheduler, it sets up corresponding reducers to fetch data.

In addition, a Bandwidth Enforcement (BE) kernel module is implemented between the TCP/IP stack and Linux Traffic

Control (TC). This module includes a netfilter hook, a flow table, and a packet modifier. An enforcement daemon at the user space communicates with the BE module via the ioctl to manage the flow table. Based on the rules in the flow table, the packet modifier leverages the netfilter hook to intercept all outgoing packets and modifies the nfmark field of the socket buffer. Then, the modified packets are sent to the TC for rate limiting. A two-level Hierarchical Token Bucket (HTB) is used in the TC. The root node classifies packets to their corresponding leaf nodes based on the nfmark field, and the leaf nodes enforce per-flow rates.

In our prototype, we have the Linux root privileges, and hence we can control the TC tool. If we do not have the root privileges, we can also realize such a rate limitation by controlling the rate to write data into the socket buffer.

VI. Performance Evaluation

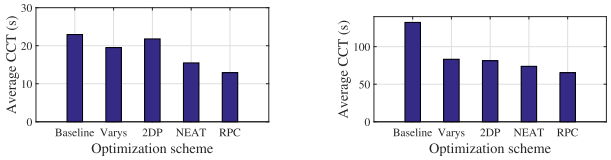
We evaluate RPC through two small-scale testbed experiments as well as extensive large-scale simulations. We compare the following schemes with RPC:

- **Baseline:** All the reducers are randomly placed and all the flows are sharing the bandwidth fairly.
- **Scheduling-only (Varys):** All the reducers are randomly placed, but coflows are scheduled according to SRTF. This is the state-of-the-art scheduling scheme Varys [8].
- **Scheduling-aware reducer placement (NEAT):** Given the scheduling scheme SRTF, the reducers are placed onto the hosts that can minimize the impact on the completion time of other coflows. This exactly follows the concept of NEAT [18].
- **Placement of both mapper and reducer (2D-Placement, abbreviated as 2DP in all the figures):** Both mappers and reducers are placed following 2D-Placement [19] assuming the size of each coflow is known before the completion of associated mappers. Different from [19], the data transmission from DFS to mappers is counted in the CCT. All the coflows are scheduled following SRTF.

Metrics: In this section, we define the performance improvement of scheme 1 compared with scheme 2 as $\frac{CCT_2-CCT_1}{CCT_2}$, where CCT_1 and CCT_2 are the average CCT derived by scheme 1 and scheme 2, respectively.

The **summary of the main results** is as follows:

- Through the experiments on the small-scale testbed, we observe that RPC reduces the average CCT by 43.76% and 16.67% compared with the baseline and NEAT, respectively.
- Compared with the random reducer placement, optimizing the reducer placement reduces the average CCT by more than 99% in a heterogeneous environment.
- RPC could reduce the average CCT by up to 64.98%, even compared with the state-of-the-art technology that takes both task placement and coflow scheduling into consideration, namely, NEAT.
- To pursue the minimum average CCT, RPC prolongs the CCT of less than 10% of the coflows.



(a) Average CCT based on synthesized jobs. (b) Average CCT based on real jobs.

Fig. 4. Testbed experiment results.

A. Testbed Experiments

In our experiments, we use three of the hosts as coflow senders and place reducers on the remaining three hosts. To emulate the heterogeneous hosts, we limit the bandwidths of two of the switch ports connecting to the receivers (server D and E) to be 500 Mbps. To study the performance of RPC in detail, we first inject three coflows into the network and assume every flow should be processed by a specific reducer. The coflow information is shown in Tab. I. For comparison purpose, we also evaluate the performances derived by the baseline, Varys, NEAT, and 2D-Placement, respectively. The results of this experiment are shown in Fig. 4(a). From this figure, we can see that RPC can save $\frac{22.94-12.9}{22.94} = 43.76\%$ of the average CCT compared with the baseline scheme, and it can reduce the average CCT by $\frac{15.48-12.9}{15.48} = 16.67\%$ compared with NEAT on our small-scale testbed. An interesting observation is that NEAT achieves an average CCT that is $\frac{21.79-15.48}{21.79} = 28.96\%$ smaller than 2D-Placement; *i.e.*, optimizing the placement of both mappers and reducers yields a worse solution than only optimizing the placement of reducers.

To study the reasons behind the above observations, we show the reducer placement results in Tab. II. In the results of 2D-Placement, the placement result C→E means placing the mapper onto server C, while placing the reducer onto server E. Beside each placement result in Tab. II, we present the completion time of the corresponding flow in the parentheses. Comparing RPC with NEAT, we can see that by reducing the impact to future coflows, NEAT reduces the completion time of coflow 2 (from 12.90 s to 12.04 s compared with RPC), but it does not optimize the average CCT in the global view. When coflow 3 is taken into consideration, the average CCT increases.

Comparing NEAT with 2D-Placement, we can see that if we only consider the time for the data transfer between mappers and reducers, the average completion time induced by 2D-Placement is (8.6+17.2+18.93)/3=14.91 s, which outperforms NEAT. However, since we need to transfer data from DFS to mappers, the CCT increases. We should also note that since 2D-Placement solves the flow contentions, it fully utilizes the network bandwidth, and hence it serves all the flows quickest if the time to transfer data from DFS to mappers is not taken into account.

For Varys, the random reducer placement may place a reducer receiving large flows onto a server with a small incoming bandwidth, *e.g.*, placing the reducer of flow 2 in coflow 1 onto server D, which results in a large CCT. As for

TABLE I
COFLOWS INJECTED INTO NETWORK

Coflow ID	Flow ID	Flow Volume	Source host
Coflow 1	Flow 1	200 MB	A
	Flow 2	500 MB	B
Coflow 2	Flow 1	500 MB	A
	Flow 2	1 GB	C
Coflow 3	Flow 1	1 GB	B
	Flow 2	1 GB	C

the baseline scheme, though it may get a better reducer placement than Varys with some probability, *i.e.*, the completion time of the last coflow is smaller, it still suffers a larger average CCT as all the flows are sharing the bandwidth equally, which hurts the completion time of all the coflows.

In addition to the above synthesized workload, we also choose 10 jobs from the Facebook MapReduce traffic trace in [37] to test the performance of RPC on our testbed. Since we only have three hosts for the coflow senders and three hosts for the reducer placement, each job is divided into nine flows among three mappers and three reducers, and all these flows belong to the same coflow. Repeating the above experiment based on the real traffic trace, we get the results shown in Fig. 4(b). Compared with the synthesized traffic results, we can see that the baseline scheme performs worse, since more coflows are sharing the bandwidth. We can also observe that the performance improvement of RPC compared with NEAT decreases from 16.67% to $\frac{73.84-65.32}{73.84} = 11.54\%$. This is due to the definition of the performance improvement. When the CCT increases, the value of the performance improvement decreases if the CCT reduction stays the same. In fact, the average CCT reduced by RPC increases from 2.58 s to 8.52 s.

B. Large-Scale Simulations

Simulation Methodology: Similar to [8], [29], we build a flow-level simulator, which records flow arrival and departure events. Whenever such an event occurs, the simulator not only updates the remaining amount of each existing flow, but also invokes the algorithms we proposed to calculate the reducer placement for newly arrived coflows and update the flow transmission rate. To solve LP models in RPC, we embed the API provided by CPLEX 12.3 into our simulator.

In the simulations, we use the Facebook MapReduce traffic trace provided in [37]. Since our system is applied to data-intensive applications, we pick out all 96 jobs whose shuffle traffic amount is more than 20 Gbits. Based on the traffic amount distribution, we generate 1000 candidate jobs to inject into the system. Given the number of mapper hosts, we randomly split the shuffle traffic onto these hosts and generate coflows. Accordingly, the more mapper hosts that are in the system, the more flows there are in a coflow, and correspondingly, the average size of these flows will be smaller. We assume the NIC bandwidth on each host is one of the values in {0.1, 0.2, 0.5, 1, 10, 40} Gbps. We set such a heterogeneous NIC bandwidth for two reasons: 1) in a cluster, the hosts should be deployed with different kinds of NICs, with different bandwidths; and 2) the network is shared by multiple applications. Some of the bandwidth could be assigned to other applications. We keep the hosts with a small bandwidth in the system, since most of the flows in a cluster are very small. The hosts with a small bandwidth can be used to serve the small flows, and their computational resources can be fully utilized. In addition, a good algorithm should avoid placing a reducer that needs to receive large flows onto those hosts with a small bandwidth. This setting can be used to show the advantage of RPC. Since the flow source and host NIC bandwidth distributions may impact the reducer placement and coflow bandwidth scheduling, the simulation results in this section are averaged over 20 trials. The overall simulation results are shown in Fig. 5–7. In general, we can see that RPC outperforms all other schemes in all scenarios. The baseline scheme performs worst as there is no optimization in it, while Varys performs better than only the baseline scheme since it absolutely misses optimizing the mapper/reducer placement. By introducing the placement of mappers and reducers, 2D-Placement outperforms Varys. However, since it introduces additional data transfer into the system, its performance is worse than that of NEAT.

Impact of Coflow Width: The coflow width is defined as the number of flows in a coflow [8]. In each simulation round, we change the number of mapper and reducer hosts (keep the number of mappers and reducers the same) in the system and observe how the average CCT changes with the number of mapper/reducer hosts in the system. The more hosts in the system, the more flows the shuffle data should be split into, and hence the wider the coflows are. In addition, we assume the coflow arrival rate is 20 coflows/second.

The simulation results are shown in Fig. 5, and we make the following observations. First, RPC outperforms the schemes without optimizing reducer placement, *i.e.*, baseline and Varys, by more than 99%. This is because a suboptimal reducer placement will result in an extremely large completion time for some coflows, especially when they place a reducer receiving large flows onto a host with a small incoming bandwidth.

Second, 2D-Placement leads to about 2–3x the average CCT that NEAT achieves. For a given mapper placement, 2D-Placement is almost the same as NEAT. However, to optimize the mapper location in 2D-Placement, additional data transfer is required, and this data transfer phase is not optimized. Accordingly, it results in an average CCT more than 2x that of NEAT.

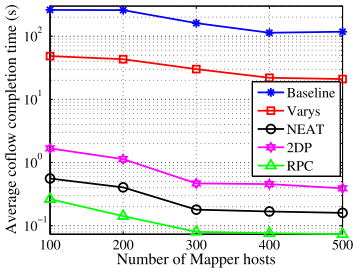
Third, compared with NEAT, RPC reduces the average CCT by 46.52%–64.98%. In order to consider the impact on other coflows, NEAT optimizes the placement of reducers one by one, rather than optimizing reducer placement from the global perspective. Accordingly, it derives a performance worse than RPC.

Fourth, regardless of which scheme is adopted, the average CCT reduces with the increase of coflow width. This observation is intuitive since, when there are more hosts, more bandwidth can be used to serve flows.

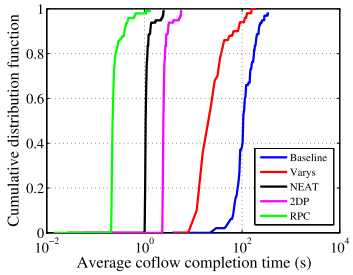
Last, Fig. 5(b) shows the CDF of CCTs when there are 500 mapper hosts and 500 reducer hosts in the system. From

TABLE II
REDUCER PLACEMENT

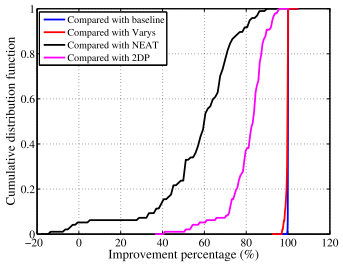
Coflow ID	Flow ID	RPC	NEAT	2DP	Varys	Baseline
Coflow 1	Flow 1	D (3.44)	D (3.44)	C $\rightarrow$ F (1.72)	D (3.44)	F (3.44)
	Flow 2	F (4.30)	E (8.60)	B $\rightarrow$ E (8.60)	F (8.60)	D (17.20)
Coflow 2	Flow 1	D (12.04)	D (12.04)	B $\rightarrow$ E (17.20)	F (2.58)	E (17.20)
	Flow 2	F (12.90)	F (12.04)	C $\rightarrow$ F (10.32)	D (20.65)	E (25.80)
Coflow 3	Flow 1	E (21.51)	E (17.20)	A $\rightarrow$ D (17.20)	E (25.81)	F (25.80)
	Flow 2	F (21.51)	F (25.80)	C $\rightarrow$ F (18.93)	F (29.25)	D (25.81)



(a) Average CCT.

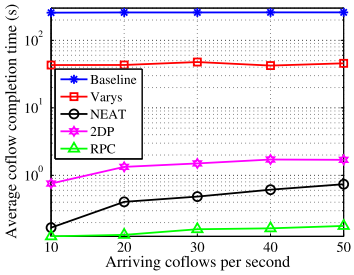


(b) CDF of CCTs with 500 mapper hosts.

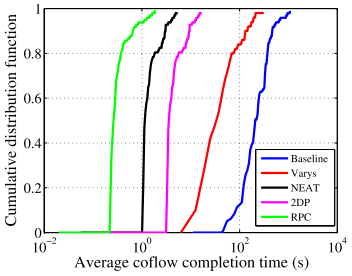


(c) CDF of performance improvements of RPC with 500 mapper hosts.

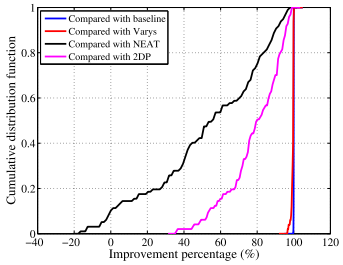
Fig. 5. The impact of coflow width.



(a) Average CCT.



(b) CDF of CCTs when the average arrival rate is 50 coflows per second.

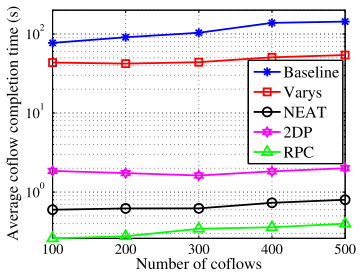


(c) CDF of performance improvements of RPC when the average arrival rate is 50 coflows per second.

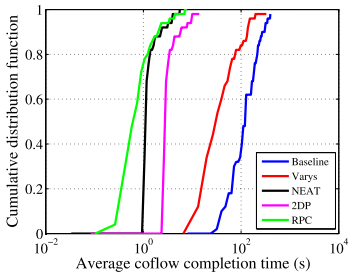
Fig. 6. The impact of coflow arrival rate.

this figure, we can see that RPC does not result in a long-tail effect on the CDF of CCTs. This means that RPC optimizes the average CCT without significantly sacrificing the CCT of some individual coflows. The simulation results in Fig. 5(c) again demonstrate this conclusion. From this figure, we can see that even compared with NEAT, which performs the best among all the comparison schemes, less than 10% of the coflows suffer the CCT increase and the maximum performance degradation is about 15%. However, more than half of the coflows enjoy a performance improvement larger than 50%.

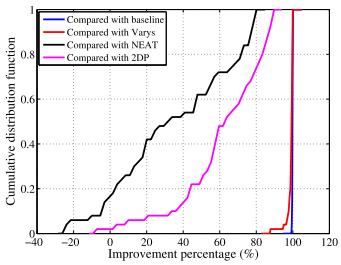
Impact of Coflow Arrival Rate: To investigate the impact of coflow arrival rate, we send out 1000 coflows into a system consisting of 200 mapper hosts and 200 reducer hosts, and observe the relationship between the average CCT and the coflow arrival rate. We investigate the coflow arrival rate from 10 coflows/second to 50 coflows/second, since if every coflow can monopolize the network, the average CCT is 0.141 s, and there is almost no remaining bandwidth in the network when the coflow arrival rate is 50 coflows/second.



(a) Average CCT.



(b) CDF of CCTs with 500 coflows.



(c) CDF of performance improvements of RPC with 500 coflows.

Fig. 7. The impact of coflow number.

From Fig. 6, we make the following observations. Firstly, the average CCT increases with the coflow arrival rate. When more coflows are arriving in a specific interval, there are more coflows queuing in the system as there are not enough resources to deal with the coflows immediately when they arrive. This results in a larger average CCT.

Secondly, as we have discussed above, NEAT outperforms 2D-Placement by 2–3x when the coflow arrival rate changes. However, with the increase of the coflow arrival rate, the performance gap between NEAT and 2D-Placement decreases. When the coflow arrival rate increases, more coflows are queuing in the system, and this results in a longer completion time for the queuing coflows. Accordingly, the percentage of time required to transfer data from DFS to mappers is relatively reduced.

Thirdly, when the coflow arrival rate is small, the performance of NEAT and RPC is close. With the increase of the coflow arrival rate, the performance gap between these two schemes also increases. When the coflow arrival rate is small, the later coflow comes when the previous one has almost completed. Both schemes optimize the average CCT by placing the reducers receiving larger flows onto the hosts with larger bandwidth. Accordingly, they derive similar performance. When the coflow arrival rate is large, there are more coflows in the network and we should carefully schedule the bandwidth to different flows. Hence, RPC derives a better performance.

Fourthly, comparing Fig. 6(b) with Fig. 5(b), we can see that with a larger coflow arrival rate, the CCT spreads in a wider interval. This is because more coflows queuing in the system results in a longer completion time for the large coflows, while it does not impact the completion time of small coflows as RPC schedules coflows following the SRTF principle.

Lastly, comparing Fig. 6(c) with Fig. 5(c), a larger coflow arrival rate results in performance degradation to more coflows, and the largest performance degradation also gets worse. However, such performance degradation is fairly slight.

Impact of Coflow Number: To investigate how the performance of RPC is influenced by the number of coflows in the system, we assume there are 200 mapper hosts and 200 reducer hosts and inject different numbers of coflows into the network simultaneously.

The simulation results are shown in Fig. 7. From this figure, we make the following observations. First, the average CCT increases with the number of coflows in the system. This is obvious because, as explained above, more coflows are injected into the system simultaneously, so there will be more coflows queuing in the system, which results in a larger average CCT. Furthermore, RPC can reduce the average CCT by up to 56.52% compared with NEAT.

Second, the performance gap between NEAT and 2D-Placement slightly reduces with the increase of the number of coflows in the system. This is again because a longer queuing delay mitigates the impact of data transfer between DFS and mappers.

Third, from Fig. 7(b), we can see that the largest CCT under RPC is similar to that under NEAT. This is because when the system is heavily loaded and all the coflows arrive simultaneously, the largest coflow cannot be sent out till all other coflows complete or its waiting time exceeds the threshold. Therefore, the largest coflow will wait for almost the same amount of time before the system starts to serve it under both schemes. Though RPC and 2D-Placement can still reduce the completion time of the largest coflow, the queuing time dominates the CCT, and hence the largest CCTs under both schemes are similar.

In Fig. 7(c), we can see that even though all the coflows arrive simultaneously, *i.e.*, with the maximum work intensity, the CCT of only about 20% of the coflows increases. This shows that RPC can improve the average CCT without incurring performance degradation to too many coflows in the system.

Takeaways: To reduce the average CCT in a heterogeneous environment, careful reducer placement is very important. In addition, we should treat each coflow as a whole to optimize the average CCT, and it is necessary to jointly optimize reducer placement and bandwidth scheduling.

Work Conservation Issue: In RPC, we propose not to pursue work conservation when we optimize the average CCT. This is a proposition different from most of the previous works. We verify this strategy through simulations. To this end,

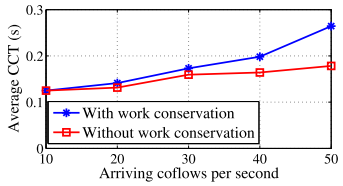


Fig. 8. Impact of pursuing work conservation.

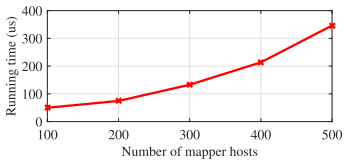


Fig. 9. Algorithm running time.

we always serve more flows if there is remaining bandwidth. To benefit the completion of a coflow, the flows with a larger size have a higher priority to get the idle bandwidth. A performance comparison between the schemes with and without pursuing work conservation is shown in Fig. 8.

We can see that when the system is lightly loaded, RPC achieves similar performance regardless of whether it pursues work conservation or not. This is because there are very few coflows sharing the bandwidth and the reducers for every coflow can be optimally placed. However, when the system is heavily loaded, to fully utilize the bandwidth may result in suboptimal reducer placement and a larger average CCT. This confirms that not pursuing work conservation is beneficial to minimize the average CCT.

Algorithm Running Time: As an online system, RPC should run Algorithm 1 in a timely manner. Fig. 9 shows the average running time of Algorithm 1 with different numbers of mapper and reducer hosts in the system (*i.e.*, the average algorithm running time when we study the impact of coflow width). All the results are collected from a desktop carrying an Intel i7-2600 CPU with 8 GB memory. From Fig. 9, we can see that even when there are 500 mappers and 500 reducers in the system, the average running time of RPC is only about 350 us. Accordingly, we can conclude that the computation overhead introduced by RPC is negligible.

VII. Conclusion

This work has proposed RPC, a framework to minimize the average CCT by jointly optimizing reducer placement and coflow bandwidth scheduling. To the best of our knowledge, RPC is the first solution that minimizes the average CCT by integrating reducer placement and coflow bandwidth scheduling. Through experiments on a real testbed and extensive simulations, we demonstrate that RPC exhibits remarkable performance advantages over existing technologies.

References

[1] Y. Zhao, C. Tian, J. Fan, T. Guan, and C. Qiao, "RPC: Joint online reducer placement and coflow bandwidth scheduling for clusters," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 187–197.

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

[3] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. Int. Conf. Manage. Data*, 2010, pp. 135–146.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.

[5] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 98–109.

[6] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proc. USENIX NSDI*, 2015, pp. 293–307.

[7] A. Trivedi *et al.*, "On the [ir]relevance of network performance for data processing," in *Proc. USENIX HotCloud*, 2016, pp. 1–5.

[8] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 443–454.

[9] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 393–406.

[10] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li, "Towards practical and near-optimal coflow scheduling for data center networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3366–3380, Nov. 2016.

[11] Y. Zhao *et al.*, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 424–432.

[12] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 431–442.

[13] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ.*, 2009, pp. 261–276.

[15] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. 9th USENIX Conf. Oper. Syst. Des. Implement.*, 2010, pp. 1–16.

[16] J. Tan *et al.*, "DynMR: Dynamic MapReduce with ReduceTask interleaving and MapTask backfilling," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–4.

[17] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "ShuffleWatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proc. USENIX*, 2018, pp. 1–13.

[18] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu, "Network scheduling aware task placement in datacenters," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2016, pp. 221–235.

[19] X. S. Huang and T. S. E. Ng, "Exploiting inter-flow relationship for coflow placement in datacenters," in *Proc. 1st Asia–Pacific Workshop Netw.*, Aug. 2017, pp. 113–119.

[20] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Oct. 2008.

[21] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, 2009.

[22] M. Alizadeh *et al.*, "CONGA: Distributed dongestion-aware load balancing for datacenters," in *Proc. ACM SIGCOMM*, 2014, pp. 503–514.

[23] W. Wang, S. Ma, B. Li, and B. Li, "Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling," in *Proc. IEEE Conf. Comput. Commun.*, May 2017, pp. 1–9.

[24] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *Proc. IEEE MASCOTS*, Dec. 2011, pp. 390–399.

[25] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production MapReduce cluster," in *Proc. 10th IEEE/ACM Int. Conf. Cluster, Cloud Grid Comput.*, 2010, pp. 94–103.

[26] Y. Peng *et al.*, "Towards comprehensive traffic forecasting in cloud computing: Design and application," *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 2210–2222, Aug. 2016.

[27] H. Wang *et al.*, "FLOWPROPHET: Generic and accurate traffic prediction for data-parallel cluster computing," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, Jun. 2015, pp. 349–358.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Oper. Syst. Princ.*, 2003, pp. 29–43.
[29] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. USENIX NSDI*, 2010, pp. 89–92.
[30] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 127–138.
[31] C. Banino-Rokkones, O. Beaumont, and H. Rejeb, "Scheduling techniques for effective system reconfiguration in distributed storage systems," in *Proc. IEEE ICPADS*, Dec. 2008, pp. 80–87.
[32] J. K. Lenstra, D. B. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," in *Proc. IEEE FOCS*, Oct. 1987, pp. 217–224.
[33] D. P. Palomar and J. R. Fonollosa, "Practical algorithms for a family of waterfilling solutions," *IEEE Trans. Signal Process.*, vol. 53, no. 2, pp. 686–695, Feb. 2005.
[34] R. Mittal *et al.*, "TIMELY: RTT-based congestion control for the datacenter," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 537–550, Sep. 2015.
[35] R. Cziva, S. Jouet, K. J. S. White, and D. P. Pezaros, "Container-based network function virtualization for software-defined networks," in *Proc. IEEE ISCC*, Jul. 2015, pp. 415–420.
[36] J. Fulkerson. (2017). *Traffic Shaping With TC*. [Online]. Available: https://www.badunetworks.com/traffic-shaping-with-tc/
[37] Y. Chen, S. Alspaugh, A. Ganapathi, R. Griffith, and R. Katz. (2013). *Statistical Workload Injector for Mapreduce (SWIM)*. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository

Yangming Zhao (Member, IEEE) received the B.Eng. degree in communication engineering and the Ph.D. degree in communication and information system from the University of Electronic Science and Technology of China in 2008 and 2015, respectively. He is currently a Research Scientist with the University at Buffalo. His research interests include network optimization, data center networks, edge computing, and transportation systems.

Chen Tian (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He was an Associate Professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology. From 2012 to 2013, he was a Post-Doctoral Researcher with the Department of Computer Science, Yale University. He is currently an Associate Professor with the State Key Laboratory for Novel Software Technology, Nanjing University, China. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming, and urban computing.

Jingyuan Fan (Member, IEEE) received the B.Eng. degree from Fudan University, China, in 2012, the M.S. degree from the University of California at Los Angeles, Los Angeles, CA, in 2014, and the Ph.D. degree in computer science from The State University of New York at Buffalo, Buffalo, NY, in 2019. His research interest includes computer networks.

Tong Guan received the Ph.D. degree from the Department of Computer Science and Engineering, University at Buffalo, in 2018. His research interests include wireless sensor networks, mobile networks, and social networks, with an emphasis on mathematical modeling and performance analysis.

Xiaoning Zhang received the B.S., M.S., and Ph.D. degrees in communication and information engineering from the University of Electronic Science and Technology of China, Chengdu, China, in 2002, 2005, and 2007, respectively. He is currently an Associate Professor with the Key Laboratory of Broadband Optical Fiber Transmission and Communication Networks, School of Communication and Information Engineering, University of Electronic Science and Technology of China. His research interests include network design and optical and broadband networks.

Chunming Qiao (Fellow, IEEE) is currently a SUNY Distinguished Professor and also the current Chair of the Computer Science and Engineering Department, University at Buffalo. He has published extensively with an H-index of over 73 (according to Google Scholar). He also has seven U.S. patents and has served as a Consultant for several IT and Telecommunications companies since 2000. His current research interest includes connected and autonomous vehicles. His research has been funded by a dozen of major IT and telecommunications companies, including Cisco and Google, and more than a dozen NSF grants. He was elected to an IEEE Fellow for his contributions to optical and wireless network architectures and protocols. Two of his papers received the Best Paper Awards from IEEE and Joint ACM/IEEE venues.