2018 IEEE 26th International Conference on Network Protocols

RPC: Joint Online Reducer Placement and Coflow Bandwidth Scheduling for Clusters

Yangming Zhao[1], Chen Tian[2], Jingyuan Fan[1], Tong Guan[1], Chunming Qiao[1]
[1]Department of Computer Science and Engineering
University at Buffalo, The State University of New York
[2]State Key Laboratory for Novel Software Technology, Nanjing University

***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 paper, 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, then schedule all the coflows following the Shortest Remaining Time First (SRTF) principle. We use a real testbed implementation and extensive large-scale simulations to demonstrate that RPC can reduce the average CCT by 64.98% compared with state-of-the-art technologies.**

I. Introduction

Data transfer has a significant impact on application performance in data-parallel frameworks, such as MapReduce [1], Pregel [2] and Dryad [3]. 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 (*i.e.*, shuffle in Hadoop) before generating the final results. For some applications, 50% of the job completion time is spent on transferring data across the network [4]. Though a work claimed that improving the data transfer cannot greatly improve the job performance [5], its application scenario is found to be very limited. In many cases, optimizing the network performance can greatly speed up the job completion [6]. Accordingly, we focus on reducing the time for data transfer in this work.

Usually in data-parallel frameworks, a network transfer phase is not considered complete till all its constituent flows have finished. For example in MapReduce [1], a computation stage cannot complete, or sometimes even start, before it receives all the flows from the previous stage. These flows between two stages are known as a *coflow*. Minimize the average Coflow Completion Time (CCT) can improve both responsiveness and throughput [7].

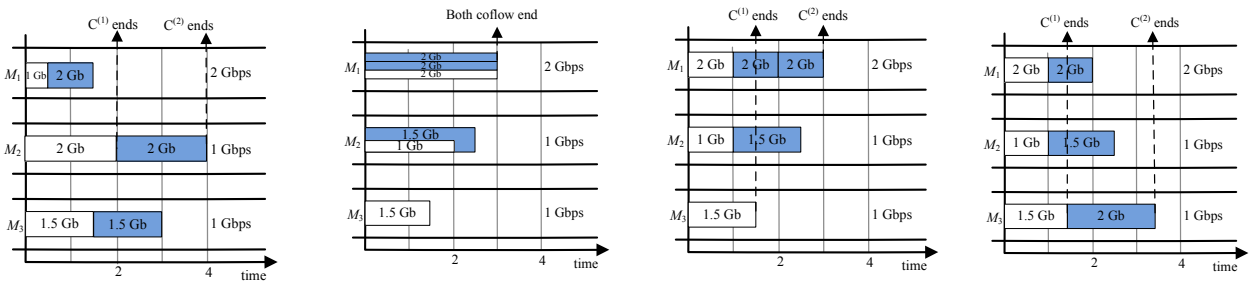
Prior works on minimizing data transfer time have focused on either task placement or coflow bandwidth scheduling, but not both. In those task placement schemes [8], [9], [10], [11], the main idea is to allocate each task to a place such that the data locality can be improved, which can reduce the amount of data that needs to be transferred, and also the time for the data transfer phase. Coflow scheduling [12], [7], [13], [14], [15] controls 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 CCT can be further optimized by jointly optimizing the reducer placement (i.e. task placement) and coflow 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 with 1 Gb, 1.5 Gb and 2 Gb, respectively. There are three reducers in $C^{(2)}$ to fetch flows with 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 of these three hosts are 2 Gbps, 1 Gbps and 1 Gbps, respectively (note that such a heterogeneous bandwidth scenario can happen if some of the bandwidth has been assigned to other applications). For simplicity, we assume that the outgoing bandwidth of the hosts generating these flows are 10 Gbps and the hosts generating data and receiving data are not overlapped. In this case, the network bottleneck only exists on the incoming links of the hosts receiving data.

Fig. 1(a) shows the case that we schedule these two coflows optimally but with a 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 rate. By scheduling the coflows following the Shortest-Remaining-Time-First (SRTF) principle to minimize the average CCT [12], the completion times of these two coflows are 2s and 4s, respectively. The average CCT is 3s. For comparison, in Fig. 1(b), we place the reducers in an optimal way but let all flows share the bandwidth equally. In this case, both coflows finish in 3s. Finally, we can optimize both reducer placement and coflow 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 all the coflows following the SRTF principle. In this case, $C^{(1)}$ completes in 1.5s and $C^{(2)}$ completes in 3s. The average CCT is 2.25s.

In this paper, we propose and implement RPC, a *joint online Reducer Placement and Coflow bandwidth scheduling* framework, to minimize the average CCT. It is worth noting that RPC does not optimize the placement of both mappers and reducers as in [16], since optimizing the mapper placement would introduce more traffic into the network and prolong the data transfer phase. The key idea of RPC is to first minimize the completion time of each single coflow through reducer

978-1-5386-6043-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICNP.2018.00028



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

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.

placement and flow transmission rate control, and then schedule all the 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 scheduling, we develop a 2-approximation algorithm (Section IV). We use a real testbed implementation and extensive large-scale simulations to show that RPC can reduce the average CCT by 64.98% compared with 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 of task placement and coflow scheduling. We review the most closely related ones.

Coflow Scheduling: Orchestra [4] is perhaps the first work that takes the coflow concept into consideration when optimizing flow transfers in data centers. After that, Varys [12] and Baraat [15] start to apply the coflow concept in their network optimization. D-CAS [14] proposes a distributed coflow scheduling scheme, and Aalo [13] extends the work to scenarios where flow sizes are not known in advance. All these works assume that task placement is already determined.

Task Placement: Most task placement approaches follow the "maximizing data locality" principle. DelayScheduling [8] and Quincy [9] try to place tasks on the hosts or racks where most of their input data are located. ShuffleWatcher [17] attempts to localize map tasks of a job to one or a few racks, and thus reduces cross-rack shuffling. They do not take network scheduling into consideration.

Given a network scheduling algorithm, NEAT [18] chooses the best task placement for new requests. Without joint scheduling, its performance is suboptimal, as we will show later in our evaluations. 2D-Placement [16] also leverages task placement to balance network load for future scheduling. However, it assumes that both the source and destination of each constituent flow can be arbitrarily optimized. The source of each constituent flow has only a few choices (*e.g.*, there are only 3 copies of each data chunk in HDFS), or even only one choice (*e.g.*, the intermediate data generated by the mappers). This is exactly why previous task placement approaches pursue data locality to reduce network traffic.

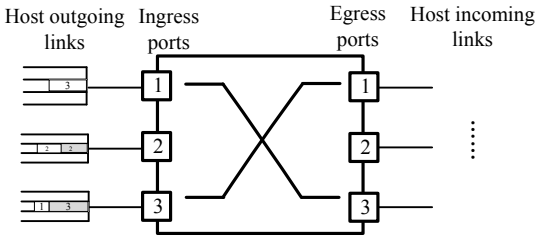


Fig. 2. Data center fabric with 3 ingress/egress ports

Move the source would inevitably increase network traffic, and negate any possible advantage brought by scheduling. Also, it only proposes a heuristic algorithm without any theoretic analysis. Our evaluations demonstrate that, its performance is even lower than NEAT.

B. System Model

Network Model. Given the recent progress in data center fabrics [19], [20], [21], we can abstract the network as a giant nonblocking switch that interconnects all physical hosts (as shown in Fig. 2). The network bottlenecks only exist at the NICs of physical hosts. Therefore, we focus on how to assign the bandwidth at the outgoing/incoming links of each physical host when we schedule the coflows, without paying attention to the flow routing and bandwidth assignment in the fabric. It should be noted that since the physical hosts in a data center can be heterogeneous, or some of the bandwidth has been occupied by other applications, the capacity of each outgoing/incoming link can be different.

When a coflow arrives, we should allocate the reducers for this coflow to the physical hosts. As a consequence, the destinations of its constituent flows are determined. We also determine the transmission rate of each individual flow. In this paper, we assume that network is the only bottleneck, and hence we do not consider the computation phase in our optimization.

Coflow Abstraction. In many cases, such as in MapReduce, the amount of data each flow needs to transfer can be known before the flow starts [12], [15], [4]. 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

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 rate allocation
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 allocation scheme derived in Line 5, and then update B;
16: **end while**

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 n^{th} reducer that should be set up for the next computation stage of coflow i. $k \in D_n^{(i)}$ denotes that $v_k^{(i)}$ should be fetched by reducer n. $C^{(i)}$ and $\{D_n^{(i)}\}$ can be reported by coflow i through the Coflow API [12].

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 [4], [12], RPC works in a centralized, cooperative manner. This is also coherent with many recent centralized data center designs such as [22], [23], [1], [19], [20], etc.

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

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, RPC is invoked to compute its reducer placement and the transmission rate for its each constituent flow. When an existing coflow finishes, RPC is invoked to determine which coflows should take up the released network resources. The underlying scheduling policy RPC takes is Shortest Remaining Time First (SRTF) [12], [7], [24].

The inputs of Algorithm 1 are all the uncompleted coflows Ω and the 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 information should be updated when we recompute the coflow scheduling order. It should be noted that when an individual flow starts, the location of the reducer to fetch this flow cannot be changed anymore. To prevent starvation, RPC first schedules the coflows that are waiting for a long time (Lines 6 – 9). Otherwise, RPC turns to the coflow with the shortest completion time (Lines 10 – 12). When the coflow to schedule is selected, RPC sets up the corresponding reducers, assigns bandwidth to its constituent flows and updates the network resource utilization (Line 15). Till now, one of the uncompleted coflows is scheduled and RPC continues to schedule the next one.

It is worth noting that RPC does not pursue work conservation property in its framework. Though it is a common property to pursue in most of the works to minimize the average CCT, it is not the case when the reducer placement is also an ingredient to optimize. Recall the example we have discussed in Fig. 1(c), at the time 1.5s, the reducer on host M_3 completes and there is still one reducer that has not started. To pursue the work conservation property, we should place the unstarted reducer on host M_3 to fully utilize the network resources. As shown in Fig. 1(d), though the completion time of $C^{(1)}$ is still 1.5s, pursuing the work conservation would delay the completion time of $C^{(2)}$ to 3.5s, which increases the average CCT. Accordingly, *work conservation is not an objective to pursue when we jointly schedule the reducer placement and coflow bandwidth to minimize the average CCT.*

The key algorithm in RPC is to calculate the minimum completion time for each coflow given the information of all its constituent flows and the network resource that can be used (Line 5). In the next section, we will discuss this problem in detail.

IV. Algorithm Details

In Section IV-A, we discuss how to minimize the completion time for a single coflow by integrating the reducer placement and flow scheduling. After that, we analyze the approximation ratio of our algorithm in Section IV-B. Since the available bandwidth for a single coflow is dynamically changing in practice, we discuss how to adjust the CCT for more efficient scheduling in Section IV-C. At last, we discuss a few practical issues in our system in Section IV-D.

A. Minimize Single Coflow Completion Time

Given the information of a coflow, we formulate the problem of minimizing the completion time of coflow i as follows:

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

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 traffic rate of the k^{th} flow in coflow i sending to host j at time t, and $b_j^{in}(t)$ denoting the available incoming bandwidth of host j at time t, the first constraint says the rate sum of all the individual flows sent to host j cannot exceed the available incoming bandwidth of host j at any time. (1b) is used to limit the outgoing rate of each host u, where $s(i,k)$ is the source host of the k^{th} flow of 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 of coflow i (also denoting this flow), (1c) means that all the data of $v_k^{(i)}$ should be sent out before the CCT. In (1d), $x_{nj}^{(i)}$ is a binary variable to denote if reducer $D_n^{(i)}$ is placed on host j. This constraint says 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 generated by the same host and the incoming and outgoing rate of each host is constant [25]. It is difficult to solve due to three reasons: 1) the remaining bandwidth on each host is time varying; 2) the upper bound of the integration in constraint (1c) is a variable; and 3) $x_{nj}^{(i)}$ are binary variables. Hereafter, we present how to address these issues.

To address the first challenge, we try two cases: 1) the coflow only uses the remaining bandwidth left by the coflows already scheduled, and can start transmission at once; 2) the coflow can use the entire bandwidth to/from each host, but should wait for the completion of previous flows to/from this host. With these two cases, we calculate the reducer placement and flow scheduling scheme. Then, we adjust the bandwidth assignment to derive a more accurate minimum CCT estimation (see detail in Section IV-C).

To eliminate the variable in the upper bound of the integration in constraint (1c), we propose the following theorem:

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

$$\textit{maximize} \qquad f^{(i)} \tag{2}$$

Subject to:

$$\sum_k r_{kj}^{(i)} \le b_j^{in}, \quad \forall j \tag{2a}$$

$$\sum_j \sum_{k:s(i,k)=u} r_{kj}^{(i)} \le b_u^{out}, \quad \forall u \tag{2b}$$

$$\sum_j r_{kj}^{(i)} = v_k^{(i)} f^{(i)}, \quad \forall k \tag{2c}$$

$$r_{kj}^{(i)} \le x_{nj}^{(i)} b_j^{in}, \quad \forall j,k,n : k \in D_n^{(i)} \tag{2d}$$

$$\sum_j x_{nj}^{(i)} = 1, \quad \forall n \tag{2e}$$

$$x_{nj}^{(i)} \in \{0,1\}, \quad \forall n,j \tag{2f}$$

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

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

Above discussion shows that $r_{kj}^{(i)} = \frac{\int_0^{T_{opt}} r_{kj}^{opt}(t)dt}{T_{opt}}$ and $f^{(i)} = \frac{1}{T_{opt}}$ is a feasible solution 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 IV.1 is a feasible solution of (1). Therefore, we have

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

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

As $f^{(i)}$ is the inverse of minimum CCT, we say it is the coflow transmission *frequency*. Theorem IV.1 shows that we can solve problem (2) instead of (1) to calculate the reducer placement and individual flow transmission rate, such that the CCT of coflow i can be minimized when all the available bandwidth is constant. However, there is still a binary variable in problem (2), which makes the problem intractable in large size networks.

To solve this problem, we first consider combining all the hosts for reducer placement to be a single "big" host. In this case, all reducers should be placed on the unique "big" host, and the binary variable $x_{nj}^{(i)}$ is eliminated:

$$\text{maximize} \qquad f^{(i)} \tag{3}$$

Subject to:

$$\sum_{k} r_k^{(i)} \leq \sum_{j} b_j^{in} \tag{3a}$$

$$\sum_{k:s(i,k)=u} r_k^{(i)} \leq b_u^{out}, \quad \forall u \tag{3b}$$

$$r_k^{(i)} = v_k^{(i)} f^{(i)}, \quad \forall i \tag{3c}$$

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 combining all the hosts for reducer placement as a "big" one is a relaxation of the original problem. To derive a feasible solution, we must assign reducers to different hosts. To this end, it is inevitable to scale down the flow transmission rate. Say the *scale down ratio* is α, i.e. transmitting $v_k^{(i)}$ in the rate $r_k^{(i)}/\alpha$, we should minimize such scale down ratio to reduce the CCT:

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

Subject to:

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

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

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

It should be noted that $r_k^{(i)}$ is a constant parameter, which is derived by solving (3). By defining $e_{nj}^{(i)} = \sum_{k \in D_n^{(i)}} r_k^{(i)} / b_j^{(in)}$, which presents the scale down ratio that should be enforced on all flows $k \in D_n^{(i)}$ if reducer n is the only reducer placed on host j, we have

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

Subject to:

$$\sum_{n} e_{nj}^{(i)} x_{nj}^{(i)} \leq \alpha, \quad \forall j \tag{5a}$$

$$(4b), (4c)$$

This is a classic unrelated parallel machine scheduling problem that can be solved by relaxation and rounding [26]. To solve (5), we first relax the binary variable constraint on $x_{kj}^{(i)}$ and get

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

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

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

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

where $E_j(\alpha)$ is the set of reducers $\{n | e_{nj}^{(i)} \leq \alpha\}$, and $H_n(\alpha)$ is the set of hosts $\{j | e_{nj}^{(i)} \leq \alpha\}$. Though (6) is not an LP model with variable α in the summation operator, it is an LP model for a fixed α. Accordingly, this model can be solved by a binary search with logarithmic iterations.

By solving (6), we can derive a fractional solution of the reducer placement problem. Then, we should round the solution to derive a feasible reducer placement scheme. To this end, we first propose the following lemma:

Lemma IV.1. *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. In this case, the feasible region is a single point which is determined by v linearly independent rows of the constraint matrix such that each of these constraints is satisfied with the equality, where v is the number of variables in (6) when α is fixed.

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

From Lemma IV.1, we can get the following corollary.

Corollary IV.1. *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.*

Without ambiguity, we say reducer/host v instead of the reducer/host associating with node v hereafter for brevity.

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 IV.1, the non-zero variable number in the solution is at most the

Algorithm 2: Reducer Placement

Input: The solution of problem (6), $\{x_{nj}^{(i)}\}$
Output: Reducer assignment

1: Construct a bigraph BG according to $\{x_{nj}^{(i)}\}$ as in Corollary IV.1
2: Remove all the reducer nodes with only one node degree and place these reducers to 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 assign each reducer to 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: Assign each reducer to its child responder host that services most fraction of this reducer
11: **end for**
12: **else**
13: Treat arbitrary reducer as the root to form a tree and assign each reducer to its child host that services most of this reducer
14: **end if**
15: **end for**

Algorithm 3: Minimize CCT through reducer placement and coflow scheduling

Input: The size of individual flows $v_k^{(i)}$, incoming/outgoing rate 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 $\{r_{kj}^{(i)}\}$

number of nodes in P. Therefore, P can be modified to a pseudo tree by changing the edges according to x'. ■

Based on Corollary IV.1, Algorithm 2 is designed for reducer placement. Line 2 is to handle 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 by Depth-First Search (DFS), and determine the reducer assignment 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 on the resulting tree, we can root the tree at this reducer leaf node, and assign the reducer to its child host that serves most fraction of this reducer. Otherwise, we root the tree at arbitrary reducer node and do the reducer assignment to its child host (Lines 8 – 11). In this way, *each host receives at most one split reducer 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 schedule bandwidth. 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 to different hosts based on Algorithm 2 (Line 2). According to 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 the flows with scale down ratio smaller than 1, we should enlarge its transmission rate to fully utilize the bandwidth at the reducer host side. It means that the network bottleneck exists at the mapper host side. Accordingly, we cannot enlarge the transmission rate.

B. Approximation Bound Analysis

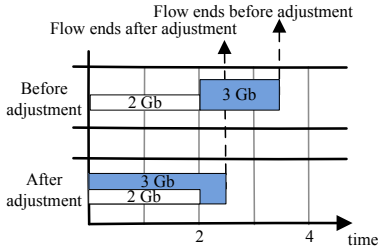
In this section, we are to analyze the approximation ratio of the algorithms we proposed to estimate the CCT, i.e. Algorithm 3. However, before we present the approximation ratio of Algorithm 3, we first analyze the approximation ratio of Algorithm 2, which is an important component to calculate reducer placement.

Theorem IV.2. *The approximation ratio of Algorithm 2 is 2.*

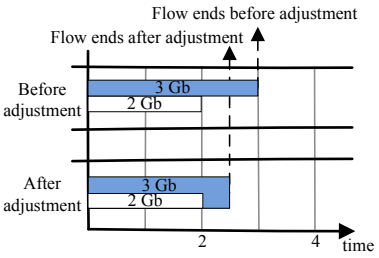
Proof: Let α_{min} is the optimal objective of problem (6), which is clearly the lower bound of the optimal objective of problem (4). In Line 2 of Algorithm 2, we place all the unsplit reducers accordingly the solution of (6), apparently, all these unsplit reducers should result in an objective value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

Theorem IV.3. *The approximation ratio of Algorithm 3 is 2.*

Proof: There are two cascading bottlenecks in our system, the bandwidth of the outgoing port and the bandwidth of the incoming port. When we solve the problem (3), we can get the



(a) Calculate CCT based on full bandwidth



(b) Calculate CCT based on remaining bandwidth

Fig. 3. Flow completion time adjustment

optimal solution which solves the bottleneck of the outgoing ports. Then, Algorithm 3 scales down the flow transmission rate to solve the bottleneck at the incoming ports, which may result in a loss of approximation factor of 2. Accordingly, the approximation ratio of Algorithm 3 is 2. ■

Theorem IV.4. *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 these reducers. For $M(M-1)$ of these reducers, we have $e_{nj}^{(i)} = T$ for all j and $0 \leq n \leq M(M-1)-1$, while $e_{M(M-1),j}^{(i)} = MT$ for all j for the remaining one reducer. Apparently, the optimal solution to problem (5) is that $x_{M(M-1),M-1}^{(i)} = 1$ and $x_{n,n \mod (M-1)}^{(i)} = 1$, otherwise $x_{nj}^{(i)} = 0$, with the objective value MT. With Algorithm 2, the optimal solution of (6) can be $x_{M(M-1),j}^{(i)} = 1/M$ and $x_{n,n \mod M}^{(i)} = 1$, otherwise $x_{nj}^{(i)} = 0$. In this case, 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 infinite, 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 the approximation is leveraging Algorithm 2 to determine the 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 calculate the CCT of a coflow under the assumption that the bandwidth is constant, however, it 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 the later coflow should wait for the completion of previous one, and hence it can use all the bandwidth; the second one is that the later coflow starts once it arrives, but each flow only uses the remaining bandwidth. Then, we use the smaller CCT between these two cases as the estimated CCT to determine the order of coflow scheduling.

However, both methods do not fully utilize the bandwidth. For the first case, we can first transmit the flow with a smaller transmission rate. As shown in Fig. 3(a), say the incoming bandwidth is 2 Gbps and there is already a flow scheduled with 1 Gbps from 0s to 2s, if the later flow waits for the completion of previous one, it will end in 3.5s. However, we can transmit the later flow with the remaining bandwidth 1 Gbps from 0s to 2s, and transmit it with 2 Gbps after the first flow ends. In this case, the later flow will end in 2.5s.

If we calculate the CCT only with the remaining bandwidth, the flow scheduling may be as shown in Fig. 3(b). In this case, we can enlarge the flow transmission rate when previous flows end. After adjusting the transmission rate of all the individual flows in the coflow, we set the completion time of the latest flow as the CCT of the coflow. All the bandwidth adjustment can be done with the classic water-filling algorithm [27].

D. Discussions

Started reducer: Since RPC updates the scheduling scheme whenever a coflow arrives or some flow completes, some of the reducers may already be started on a certain host. If reducer n has already started on host j, we can add a constraint $x_{nj}^{(i)} = 1$ into the problem (4). All the algorithms to calculate the CCT of coflow i and the approximation ratio will not change.

Reducer number constraint: So far, we assume each host can support many reducers simultaneously, however, a host can support only a few reducers in practice. In most of the cases, this would not be a problem since we should 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 assigned to each host. When the number meets the constraint on host j, we first fix all the reducer placement that has already determined, and execute Algorithm 2 again by setting $x_{nj}^{(i)} = 0$ for all the reducers that have not been placed onto any hosts.

Local reducer: When the host issuing flow can host the compute reducers, we can first fix some of the reducers to the host that contains most of its required data. If no such host exists, we still use Algorithm 2 to calculate the reducer placement as we cannot greatly improve the CCT by leveraging the data locality.

Multi-wave reducers: When the reducers are executed in multi-wave [28], only the last wave of reducers determines the CCT. Accordingly, we have at least two solutions to deal with this case. First, we can place all but the last waves of reducers and transmit the corresponding flows by pursuing work conservation, and only optimize the last wave of reducers with the algorithms proposed in this paper. Second, we can

enforce our algorithms to every wave of reducers to minimize the CCT.

Scalability: In Algorithm 1, we schedule all the coflows and let them queue at each physical host, which may be time consuming. In practice, we can pause the algorithm when each physical host takes at least one reducer that has not been assigned any bandwidth. This can reduce the computation time and ensure that when some resources are released, there are reducers to take up them. When the waiting reducer is started, RPC invokes Algorithm 1 to calculate the further scheduling.

In addition, there should be millions of hosts in a data center network, and hence we may need to solve large scale LP models which is computation expensive. To save computation cost, we can limit each job to be executed on only a few hosts, i.e. the number of hosts that reducers can be placed onto is limited. In this way, we can reduce the computation complexity and enhance the algorithm scalability.

V. Implementation

We implement the RPC on a testbed with 7 hosts. One of them is working as job scheduler to execute the algorithms in RPC. The remaining 6 hosts are used to transfer coflows. All of these hosts are connected by a switch, and the NIC rate on each host is 1 Gbps.

Our scheduling algorithms are implemented on the scheduler with CPLEX 12.3 as the linear programming solver. Whenever a coflow generates, the corresponding hosts will notify the scheduler through coflow API. The scheduler calculates the reducer placement and coflow bandwidth scheduling scheme and responses to the hosts. Whenever a host receives the signal from the scheduler, it sets up corresponding reducers to fetch data from the source hosts.

As we need to control the traffic rate of each flow, rate limitation process should be triggered on every host for each reducer. We use two methods. If the scheduler has root privileges, a user-space process is used to control the tc tool in Linux; otherwise, this control is realized by controlling the rate to write the data to the socket buffer.

VI. Performance Evaluation

We evaluate RPC through a small-scale testbed emulation as well as large-scale simulations. We compare the following schemes with RPC.

- **Baseline:** all the reducers are randomly placed and all the flows are fairly competing for bandwidth.
- **Scheduling-only (Varys):** randomly placed all the reducers but schedules them according to SRTF, which is the state-of-the-art scheduling scheme Varys [12].
- **Scheduling-aware reducer placement (NEAT):** Given the scheduling scheme follows SRTF, we place the reducers on the hosts that can minimize the impact on the completion time of other coflows. It exactly follows the thought of NEAT [18].
- **Placement of both mapper and reducer (2D-Placement, abbreviated as 2DP in all the figures):** Place both of the mappers and reducers when a coflow arrives following 2D-Placement [16], and send data to each mapper if it is not placed at the host having the required data. After that, we schedule the coflows following SRTF.

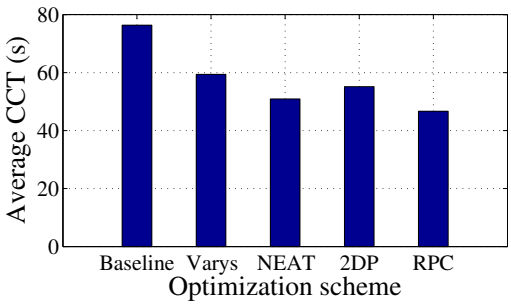


Fig. 4. Testbed experiment results.

Metrics: In this section, we define the performance improvement of scheme 1 compared to 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.

Summary of the main results is as follows:

- Through the experiment on the small scale testbed, we can see that 39.58% and 9.38% of the average CCT can be reduced by RPC, compared with the baseline and NEAT, respectively.
- Compared with the random reducer placement, optimizing the reducer placement can reduce the average CCT by more than 99% in a heterogeneous environment.
- RPC can reduce up to 64.98% of the average CCT, even compared with the state-of-the-art technology (NEAT) to optimize the average CCT.

A. Testbed Experiment

In our experiment, we use 3 of the hosts as coflow senders and place reducers on the remaining 3 hosts. To emulate the heterogeneous hosts, we limit 2 of the switch ports connecting to the receivers to be 500 Mbps. To evaluate the performance of RPC, we inject 3 coflows into the network and assume every flow should be processed by a specific reducer. The coflow information is shown in Tab. I. As a comparison, we also evaluate the performance derived by the baselines, Varys, NEAT and 2D-Placement. The result of this experiment is shown in Fig. 4. From this figure, we can see that RPC can save $\frac{78.6-46.4}{78.6} = 39.58\%$ of the average CCT compared to the baseline scheme, and it can reduce the average CCT by $\frac{51.2-46.4}{51.2} = 9.38\%$ compared to the state-of-the-art technology NEAT on our small-scale testbed. An interesting observation is that NEAT achieves a smaller average CCT than 2D-Placement by $\frac{55.1-51.2}{55.1} = 7.08\%$ *i.e.* optimizing the placement of both mappers and reducers derive a worse solution than only optimizing the placement of reducers. This is because that 2D-Placement needs to transfer data among mapper hosts, which delays the completion of coflows.

TABLE I
COFLOWS INJECTED INTO NETWORK.

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

B. Large Scale Simulations

Simulation methodology: Similar to [23], [12], we build a flow-level simulator, which accounts for the flow arrival and departure events. Whenever such event occurs, the simulator not only updates the remaining amount of each existing flow, but also invokes algorithms we proposed to calculate the reducer placement for newly arrival coflow and update flow transmission rate. To solve linear programming problems in RPC, we embed the API provided by CPLEX 12.3 in our simulator.

In the simulation, we use the MapReduce traffic trace provided in [29]. Since our system is applied to the data-intensive applications, we pick out all the 96 jobs whose shuffle traffic is larger than 20 Gbit. 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 are in the system, the more flows are in a coflow. Correspondingly, the average size of these flows will be smaller. To emulate the heterogeneous environment, we assume the NIC rate of each host is one of the value in $\{0.1, 0.2, 0.5, 1, 10, 40\}$ Gbps. Since the flow source and host NIC rate distributions may impact the reducer placement and coflow scheduling, the simulation results in this section are averaged by 20 tries. The overall simulation results are shown in Fig. 5–7. In general, we can see that RPC outperforms all other schemes in all scenarios. Among the comparison schemes, NEAT derives the best performance, and hence we treat it as the state-of-the-art technology in the simulations. Baseline scheme performs worst as there is no optimization in it, while Varys performs only better than baseline scheme since it absolutely misses optimizing the mappers/reducers placement. By introducing the placement of mappers and reducers, 2D-Placement outperforms Varys. However, since it introduces additional data transfer into the system, it cannot work as well as NEAT.

Impact of Coflow Width: The coflow width is defined as the number of flows in a coflow [12]. In each round of simulations, we change the number of hosts (the number of mapper hosts and that of reducer hosts are kept the same) in the system and observe how the average CCT changes with the number of mapper hosts in the system. The more hosts in the system, the shuffle data should be split into more flows, and hence the wider the coflows are. In addition, we assume the coflow arriving rate is 20 coflows/second. The simulation results are shown in Fig. 5. From this figure, we make following observations.

First, RPC outperforms the schemes without optimizing the reducer placement, i.e. baseline and Varys, by more than 99%. This is because that a bad reducer placement in the baseline scheme and Varys will result in an extreme large completion time to some coflows, especially when they place a reducer receiving a large flow on a host with small incoming bandwidth.

Second, 2D-Placement needs about 2-3x of the average CCT that NEAT needs. For given mapper placement, the 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 more than 2x of the average CCT compared with NEAT.

Third, compared with state-of-the-art scheme (*i.e.*, NEAT), RPC reduces the average CCT by 46.52%–64.98%. In order to consider the impact to other coflows, NEAT optimizes the placement of reducers one by one, rather than optimizing all the reducers in an overall view. Accordingly, it derives a performance worse than RPC.

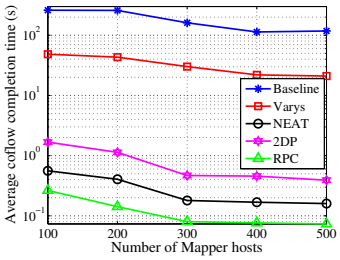
Fourth, regardless of which scheme is adopted, the average CCT is reducing with the increase of the coflow width. This observation is intuitive since there are more hosts, and hence larger bandwidth can be used to serve flows.

Last, Fig. 5(b) shows the CDF of CCT when there are 500 mapper hosts and 500 reducer hosts in the system. From this figure, we can see that RPC does not result in a long-tail effect on the CDF of CCT. It means that RPC optimizes the average CCT without greatly sacrificing the CCT of some individual coflows.

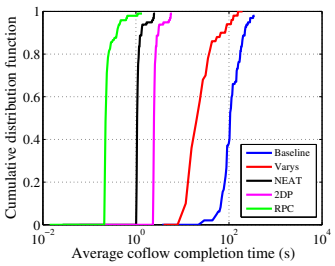
Impact of Coflow Arrival Rate: To investigate the impact of coflow arrival rate, we send out 1000 coflows into the system with 200 mapper hosts and 200 reducer hosts, and observe the relationship between the average CCT and the coflow arrival rate. We investigate the coflow arrival rate from 10 coflows/second to 50 coflows/second, since if every coflow can monopolize the network, the average CCT is 0.141s; and there is almost no remaining bandwidth in the network when the coflow arrival rate is 50 coflows/second. From Fig. 6, we make following observations.

First, the average CCT is increasing 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 at once when they arrive. It results in the larger average CCT.

Second, as we have discussed above, NEAT outperforms 2D-placement by 2-3x when the coflow arrival rate changes. However, with the increase of 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 it results in larger completion time for each coflow. In this case, the percentage of time to transfer data among mapper hosts is relatively reduced.

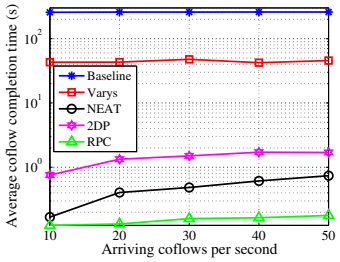


(a) Average CCT.

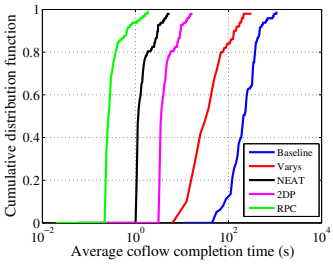


(b) CDF of CCT with 500 mapper hosts.

Fig. 5. The impact of coflow width.

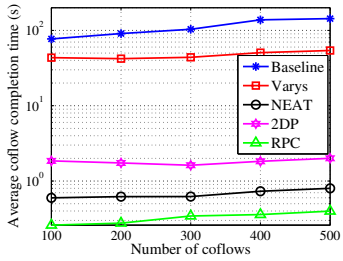


(a) Average CCT.

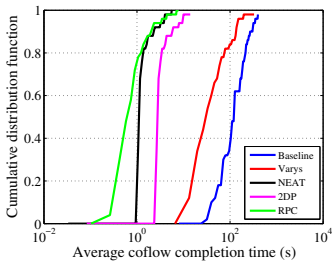


(b) CDF of CCT when the average arrival rate is 50 coflows per second.

Fig. 6. The impact of coflow arrival rate.



(a) Average CCT.



(b) CDF of CCT with 500 coflows.

Fig. 7. The impact of coflow number.

Third, 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 almost completes. Both schemes optimize the average CCT by placing the reducers associated with larger flows to the hosts with larger bandwidth. Accordingly, they derive similar performance. When the coflow arrival rate is large, more coflows in the network and we should carefully assign the bandwidth to different flows. Hence, RPC derives the better performance.

Last, compare Fig. 6(b) with Fig. 5(b), we can see that with larger coflow arrival rate, the CCT spreads in a wider interval. This is because more coflows queuing in the system results in a larger 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.

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 a different number of coflows into the network simultaneously. The simulation results are shown in Fig. 7. From this figure, we make following observations.

First, the average CCT is increasing with the number of coflows in the system. This is obvious because, as explained above, more coflows are injected into the system simultaneously, there will be more coflows queuing in the system, which increases the average CCT. Furthermore, RPC can reduce average CCT by up to 56.52% compared with NEAT.

Second, the performance gap between NEAT and 2D-Placement is slightly reducing with the increase of the number of coflows in the system changes. This is again because that larger queuing delay mitigates the impact of data transfer among mapper hosts.

Third, from Fig. 7(b), we can see that the largest CCT under RPC is similar to that under NEAT. This is because that 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. In this case, the largest coflow should wait for almost the same time before the system starts to serve it under different 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 CCT under different schemes is similar.

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

Work Conservation Issue: In RPC, we propose not to pursue work conservation when we optimize the average CCT, which is a proposition different from most of the previous works. We verify this strategy through some simulations. To this end, we always serve more flows if there is remaining bandwidth. To benefit the completion of coflow, we always assign the remaining bandwidth to the flow with the largest size in a coflow if we cannot serve all the flows. The performance comparison between the scheme with and without pursuing work conservation is shown in Fig. 8.

We can see that when the system is lightly loaded, pursuing work conservation or not achieves the similar performance, since there is little bandwidth competition among coflows and the reducers for each coflow can be optimally placed.

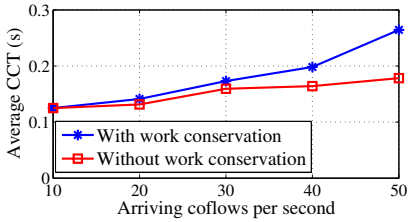


Fig. 8. Impact of pursuing work conservation.

However, when the system is heavily loaded, to fully utilize the bandwidth may result in a bad reducer placement and hence increases the average CCT. This certifies not to pursue work conservation is good for minimizing the average CCT.

VII. Conclusions

This work proposed a framework to joint reducer placement and coflow bandwidth scheduling to minimize the average Coflow Completion Time (CCT). To the best of our knowledge, RPC is the first work that minimizes the average CCT by integrating reducer placement and coflow scheduling. Through real implementation and extensive simulations, we demonstrate that RPC preserves remarkable performance advantages over state-of-the-art technologies.

Acknowledgment

This research is partially supported by the National Key R&D Program of China 2018YFB1003202; NSF grants CNS-1626374, EFMA-1441284 and CNS-1737590, NSFC grants 61671130 and 61671124.

References

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

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the ACM SIGMOD*, 2010, pp. 135–146.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 59–72.

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

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

[6] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou, "On the [ir]relevance of network performance for data processing," in *USENIX HotCloud*, 2016.

[7] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *Proceedings of the IEEE INFOCOM*, 2015.

[8] 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 *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 265–278.

[9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 261–276.

[10] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.

[11] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang, "Dynmr: Dynamic mapreduce with reducetask interleaving and maptask backfilling," in *Proceedings of the EurSys*, 2014.

[12] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the ACM SIGCOMM*, 2014, pp. 443–454.

[13] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proceedings of the 2015 ACM SIGCOMM*, 2015, pp. 393–406.

[14] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li, "Towards practical and near-optimal coflow scheduling for data center networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3366–3380, 2016.

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

[16] X. S. Huang and T. S. E. Ng, "Exploiting inter-flow relationship for coflow placement in datacenters," in *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017, pp. 113–119.

[17] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *USENIX ATC 14*, 2014, pp. 1–13.

[18] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu, "Network scheduling aware task placement in datacenters," in *Proceedings of the CoNext*, 2016, pp. 221–235.

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

[20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, Aug. 2009.

[21] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "CONGA: distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM 2014*, pp. 503–514.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the SOSP 2003*, pp. 29–43.

[23] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the USENIX NSDI*, 2010.

[24] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proceedings of the ACM SIGCOMM*, 2012, pp. 127–138.

[25] C. Banino-Rokkones, O. Beaumont, and H. Rejeb, "Scheduling techniques for effective system reconfiguration in distributed storage systems," in *IEEE ICPADS 2008*, 2008, pp. 80–87.

[26] J. K. Lenstra, D. B. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," in *IEEE FOCS*, Oct 1987, pp. 217–224.

[27] D. P. Palomar and J. R. Fonollosa, "Practical algorithms for a family of waterfilling solutions," *IEEE Transactions on Signal Processing*, vol. 53, no. 2, pp. 686–695, Feb 2005.

[28] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *Proceedings of the IEEE/ACM CCGRID*, 2010, pp. 94–103.

[29] Y. Chen, S. Alspaugh, A. Ganapathi, R. Griffith, and R. Katz, "Statistical workload injector for mapreduce (swim)," 2013. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository