Co-Scheduler: A Coflow-Aware Data-Parallel Job Scheduler in Hybrid Electrical/Optical Datacenter Networks

Zhuozhao Li and Haiying Shen, *Senior Member, IEEE, ACM*

***Abstract*— To support higher demand for datacenter networks, networking researchers have proposed hybrid electrical/optical datacenter networks (Hybrid-DCN) that leverages optical circuit switching (OCS) along with traditional electrical packet switching (EPS). However, due to the high *reconfiguration* delay of OCS, OCS is used only for bulk data transfers between racks to amortize the reconfiguration delay. Existing job schedulers for data-parallel frameworks are not designed 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 describe the mismatch between existing job schedulers and the advanced Hybrid-DCN, introduce the requirements for the new scheduler, and present the implementation of *Co-scheduler*, a job scheduler for data-parallel frameworks that aims to improve job performance by placing the tasks of jobs to aggregate enough data traffic to better leverage OCS to minimize the CCT in Hybrid-DCN. Specifically, for every job, Co-scheduler computes *guidelines* on how many racks to place the job's input data and the job's tasks. The guidelines are dynamically generated based on the real-time job characteristics or predictable job characteristics from prior runs, with the aim of leveraging OCS whenever possible and efficient and minimizing CCT of jobs. Co-scheduler then schedules the tasks of jobs based on the guidelines. We evaluate the effectiveness of Co-scheduler using trace-driven simulation. The evaluation demonstrates that Co-scheduler can improve makespan, average job completion time, and average CCT of a workload by up to 56%, 61%, and 79%, respectively, compared to the state-of-the-art schedulers.**

***Index Terms*— Optical circuit switching, Hybrid-DCN, job scheduler, traffic aggregation.**

I. Introduction

In THE past decade, many organizations (e.g., Facebook and Yahoo!) have deployed data-parallel frameworks such as MapReduce [1] and Spark [2] to process the increasingly large volume of data. Their applications often involve network-intensive stages (e.g., shuffle in MapReduce) that transfer a large amount of data. 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]. In addition, it has been showed that the background data transfer (e.g., data replications) can use up to 50% of the cross-rack bandwidth [8], which further decreases the available bandwidth for the data-parallel frameworks. As a result, these frameworks often suffer performance degradation due to cross-rack network bottlenecks.

To provide sufficient network bandwidth, several studies proposed a hybrid electrical/optical datacenter network (in short *Hybrid-DCN*) [9]–[12] architecture, which augments the traditional *electrical packet switching* (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) [9]–[12]. However, OCS has a *port constraint*: one input port can setup only one *circuit* to an output port at a time. To change the input-to-output connection for data transfers, one needs to *reconfigure* the circuit connection in OCS, which results in a *reconfiguration delay* on the order of μs-to-ms [9], [10]. This reconfiguration overhead is not acceptable for small flows with a small data size, which generally completes in several microseconds within the datacenter networks. This constraint implicates that OCS should be used only for elephant flows (e.g., flows with at least 1.125 GB [9], [10], [13], called elephant flow threshold later on) between racks in Hybrid-DCN, so that the reconfiguration delay becomes negligible [9], [10], [14], compared to the data transfer time.

However, we need to develop new job schedulers for data-parallel frameworks to keep pace of the needs for Hybrid-DCN due to the two reasons below.

Current state-of-the-art job schedulers fail to *aggregate sufficient data traffic to take advantage of OCS* (take advantage of OCS in short later in the paper) to accelerate data transfers. Existing schedulers can be classified into two groups. A group of schedulers (e.g., Fair [15] and Delay [4]) reduces the data transfers by increasing the data locality during map stages. However, since the input data of a job is

Manuscript received 12 June 2021; revised 26 November 2021 and 4 January 2022; accepted 9 January 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C. Wu. Date of publication 28 January 2022; date of current version 18 August 2022. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant NSF-1827674, Grant CCF-1822965, Grant OAC-1724845, Grant ACI-1719397, and Grant CNS-1733596; and in part by the Microsoft Research Faculty Fellowship under Grant 8300751. The preliminary work was published in [62] [DOI: 10.1109/ICDCS.2019.00027]. *(Corresponding author: Zhuozhao Li.)*

Zhuozhao Li is with the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: lizz@sustech.edu.cn).

Haiying Shen is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903 USA (e-mail: hs6ms@virginia.edu).

Digital Object Identifier 10.1109/TNET.2022.3143232

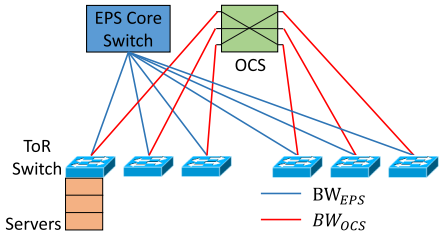


Fig. 1. A typical Hybrid-DCN architecture. The link rate between ToR and core EPS is BW_{EPS}, while the link rate between ToR and OCS is BW_{OCS}.

randomly placed across all the racks, these schedulers schedule the tasks *randomly* (in most case *evenly* as well for load balancing) across all the racks, which does not attempt to aggregate the data transfers and generates many small flows that cannot be sent via OCS. Another group of schedulers (e.g., ShuffleWatcher [6], Corral [16], and NAS [17]), 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 a new job scheduler to aggregate the data transfers of jobs to reach the elephant flow threshold (e.g., 1.125 GB).

Modern data-parallel applications often have communication stages between computation stages, and each communication stage is deemed completed only when all of its communication flows have completed. Recent studies [18], [19] proposed *Coflow* to abstract such a collection of parallel flows. For example, the shuffle between map and reduce stages 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 for OCS [20] minimize the CCTs of jobs by optimally prioritizing the Coflows, assuming the source and destination of each flow in a Coflow is *known priori*. However, different job schedulers may place the map and reduce tasks very differently, and hence the source-destination pair of a flow may vary significantly and so does the CCT of a job. This implies that the scheduling of a job may affect its CCT. **Unfortunately, existing job schedulers do not schedule tasks in a way that facilitates the Coflow schedulers to easily minimize CCTs.**

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, while placing the tasks to minimize CCTs of the jobs.

Co-scheduler consists of four main steps for *every* job.

- Co-scheduler computes a *guideline* on the number of racks to place the job's input data, so that the job can potentially take advantage of OCS to transfer its data. Many jobs in modern workloads are recurring while others are not. Co-scheduler leverages this feature and uses different models to generate different guidelines with respect to the input data placement of recurring and non-recurring jobs.
- Based on the input data placement guideline, Co-scheduler further generates the map task placement guideline to maintain high data locality, while still enabling the job to potentially take advantage of OCS. When the map tasks of the job complete, Co-scheduler finds out all the *possible schedules* of the reduce tasks of the job based on its map output distribution. Each possible schedule includes the number of racks to run the job's reduce tasks, 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, Co-scheduler selects the *best schedule* among them that enables the reduce tasks of the job to finish the data fetching and to start the computation stage the earliest.
- Finally, Co-scheduler schedules the job based on generated data placement and task placement guidelines, which maximize the probability of the job to take advantage of OCS in Hybrid-DCN and minimizes CCT of the job.

We have evaluated Co-scheduler via a trace-driven simulation. The experimental results demonstrate that Co-scheduler outperforms the state-of-the-art schedulers by up to 56% makespan, 61% average job completion time, and 79% average CCT.

The rest of the paper is organized as follows. Section II introduces the background of Hadoop and Hybrid-DCN. Section III presents two motivation examples. Section IV introduces the main design of Co-scheduler. Section V presents the performance evaluation. Section VI discusses how to extend Co-scheduler to other general scenarios. Section VII discusses the related work. Section VIII concludes this paper with remarks on our future work.

II. Background

A. Hadoop MapReduce

A MapReduce job generally consists of map and reduce stages. The input data of a MapReduce job is split into small data blocks. The map stage consists of a number of *map tasks*, each of which applies a "map" function on an input data block to generate *intermediate data* (called *shuffle data*). The reduce stage consists of two phases, shuffle and reduce phases. In the shuffle phase, all shuffle data belonging to one key is grouped together and transferred to the corresponding worker node. The reduce phase also consists of a number of *reduce tasks*, each of which applies the "reduce" function to process a group of the shuffle data.

Hadoop YARN [21] is a widely-used and open-source distributed data processing framework that enables running MapReduce jobs. The basic processing unit in YARN is called a *resource container*. Each (map or reduce) task is processed by a container, each of which has a certain amount of CPU and memory resources [16]. YARN has a job scheduler that is responsible for scheduling the tasks of jobs to available resource containers on different nodes, i.e., the task placements of every job and the orders of tasks among jobs.

When a certain fraction of map tasks (e.g., by default 5% [22]) for a job have completed, YARN can start to schedule the reduce tasks in the reduce stage. After a reduce task is scheduled, the shuffle phase of this reduce task start immediately. However, the reduce task cannot start until all the map tasks of the job complete, i.e., until the shuffle data needed by the reduce task has completed to transfer.

B. Hybrid-DCN

In this paper, we assume a cluster has R racks and assume that Hybrid-DCN is the same as c-Through [9], as shown in Figure 1. As in [10] and [9], we assume that only the flow with size larger than the elephant flow threshold T_e (empirically set by network researchers [9], [10], e.g., 1.125 GB) 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 a 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 fixed time δ called *reconfiguration delay*.

We assume that circuit switching model of OCS is *not-all-stop* model, as in [20]. 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 as 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, $i, j = 1, 2, \ldots, R$. 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 circuit. Thus, each flow incurs at least one reconfiguration delay δ and the minimum time to transfer a flow with size C_{ij} is

$$t_{ij} = \begin{cases} \dfrac{C_{ij}}{BW_{OCS}} + \delta, & \text{if } C_{ij} > 0 \\ 0, & \text{if } C_{ij} = 0 \end{cases}, \tag{1}$$

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

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

$\sum_j t_{ij}$ 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. Note that this theoretical lower bound CCT is commonly used in several previous work [20], [23]. In fact, previous work [23] showed that Coflow C can be completed in exactly $T(\mathbf{C})$ time by using the *optimal clearance algorithm* in [24].

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 (15).

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. For example, let us consider a typical shuffle-heavy MapReduce job, TeraSort. Assume we need to sort 10 GB of data. In a MapReduce TeraSort job, the shuffle data size is the same as the input data size, i.e., 10 GB. A typical input data block is 256 MB, which results in 10 GB / 256 MB $= 40$ map tasks. Suppose the number of reduce tasks is 10 (a typical setting with this input data size). This means there are in total $40 * 10 = 400$ flows (all-map-to-all-reduce communication), with each flow of size 25 MB (on average). If we do not attempt to aggregate the flows, this flow size is much smaller than the elephant flow threshold. This example illustrates that even many shuffle-heavy jobs have a huge shuffle data size (e.g., 10 GB), it still may not be able to take advantage of OCS.

However, if we *proactively* place the map tasks in 2 racks and the reduce tasks in 2 racks and batch the flows to 4 "elephant" flows across the 4 racks (batching can be enabled in Hybrid-DCN as introduced in c-Through [9]), then each of these 4 elephant flows would be 2.5 GB and it is sufficient to enable the flows to use OCS. Thus, we have the first goal of designing the job scheduler for Hybrid-DCN:

Goal-1: *The job scheduler needs to schedule the map and reduce tasks of a job on as few racks as possible to aggregate the data transfer to take advantage of OCS.*

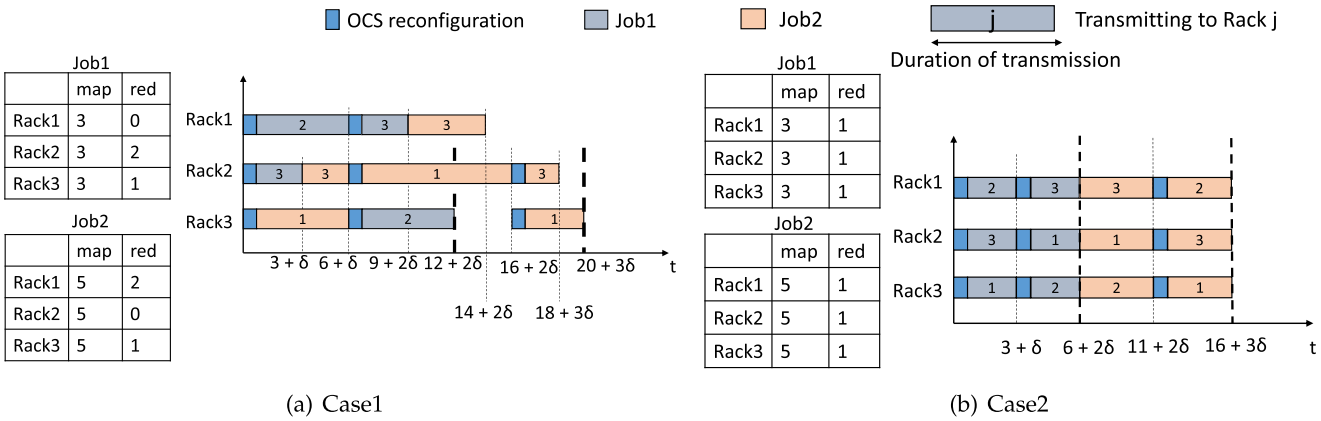
B. Maximizing the Number of Circuits for Coflow

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 shuffle-heavy 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 (i.e., in total 27 and 45 flows with data size of 1 in $Job1$ and $Job2$, respectively). In each unit time, OCS can transfer 1 unit of data. 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 [20] 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 a higher priority to use the circuits non-preemptively. Thus, the Coflow of $Job1$ has a higher priority in this example.

Case1 (Figure 2(a)): The map and reduce tasks of $Job1$ and $Job2$ are scheduled as shown in the two tables in Figure 2(a). According to the Gantt chart on the right, the CCTs of $Job1$ and $Job2$ are $12 + 2\delta$ and $20 + 3\delta$, respectively. We describe below how to interpret the Gantt chart in these



(a) Case1 (b) Case2

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

figures. For example, the second row of the Gantt chart can be interpreted as follows: (1) this row represents the durations of flows sent out from Rack2; (2) the OCS circuit is first configured from Rack2 to Rack3. $Job2$ needs to transfer 3 units of data from Rack2 to Rack3 (3 map on Rack2 to 1 reduce on Rack3). The remaining circuit time is used by $Job2$ to transfer 3 units of data from Rack2 to Rack3 (out of 5 units in total, 5 map on Rack2 to 1 reduce on Rack3); (2) the OCS circuit is then reconfigured from Rack2 to Rack1 to transfer 10 units of data by $Job2$ (5 map on Rack2 to 2 reduce on Rack1); (3) the OCS circuit is then reconfigured from Rack2 to Rack3 again to transfer the remaining 2 units of data leftover in step (2) above.

Case2 (Figure 2(b)): The map and reduce tasks of $Job1$ and $Job2$ are scheduled as shown in the two tables in Figure 2(b). According to the Gantt chart on the right, 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 those in Case1, which 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. Thus, we have:

Goal-2: *The job scheduler needs to distribute the data transfer of a job to as many 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 many circuits as possible to minimize the CCT.*

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

- **Issue1:** How many racks should the map tasks and reduce tasks of a job be placed?
- **Issue2:** How many tasks should we place on each rack to minimize the CCT of a job?
- **Issue3:** 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. Co-Scheduler Design

In this section, we propose Co-scheduler, a job scheduler that schedules the tasks of jobs on Hybrid-DCN and enables the jobs to take advantage of OCS in Hybrid-DCN while minimizing CCTs. We note that Co-scheduler is only for the "scheduling" of jobs but not Coflows. It determines the task and data placement of jobs (Sections IV-D, IV-E, and IV-F) and the order to execute the tasks among jobs (Section IV-G). Coflow scheduling is orthogonal to our work, and can be combined with Co-scheduler to further improve the job performance.

A. Rethinking the Overlapping of Map and Reduce Tasks

In YARN, a reduce task is associated with its corresponding shuffle and the shuffle starts fetching data once the corresponding reduce task is scheduled, while fractions of the map tasks of the job may be still running. The goal of this mechanism is to *overlap* the shuffle data transfers with the map progress to shorten the execution time of the job. However, this mechanism may cause two problems.

- In a highly busy cluster, it results in low resource utilization since the reduce tasks take up the containers to just wait for data transfers when these resources could otherwise process other jobs.
- It does not facilitate shuffle traffic aggregation, since it starts each shuffle data flow individually and immediately after its reduce, instead of waiting for data aggregation to take advantage of OCS.

Due to these problems, we raise a question: *With high-bandwidth OCS, is this overlapping mechanism still beneficial?* We argue that abandoning the overlapping mechanism would be more beneficial for workloads with a large portion of shuffle-heavy jobs, since it could enable the data transfers of jobs to take advantage of OCS in Hybrid-DCN.

We propose to start scheduling the reduce tasks of a job after all its map tasks are completed. In addition, unlike YARN that starts the corresponding shuffle data transfer *immediately* after each corresponding reduce task is scheduled, we further

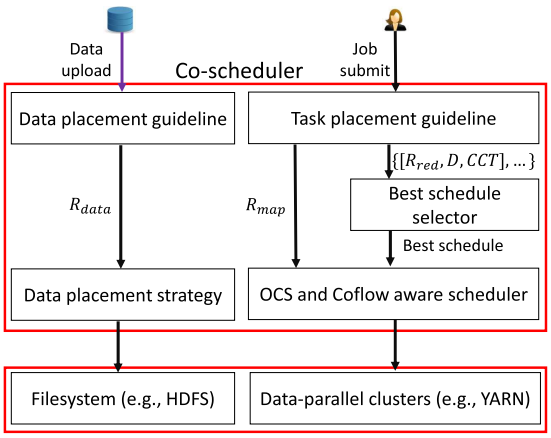


Fig. 3. Co-scheduler system architecture.

propose to start the shuffle data transfer of the job *after all* the reduce tasks are assigned to containers. This late start mechanism has several benefits:

- It relieves the symptoms that reduce tasks take up 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.

B. Opportunity

Several previous studies [16], [25]–[27] show that cluster workloads contain a large portion of *recurring* jobs (e.g., 40%), whose job characteristics (e.g., input/shuffle/output data sizes, job arrival time, the number of map/reduce tasks, and the map/reduce task execution time) can be predicted with a small error (e.g., 6.5% [16]). This provides an opportunity to better determine the number of racks to place the input datasets and to run the tasks for the recurring jobs *before* the input datasets and the jobs are submitted to the cluster.

C. Overall Architecture

Co-scheduler has different strategies for recurring and non-recurring jobs. Figure 3 shows the overall architecture. The ultimate goal of Co-scheduler is to aggregate the data transfers of jobs to a few racks to take advantage of OCS. To achieve so, Co-scheduler consists of four main components and we describe how each component works below.

Data Placement Guideline Generator (Section IV-D). To aggregate the data transfers of a job, we need to place its map and reduce tasks on a few racks, as aforementioned. Due to the data locality (a map task and its input data block are on the same rack) preference, the input data placement should also be constrained in a few racks; otherwise, we either lose data locality or fail to aggregate the map tasks, which may hurt the job performance. Thus, when the input data of a job is submitted to the cluster, Co-scheduler generates a data placement guideline on how many racks to place its input data, so that the job can take advantage of OCS while achieving high data locality. For non-recurring jobs, Co-scheduler dynamically infers from the job characteristics to maximize the possibility of taking advantage of OCS. For recurring jobs, Co-scheduler leverages an improved latency response model to derive the number of racks that can maximize the benefits of OCS.

Task Placement Guideline Generator (Section IV-E). Task Placement Guideline Generator is to generate the *possible schedules* for both map and reduce tasks of a job, so that the job can take advantage of OCS to transfer its shuffle data. For map tasks, the guideline is to select R_{map} racks randomly among those racks that have the job's input data, where R_{map} is the number of racks to run its map tasks on. For reduce tasks, the guideline is a set of $\{[R_{red}, \mathcal{D}, CCT]\}$, where R_{red} is the number of 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.

Best Reduce Task Schedule Selector (Section IV-F). For each job, Co-scheduler uses an *ExploreSchedule* function to explore each *possible schedule* of reduce tasks: for each possible schedule, it aims to find out how to place tasks that yields the shortest job completion time. Co-scheduler then selects the *best schedule* among them, so that the reduce tasks of the job can start the earliest and hence the job completion time is minimized.

OCS and Coflow Aware Scheduler (Section IV-G). Finally, Co-scheduler schedules each job by following the guidelines of its map tasks and the best schedule of its reduce tasks.

D. Data Placement Guideline Generator

Suppose R_{data}, R_{map} and R_{red} denote the number of racks to place the input data, the map tasks and the reduce tasks, respectively. As mentioned above, we need to place its map and reduce tasks on a few racks to aggregate the data transfers. Ideally, it is desired to have data locality for a job, which means that the map tasks and their input data blocks are on the same set of racks, i.e.,

$$R_{data} = R_{map}. \tag{3}$$

In this section, we present the detail on how to compute a guideline, i.e., the number of racks R_{data} (hence R_{map}), to place the input data for *every* recurring and non-recurring job, respectively.

1) Non-Recurring Job: For a non-recurring job, the only characteristic that is known priori is the input data size, hence we need to dynamically infer its characteristics. Let us denote Shuffle data size to Input data size Ratio as SIR. 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 initial ratio could be changed accordingly based on the workload characteristics in the cluster.

To ensure that a job can use OCS to transfer its data, it requires

$$\frac{Input * SIR}{R_{map} * R_{red}} \geq T_e, \tag{4}$$

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

Based on Section III-B, it is always desired to use more optical circuits at a time to transfer data concurrently, so that the CCT of the job is minimized. 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, hence the number of circuits that can be used at a time is at most $\min(R_{map}, R_{red})$. As a result, maximizing the number of circuits can be interpreted as

$$\text{maximize} \qquad \min(R_{map}, R_{red}), \tag{5}$$

Based on Equation (4), we have

$$(\min(R_{map}, R_{red}))^2 \leq R_{map} * R_{red} \leq \frac{Input * SIR}{T_e}. \tag{6}$$

$(\min(R_{map}, R_{red}))^2 = R_{map} * R_{red}$ occurs only when $R_{map} = R_{red}$. Hence, the number of circuits that can be used must satisfy

$$\min(R_{map}, R_{red}) \leq \sqrt{\frac{Input * SIR}{T_e}}. \tag{7}$$

Equation (7) implies that $\sqrt{\frac{Input*SIR}{T_e}}$ is the theoretical upper bound for the maximum number of circuits a job can use to transfer its data. However, the achievable value in real time may be lower than this upper bound. To *maintain the potential* for a job to use the maximum number of circuits, Co-scheduler initializes $R_{data} = R_{map} = \lfloor\sqrt{\frac{Input*SIR}{T_e}}\rfloor$ for a job.

2) Recurring Job: We use an improved version of *latency response function* (LRF) [16] to model the latency for every recurring job j. LRF takes the number of racks allocated to job j as an input and predicts the latency of job j. We describe below why the LRF model is suitable for Co-scheduler and what we have done to improve the LRF model in our paper:

• To simplify the latency predictions in LRF model, the previous paper [16] assumes that the map, shuffle, and reduce stages run sequentially. This assumption may not be perfect for the default Hadoop MapReduce setting. However, in this paper, we propose a specialized mechanism to decouple the shuffle from overlapping with the reduce stage to take advantage of OCS in Section IV-A, which matches perfectly with the assumption in LRF model.

• The LRF model in the previous paper [16] assumes that the map and reduce tasks of a job are scheduled on the same set of racks (i.e., $R_{map} = R_{red}$), which avoids using the network by decreasing the cross-rack data transfers between map and reduce tasks. However, this assumption is no longer suitable for Co-scheduler. With the high bandwidth of OCS, we would like to aggregate data transfers to take advantage of OCS, rather than avoiding using the network. Hence, we improve LRF to predict the latency as a model of both R_{map} and R_{red}.

Below is the detail of the improved LRF model.

$$L(R_{map}, R_{red}) = l^{map}(R_{map}) + l^{shu}(R_{map}, R_{red}) + l^{red}(R_{red}), \tag{8}$$

where $l^{map}(R_{map})$, $l^{shu}(R_{map}, R_{red})$ and $l^{red}(R_{red})$ denote the latency for the three stages, respectively. $l^{map}(R_{map})$ and $l^{red}(R_{red})$ can be easily computed from the estimated job characteristics (input/shuffle/output data sizes and the number of tasks) [16]. $l^{shu}(R_{map}, R_{red})$ can be computed as

$$l^{shu}(R_{map}, R_{red}) = \frac{D_(R_{map}, R_{red})}{BW}, \tag{9}$$

$$D(R_{map}, R_{red}) = \frac{D_{shu}}{R_{map} \cdot R_{red}} * (R_{red} - 1), \tag{10}$$

where BW is bandwidth, D_{shu} is shuffle data size of job j, and $D(R_{map}, R_{red})$ is cross-rack shuffle data size. Whether BW_{EPS} or BW_{OCS} should be used for BW depends on whether OCS or EPS is used in the shuffle stage of job j. We check if the shuffle data size of job j divided by $R_{map} * R_{red}$ (i.e., $\frac{D_{shu}}{R_{map}*R_{red}}$) is greater than the elephant flow threshold. If yes, OCS is used; otherwise, EPS is used. Based on Equ. (8), we can compute a latency matrix for every job j under different combinations of R_{map} and R_{red}.

R^{red}

R^{map}	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	630	630	630	630	630	630	630	630	630	630	630	630	630	630	630
2	379	373	373	373	373	373	373	373	373	373	373	373	373	373	373
3	295	289	287	287	287	287	287	287	287	287	287	287	287	287	287
4	295	289	287	286	286	286	286	286	286	286	286	286	286	286	286
5	295	289	287	286	285	285	285	285	285	285	285	285	285	285	285
6	211	205	203	202	202	201	201	201	201	201	201	201	401	401	401
7	211	205	203	202	202	201	201	201	201	201	372	372	372	372	372
8	211	205	203	202	202	201	201	201	201	351	351	351	351	351	351
9	211	205	203	202	202	201	201	201	334	334	334	334	334	334	334
10	211	205	203	202	202	201	201	320	320	320	320	320	320	320	320
11	211	205	203	202	202	201	309	309	309	309	309	309	309	309	309
12	211	205	203	202	202	201	300	300	300	300	300	300	300	300	300
13	211	205	203	202	202	292	292	292	292	292	292	292	292	292	292
14	211	205	203	202	202	286	286	286	286	286	286	286	286	286	286
15	211	205	203	202	202	280	280	280	280	280	280	280	280	280	280

Fig. 4. Latency matrix of an example job under different assignments. The job consists of 3472 map tasks and 169 reduce tasks. The row and column represent R_{map} and R_{red}.

For instance, Figure 4 shows the latency matrix of an example shuffle-heavy job under different combinations on a 15-rack cluster, with 600 task slots on each rack. The number of map tasks for this job is 3472, which is greater than the total number of containers in 5 racks ($5 * 600 = 3000$). Therefore, as R_{map} increases from 1 to 5, the latency of the job drops significantly due to higher parallelism. Since the job has only 169 reduce tasks, there are sufficient slots (i.e., 600 per rack) to run all the reduce tasks on one rack concurrently (i.e., $R_{red} = 1$). When R_{red} increases to 12 racks, the shuffle data is too spread to exploit OCS and hence we see a significant latency jump when $R_{red} > 11$.

With this improved LRF model, Co-scheduler can find out all the feasible combinations of R_{map} and R_{red} for job j that can leverage OCS while achieving high parallelism, as shown in the green zone of Figure 4. We see that the latencies in this zone are much smaller than the other entries. For simplicity, Co-scheduler selects the smallest R_{map} in this zone, since Co-scheduler aims to aggregate the data transfers to take advantage of OCS and the smallest R_{map} leads to the highest degree of data aggregation.

3) Input Data Placement: In general, each data block has three replicas. As mentioned above, it is always desired to have data locality for a map task, which means that the map task and its input data block are on the same set

of racks. Having three replicas of every input data block can not only increase data reliability, but can also increase the chance to achieve data locality of the corresponding map task.

In YARN, input data blocks are randomly stored in the cluster, *without a constraint on how many racks to place them*. We have shown in the above two subsections that the input data placement needs to place the input data on R_{data} racks to take advantage of OCS while achieving sufficient parallelism. Thus, for the input data of a job, Co-scheduler first randomly chooses R_{data} racks and places the first replica of its input data blocks *evenly* onto the R_{data} racks. For the second and third replicas, Co-scheduler randomly selects two other distinct 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 completely *disjoint* with each other (hence in total $3 * R_{data}$ distinct racks to place all replicas). Such an input data placement strategy can help facilitate the job to take advantage of OCS, while still maintaining three replicas.

E. Task Placement Guideline Generator

In this section, we describe how Co-scheduler generates the task placement guideline for every job, i.e., the number of racks to run the map and reduce tasks, which can aggregate data transfers sufficiently to take advantage of OCS. When the map (or reduce) tasks are available to schedule, the generator is used to compute a guideline for them. Then the OCS and Coflow aware scheduler (details in Section IV-G) follows the guideline to schedule those tasks.

1) Map Task Guideline: It is desired to maintain high data locality for any job to achieve high performance. Thus, the guideline for map tasks of a job is straightforward: Since Co-scheduler generates a guideline to place the input data, as described in Section IV-D, the guideline for map tasks of a job is to schedule them on any R_{map} racks selected from the $3 * R_{data}$ distinct racks that contain the job's input data.

2) Reduce Task Guideline: Recall in Section IV-A, we propose to delay scheduling the reduce tasks of a job *until* the last map task of the job is completed. One of the benefits is that the scheduler can know the distribution of data transfer sizes $\{SM_1, SM_2, \ldots, SM_{R_{map}}\}$ on the R_{map} racks of the job, as well as whether the job is shuffle-heavy or not. If any $SM_i, i = 1, 2, \ldots, R_{map}$ is smaller than T_e, we can disregard it in the computation because i) regardless of the reduce task placement, the data transfer from this rack cannot use 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, let us assume that all $SM_i, i = 1, 2, \ldots, R_{map}$ are greater than the elephant flow threshold T_e, and $\{SM_1, SM_2, \ldots, SM_{R_{map}}\}$ are sorted in ascending order.

Given the data transfer sizes $\{SM_1, SM_2, \ldots, SM_{R_{map}}\}$ on R_{map} racks of a job, Co-scheduler needs to solve the following two problems.

- *First*, determine all the possible values of R_{red} that can take advantage of OCS.
- *Second,* for each value of R_{red}, find out how many reduce tasks to place on each of the R_{red} racks, denoted as $\mathcal{D} = \{d_1, d_2, \ldots, d_{R_{red}}\}$, where d_i indicates the number of reduce tasks to place on a rack, so that the CCT is minimized.

All possible values of $\mathbf{R_{red}}$ for every non-recurring job. We describe below how we determine all possible values of $\mathbf{R_{red}}$ for a non-recurring job. Obviously, the lower bound of R_{red} is 1. As to the upper bound, since we expect that the sizes of as many flows as possible of the job are larger than the elephant flow threshold T_e, we have

$$R_{red} \leq \lfloor \frac{SM_1}{T_e} \rfloor. \tag{11}$$

Here, we use SM_1 because it is the minimum among $\{SM_1, SM_2, \ldots, SM_{R_{map}}\}$ (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, R_{red} must be in the range of $[1, \lfloor \frac{SM_1}{T_e} \rfloor]$.

All possible values of $\mathbf{R_{red}}$ for every recurring job. For a recurring job, all possible values of $\mathbf{R_{red}}$ can be derived from Equ. 8 (e.g., green zone in Figure 4). In practice, the predicted job characteristics may have some estimation variances, such as the number of tasks and shuffle data size. At this point, Co-scheduler can adjust the range of possible R_{red} to accommodate the estimation variances.

Find the placement $\mathcal{D}$ for each possible $\mathbf{R_{red}}$ that minimizes CCT. We use the following algorithm to find $\mathcal{D}$ for each R_{red} value in a range of $[1, \alpha]$. First, for each of the R_{red} racks, we *simulate* to keep placing the reduce tasks to each rack, until the data transfers from R_{map} racks to R_{red} racks 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 (15), the CCT lower bound $T(\mathbf{C})$ is proportional to 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 computed based on Equations (1) and (15).

Summary for reduce task guideline. For each job, a set of *possible schedules* $\{[R_{red}, \mathcal{D}, CCT], \ldots\}$ is generated for the reduce tasks. In the next subsection, Co-scheduler selects the *best schedule* among them.

F. Best Reduce Task Schedule Selector

We present the details of selecting the best schedule, as shown in Algorithm 1. Given all the *possible schedules* of the reduce tasks of a job in Section IV-E, 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*

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: ▷ Note: selected rack cannot be used again in the search
10: **end for**
11: $t_{max} = \max\{t_1, t_2, \ldots, t_{R_{red}}\}$
12: **return** selected racks $\mathcal{R} = \{r_1, r_2, \ldots, r_{R_{red}}\}$ 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.

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:

$$T_{rem} = t_{elapsed} * \frac{1-P}{P}. \tag{12}$$

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.

In the following, we describe the details of *ExploreSchedule* function. Without loss of generality, let us assume the sorted $\mathcal{D}$ (descending order) is $\{d_1, d_2, \ldots, d_{R_{red}}\}$ (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 $\mathcal{R} = \{r_1, r_2, \ldots, r_{R_{red}}\}$ and their estimated released times $\mathcal{T} = \{t_1, t_2, \ldots, t_{R_{red}}\}$ to run $\{d_1, d_2, \ldots, d_{R_{red}}\}$ reduce tasks. The problem is interpreted as selecting the set of racks $\mathcal{R}$ in the cluster with the goal

$$\text{minimize} \qquad \max\{t_1, t_2, \ldots, t_{R_{red}}\}. \tag{13}$$

Given a possible schedule $[R_{red}, \mathcal{D} = \{d_1, \ldots, d_{R_{red}}\}, CCT]$, 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 $r_2, \ldots, r_{R_{red}}$ for the subsequent $d_2, \ldots, d_{R_{red}}$ reduce tasks using the same method (lines 3-10). Finally, it outputs the selected racks $\mathcal{R} = \{r_1, r_2, \ldots, r_{R_{red}}\}$ and the estimated time of when all reduce tasks complete their shuffle data transfers, i.e., $CCT + t_{max}$, where $t_{max} = \max\{t_1, t_2, \ldots, t_{R_{red}}\}$ (lines 11-12).

Next, we prove that *ExploreSchedule* can always find the optimal solution that matches the schedule.

Proof: Without loss of generality, assume that $t_i = \max\{t_1, t_2, \ldots, t_{R_{red}}\}$. First, rack r_i cannot be replaced by any racks in $\{r_{i+1}, \ldots, r_{R_{red}}\}$ or any other racks in the cluster (i.e., $R/\{r_1, \ldots, r_{R_{red}}\}$), 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 in $\{r_1, \ldots, r_{i-1}\}$ 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. In this case, since $d_j \geq d_i$ (recall $\mathcal{D} = \{d_1, d_2, \ldots, d_{R_{red}}\}$ 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 $\mathcal{R} = \{r_1, r_2, \ldots, r_{R_{red}}\}$, t_i is larger. □

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

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**

G. OCS and Coflow Aware Scheduler

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] and [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.

H. Computation Complexity

In this section, we discuss the computation complexity of each component in Co-scheduler. For each job, generating the guidelines for input data placement and map task placement (Sections IV-D and IV-E) takes $O(R^2)$ time at worse, where R is the number of racks in a cluster. The reduce task placement generator needs to select the number of racks to place the reduce tasks, i.e., R_{red}. We propose a heuristic Algorithm 1 to speed up this process (Sections IV-E and IV-F), which takes $O(R * R_{red}^2)$ at worse for each job. In practice, the number of racks in a cluster is generally small ($<$100) and R_{red} is even smaller (as Co-scheduler would like to aggregate the reduce tasks), so the computation complexities of all the above components are acceptable.

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 1.125 GB, which is inferred empirically from previous studies [10], [20], [32]. The reconfiguration delay of OCS was set to 10 ms, which is a typical delay of a 3D-MEMS OCS [10].

Baselines. We compared Co-scheduler with two baselines.

(1) *Fair scheduler* [15] (Fair in short) is the most widely used scheduler in current production clusters [15], and it assigns containers to jobs so that each job roughly receives an equal share of containers over time.

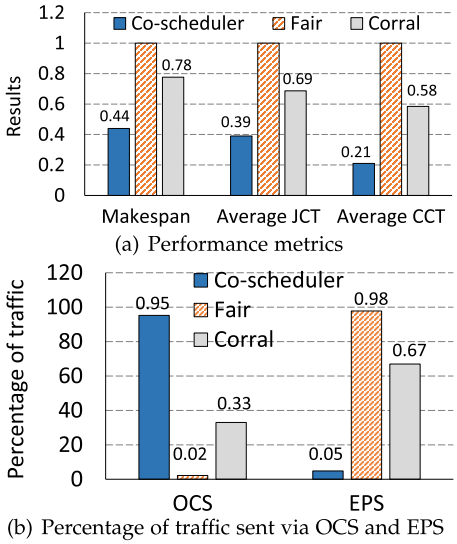


Fig. 5. Experimental results.

(2) *Corral* [16] 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 [20] 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

$$\frac{|\text{Metric}_{Baseline} - \text{Metric}_{Co-scheduler}|}{\text{Metric}_{Baseline}}, \tag{14}$$

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

B. Experimental Results

We present the experimental results below. The 1000 jobs arrived uniformly at random in $[0, 90]$ minutes. We assume 50% of the jobs are recurring in the following experiments. 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 5(a) shows the makespan of the workload, average JCT, and average CCT with different schedulers. Co-scheduler reduces the makespan of Fair and Corral by 56% and 44%, respectively. Co-scheduler achieves 61% and 43% reduction on the average job completion time, compared with Fair

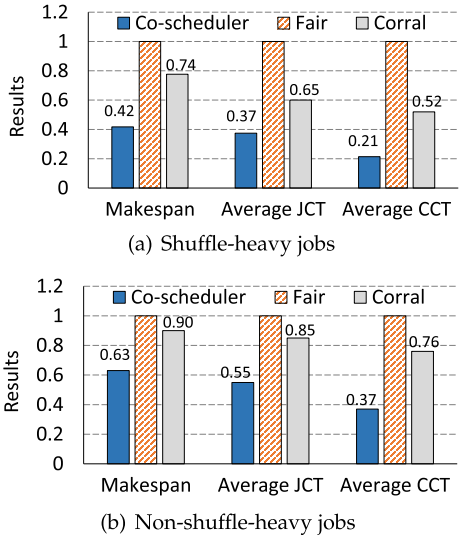


Fig. 6. Performance improvements of shuffle-heavy and non-shuffle-heavy jobs.

and Corral, respectively. Compared with Fair and Corral, Co-scheduler has 79% and 63% reduction on the average CCT, respectively.

These 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 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 explanations can also be demonstrated through Figure 5(b), which shows the network traffic sent via OCS and EPS. We see that 95.2% 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-scheduler over Fair and Corral for shuffle-heavy and non-shuffle-heavy jobs, respectively. Figures 6(a) and 6(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 Co-scheduler enables the use of OCS. As most of the data transfers in shuffle-heavy jobs take advantage of OCS, finish faster, and release the containers earlier, non-shuffle-heavy jobs also benefit from the more network bandwidth and computing resources. 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.

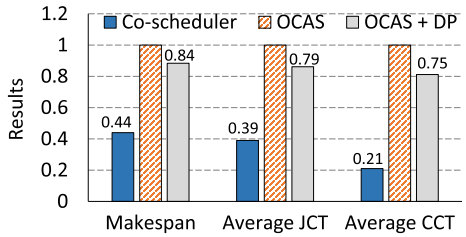


Fig. 7. Effectiveness of different mechanisms.

C. Effectiveness of Different Mechanisms

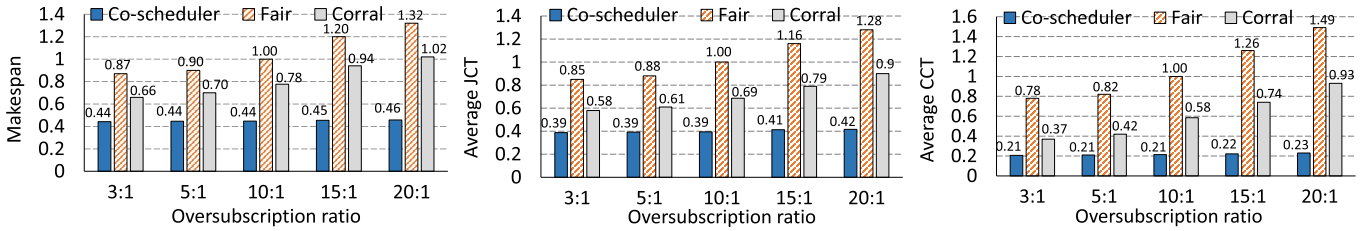
We also evaluate the impact of different mechanisms in Co-scheduler: data placement guideline generator (DP), task placement guideline generator (TP), best reduce task schedule selector (BSS), and OCS and Coflow aware scheduler (OCAS). We only evaluate the performance of OCAS, and DP + OCAS, compared with DP + TP + BSS + OCAS, where OCAS is Fair scheduler when there is no guideline of map and reduce tasks for shuffle-heavy jobs, DP + OCAS consists of data placement and map task guideline, and DP + TP + BSS + OCAS is Co-scheduler. We can only evaluate these combinations of Co-scheduler components because (i) DP cannot work without OCAS; and (ii) TP and BSS cannot work without DP.

Figure 7 shows the contributions of different mechanisms, in terms of the makespan of the workload, average JCT, and average CCT. Compared with OCAS, DP + OCAS achieves performance improvements on makespan, average job completion time, and average CCT by 16%, 21%, and 25%, respectively. This is because DP + OCAS attempts to place the input data and map tasks in a limited number of racks, which aggregates the shuffle data transfers of the shuffle-heavy jobs to some extent. However, DP + OCAS has much worse performance than DP + TP + BSS + OCAS (i.e., Co-scheduler), since DP + 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, DP + OCAS cannot effectively enable shuffle-heavy jobs to take advantage of OCS, leading to worse performance than Co-scheduler.

D. Sensitivity Analysis

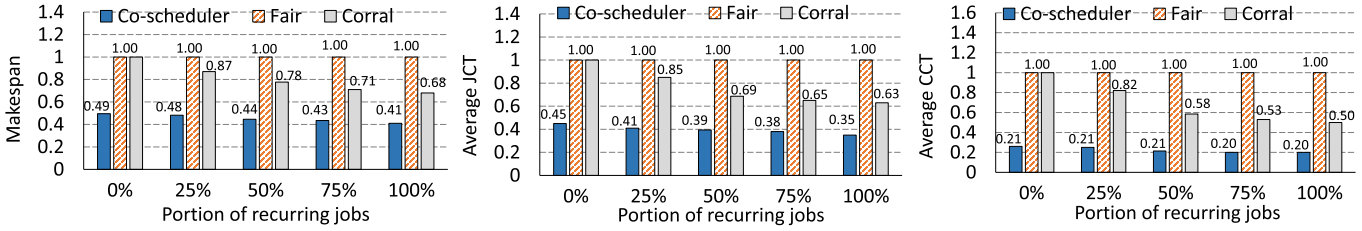
In this section, we conduct several sensitivity tests of Co-scheduler. The experiment settings are the same as Section V-A unless otherwise specified.

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 10(a), 10(b), 10(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 most of the shuffle network traffic is sent via OCS and only a small amount of network traffic is sent via EPS in Co-scheduler. However, as the oversubscription ratio increases, the performance of Fair and Corral are significantly degraded, since a large portion of



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

Fig. 8. Sensitivity analysis of oversubscription ratio.



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

Fig. 9. Sensitivity analysis of portion of recurring jobs.

network traffic is still sent via EPS in Fair and Corral and they cannot take full advantage of OCS.

Sensitivity to the portion of recurring jobs in the workload. We varied the portion of recurring jobs from 0% to 100% in the workload. Figures 9(a), 9(b), 9(c) show the makespan, average JCT, and average CCT versus the portion of recurring jobs. Note that Corral is equivalent to Fair when there are not any recurring jobs (i.e., 0%). The performance of Fair does not change with the portion of recurring jobs, since it does not rely on the predictable characteristics of recurring jobs. On the contrary, both Co-scheduler and Corral leverage the predictable feature to help with their decisions, hence it is not surprising that the performance of both schedulers increases as the portion of recurring jobs increases. Co-scheduler benefits less than Corral does as the portion of recurring jobs increases, primarily because Corral does not have any specific handling to non-recurring jobs, while Co-scheduler generates guidelines for non-recurring jobs as well as recurring jobs and thus relies less heavily on the predictable feature.

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}.

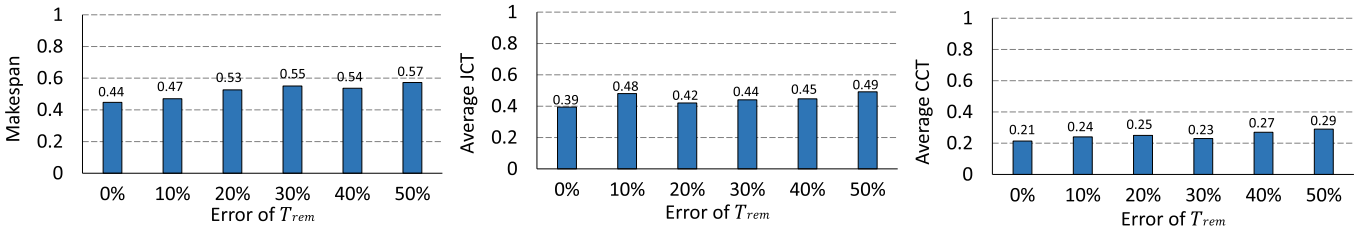
Figures 10(a), 10(b), 10(c) show the makespan, average JCT, and average CCT versus the estimation error. The results of Fair and Corral are not shown in the figures as they do not rely on the estimation of T_{rem}. We see that the performance improvements of Co-scheduler on makespan and average job completion 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, 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 43% makespan and 51% 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. Discussions

General DAGs. While we present our designs and results in the Hadoop MapReduce framework in this paper, the ideas can be generalized to many other DAG-based data-parallel frameworks. Some frameworks (e.g., Hive [33] and Pig [34]) allow an application to be partitioned into several MapReduce jobs, and thus Co-scheduler can be directly applied there to each of the partitioned MapReduce jobs.

Other frameworks (e.g., Tez [35] and Spark [36]) allow a job to be expressed as a complex DAG. Compared with a MapReduce job, a DAG job typically involves multiple communication stages. We can also extend Co-scheduler to adapt these DAG-based frameworks. In this scenario, the aim of Co-scheduler remains the same—enabling the communication stages in a DAG job to take advantage of OCS to improve the job performance. To achieve so, Co-scheduler needs to apply the guideline generating mechanisms in Sections IV-E and IV-F to the parent tasks and child tasks of *every* communication stage in a DAG job, guiding the tasks to be placed on a certain number of racks and hence aggregating the traffic whenever possible. As analyzed in Section IV-H, the guideline generating mechanisms generate minimal computation costs and do not lead to high overhead on the scheduler.



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

Fig. 10. Sensitivity analysis of estimation of T_{rem}.

Wavelength selective switching (WSS). In our paper, we assume that each rack can configure only one circuit to another rack at a time. Recently, several papers [37], [38] propose to use WSS to enable a rack to switch multiple wavelengths to several racks at a time. We discuss below how to extend Co-scheduler to this scenario. The overall goal of Co-scheduler remains unchanged since the reconfiguration delay still exists and elephant flows are preferred in WSS. Co-scheduler aims to aggregate the map and reduce tasks to a few racks, meaning that one rack may have data transfers to several other racks at a time, while WSS indeed supports data transfers from one rack to several other racks at a time. Thus, Co-scheduler fits WSS perfectly.

Nevertheless, there are two aspects we need to adapt Co-scheduler slightly to WSS. First, the way to estimate the CCT of a Coflow in Co-scheduler needs to be updated:

$$T(\mathbf{C}) = \max\left(\frac{1}{W}\max_i \sum_j t_{ij}, \frac{1}{W}\max_j \sum_i t_{ij}\right), \quad (15)$$

where W is the number of wavelengths WSS can support. This is because the amount of traffic $\sum_j t_{ij}$ from rack i to all other racks can now be sent through multiple wavelengths. Second, in Algorithm 1 we select the rack with the smallest resource release time. As now there could be multiple choices of racks, we could relax this algorithm a bit to select the top W racks.

In summary, the goal of Co-scheduler does not deviate from the main advantage of WSS. On the contrary, we envision that an extended version of Co-scheduler and WSS can complement each other well to further improve the job performance and we leave this as one of our future work.

VII. Related Work

Hybrid-DCN. Previous studies [9]–[11] 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.

Job schedulers. Many schedulers [4], [6], [15]–[17], [39]–[44] have been designed to improve job performance of data-parallel clusters with different objectives, such as high data locality, deadline driven, energy efficient, and network aware. Unfortunately, none of these schedulers focus on scheduling tasks to take advantage of optical networks. For example, the current state-of-the-art schedulers in Hadoop YARN, Fair scheduler [15] and Delay scheduler [4], focus on achieving high data locality and schedule the tasks of jobs across an entire cluster. Similarly, ShuffleWatcher [6] aims to evenly distribute the shuffle network traffic spatially and temporally among different racks to reduce the network contention. The above schedulers totally disaggregate the data transfers of the jobs, which fails to take advantage of Hybrid-DCN to solve the network bottleneck for high performance. Corral [16] 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, the key idea behind Corral is to avoid using the network, rather than utilizing OCS to accelerate cross-rack traffic transfer.

Cluster configuration. Besides job scheduling, many researchers focus on how to configure the Hadoop framework on various clusters to achieve high performance [40]–[42], [45]–[47]. For example, the performance of Hadoop framework is highly dependent on the cluster configurations. Li *et al.* [40], [41] have proposed to use hybrid scale-up and scale-out heterogeneous machines to process the workloads that are dominated by many small jobs. Chen *et al.* [46] have proposed Silhouette, a system that trains a prediction model based on small subset of tasks and then selects the best configurations for analytic frameworks. Some other works [42], [45], [47] have proposed methods on selecting file systems or modeling data replica strategies on various clusters, such as HPC and virtualized clusters. Our paper is orthogonal to these papers, whose conclusions and findings can be adapted with Co-scheduler for high performance.

Coflow. The Coflow abstraction was first documented in [18], [48]. The objective of Coflow scheduling is often to minimize the average CCT. This problem is proved to be NP-hard [19], [20], as it can be reduced from the concurrent open-shop scheduling problem [49]. Many heuristic scheduling algorithms [19], [23], [50]–[57] were proposed to minimize the CCT in traditional EPS network to improve the performance of data-parallel jobs. As optical networks become more and more popular in datacenter networks, researchers extend the Coflow abstraction into optical networks and proposes several algorithms to schedule Coflows in optical networks [20], [58]–[61].

The above papers 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 facilitates the minimization of CCTs, which may lead to poor job performance. Different job scheduling algorithms may result in different source-destination pairs of flows and thus 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.

VIII. Conclusion

We propose Co-scheduler, a job scheduler that aims to improve job performance by exploiting OCS in Hybrid-DCN and minimizing the CCT. Co-scheduler leverages the fact that modern workloads consist of both recurring and non-recurring jobs, and uses pre-generated guidelines to guide input data and task placements. For a non-recurring job, Co-scheduler dynamically infers its job characteristics to maximize the possibility to take advantage of OCS for its shuffle data transfers and thus minimizing the CCT. For a recurring job, Co-scheduler uses an improved latency response model to guide the predictions of optimal data and task placements. 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 on Hybrid-DCN. In the future, we will explore using machine learning techniques to automatically learn how to conduct scheduling and investigate more sophisticated methods of estimating the remaining processing time.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. OSDI*, 2004, pp. 107.

[2] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. NSDI*, 2012, pp. 15–28.

[3] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *Proc. IEEE 19th Annu. Int. Symp. Modeling, Anal., Simulation Comput. Telecommun. Syst.*, Jul. 2011, pp. 15–28.

[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. EuroSys*, 2010, pp. 265–278.

[5] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Conf. Data Commun. (SIGCOMM)*, 2009, pp. 51–62.

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

[7] H. Shen and Z. Li, "New bandwidth sharing and pricing policies to achieve a win-w*in situ*ation for cloud provider and tenants," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2682–2697, Sep. 2016.

[8] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, Aug. 2013, pp. 231–242.

[9] G. Wang *et al.*, "C-through: Part-time optics in data centers," in *Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM)*, 2010, pp. 327–338.

[10] N. Farrington *et al.*, "Helios: A hybrid electrical/optical switch architecture for modular data centers," in *Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM)*, 2010, pp. 339–350.

[11] K. Chen *et al.*, "OSA: An optical switching architecture for data center networks with unprecedented flexibility," in *Proc. NSDI*, 2012, pp. 498–511.

[12] G. Porter *et al.*, "Integrating microsecond circuit switching into the data center," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, Aug. 2013, pp. 447–458.

[13] X. Wang, M. Veeraraghavan, Z. Lin, and E. Oki, "Optical switch in the middle (OSM) architecture for DCNs with Hadoop adaptations," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2017, pp. 1–7.

[14] Z. Li and H. Shen, "Job scheduling for data-parallel frameworks with hybrid electrical/optical datacenter networks," in *Proc. Symp. Cloud Comput.*, Sep. 2017, p. 662.

[15] *Fair Scheduler*. Accessed: Jan. 1, 2022. [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[16] 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. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 407–420.

[17] Z. Li, H. Shen, and A. Sarker, "A network-aware scheduler in data-parallel clusters for high performance," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2018, pp. 1–10.

[18] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. 11th ACM Workshop Hot Topics Netw. (HotNets-XI)*, 2012, pp. 31–36.

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

[20] X. S. Huang, X. S. Sun, and T. S. E. Ng, "Sunflow: Efficient optical circuit scheduling for coflows," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2016, pp. 297–311.

[21] *Apache Hadoop*. Accessed: Jan. 1, 2022. [Online]. Available: http://hadoop.apache.org/

[22] *Default Slowstart Threshold*. Accessed: Jan. 1, 2022. [Online]. Available: https://hadoop.apache.org/docs/r3.3.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml

[23] Q. Liang and E. Modiano, "Coflow scheduling in input-queued switches: Optimal delay scaling and algorithms," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2017, pp. 1–9.

[24] T. Inukai, "An efficient SS/TDMA time slot assignment algorithm," *IEEE Trans. Commun.*, vol. COM-27, no. 10, pp. 1449–1455, Oct. 1979.

[25] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proc. USENIX ATC*, 2012, pp. 281–294.

[26] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proc. 7th ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2012, pp. 99–112.

[27] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *Proc. OSDI*, 2016, pp. 65–80.

[28] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. OSDI*, 2008, p. 7.

[29] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using Mantri," in *Proc. OSDI*, vol. 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. NSDI*, 2011, p. 24.

[31] M. Zaharia, "Job scheduling with the fair and capacity schedulers," *Hadoop Summit*, vol. 9, p. 592, Jun. 2009.

[32] H. Liu *et al.*, "Scheduling techniques for hybrid circuit/packet networks," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2015, pp. 1–13.

[33] *Apache Hive*. Accessed: Jan. 1, 2022. [Online]. Available: https://hive.apache.org/

[34] *Apache Pig*. Accessed: Jan. 1, 2022. [Online]. Available: https://pig.apache.org/

[35] *Apache Tez*. Accessed: Jan. 1, 2022. [Online]. Available: https://tez.apache.org/

[36] *Apache Spark*. Accessed: Jan. 1, 2022. [Online]. Available: https://spark.apache.org/

[37] N. Farrington *et al.*, "A multiport microsecond optical circuit switch for data center networking," *IEEE Photon. Technol. Lett.*, vol. 25, no. 16, pp. 1589–1592, Aug. 15, 2013.

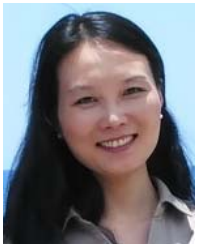
[38] K. Chen *et al.*, "OSA: An optical switching architecture for data center networks with unprecedented flexibility," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 498–511, Apr. 2014.

[39] C. Chen, W. Wang, S. Zhang, and B. Li, "Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2017, pp. 1–9.

[40] Z. Li and H. Shen, "Designing a hybrid scale-up/out Hadoop architecture based on performance measurements for high application performance," in *Proc. 44th Int. Conf. Parallel Process.*, Sep. 2015, pp. 21–30.

[41] 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 Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 386–400, Feb. 2017.
[42] Z. Li and H. Shen, "Measuring scale-up and scale-out Hadoop with remote and local file systems and selecting the best platform," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3201–3214, Nov. 2017.
[43] P. Jin, X. Hao, X. Wang, and L. Yue, "Energy-efficient task scheduling for CPU-intensive streaming jobs on Hadoop," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 6, pp. 1298–1311, Jun. 2019.
[44] D. Cheng, X. Zhou, Y. Xu, L. Liu, and C. Jiang, "Deadline-aware MapReduce job scheduling with dynamic resource availability," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 4, pp. 814–826, Apr. 2019.
[45] Z. Li and H. Shen, "Performance measurement on scale-up and scale-out Hadoop with remote and local file systems," in *Proc. IEEE 9th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2016, pp. 456–463.
[46] Y. Chen, L. Lin, B. Li, Q. Wang, and Q. Zhang, "Silhouette: Efficient cloud configuration exploration for large-scale analytics," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 2049–2061, Aug. 2021.
[47] C. Guerrero, I. Lera, B. Bermejo, and C. Juiz, "Multi-objective optimization for virtual machine allocation and replica placement in virtualized Hadoop," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 11, pp. 2568–2581, Nov. 2018.
[48] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 98–109, Aug. 2011.
[49] T. A. Roemer, "A note on the complexity of the concurrent open shop problem," *J. Scheduling*, vol. 9, no. 4, pp. 389–396, Aug. 2006.
[50] H. Tan, S. H.-C. Jiang, Y. Li, X.-Y. Li, C. Zhang, Z. Han, and F. C. M. Lau, "Joint online coflow routing and scheduling in data center networks," *IEEE/ACM Trans. Netw.*, vol. 27, no. 5, pp. 1771–1786, Oct. 2019.
[51] Y. Zhao, C. Tian, J. Fan, T. Guan, X. Zhang, and C. Qiao, "Joint reducer placement and coflow bandwidth scheduling for computing clusters," *IEEE/ACM Trans. Netw.*, vol. 29, no. 1, pp. 438–451, Feb. 2021.
[52] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 393–406.
[53] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: Near-optimal network design for coflows," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 16–29.
[54] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "CODA: Toward automatically identifying and scheduling coflows in the dark," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 160–173.
[55] Y. Li *et al.*, "Efficient online coflow routing and scheduling," in *Proc. 17th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, Jul. 2016, pp. 161–170.
[56] Y. Zhao *et al.*, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 424–432.
[57] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *Proc. IEEE INFOCOM 35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.
[58] T. Zhang, F. Ren, J. Bao, R. Shu, and W. Cheng, "Minimizing coflow completion time in optical circuit switched networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 457–469, Feb. 2021.
[59] H. Wang, X. Yu, H. Xu, J. Fan, C. Qiao, and L. Huang, "Integrating coflow and circuit scheduling for optical networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 6, pp. 1346–1358, Jun. 2019.
[60] H. Tan, C. Zhang, C. Xu, Y. Li, Z. Han, and X.-Y. Li, "Regularization-based coflow scheduling in optical circuit switches," *IEEE/ACM Trans. Netw.*, vol. 29, no. 3, pp. 1280–1293, Jun. 2021.
[61] C. Xu, H. Tan, J. Hou, C. Zhang, and X.-Y. Li, "OMCO: Online multiple coflow scheduling in optical circuit switch," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2018, pp. 1–6.
[62] Z. Li and H. Shen, "Co-scheduler: Accelerating data-parallel jobs in datacenter networks with optical circuit switching," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 186–195.

Zhuozhao Li received the Ph.D. degree in computer science from the University of Virginia. He was a Post-Doctoral Scholar with The University of Chicago. He is currently an Assistant Professor with the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology. His research interests include the broad areas of distributed systems, cloud computing, and high-performance computing, with an emphasis on developing working prototypes for real-world problems and designing the foundation methodologies to optimize system performance for efficient computing.



Haiying Shen (Senior Member, IEEE) received the B.S. degree in computer science and engineering from Tongji University, China, in 2000, and the M.S. and Ph.D. degrees in computer engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Associate Professor with the Department of Computer Science, University of Virginia. Her research interests include distributed computer systems, cloud computing, big data, and cyber-physical systems. She is a Microsoft Faculty Fellow of 2010, and a Senior Member of the ACM.