Co-scheduler: Accelerating Data-Parallel Jobs in Datacenter Networks with Optical Circuit Switching

Zhuozhao Li
Department of Computer Science
University of Chicago
Email: zhuozhao@uchicago.edu

Haiying Shen
Department of Computer Science
University of Virginia
Email: hs6ms@virginia.edu

***Abstract*—The optical circuit switch (OCS) in recently proposed hybrid electrical/optical datacenter networks (Hybrid-DCN) can only be used to transfer large flows (i.e., flows with a large size of data). Current job schedulers for data-parallel frameworks are not suitable for Hybrid-DCN, since they neither place tasks to aggregate data traffic to take advantage of OCS nor schedule tasks to minimize the Coflow completion time (CCT). In this paper, we propose *Co-scheduler*, a job scheduler for data-parallel frameworks that aims to improve job performance by attempting to place the tasks of a job to aggregate enough data traffic to take advantage of OCS and minimize the CCT in Hybrid-DCN. Specifically, to achieve this goal, for each job, Co-scheduler computes *guidelines* on the number of racks to place the job's input data and the job's map and reduce tasks, and schedules the map and reduce tasks of the job based on the computed guidelines. The evaluation demonstrates that compared to the state-of-the-art schedulers, Co-scheduler achieves performance improvements on makespan, average job completion time, and average CCT by up to 51.2%, 54.6% and 73.6%, respectively.**

I. Introduction

In the past decade, many organizations (e.g., Facebook, Google, Microsoft, and Yahoo!) have deployed data-parallel frameworks such as MapReduce [1] and Spark [2] to process the increasingly large volume of data. These frameworks often involve network-intensive stages (e.g., shuffle in MapReduce) that transfer a large amount of data in their workflows. For example, 60% and 20% of the jobs on the Yahoo! [3] and Facebook MapReduce clusters [4] are shuffle-heavy jobs (i.e., jobs with a large shuffle data size). However, the network from Top-of-Rack (ToR) switch to the core switch in current datacenter commonly have link oversubscription ranging from 3:1 to 20:1 [1, 4–7]. As a result, these frameworks often suffer from cross-rack network bottlenecks.

To supply sufficient network bandwidth, several studies proposed hybrid electrical/optical datacenter networks (in short *Hybrid-DCN*) [8–11], which augment the traditional *electrical packet switch* (EPS) datacenter network with an optical network using *optical circuit switch* (OCS), as shown in Figure 1. The ToR switches are connected with a core EPS and an OCS. Compared with EPS, OCS has significant lower capital expenditures (CapEx) and operating expenditures (OpEx) [8–11]. However, OCS has a *port constraint*: one input port can setup only one *circuit* to an output port at a time. To change

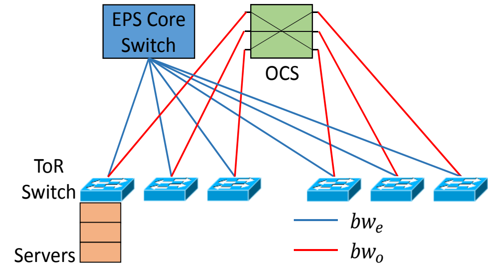


Fig. 1: Architecture of Hybrid-DCN. The link rate between ToR and core EPS is bw_e, while the link rate between ToR and OCS is bw_o

the input-to-output connection, one needs to *reconfigure* the *circuit* connection in OCS, which results in a *reconfiguration delay* on the order of μs-to-ms [8, 9]. Such a delay is too large for a small flow (i.e., flows with a small data size). Thus, in Hybrid-DCN, OCS is used only for large data (e.g., 1.125GB) transfers between racks so that the μs-to-ms reconfiguration delay becomes negligible [8, 9, 12].

In the advanced Hybrid-DCN, the job schedulers for data-parallel jobs must keep pace to meet the needs of such hybrid networks due to the following two reasons.

First, existing schedulers in data-parallel frameworks fail to *aggregate enough data traffic to take advantage of OCS* (*take advantage of OCS* in short later in the paper) to accelerate data transfers. On the one hand, a large group of schedulers (e.g., Fair [13] and Delay [4]) schedule the tasks of a job among all the racks, which generates many small flows that cannot be sent via OCS. On the other hand, another group of schedulers (e.g., ShuffleWatcher [6], Corral [14] and NAS [15]), so called network-aware schedulers, schedule the tasks of a job to avoid using the network to reduce the cross-rack network traffic and cross-rack congestion, rather than taking advantage of the high-bandwidth OCS. Thus, we need to develop new job schedulers to aggregate the data transfers of jobs to reach the elephant flow threshold (e.g., 1.125GB), so that the data transfers can take advantage of OCS.

Second, current data-parallel applications often have communication between different computation stages, and a communication stage finishes only after all of its flows have completed. Recent studies [16, 17] proposed *Coflow* to represent

such a collection of parallel flows in a job. For example, the shuffle between the map tasks and reduce tasks in MapReduce is a Coflow. The *completion time of a Coflow* (CCT) is defined as the time duration between the beginning of its first flow and the completion of its last flow. Existing Coflow schedulers in OCS [18] attempt to minimize the CCTs of jobs by optimally prioritizing the Coflows, assuming that the placement of the job tasks is *fixed* and determined by the job scheduling algorithm of the cluster. However, current job scheduling algorithms do not schedule the tasks of jobs in a way that attempts to minimize the CCTs, which may lead to poor job performance. With different job scheduling algorithms, the source-destination pairs of the flows may vary significantly and hence the CCT of a job may also vary significantly. Therefore, we aim to further improve the job performance by designing job scheduler to minimize CCTs of the jobs.

In this paper, we propose *Co-scheduler*, a job scheduler that aims to improve job performance by enabling the data transfers of jobs to take advantage of OCS and placing the tasks to minimize CCTs of the jobs. While we present our designs and results in the Hadoop MapReduce framework, the ideas can be generalized to any data-parallel frameworks.

Co-scheduler consists of four main steps.

- For each job, Co-scheduler first computes a *guideline* on the number of racks to place the job's input data and to run the job's map tasks, so that the job can potentially take full advantage of OCS to transfer its data.
- When the map tasks of the job complete, based on the map output distribution, Co-scheduler then finds out all the *possible schedules* of the reduce tasks of the job. Each possible schedule includes the number of racks to run the job's reduce tasks that enables the data transfers of the job to take advantage of OCS, and the number of reduce tasks to place on each of the racks that minimizes CCT of the job.
- After finding out all the *possible schedules* for the job that indicate which racks to run the reduce tasks, Co-scheduler selects the *best schedule* among them so that the reduce tasks of the job can finish the data fetching and start the computation stage the earliest, and hence the job completion time is minimized.
- Finally, Co-scheduler schedules the job based on its guideline of map tasks and the best schedule of the reduce tasks, which enables the data transfers of the job to take advantage of OCS in Hybrid-DCN and minimizes CCT of the job.

We have evaluated Co-scheduler through a trace-driven simulation. The experimental results demonstrate that Co-scheduler outperforms the state-of-the-art schedulers by up to 51.2% makespan, 54.6% average job completion time, and 73.6% average CCT.

The rest of the paper is organized as follows. Section II introduces the Hadoop and Hybrid-DCN. Section III presents motivation examples. Section IV introduces the main design of Co-scheduler. Section V presents the performance evaluation. We discuss the related work in Section VI. Section VII concludes this paper with remarks on our future work.

II. BACKGROUND

A. Hadoop MapReduce

A MapReduce job consists of map and reduce stages, which contain multiple map and reduce tasks respectively. Each task is processed by a *container*, which has a certain amount of CPU and memory resource [14]. Each map task processes one input data block and generates map output data (called shuffle data). Each reduce task consists of two steps: shuffle and reduce. In the shuffle, all shuffle data with the same key on all map outputs is transferred to the same reduce task.

Hadoop YARN [19] is a popular open-source implementation of MapReduce [1]. YARN has a centralized scheduler, which determines how to schedule the tasks of the jobs to the containers of different nodes.

B. Hybrid-DCN

In this paper, we assume there are R racks in the cluster and assume the same Hybrid-DCN architecture as in c-Through [8]. As in [8, 9], we assume that only the flow with size larger than the elephant flow threshold T_e (empirically set by network researchers [8, 9], e.g., 1.125GB) is sent via OCS; otherwise, it communicates through EPS. We define *shuffle-heavy jobs* as the jobs with shuffle data size no smaller than the elephant flow threshold.

We abstract OCS as one non-blocking R-port switch (R input ports and R output ports). Each port is connected to a ToR switch and each ToR switch is connected to a rack of machines. Since each input port of OCS can configure one *circuit* to only one output port at a time, one rack can send data via OCS to only one another rack at a time. To change the rack to rack connection, OCS needs to reconfigure the circuit with a fix time δ called *reconfiguration delay*.

We assume that circuit switching model of OCS is *not-all-stop* model, as in [18]. In other words, during the reconfiguration period δ, the communication stops *only* on the affected racks, including the racks to be setup circuits, as well as the racks to be torn down circuits.

C. Lower Bound of Coflow Completion Time with OCS

Assume there are R racks in the cluster. We present a Coflow C by a traffic matrix $\mathbf{C} = (C_{ij})$, where C_{ij} is the size of the flow that needs to be transmitted from rack i to rack j, $i, j = 1, 2, ..., R$. Note that C_{ij} is required to be larger than elephant flow threshold T_e to use OCS. In OCS, to send data from one rack to another rack, it requires reconfiguration to setup the circuits. Thus, each flow incurs at least one reconfiguration delay δ. The minimum time to transfer a flow with size C_{ij} is

$$t_{ij} = \begin{cases} \frac{C_{ij}}{BW_{OCS}} + \delta, & \text{if } C_{ij} > 0 \\ 0, & \text{if } C_{ij} = 0 \end{cases}, \qquad (1)$$

where BW_{OCS} is the link bandwidth of OCS. We can derive that the CCT lower bound in OCS is

$$T(\mathbf{C}) = \max\left(\max_i \sum_j t_{ij}, \max_j \sum_i t_{ij}\right). \qquad (2)$$

$\sum_j t_{ij}$ is the minimum time used to complete the flows sent out from rack i, while $\sum_i t_{ij}$ is the minimum time used to complete the flows received by rack j. Thus, $T(\mathbf{C})$ serves as a lower bound CCT for Coflow C. In fact, previous work [20] showed that Coflow C can be completed in exactly $T(\mathbf{C})$ time by using the *optimal clearance algorithm* in [21].

It is worth mentioning that the lower bound $T(\mathbf{C})$ of a Coflow is actually determined by the maximum data size sent or received of a rack in the Coflow, according to Equation (2).

III. Motivations and Challenges

A. Traffic Aggregation to Take Advantage of OCS

To take full advantage of OCS in Hybrid-DCN, we need to schedule the tasks so that the network traffic between map and reduce stages is aggregated enough to reach the elephant flow threshold. To achieve this, we have the first goal of designing the job scheduler for Hybrid-DCN:

Goal-1: *The job scheduler needs to aggregate the data transfer by scheduling the map and reduce tasks of a job on only a few racks.*

B. Maximizing the Number of Circuits for Coflow

The scheduling of the tasks of a job impacts its CCT, since it affects the number of circuits that can be used to transfer the Coflow of the job. Intuitively, if a job uses more circuits, given the same size of shuffle data to transfer, it takes shorter time to complete the Coflow transfer of the job. Let us consider the following example.

For example, assume there are two jobs $Job1$ and $Job2$. $Job1$ has 9 map tasks and 3 reduce tasks, and $Job2$ has 15 map tasks and 3 reduce tasks. Each map task needs to transfer 1 unit data size to each reduce task. In each unit time, OCS can transfer 1 unit of data. Recall that the OCS reconfiguration delay is δ. The cluster has three racks, and each rack can communicate with one another rack at a time.

To schedule the Coflows, Sunflow [18] is used here. Specifically, Sunflow uses shortest Coflow first algorithm (i.e., shortest lower bound CCT as introduced in Section II-C) and allows the Coflow with higher priority to first use the circuits non-preemptively. Thus, in this example, the Coflow of $Job1$ has a higher priority.

- **Case1 (Figure 2(a)):** The map and reduce tasks of $Job1$ and $Job2$ are scheduled as shown in Figure 2(a). In Case1, the CCTs of $Job1$ and $Job2$ are $12+2\delta$ and $20+3\delta$, respectively.
- **Case2 (Figure 2(b)):** The map and reduce tasks of $Job1$ and $Job2$ are scheduled as shown in Figure 2(b). In Case2, the CCTs of $Job1$ and $Job2$ are $6+2\delta$ and $16+3\delta$, respectively.

We see that the CCTs of the two jobs in Case2 are shorter than the CCTs in Case1, which further demonstrates that the scheduling of the tasks of a job impacts its CCT. To shorten the CCT of a job, the ideal way is to distribute the data transfer of the job to more racks so that more circuits can be used to transfer data concurrently. To achieve this, we have the second goal of designing the job scheduler for Hybrid-DCN:

Goal-2: *The job scheduler needs to distribute the data transfer of a job to as more racks as possible, so that more circuits can be used to transfer data concurrently to shorten the CCT of the job.*

C. Summary and Issues

Takeaway: *We need to design a job scheduler to (i) place the map and reduce tasks of each job on a few racks to aggregate the data transfer to take advantage of OCS; and (ii) allow each job to transfer its data using as more circuits as possible to minimize the CCT.*

Issues: There are several issues we need to resolve in designing such a job scheduler.

- **I1:** How many racks should the map tasks and reduce tasks of a job be placed?
- **I2:** How many tasks should we place on each rack to minimize the CCT of a job?
- **I3:** How to select the set of racks to place the tasks of a job?

In the next section, we present the Co-scheduler design to solve the issues.

IV. The Design of Co-scheduler

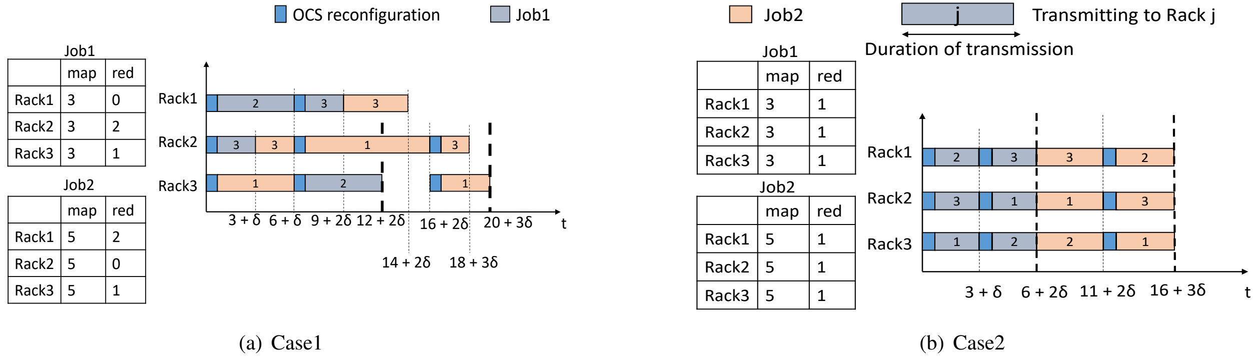
A. Rethinking the Overlapping of Map and Reduce Tasks

In the conventional Hadoop framework, the shuffle data transfers of a job can start *immediately after* its reduce tasks are scheduled while there are still map tasks of the job running. The reduce tasks cannot start until all map tasks complete. However, while this mechanism can shorten the execution time of each single job, it also has a significant drawback. That is, in a highly loaded cluster, this mechanism results in low resource utilization since the reduce tasks take up the containers that could otherwise process other jobs.

With the use of high-bandwidth OCS, *is it still appropriate to use this overlapping mechanism?* We argue that the only advantage of this overlapping mechanism no longer exists, if we could enable the data transfers of the jobs to take advantage of OCS in Hybrid-DCN. On the one hand, shuffle-heavy jobs can exploit OCS to transfer data and hence the transfer delay is relatively small. On the other hand, though shuffle-light jobs cannot exploit OCS, their transfer delay is relatively small even using EPS since they have small size of shuffle data.

Therefore, we propose to start scheduling the reduce tasks of a job after all the map tasks of the job are completed. In addition, unlike conventional Hadoop that starts the corresponding shuffle data transfer *immediately* after each corresponding reduce task is assigned containers, we further propose to start the shuffle data transfer of the job *after all* the reduce tasks are assigned to containers. This new mechanism has several advantages:

- It disallows the reduce tasks from taking up the containers that could be used to process the tasks of other jobs.
- It facilitates the aggregation of data transfer within a job to exploit OCS. Reduce tasks on the same rack can fetch shuffle data from map tasks simultaneously using OCS.
- Since reduce tasks are scheduled after all the map tasks of the job complete, the job scheduler can exploit the information of map output distribution to better schedule the reduce tasks to take advantage of OCS.



(a) Case1

(b) Case2

Fig. 2: Motivation example. The box with number j indicates that a circuit to rack j is configured on the corresponding rack.

B. Overview of Design

In this section, we describe a high-level overview of Co-scheduler. To solve the issues in Section III-C, Co-scheduler consists of four main components:

- **Input Data Placement and Map Task Scheduling Guideline (Section IV-C).** When the input data of a job is submitted to the cluster, Co-scheduler computes the input data placement and map task scheduling guideline for the job, which determines how many racks to place its input data on and how many racks to run its map tasks on, so that the job can potentially take advantage of OCS to transfer its shuffle data. This component is to solve issues I1 and I2.
- **Determining Possible Schedules of Reduce Tasks (Section IV-D).** After the map tasks of a job complete, Co-scheduler computes the *possible schedules* of its reduce tasks that enable the data transfers of the job to take advantage of OCS. The *possible schedules* are a set of $\{[R_{red}, \mathcal{D}, CCT]\}$, where R_{red} is how many racks to run its reduce tasks on, $\mathcal{D}$ indicates the number of reduce tasks to place on each of the R_{red} racks, and CCT is the CCT with such placement of reduce tasks. This component is to solve issues I1 and I2.
- **Selecting the Best Schedule (Section IV-E).** Next, for each job, Co-scheduler uses a *ExploreSchedule* function to explore each *possible schedule* (i.e., finding out how to place tasks according to the possible schedule that yields the shortest job completion time) and then selects the *best schedule* from all the *possible schedules*. The *best schedule* includes which racks to schedule the reduce tasks on so that the reduce tasks of the job can start the earliest and hence the job completion time is minimized. This component is to solve issue I3.
- **OCS and Coflow Aware Scheduling (Section IV-F).** Finally, Co-scheduler attempts to schedule each job following the guideline of its map tasks and the best schedule of its reduce tasks. This component is to solve issue I3.

C. Input Data Placement and Map Task Scheduling Guideline

As mentioned in Section III, in order to exploit the OCS, we need to aggregate the data transfers of the job by placing both of its map and reduce tasks on a few racks. Suppose R_{data}, R_{map} and R_{red} are the number of racks to place the input data, the map tasks and the reduce tasks, respectively. It is expected that each map task can maintain data locality, which means that the map task and its input data block are located on the same rack. Thus, to run the map tasks of the job on R_{map} racks, we also need to place the input data blocks of the job on R_{map} racks, which indicates $R_{data} = R_{map}$.

In this section, we present the details of input data placement and map task scheduling guideline, which computes a guideline on how to place the input data and map tasks of *each submitted job*. The guideline includes how to determine R_{map} (hence R_{data}) and how to place the input data accordingly.

Determining $\mathbf{R_{map}}$. Let us denote Shuffle data size to Input data size Ratio as SIR. Then, the shuffle data size of the job equals $Input * SIR$, where $Input$ is the input data size. Our current implementation initializes SIR to be 1.0 as in [6], and dynamically updates the value as the Map phase of a job progresses. This ratio could also be provided by the user if known in advance, tracked from previous runs of the job, or changed accordingly based on the workload characteristics in the cluster. Many previous studies [14, 22–27] show that cluster workloads contain a large number of recurring jobs, whose job characteristics can be predicted with a small error (e.g., 6.5% [14]).

Assume the map tasks and reduce tasks are evenly distributed to R_{map} and R_{red} racks, to ensure that a job can use OCS to transfer its data, it requires

$$\frac{Input * SIR}{R_{map} * R_{red}} \geq T_e, \tag{3}$$

which means that the data size sent from any map rack to any reduce rack must exceed the elephant flow threshold T_e.

Moreover, based on Section III-B, we expect to use more circuits to transfer data at a time so that the CCT of the job is minimized. Since we can setup at most R_{map} circuits for the R_{map} map racks and at most R_{red} circuits for the R_{red} reduce racks, the number of circuits that can be used to transfer the shuffle data of a job at a time is $\min(R_{map}, R_{red})$. As a result, maximizing the number of circuits can be interpreted as

$$\text{maximize} \quad \min(R_{map}, R_{red}), \tag{4}$$

Based on Equations (3) and (4), we have

$$(\min(R_{map}, R_{red}))^2 \leq R_{map} * R_{red} \leq \frac{Input * SIR}{T_e}. \tag{5}$$

Hence, the number of circuits that can be used is guaranteed to satisfy

$$\min(R_{map}, R_{red}) \leq \sqrt{\frac{Input * SIR}{T_e}}. \quad (6)$$

Jointly considering Equations (5) and (6), we derive that the "=" sign holds true only when $R_{map} = R_{red}$.

Thus, the maximum number of circuits a job can use to transfer its data is $\sqrt{\frac{Input*SIR}{T_e}}$. We initialize $R_{map} = \sqrt{\frac{Input*SIR}{T_e}}$, which ensures that the job has the potential to use the maximum number of circuits to transfer its data.

Input data placement and map tasks scheduling. In Hadoop, each data block has three replicas. Conventionally, the input data blocks are placed randomly in the entire cluster *when the input data of the job is uploaded to the cluster*. The difference of Co-scheduler is that the input data blocks are placed onto only R_{data} racks.

To achieve this, we first randomly choose R_{data} racks and place the first replica of the input data blocks *evenly* onto the R_{data} racks. For the second and third replicas, we need to randomly select two other sets of R_{data} racks and place the second and third replicas of the input data blocks *evenly* onto the R_{data} racks, respectively. The three sets of R_{data} racks are *disjoint* with each other (in total $3 * R_{data}$ distinct racks). Based on such an input data placement of the job, to achieve data locality, we can schedule the map tasks of the job on any R_{map} racks selected from the three *disjoint* sets of R_{data} racks that contain the job's input data (recall $R_{data} = R_{map}$).

D. Determining Possible Schedules of Reduce Tasks

In this section, we describe the details of how to compute the *possible schedules* of reduce tasks of a job. The *possible schedules* of reduce tasks refer to how many racks to run the reduce tasks on to enable the data transfers of the job to take advantage of OCS.

Recall in Section IV-A, we propose to start scheduling the reduce tasks of a job after the last map task of the job is completed. Thus, we can know the distribution of the map output sizes on R_{map} racks of the job, as well as whether the job is shuffle-heavy or not. Notice that if any $SM_i, i = 1, 2, ..., R_{map}$ is smaller than T_e, we can disregard it in the computation because i) regardless of the reduce task placement, the transfer of the map output data cannot take advantage of OCS due to its small size, and ii) the small amount of map output data only takes a short time to transfer even with EPS. Therefore, without loss of generality, we assume that all $SM_i, i = 1, 2, ..., R_{map}$ are greater than elephant flow threshold T_e. We use $\{SM_1, SM_2, ..., SM_{R_{map}}\}$ to denote the distribution of map output data sizes, sorted in ascending order.

Problems: Given a distribution of map output data size $\{SM_1, SM_2, ..., SM_{R_{map}}\}$ on R_{map} racks of a job, Co-scheduler needs to solve the following two problems.

- *First*, determining all the possible values of R_{red} that enable the data transfers of the job to take advantage of OCS.
- *Second,* for each possible value of R_{red}, finding out the number of reduce tasks to be placed on each of the R_{red} racks so that the CCT is minimized, denoted as $\mathcal{D} = \{d_1, d_2, ..., d_{R_{red}}\}$, where d_i indicates the number of reduce tasks placed on a rack.

Determining all the possible values of $\mathbf{R_{red}}$. Obviously, the lower bound of R_{red} is 1. As to the upper bound, since we need to guarantee that the sizes of as many flows as possible of the job are larger than the elephant flow threshold T_e, we have

$$R_{red} \leq \lfloor \frac{SM_1}{T_e} \rfloor. \quad (7)$$

Here, we use SM_1 because it is the minimum among $\{SM_1, SM_2, ..., SM_{R_{map}}\}$ (sorted in ascending order as aforementioned). Setting R_{red} to a value smaller than $\lfloor \frac{SM_1}{T_e} \rfloor$ guarantees all the flows can use OCS. Thus, the possible R_{red} is in the range of $[1, \lfloor \frac{SM_1}{T_e} \rfloor]$.

Finding the placement $\mathcal{D}$ for each possible $\mathbf{R_{red}}$ that minimizes CCT and the corresponding CCT. For each possible R_{red} in the range of $[1, \alpha]$, we use the following algorithm to find its corresponding $\mathcal{D}$. First, for each of the R_{red} racks, we keep placing the reduce tasks to each rack, until the data transfers are aggregated sufficiently, i.e., the smallest map output data SM_1 can be sent via OCS to each of these R_{red} racks. Based on Equations (1) and (2), we expect to minimize the lower bound $T(\mathbf{C})$, which is determined by the maximum data size sent or received of a rack. Thus, we place the remaining reduce tasks one by one to the rack that has the smallest data size among the R_{red}. By doing this, the maximum data size sent or received of a rack is minimized. Using this placement can enable the data transfers of the job to take advantage of OCS, while minimizing CCT. The CCT can be obtained based on Equations (1) and (2).

Output. For each job, Co-scheduler outputs a set of *possible schedules* $\{[R_{red}, \mathcal{D}, CCT], ...\}$ to schedule the reduce tasks, and selects the *best schedule* among them, which will be introduced in the next section.

E. Selecting the Best Schedule

In this section, we present the details of how to select best schedule, as shown in Algorithm 1. Given all the *possible schedules* of the reduce tasks of a job in Section IV-D, Co-scheduler uses a function called *ExploreSchedule* to select the *best schedule* (lines 1-13), so that the reduce tasks can finish the data transfers and start their computation the earliest, and hence the job completion time is minimized.

Specifically, the input of *ExploreSchedule* is a possible schedule $[R_{red}, \mathcal{D}, CCT]$, and the output of *ExploreSchedule* includes the selection of a set of racks for this possible schedule and the estimated time when all the reduce tasks complete their shuffle data transfer. The estimated time in *ExploreSchedule* function is based on the estimated remaining processing time T_{rem} of every running task in the cluster.

Specifically, we exploit a linear model to estimate T_{rem} of every running task periodically. In Hadoop, the status of every running task, including the time elapsed $t_{elapsed}$ and the progress P of the task, is reported periodically. We estimate T_{rem} using a simple heuristic:

$$T_{rem} = t_{elapsed} * \frac{1-P}{P}. \quad (8)$$

This heuristic assumes that the tasks make progress at a constant rate. Previous studies [28, 29] show that such a model works well in practice and the estimation error of T_{rem} is within 2.9% of the actual completion time.

Algorithm 1 Pseudocode of selecting the best schedule.

1: **function** EXPLORESCHEDULE($[R_{red}, \mathcal{D}, CCT]$)
2: Sort $\mathcal{D}$ in descending order
3: **for** d_i in $\mathcal{D}$ **do**
4: **for** Rack R in all non-selected racks **do**
5: T_R = the time to schedule d_i reduce tasks based on T_{rem}
6: **end for**
7: t_i = the smallest T_R
8: r_i = the rack with t_i, and mark r_i as selected rack
9: ▷ selected rack cannot be used again in the search
10: **end for**
11: $t_{max} = \max\{t_1, t_2, ..., t_{R_{red}}\}$
12: **return** selected racks $\mathcal{R} = \{r_1, r_2, ..., r_{R_{red}}\}$ and $CCT + t_{max}$
13: **end function**
14:
15: **for** P in all possible schedules **do**
16: $\mathcal{R}_P$, t_P = ExploreSchedule(P)
17: **end for**
18: Select best schedule as $\mathcal{R}_B$ that has the smallest t_p.

In the following, we describe the details of *ExploreSchedule* function. Without loss of generality, let us assume the sorted $\mathcal{D}$ (descending order) is $\{d_1, d_2, ..., d_{R_{red}}\}$ (line 2). Assume that the rack to run d_i reduce tasks is r_i and the estimated time when sufficient containers on rack r_i are available is t_i (namely released time of r_i). Thus, we have the selected racks $\mathcal{R} = \{r_1, r_2, ..., r_{R_{red}}\}$ and their estimated released times $\mathcal{T} = \{t_1, t_2, ..., t_{R_{red}}\}$ to run $\{d_1, d_2, ..., d_{R_{red}}\}$ reduce tasks. The problem is interpreted as selecting the set of racks $\mathcal{R}$ in the cluster with the goal

$$\text{minimize} \quad \max\{t_1, t_2, ..., t_{R_{red}}\}. \quad (9)$$

Given a possible schedule $[R_{red}, \mathcal{D} = \{d_1, ..., d_{R_{red}}\}, CCT]$, the *ExploreSchedule* function first checks which rack in the cluster is the earliest rack that has available containers to run d_1 reduce tasks and select this rack as r_1. Similarly, *ExploreSchedule* selects the racks $r_2, ..., r_{R_{red}}$ for the subsequent $d_2, ..., d_{R_{red}}$ reduce tasks using the same method (lines 3-10). Finally, it outputs the selected racks $\mathcal{R} = \{r_1, r_2, ..., r_{R_{red}}\}$ and the estimated time of when all reduce tasks complete their shuffle data transfers, i.e., $CCT + t_{max}$, where $t_{max} = \max\{t_1, t_2, ..., t_{R_{red}}\}$ (lines 11-12).

We can prove that such an algorithm of *ExploreSchedule* can always find the optimal solution that matches the schedule.

Proof. Without loss of generality, assume that $t_i = \max\{t_1, t_2, ..., t_{R_{red}}\}$. First, rack r_i cannot be replaced by any racks in $\{r_{i+1}, ..., r_{R_{red}}\}$ or any other racks in the cluster (i.e., $R/\{r_1, ..., r_{R_{red}}\}$), since their released times of d_i containers are certainly larger than t_i (otherwise r_i will not be selected ro run d_i reduce tasks). Second, let us switch any rack r_j in $\{r_1, ..., r_{i-1}\}$ with r_i, which means that d_j reduce tasks are scheduled to rack r_i while d_i reduce tasks are scheduled to rack r_j. In this case, since $d_j \geq d_i$ (recall $\mathcal{D} = \{d_1, d_2, ..., d_{R_{red}}\}$ is in descending order), the released time of r_i to run d_j reduce tasks is no smaller than t_i, which is the released time of r_i to run d_i reduce tasks. Hence, no matter what selection of racks other than $\mathcal{R} = \{r_1, r_2, ..., r_{R_{red}}\}$, t_i will be larger. □

Algorithm 2 Pseudocode of OCS and Coflow aware scheduling.

1: Sort users based on fairness policy
2: **for** each container c in all empty containers **do**
3: Select the first user from the user list and select a task based on the following order:
4: • The reduce task from a shuffle-heavy job whose *best schedule* contains the current rack
5: • The map task from a shuffle-heavy job whose data is on this rack and whose map tasks has been placed on fewer than R_{map} racks
6: • The reduce task from a non-shuffle-heavy job
7: • Any map task from a non-shuffle-heavy job
8: • Any available reduce task
9: • Any available map task
10: **end for**

After Co-scheduler applies *ExploreSchedule* to all the possible schedules of a job, it selects the *best schedule* as the schedule that leads to the smallest $CCT + t_{max}$ (lines 15-18).

F. OCS and Coflow Aware Scheduling

We present the details of OCS and Coflow aware scheduling, which is invoked when there are available containers in the cluster. Specifically, when a container on a rack is available, Co-scheduler selects the first user based on the fairness policy in [4] and schedules a task from the user to the container following Algorithm 2. Though we assume Co-scheduler follows fairness here, other policies in [30, 31] can also be applied.

When a specific container is available, Co-scheduler schedules the tasks in the order above considering the factors below:

- Higher priorities are given to the tasks from a shuffle-heavy job that follows the guideline of the map tasks or the best schedule of the reduce tasks (lines 4-5), which enables the shuffle-heavy job to take advantage of OCS and to minimize CCT of the jobs.
- If a map or reduce task from shuffle-heavy jobs that follows the guideline or best schedule *cannot* be found, higher priorities are given to the tasks from non-shuffle-heavy jobs (lines 6-9). This is because scheduling tasks of shuffle-heavy jobs will violate the guideline and the best schedule of the shuffle-heavy jobs, which prevents them from using OCS.

V. PERFORMANCE EVALUATION

In this section, we evaluate Co-scheduler using simulation with workload traces drawn from production traces.

A. Experimental Setting

Workloads. The workload traces we used were from the SWIM Facebook workloads [3]. Since the workload traces miss important information such as task running time, we first replayed all the jobs in the traces (using the tools provided in the same project [3]) one by one on a single-node Hadoop YARN cluster and then recorded the necessary information for every job. We used this recorded log as the workload traces for simulation.

Simulation. We built a trace-driven flow-level event simulator with different job schedulers. In the simulation, there were 600 servers, organized into 60 racks with 10 servers each. Each server can run up to 20 tasks and had 10Gbps network bandwidth. The Hybrid-DCN topology was the same as in Figure 1. The link rate between the ToR switch and core EPS was 10Gbps, which yields a 10:1 oversubscription ratio. The ToR and OCS were always connected with 100Gbps link. We ran 1000 jobs selected from the workload. The number of users was set to 20 and the jobs were randomly assigned to the users. The elephant flow threshold was set to 1.125GB, which is inferred empirically from previous studies [9, 18, 32]. The reconfiguration delay of OCS was set to 10ms, which is a typical delay of a 3D-MEMS OCS [9].

Baselines. We compared Co-scheduler with two baselines.

(1) *Fair scheduler* [13] (Fair in short) is the most widely used scheduler in current production clusters [13], and it assigns containers to jobs so that each job roughly receives an equal share of containers over time.

(2) *Corral* [14] places the map and reduce tasks of the same job on the same set of racks to reduce the cross-rack shuffle data transfer.

We used Sunflow [18] as the Coflow scheduling algorithm for all the schedulers. Specifically, Sunflow uses the shortest Coflow first algorithm (i.e., shortest lower bound CCT as introduced in Section II-C) and allows the Coflow with higher priority to use all the circuits non-preemptively.

Metrics. We used the three metrics below for the evaluation.

(1) *Makespan*: Makespan is the time to finish all the jobs in the workload.

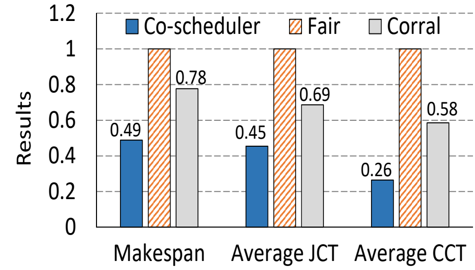
(2) *Average job completion time (JCT)*: The JCT of a job is the time from the arrival of the job until its completion. Average JCT is the average of all the jobs' JCTs.

(3) *Average CCT*: It is the average of all the jobs' CCTs.

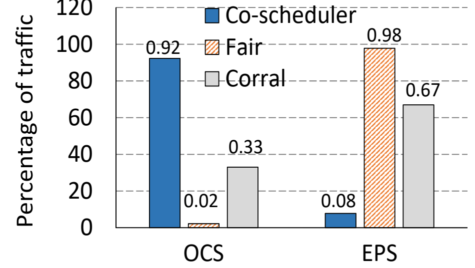
We define the performance comparison between Co-scheduler and each baseline by

$$\frac{|\text{Metric}_{Baseline} - \text{Metric}_{Co-scheduler}|}{\text{Metric}_{Baseline}}, \qquad (10)$$

where $\text{Metric}_{Baseline}$ and $\text{Metric}_{Co-scheduler}$ are results for a specific metric of the baseline scheduler and Co-scheduler, respectively.



(a) Performance metrics



(b) Percentage of traffic sent via OCS and EPS

Fig. 3: Experimental results.

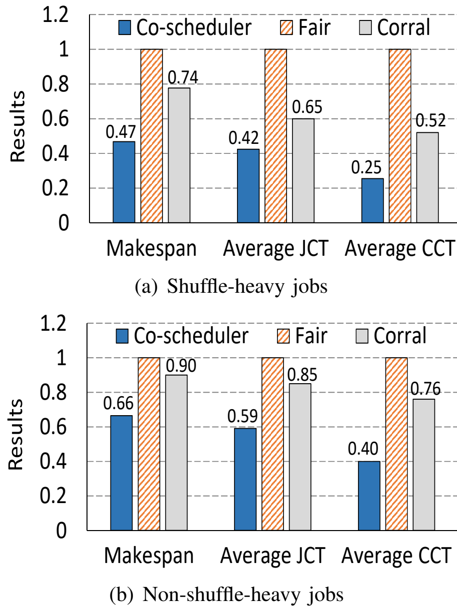
B. Experimental Results

We present the experimental results below. The 1000 jobs arrived uniformly at random in $[0, 90]$ minutes. The experiments were repeated 20 times and the average results were reported. All the results were normalized by the results of Fair scheduler for ease of comparison.

Figure 3(a) shows the makespan of the workload, average JCT, and average CCT with different schedulers. We see that Co-scheduler reduces the makespan of Fair and Corral by 51.2% and 37.2%, respectively. Co-scheduler achieves 54.6% and 33.8% reduction on the average job completion time, compared with Fair and Corral, respectively. Compared with Fair and Corral, Co-scheduler has 73.6% and 54.8% reduction on the average CCT, respectively.

All the results demonstrate superior performance of Co-scheduler in terms of minimizing makespan, average job completion time, and average CCT, since Co-scheduler aggregates the shuffle data transfers of the shuffle-heavy jobs to take advantage of OCS and schedules the tasks of jobs to minimize the CCTs of the jobs. Co-scheduler outperforms Fair since Fair does not *intentionally* aggregate the network traffic to take full advantage of OCS in Hybrid-DCN. Co-scheduler outperforms Corral because (i) Corral attempts to place both map and reduce tasks on the same set of racks to reduce shuffle network traffic, which imposes significant container contentions on the set of racks; and (ii) Corral neither aggregates the data transfers of the jobs to take full advantage of OCS, nor attempts to maximize the number of circuits to shorten the CCT. The above reasons can be demonstrated through Figure 3(b), which shows the network traffic sent via OCS and EPS. We see that 92.2% of the network traffic for Co-scheduler is sent via OCS, while only 2.2% and 33.0% of the network traffic for Fair and Corral is sent via OCS.

We also evaluate the performance improvements of Co-



(a) Shuffle-heavy jobs

(b) Non-shuffle-heavy jobs

Fig. 4: Performance improvements of shuffle-heavy and non-shuffle-heavy jobs.

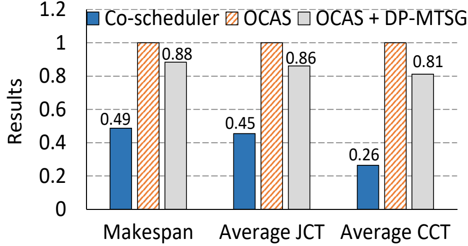


Fig. 5: Effectiveness of different mechanisms.

scheduler over Fair and Corral for shuffle-heavy and non-shuffle-heavy jobs, respectively. Figures 4(a) and 4(b) show that Co-scheduler significantly improves the performance of both shuffle-heavy and non-shuffle-heavy jobs. The shuffle-heavy jobs have significant performance improvements because of the use of OCS in Co-scheduler. As the shuffle-heavy jobs will finish faster and release the containers earlier, non-shuffle-heavy jobs also finish earlier. We also observe that the performance improvements of the shuffle-heavy jobs are more significant than those of non-shuffle-heavy jobs, since non-shuffle-heavy jobs are not dominated by the shuffle, which is the main phase optimized by the Co-scheduler.

C. *Effectiveness of Different Mechanisms*

We also evaluate the impact of different mechanisms in Co-scheduler: input data placement and map task placement guideline (MTS), determining possible schedules of reduce tasks (PSRT), selecting the best schedule (SBS), and OCS and Coflow aware scheduling (OCAS). We only evaluate the performance of OCAS, and MTS + OCAS, compared with MTS + PSRT + SBS + OCAS, since (i) MTS cannot work without OCAS; and (ii) PSRT and SBS cannot work without MTS. Notice that OCAS is actually Fair when there is no guideline of map and reduce tasks for shuffle-heavy jobs, and MTS + PSRT + SBS + OCAS is actually Co-scheduler.

Figure 5 shows the contributions of different mechanisms, in terms of the makespan of the workload, average JCT, and average CCT. Compared with OCAS, MTS + OCAS achieves performance improvements on makespan, average job completion time, and average CCT by 12%, 14%, and 19%, respectively. This is because MTS attempts to place the input data and map tasks in a limited number of racks, which somehow aggregates the shuffle data transfers of the shuffle-heavy jobs. However, MTS + OCAS results in much worse performance than MTS + PSRT + SBS + OCAS (i.e., Co-scheduler), since MTS + OCAS only aggregates the map tasks but does not have a mechanism to aggregate the reduce tasks. Without the guideline for reduce tasks of the jobs, MTS + OCAS cannot enable all the shuffle-heavy jobs to use OCS, leading to worse performance than Co-scheduler.

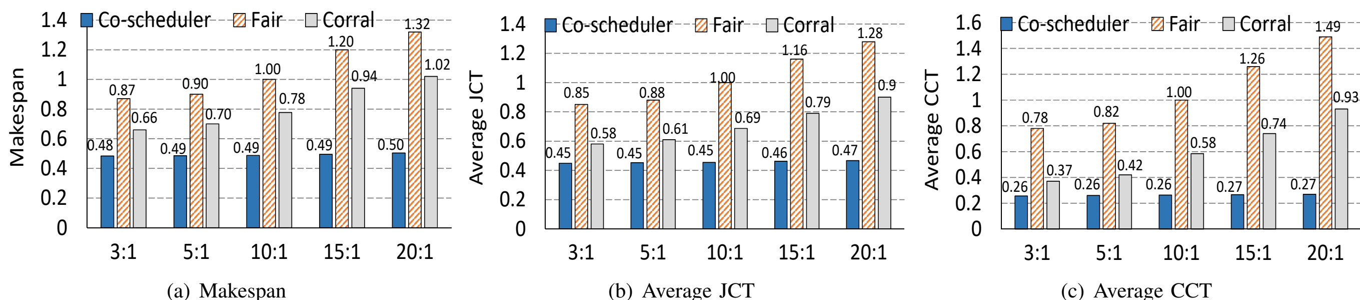
D. *Sensitivity Analysis*

In this section, we conduct several sensitivity tests of Co-scheduler. The experiment settings are the same as Section V-A unless otherwise specified.

Sensitivity to oversubscription ratio. In this experiment, we varied the oversubscription ratio in the EPS network from 3:1 to 20:1. All the results are normalized by the results of Fair scheduler with oversubscription ratio of 10:1. Figures 6(a), 6(b), 6(c) show the makespan, average JCT, and average CCT versus different oversubscription ratio. We see that the makespan, average job completion time, and average CCT with Co-scheduler are not sensitive to the oversubscription ratio. This is because with Co-scheduler, most of the shuffle network traffic is sent via OCS and only a small amount of network traffic is sent via EPS. As a result, the variation of oversubscription ratio does not impact the performance of Co-scheduler significantly. However, as the oversubscription ratio increases, the performance of Fair and Corral are significantly degraded, since with Fair and Corral, a large portion of network traffic is still sent via EPS.

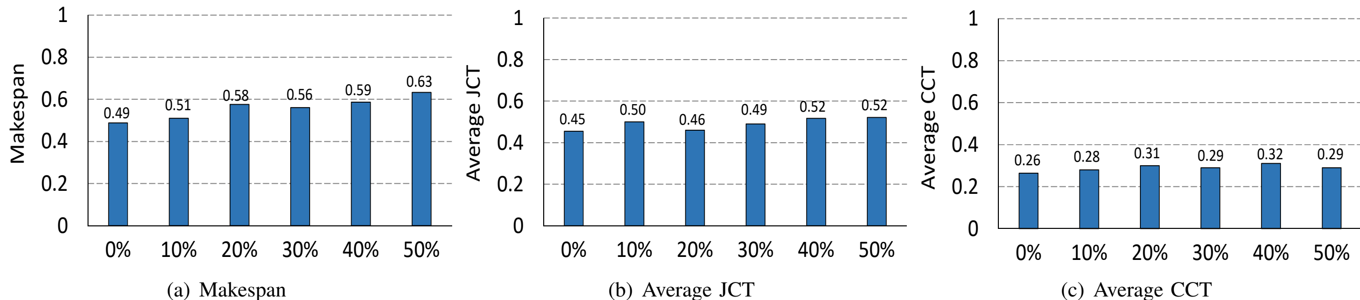
Sensitivity to estimation error of T_{rem}. Recall that we exploit the estimation of T_{rem} to schedule the tasks in Co-scheduler. In this experiment, we varied the error rate of the estimation of T_{rem} by up to 50% to see how Co-scheduler performs. We define the estimation error rate as $\frac{|real-estimation|}{real}$, where $real$ is the actual T_{rem} of the job and $estimatation$ is the estimated T_{rem}.

Figures 7(a), 7(b), 7(c) show the makespan, average JCT, and average CCT versus the estimation error. The results of Fair and Corral are not shown in the figures as they do not rely on the estimation of T_{rem}. We see that the performance improvements of Co-scheduler on makespan and average job completion time decrease as the error rate increases, while the variation of the error rate has less significant impact on the average CCT. This is because of two reasons. First, as the error rate increases, Co-scheduler cannot accurately select the best set of racks to run reduce tasks of the jobs, which degrades the JCTs of the jobs (and hence makespan). Second, although Co-scheduler cannot accurately select the best set of racks,



(a) Makespan (b) Average JCT (c) Average CCT

Fig. 6: Sensitivity analysis of oversubscription ratio.



(a) Makespan (b) Average JCT (c) Average CCT

Fig. 7: Sensitivity analysis of estimation of T_{rem}.

selecting different possible schedules have slight variation on the CCTs of the jobs.

However, even with high error rate, Co-scheduler still outperforms Fair by 36% makespan and 46% average JCT, and outperforms Corral by 18% makespan and 21% average JCT. This demonstrates the robustness of Co-scheduler regarding to the error in estimating T_{rem}. Nevertheless, recent studies [28, 29] show that the estimation of T_{rem} can be estimated with a low error rate around 2.9%.

VI. Related Work

Hybrid-DCN. Previous studies [8–10] have demonstrate the feasibility of utilizing OCS in datacenter networks to improve network capacity. One common assumption in these proposals is that in a datacenter, a rack has elephant flows to only a few racks and mice flows to other racks, which may not be true for the data-parallel frameworks with current job schedulers. In this paper, we aim to design a job scheduler to take advantage of OCS in Hybrid-DCN to improve job performance.

Schedulers. Many researchers have designed various schedulers [4, 6, 13–15, 26, 33–38] for data-parallel clusters to improve job performance. Unfortunately, none of these schedulers schedule tasks of the jobs to take advantage of Hybrid-DCN. For example, the current state-of-the-art schedulers in Hadoop YARN, Fair scheduler [13] and Delay scheduler [4], focus on achieving high data locality and schedule the tasks of the jobs across the entire cluster. Similarly, ShuffleWatcher [6] aims to distribute the shuffle network traffic spatially among different racks. The above schedulers totally disaggregate the data transfers of the jobs, which fails to take advantage of Hybrid-DCN to solve the network bottleneck to improve job performance. Corral [14] is a network-aware scheduler that attempts to reduce the data transfers between map and reduce stages of a job by placing the map and reduce tasks of the job together on the same racks. Although Corral somehow aggregates the data transfers of the job to a fewer racks, it causes significant container contention in these racks among these map and reduce tasks.

Coflow. The Coflow abstraction was first documented in [16], although the similar idea was present in previous paper [39]. The objective of Coflow scheduling is often to minimize the average CCT. This problem is proved to be NP-hard [17, 18], as it can be reduced from the concurrent open-shop scheduling problem [40]. Thus, many heuristic scheduling algorithms [17, 18, 20, 41, 42] were proposed to minimize the CCT to improve the performance of data-parallel jobs.

All of these techniques schedule the Coflows assuming that the source-destination pairs of the Coflows are *fixed*. However, current job scheduling algorithms do not schedule the tasks of jobs in a way that attempts to minimize the CCTs, which may lead to poor job performance. Different job scheduling algorithms may result in different source-destination pairs of the flows, which significantly impacts on the CCTs of jobs. In this paper, we propose a job scheduler that improves the job performance by coordinating the task placement to minimize the CCTs in OCS.

VII. Conclusion

Existing job schedulers for data-parallel frameworks cannot take advantage of Hybrid-DCN to accelerate the data transfers of the jobs. Moreover, current job schedulers do not schedule the tasks of jobs to minimize the CCTs of the jobs. We propose Co-scheduler, a job scheduler that aims to improve job performance by exploiting OCS in Hybrid-DCN and minimizing the CCT. For each job, Co-scheduler first computes a guideline to place its input data and map tasks, which guarantees the potential of the job to take advantage of OCS. Then, when the map tasks of a job complete, Co-

scheduler computes the possible schedules of the jobs, uses the *ExploreSchedule* function to explore every possible schedule, and selects the best schedule among all the possible schedules, so that the reduce tasks finish their data fetching and start the computation stage the earliest and hence the job completion is minimized. Finally, Co-scheduler schedules each job following the guideline of map tasks and the best schedule of reduce tasks. Our trace-driven simulation shows that Co-scheduler outperforms the state-of-the-art job schedulers in terms of makespan, average job completion time and average CCT. In the future, we will further explore using machine learning techniques to automatically learn how to conduct scheduling and investigate more sophisticated methods of estimating the remaining processing time.

ACKNOWLEDGMENT

This research was supported in part by U.S. NSF grants NSF-1827674, CCF-1822965, OAC-1724845, ACI-1719397 and CNS-1733596, and Microsoft Research Faculty Fellowship 8300751.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. of OSDI*, 2004.
[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of NSDI*, 2012.
[3] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites." in *Proc. of MASCOTS*, 2011.
[4] M. Zaharia, D. Borthakur, S. Sen, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of EuroSys*, 2010.
[5] 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," in *Proc. of SIGCOMM*, 2009.
[6] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "ShuffleWatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proc. of ATC*, 2014.
[7] H. Shen and Z. Li, "New bandwidth sharing and pricing policies to achieve a win-win situation for cloud provider and tenants," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2682–2697, 2016.
[8] G. Wang, D. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan, "c-Through: Part-time optics in data centers," in *Proc. of SIGCOMM*, 2010.
[9] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: A hybrid electrical/optical switch architecture for modular data centers," in *Proc. of SIGCOMM*, 2010.
[10] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen, "OSA: An optical switching architecture for data center networks with unprecedented flexibility," in *Proc. of NSDI*, 2012.
[11] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat, "Integrating microsecond circuit switching into the data center," in *Proc. of SIGCOMM*, 2013.
[12] Z. Li and H. Shen, "Job scheduling for data-parallel frameworks with hybrid electrical/optical datacenter networks," in *Proceedings of the Symposium on Cloud Computing, Poster*, 2017, pp. 662–662.
[13] "Fair Scheduler," https://hadoop.apache.org/docs/r2.7.2/hadoop -yarn/hadoop-yarn-site/FairScheduler.html.
[14] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proc. of SIGCOMM*, 2015.
[15] Z. Li, H. Shen, and A. Sarker, "A network-aware scheduler in data-parallel clusters for high performance," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 1–10.
[16] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. of HotNets*, 2012.
[17] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proc. of SIGCOMM*, 2014.
[18] X. S. Huang, X. S. Sun, and T. Ng, "Sunflow: Efficient optical circuit scheduling for coflows," in *Proc. of CoNext*, 2016.
[19] "Apache Hadoop," http://hadoop.apache.org/.
[20] Q. Liang and E. Modiano, "Coflow scheduling in input-queued switches: Optimal delay scaling and algorithms," in *Proc. of INFOCOM*, 2017.
[21] T. Inukai, "An efficient ss/tdma time slot assignment algorithm," *IEEE Transactions on Communications*, vol. 27, no. 10, pp. 1449–1455, 1979.
[22] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proc. of USENIX ATC*, 2012.
[23] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proc. of EuroSys*, 2012.
[24] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *Proc. of OSDI*, 2016.
[25] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards Automated SLOs for Enterprise Clusters," in *Proc. of OSDI*, 2016.
[26] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters," in *Proc. of OSDI*, 2016.
[27] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes," in *Proc. of SOCC*, 2016.
[28] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *Proc. of OSDI*, 2008.
[29] 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 *Proc. of OSDI*, 2010, pp. 1–16.
[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. of NSDI*, 2011.
[31] "Capacity Scheduler," https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.
[32] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky, G. Porter, and A. C. Snoeren, "Scheduling techniques for hybrid circuit/packet networks," in *Proc. of CoNext*, 2015.
[33] J. Jiang, S. Ma, B. Li, and B. Li, "Symbiosis: Network-aware task scheduling in data-parallel frameworks," in *Proc. of INFOCOM*, 2016.
[34] C. Chen and et al., "Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees," in *Proc. of INFOCOM*, 2017.
[35] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proc. of SIGCOMM*, 2014.
[36] Z. Li and H. Shen, "Designing a hybrid scale-up/out hadoop architecture based on performance measurements for high application performance," in *Proceedings of International Conference on Parallel Processing*, 2015, pp. 21–30.
[37] Z. Li, H. Shen, W. Ligon, and J. Denton, "An exploration of designing a hybrid scale-up/out hadoop architecture based on performance measurements," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 386–400, 2017.
[38] Z. Li and H. Shen, "Measuring scale-up and scale-out hadoop with remote and local file systems and selecting the best platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, pp. 3201–3214, 2017.
[39] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.
[40] T. A. Roemer, "A note on the complexity of the concurrent open shop problem," *Journal of scheduling*, vol. 9, no. 4, pp. 389–396, 2006.
[41] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. of SIGCOMM*, 2015.
[42] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: near-optimal network design for coflows," in *Proc. of SIGCOMM*, 2018.