JobPacker: Job Scheduling for Data-Parallel Frameworks with Hybrid Electrical/Optical Datacenter Networks

Zhuozhao Li
University of Chicago
zhuozhao@uchicago.edu

Haiying Shen
University of Virginia
hs6ms@virginia.edu

ABSTRACT

In spite of many advantages of hybrid electrical/optical datacenter networks (Hybrid-DCN), current job schedulers for data-parallel frameworks are not suitable for Hybrid-DCN, since the schedulers do not aggregate data traffic to facilitate using optical circuit switch (OCS). In this paper, we propose JobPacker, a job scheduler for data-parallel frameworks in Hybrid-DCN that aims to take full advantage of OCS to improve job performance. JobPacker aggregates the data transfers of a job in order to use OCS to improve data transfer efficiency. It first explores the tradeoff between parallelism and traffic aggregation for each shuffle-heavy recurring job, and then generates an offline schedule including which racks to run each job and the sequence to run the recurring jobs in each rack that yields the best performance. It has a new sorting method to prioritize recurring jobs in offline-scheduling to prevent high resource contention while fully utilizing cluster resources. In real-time scheduler, JobPacker uses the offline schedule to guide the data placement and schedule recurring jobs, and schedules non-recurring jobs to the idle resources not assigned to recurring jobs. Trace-driven simulation and GENI-based emulation show that JobPacker reduces the makespan up to 49% and the median completion time up to 43%, compared to the state-of-the-art schedulers in Hybrid-DCN.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; • **Theory of computation** → *Scheduling algorithms*;

ACM Reference Format:
Zhuozhao Li and Haiying Shen. 2019. JobPacker: Job Scheduling for Data-Parallel Frameworks with Hybrid Electrical/Optical Datacenter Networks. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337880

1 INTRODUCTION

Recently, data-parallel frameworks such as MapReduce [10] and Spark [35] have been developed for analyzing large datasets. Network has been identified as a key factor for the performance of data-parallel frameworks for several reasons [22]. First, the jobs have network-intensive stages (e.g., shuffle in MapReduce) that transfer a large amount of data. For example, previous work has shown that 60% and 20% of the jobs are shuffle-heavy jobs (i.e., jobs with a large shuffle data size) on the Yahoo! [6] and Facebook MapReduce clusters [34], respectively. Second, datacenter networks commonly have link oversubscription ranging from 3:1 to 20:1 for the racks to the core [4, 10, 20, 29, 30, 34].

ICPP 2019, August 5–8, 2019, Kyoto, Japan
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337880

To increase network capacity, optical circuit switch (OCS) [14, 33] is an alternative to traditional packet switch in datacenter network due to its low capital expenditures (CapEx) and operating expenditures (OpEx). OCS has a certain number of input and output ports, and one input port can be connected to *only* one output port at a time. To change the input-to-output connection, one needs to *reconfigure* the OCS connection, causing a *reconfiguration delay* on the order of μs-to-ms, which is significantly higher than the latency of packet switching that is in the order of ns.

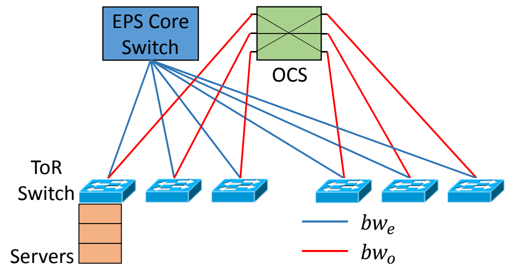


Figure 1: Architecture of Hybrid-DCN.

Recently, several studies propose hybrid electrical/optical datacenter network (in short *Hybrid-DCN*) designs [5, 14, 27, 33], which augment the traditional EPS network with an on-demand *rack-to-rack* network using the OCS. Figure 1 shows a general network abstraction of Hybrid-DCN: the top-of-rack (ToR) switches are connected with a core EPS and an OCS, forming packet-switching and circuit-switching network, respectively. In Hybrid-DCN, each rack connects to one input port and one output port, which means that one rack can send data via OCS to only one other rack at a time. Due to the high reconfiguration delay, in Hybrid-DCN, OCS is only used for large data transfers (e.g., 1.125GB) between racks so that the overhead of μs-to-ms reconfiguration delay is amortized.

Current state-of-the-art schedulers (e.g., Fair [1] and Corral [22]) in data-parallel frameworks fail to leverage OCS to accelerate the data transfer, since they either spread the tasks of a job (e.g., map and reduce tasks in MapReduce) among racks which generates many small flows or schedule the tasks of a job to avoid using cross-rack traffic which cannot exploit OCS to accelerate the data transfer. Thus, new job schedulers for data-parallel frameworks are required to meet the need of Hybrid-DCN.

To take full advantage of Hybrid-DCN, we could aggregate the data to be transferred by placing the tasks of a job in only a few racks. However, it may sacrifice the basic principle of data-parallel frameworks – parallelism (i.e., the tasks of a job running concurrently), since each rack may have a limited number of containers available at a time. If a rack does not have sufficient available resources to run all the assigned tasks concurrently, it increases the latency of the job (i.e., the duration from the start of a job until its completion). Hence, there is a tradeoff between parallelism and traffic aggregation. In this paper, we propose JobPacker to efficiently leverage OCS in Hybrid-DCN by balancing such tradeoff.

JobPacker consists of an offline scheduler (to schedule recurring jobs in the next unit period of time) and a real-time scheduler. The offline scheduler consists of a job profiler and a job manager.

The job profiler exploits the fact that many jobs are often recurring and have predictable job characteristics [3, 15, 22] to find all feasible (map-width, reduce-width) pairs (defined as the number of racks to run the map and reduce tasks) of each shuffle-heavy recurring job that can aggregate sufficient shuffle data to use OCS effectively while achieving sufficient parallelism. Then, the job manager finds the best (map-width, reduce-width) pair with the shortest completion time, and also generates a global schedule including which racks to run each recurring job and the sequence to run the map/reduce tasks of recurring jobs in each rack that yields the best performance (i.e., high throughput for batch jobs and short completion time for online jobs). The job manager also has a new sorting method to prioritize the recurring jobs in scheduling to prevent high resource contention while fully utilizing cluster resources.

Based on the determined schedule, when jobs and their datasets are submitted, the real-time scheduler places input datasets and schedules the recurring jobs to racks accordingly. It schedules non-recurring (i.e., ad-hoc) jobs to the resources not assigned to the recurring jobs. As the recurring jobs can finish earlier by more efficiently utilizing OCS, it leaves more computing resources and network bandwidth to ad-hoc jobs to complete earlier [22].

We have evaluated JobPacker using large-scale simulation and small-scale emulation on GENI based on a Facebook trace [6]. The results show that JobPacker reduces the makespan of a batch of jobs (i.e., the time to finish all the jobs) up to 49% and the median job completion time up to 43%, compared to the state-of-the-art schedulers in Hybrid-DCN.

The rest of the paper is organized as follows. Section 2 and 3 introduce the background and related work. Section 4 introduces the main design of JobPacker. Section 5 presents the performance evaluation. Section 6 concludes this paper with our future work.

2 BACKGROUND

To present use case, we use MapReduce [10] as an example for the data-parallel frameworks in this paper. However, JobPacker can be applied to other frameworks.

2.1 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 [22]. Each map task processes one input data block and generates intermediate data (called shuffle data). The reduce stage is a combination of shuffle stage and reduce stage, and each reduce task consists of two steps: shuffle and reduce. In the shuffle, all shuffle data with the same key is transferred to the same reduce task. In YARN [32], when a certain percent (called slowstart threshold) of map tasks for a job have completed, the reduce tasks of the job can be scheduled. Only after a reduce task is scheduled, the shuffle of this reduce task start immediately, which overlaps the map and shuffle stages (i.e., intra-job concurrency) to reduce the job execution time.

2.2 Hybrid-DCN

In this paper, we assume there are R racks in a cluster and one rack can send data via OCS to only another rack at a time, as in Helios [14] and c-Through [33]. As in [14, 33], we assume that only the flows with size larger than the elephant flow threshold (e.g., 1.125GB) are 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 express traffic between each rack by a matrix M of size $R * R$. We denote the traffic sent via OCS as an $R * R$ matrix M_o; the remaining traffic is sent via EPS. During one OCS reconfiguration process, it estimates the traffic, build the matrices, and accordingly compute and establishes the port connections for data transfers. As in [14, 33], we assume in this paper that OCS is reconfigured periodically with a fixed reconfiguration interval.

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 20 & 20 & 20 & 20 \\ 2 & 2.5 & 2.5 & 2.5 & 2.5 \\ 3 & 1.25 & 1.25 & 1.25 & 1.25 \\ 4 & 1.25 & 1.25 & 1.25 & 1.25 \end{array}$$

(a)

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 6.25 & 6.25 & 6.25 & 6.25 \\ 2 & 6.25 & 6.25 & 6.25 & 6.25 \\ 3 & 6.25 & 6.25 & 6.25 & 6.25 \\ 4 & 6.25 & 6.25 & 6.25 & 6.25 \end{array}$$

(b)

Figure 2: Balance-skewness of demand matrix M_o. Suppose that a shuffle-heavy job with 20 map tasks, 4 reduce tasks and 100 units of shuffle data runs in a 4-rack cluster. Suppose flows with no smaller than 1 unit are elephant flows and the speed of OCS is 1 per unit time. Assume each map task generates the same size of shuffle data and each reduce task processes the same size of data [22]. (a) This is the demand matrix when racks 1,2,3,4 process 16,2,1,1 map tasks and 1,1,1,1 reduce task, respectively. The time to complete the data transfer of this matrix is 20+20+20=60. (b) This is the demand matrix when racks 1,2,3,4 process 5,5,5,5 map tasks and 1,1,1,1 reduce task, respectively. The time to complete the data transfer of this matrix is 6.25+6.25+6.25=18.75.

To fully take advantage of OCS, the desired properties of matrices M and M_o are listed as follows.

Skewness [25]: To take full advantage of OCS, we expect that the demand from any rack is high to only a few other racks and low to the remaining racks, forming a skew demand matrix M. Thus, the high-demand entries in M can be well served by the OCS, while the low-demand entries can be served by the EPS. **Sparsity [25]:** The OCS demand matrix M_o should be sparse (with only a few non-zero entries), since a rack can send data via OCS to only one other rack at a time. **Balance-skewness:** The shuffle traffic of a shuffle-heavy job is balanced between racks to reduce the durations of shuffle data transfer, as illustrated in Figure 2.

2.3 Opportunity

Several previous studies [3, 15, 18, 22] show that cluster workloads contain a large number of recurring jobs, whose *job characteristics*, including 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% [22]). The predictability characteristics allow us to determine which racks to place the job input datasets and run the tasks for the recurring jobs *before* the input datasets and the jobs are submitted to the cluster. Many practical scenarios allow us to place data beforehand [22]. For example, in the cloud environment such as AWS, data is often stored in a dedicated storage cluster (e.g., Amazon S3). To run MapReduce on the cloud, the data is fetched from the storage cluster. At this step, the data can be placed on the pre-determined racks.

3 RELATED WORK

Previous studies [5, 14, 33] have deployed OCS with ~10ms reconfiguration delay in datacenter networks to improve performance. In this paper, we aim to design a job scheduler to use OCS efficiently in Hybrid-DCN to improve performance.

There has been much effort [1, 4, 8, 9, 17, 22, 28, 34] focusing on designing schedulers for data-parallel clusters to improve performance (e.g., throughput). However, none of these schedulers tackle the scheduling problem in Hybrid-DCN to leverage OCS effectively. For example, in Fair [1] and Delay [34] schedulers, the input data is randomly spread among racks, so that the map tasks is also spread among racks to achieve high map data locality. ShuffleWatcher [4] aims to distribute the shuffle network traffic spatially (among different racks) and temporally (during different time periods). These two schedulers above generate many mice flows, which cannot use OCS effectively. Corral [22] places all the map and reduce tasks of a job in the same set of racks to avoid cross-rack shuffle data transfer. However, it imposes an intensive container contention in the set of racks, which may sacrifice the parallelism and decrease performance. In additional, Corral cannot take full advantage of OCS in Hybrid-DCN since it attempts to avoid cross-rack traffic, rather than efficiently utilizing OCS to accelerate cross-rack traffic transfer. Unlike these previous studies, JobPacker is a new job scheduler for Hybrid-DCN that can take full advantage of the high-bandwidth OCS to achieve better job performance.

Our prior work [7, 23] also focuses on designing job schedulers to aggregate data transfers to efficiently use OCS in Hybrid-DCN. Unlike prior work that places tasks considering both Coflow completion time and data transfer aggregation, this paper attempts to leverage the predictable nature in production workloads to explore the tradeoff between parallelism and data transfer aggregation.

There have been many other works [11, 13, 19, 24] focusing on using various cluster configurations and file systems to improve the job performance. Our paper is orthogonal to these works, which can be combined to further improve the job performance.

4 DESIGN OF JOBPACKER

4.1 System Architecture

JobPacker has a shuffle data aggregation scheme that facilitates to use OCS. In addition, as shown in Figure 3, JobPacker consists of an offline scheduler and a real-time scheduler. The offline scheduler is responsible for deciding the schedule for the recurring jobs in the next unit period and has two main components – job profiler and job manager. The job profiler explores the tradeoff between parallelism and traffic aggregation, and returns all feasible *map-width* and *reduce-width* pairs of each shuffle-heavy recurring job (i.e., number of racks to run the map and reduce tasks) that can leverage OCS effectively while achieving sufficient parallelism. Then, the job manager finds out the best (map-width, reduce-width) for the shortest completion time of each shuffle-heavy recurring job, and also generates the global schedule including which racks to run the map/reduce tasks of each recurring job, and the sequence to run the map/reduce tasks of recurring jobs in each rack that yields the best performance (i.e., high throughput for batch jobs and short completion time for online jobs). For example, if the map tasks of job i, the reduce tasks of job j, and the map tasks of job k are assigned to a rack, the sequence on this rack is $Seq = \{map_i, reduce_j, map_k\}$, where map_j and $reduce_j$ means any map task and any reduce task of job j, respectively.

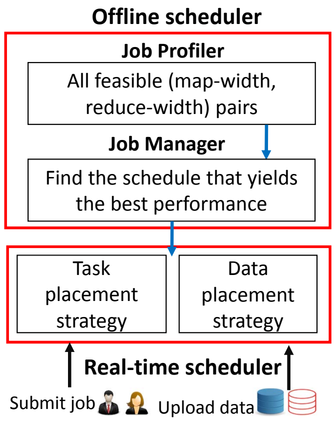


Figure 3: System architecture of JobPacker.

Based on offline schedule generated from job manager, the real-time scheduler guides the data placement and task placement of the job. The ad-hoc jobs are then scheduled based on previous scheduling scheme (e.g., Fair [1]) and use the idle resources that are not assigned to recurring jobs.

4.2 Shuffle Data Aggregation

Currently, the reduce task is associated with its shuffle and the shuffle starts fetching data once the corresponding reduce task is scheduled [32]. However, this default scheme does not facilitate shuffle traffic aggregation. Hence, we propose not to start the shuffle immediately after its corresponding reduce task is scheduled to a container. In order to aggregate the shuffle data transfers of a job, we force the shuffle to start until more reduce tasks from the same job are assigned to containers and the size of aggregated shuffle data of the reduce tasks reaches the elephant flow threshold. Then, we can use high-bandwidth OCS for shuffle data transfer to reduce the transfer delay of low-bandwidth EPS. If the size of aggregated shuffle data cannot reach the elephant flow threshold for a job (i.e., non-shuffle-heavy jobs), the shuffle data is transferred through EPS, which will not take a long time due to the small data size.

Figure 4 illustrates the shuffle data transfers for shuffle-heavy jobs with default schedulers and with JobPacker. The shuffle data aggregation in JobPacker brings two advantages. First, it enables the use of high-bandwidth OCS to accelerate the shuffle data transfers to shorten the shuffle duration for a job. On the contrary, without shuffle data aggregation, a shuffle-heavy job may have to use the low-bandwidth EPS in Hybrid-DCN since the sizes of its traffic flows are small. The shuffle data aggregation does not degrade the performance of non-shuffle-heavy jobs much, as their shuffle data is relatively small so their shuffle duration is relatively short.

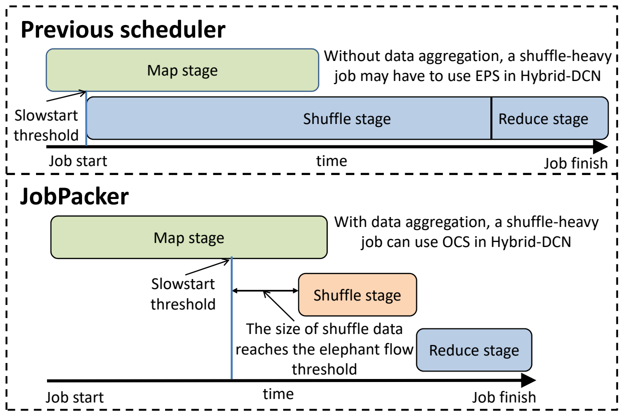

Previous scheduler
Map stage
Without data aggregation, a shuffle-heavy job may have to use EPS in Hybrid-DCN
Slowstart threshold
Shuffle stage
Reduce stage
Job start
time
Job finish
JobPacker
Map stage
With data aggregation, a shuffle-heavy job can use OCS in Hybrid-DCN
Slowstart threshold
Shuffle stage
The size of shuffle data reaches the elephant flow threshold
Reduce stage
Job start
time
Job finish

Figure 4: Illustration of shuffle data aggregation.

Second, in YARN, the slowstart threshold is set to a small value (default 5% [4]) to achieve high intra-job concurrency for high performance. However, in this case, the reduce tasks occupy the containers when they are doing nothing but transferring shuffle data, which wastes precious resources. With the data aggregation scheme in Hybrid-DCN, as shown in Figure 4, JobPacker can increase the slowstart threshold to prevent the reduce tasks from occupying resources for too long without compromising job performance, since fast data transfers through high-bandwidth OCS offsets the influence of intra-job concurrency reduction for shuffle-heavy jobs. For non-shuffle-heavy jobs, since transferring small-size data can be completed in short time duration, they do not need a low slowstart threshold for high intra-job concurrency. The slowstart threshold should be determined depending on the workloads in the system (i.e., whether reduce tasks can occupy containers for a long time without compromising other tasks' performance). If the system is lightly loaded, the slowstart threshold can be smaller for higher intra-job concurrency for high performance.

4.3 Offline Scheduler

We use r_j^{map} and r_j^{red} to denote the number of racks that are assigned to run job j's map and reduce tasks, respectively. We evenly distribute the map and reduce tasks among the r_j^{map} and r_j^{red} racks to achieve the balance-skewness property.

4.3.1 Job Profiler. We use a *latency response function* (LRF) [22] to model the latency for every job j. LRF takes the number of racks allocated to job j as input and predicts the latency of job j. LRF assumes that the map, shuffle and reduce stages run sequentially for simplicity though the shuffle stage overlaps with the map stage. This assumption matches JobPacker since it reduces the overlap (as shown in Figure 4). LRF also assumes that the map and reduce tasks of job j are scheduled on the same number of racks (i.e., $r_j^{map} = r_j^{red}$), which is not always correct in practice. In this paper, we remove this assumption to improve LRF. The latency of a job is calculated by:

$$L_j(r_j^{map}, r_j^{red}) = l_j^{map}(r_j^{map}) + l_j^{shu}(r_j^{map}, r_j^{red}) + l_j^{red}(r_j^{red}), \quad (1)$$

where $l_j^{map}(r_j^{map})$, $l_j^{shu}(r_j^{map}, r_j^{red})$ and $l_j^{red}(r_j^{red})$ denote the latency for each of the three stages. Please refer to [22] for the details of how to compute $l_j^{map}(r_j^{map})$ and $l_j^{red}(r_j^{red})$ from the estimated job characteristics (input/shuffle/output data sizes and the number of tasks). $l_j^{shu}(r_j^{map}, r_j^{red}) = \frac{D_j(r_j^{map}, r_j^{red})}{BW}$, where $D_j(r_j^{map}, r_j^{red}) = \frac{D_j^s}{r_j^{map} \cdot r_j^{red}} \cdot (r_j^{red} - 1)$ is cross-rack shuffle data size, BW is the bandwidth, and D_j^s is the shuffle data size of job j. To determine bandwidth BW, we need to determine whether OCS or EPS is used in the shuffle stage of job j. Since shuffle data is sent from all map tasks to all reduce tasks, we check if the shuffle data size of job j divided by $r_j^{map} * r_j^{red}$ (i.e., $\frac{D_j^s}{r_j^{map} \cdot r_j^{red}}$) is greater than the elephant flow threshold. If yes, OCS is used; otherwise, EPS is used.

The job scheduler needs to carefully determine r_j^{map} and r_j^{red} for each shuffle-heavy job to achieve an optimal balance between parallelism and traffic aggregation, which yields relatively low job latency. For each value assignment of r_j^{map} and r_j^{red}, we can compute the latency of job j based on Equ. (1).

Figure 5 shows the latencies of an example shuffle-heavy job under different assignment combinations on a 15-rack cluster, where each rack has 600 containers. We see that as r_j^{map} increases from 1 to 5, the latency of the job drops significantly due to higher parallelism. The number of map tasks for this job is 3472, which is greater than the total number of containers in 5 racks ($5 * 600 = 3000$). Since the job has 169 reduce tasks, running the reduce tasks on one rack (i.e., $r_j^{red} = 1$) is sufficient for all the reduce tasks to run concurrently, i.e., achieving parallelism. Additionally, the latencies in the *green* zone are considerably lower than the latencies in the other zones due to two reasons. First, the assignment combinations in this zone do not sacrifice the parallelism. Second, OCS is used for shuffle data transfer in the green zone. As a result, the green zone illustrates all feasible (map-width, reduce-width) pairs for job j that can leverage the OCS to achieve a good tradeoff between parallelism and traffic aggregation.

4.3.2 Job Manager. As [22], we consider two scenarios of job submission: *batch* and *online* scenarios. In the *batch* scenario, all jobs are submitted at the same time, and the goal is to makespan, i.e., the time to finish all the jobs in the batch. In the *online* scenario, jobs are submitted at different times and the goal is to minimize the average job completion time, i.e., the average time from the arrival of a job until its completion.

For both batch and online scenarios, we can model the job scheduling as an optimization problem to achieve different goals. Nevertheless, the optimization problems for both scenarios are analogous

r_j^{map} \ r_j^{red}	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

Figure 5: Latencies of an example job under different assignments. The job consists of 3472 map tasks and 169 reduce tasks. The row and column represent r_j^{map} and r_j^{red}. Each entry is the latency when the map and reduce tasks are evenly distribute to r_j^{map} and r_j^{red} racks.

to complex directed-acyclic-graph (DAG) structured job scheduling problem [22], which is well-known as NP-hard [26, 31]. Hence, we propose a heuristic method to solve the scheduling problem.

Batch scenario. We define a shuffle-heavy job's width as the maximum value of $r_j^{map} + r_j^{red}$ among all of its feasible (map-width, reduce-width) pairs. The width of a non-shuffle-heavy job is defined as the total number of map and reduce tasks divided by the number of containers on a rack. First, we need to determine the priority of each job in scheduling. We could use the algorithm in Corral [22] that sorts the batch of jobs in descending order of job width. This widest-job-first algorithm avoids the case that the widest job cannot find enough racks to run all of its tasks concurrently and needs to wait for the job that is allocated to only a few racks to complete, which wastes the resources [22]. However, using this sorting algorithm, the *extremely* shuffle-heavy jobs are more likely to have very high priorities as these jobs most probably have extremely huge input data size (see explanation in Section 4.4) and hence have more tasks, which requires more racks. Then, it may lead to an extremely high network load and computing load (i.e., demand for a larger number of containers) at the beginning and a light network and computing load later. This resource utilization pattern is not desired [4], because all these extremely shuffle-heavy jobs compete for the precious resource simultaneously at the beginning, which degrades the performance significantly.

In order to solve this problem, we propose to divide the workload (for shuffle data transfer and map/reduce tasks) to B sub-batches, so that each sub-batch's workload will not impose resource competition while fully utilizing cluster resource, as shown in Figure 6. The number of sub-batches B is a tunable parameter based on the entire cluster capacity and the resource demands of jobs.

To divide into B sub-batches, we use Tetris [17], which chooses a job to assign to a server with available capacity in order to increase the resource utilization of each server considering multi-resources (e.g., CPU, memory, bandwidth). Basically, based on the available

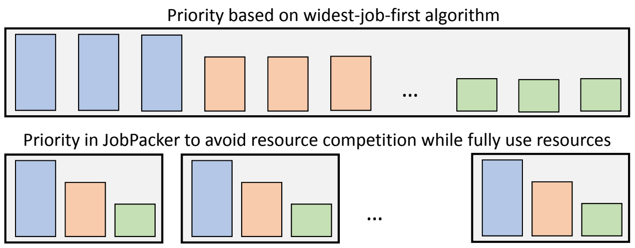

Priority based on widest-job-first algorithm
...
Priority in JobPacker to avoid resource competition while fully use resources
...

Figure 6: Priority determination based on sub-batches.

resources on a server, Tetris gives a score[1] to each job and then greedily picks the job with the highest score to run on the server. We treat each sub-batch as a server and treat the shuffle data size, the number of map tasks and the number of reduce tasks of each job as its demand on multi-resources. The capacity of each sub-batch is the capacity of the cluster on different dimensions (total cross-rack bandwidth, the number of containers). Then the batch division problem is interpreted as the job-to-server packing problem. The output includes B sub-batches, and the resource demands on each resource from all sub-batch are similar. The resource demands of a sub-batch on different resources equal the sum of shuffle data sizes, the sum of the number of map tasks and the sum of the number of reduce tasks of all the jobs in the sub-batch.

In each sub-batch, we sort the jobs in the descending order of width. The jobs with the same width are further sorted in the decreasing order of job latency, because the longest-latency-job-first first algorithm is effective for makespan minimization [16, 22]. After sorting, we combine all the sub-batches in a random order. Finally all the jobs form a list for sequential offline scheduling as shown in Figure 6.

During the offline scheduling, we keep track of the time T_{ik} when the container k on rack i completes the current task and requests the next task. We compute the time needed by the map, shuffle and reduce stage using the method in Equ. (1).

For each job j from the sorted list, we check whether it is shuffle-heavy or not and conduct the scheduling as follows. We assign the tasks of a shuffle-heavy job to the best (map-width, reduce-width) pair among all feasible pairs that yields the best performance, while assigning the tasks of a non-shuffle-heavy job to any containers that are available. We use N_j^{map} and N_j^{red} to denote the number of map tasks and reduce tasks of job j.

Non-shuffle-heavy jobs. We pick the first N_j^{map} available containers based on the next available time T_{ik} of each container. We assign these containers to the map tasks and update the T_{ik}. Next, we pick the first N_j^{red} available containers based on T_{ik} to run the reduce tasks of job j. The reduce stage start time S_j^{red} is computed by adding the completion time of the last map task (cmp_j^{map}) and the shuffle stage latency $l_j^{shu}(r_j^{map}, r_j^{red})$. Finally, the job manager updates the sequences of the racks that run job j's map and reduce tasks correspondingly, and records the set of racks that run job j's map tasks (R_j^{map}), which will be used to guide the input data placement in real-time scheduling.

[1]For example, a server has (0.2,0.3,0.5) available resources. If a job consumes (0.1,0.2,0.3), the score of this job is the dot product of the two resource vectors, i.e., 0.1*0.2+0.2*0.3+0.3*0.5=0.23.

Shuffle-heavy jobs. We enumerate each (map-width, reduce-width) pair among all feasible pairs, and find the pair that yields the earliest completion time. For each rack i, we find out the time when $\lceil N_j^{map}/r_j^{map}\rceil$ containers are available. Then we find r_j^{map} the earliest such available racks to run its map tasks, and update the corresponding T_{ik} of each assigned container. We use the same way above to compute the completion time of the last map task cmp_j^{map}. Similarly, we find r_j^{red} the earliest available racks that have $\lceil N_j^{red}/r_j^{red}\rceil$ available containers. Then we can compute the job completion time in this iteration. After we finish all interations, we find out the (map-width, reduce-width) that yields the earliest job completion for job j, and which racks to run job j's map and reduce tasks. Finally, the job manager updates the sequences of those racks correspondingly, and records the set of racks that run job j's map tasks (R_j^{map}).

As a result, in the offline schedule, a rack has large data transfer to only a few racks and has small data transfer to the other racks, which satisfies the skewness and sparsity desired properties. Also, since the tasks of each shuffle-heavy job are evenly distributed among the set of racks, the balance-skewness is achieved.

Online scenario. The objective in the online scenario is to minimize the average job completion time. We sort the jobs in an increasing order of their predicted job arrival times. When the jobs are submitted at the same time, we use the sorting algorithm for sorting jobs in each sub-batch. Other steps are the same as those in the batch scenario.

Over-provisioning. Recall that we estimate the resource demand (i.e., the number of containers needed) of each recurring job with a small error. However, in the offline scheduler, we *intentionally* assign more containers to each job than the estimation by a certain ratio (i.e., *over-provisioning ratio*) due to two reasons. First, we take the estimation variation into account. Thus, the recurring jobs can have sufficient containers during the actual job execution. Second, we attempt to leave ad-hoc jobs sufficient containers to run in the cluster. This strategy will not waste resource because during actual job execution, when the recurring jobs do not need as many containers as planned in the offline schedule, the unused containers can be used by the ad-hoc jobs or other recurring jobs assigned to the same racks. The cluster operators can adaptively determine the over-provisioning ratio based on the estimation variance and the percent of ad-hoc jobs in their clusters to achieve better performance. If a cluster has higher estimation variance or fewer recurring jobs, we can set a higher over-provisioning ratio; otherwise, it can be zero.

Summary. The job manager in JobPacker returns a sequence for each rack, which includes the recurring jobs' map or reduce tasks to run on this rack. Besides the sequence for each rack, for each job j, the job manager outputs R_j^{map} to guide the placement of its input data. The outputs of job manager are then passed to the real-time scheduler, which will be introduced in Section 4.4.

4.4 Real-time Scheduler

The real-time scheduler executes the generated offline schedule. When the input dataset of a recurring job j is uploaded to the cluster, JobPacker places one replica of each data chunk of job j in a randomly chosen rack from R_j^{map} to achieve data locality. The second and the third replicas of all data chunks are randomly placed on the other racks. This data placement strategy still obeys the default data placement in HDFS that places the replicas of each data chunk in two random racks.

We try to determine if we can judge whether an ad-hoc job is a shuffle-heavy job based on its input data size in order to avoid scheduling shuffle-heavy jobs together in a rack to achieve the sparsity property. Using the two Facebook workloads in 2009 (FB2009-1, FB2009-2) and two Facebook workloads in 2010 (FB2010-1, FB2010-2) from [6], we plot Figure 7 that shows the cumulative distribution function (CDF) of input data size of all the shuffle-heavy jobs (shuffle data size greater than 1.125GB). We observe that most shuffle-heavy jobs have an input data size larger than 1GB. Based on the above observation, when the input data of a job is submitted to the cluster, if the input data size is greater than a threshold (e.g., 1GB), we empirically treat the job as a shuffle-heavy job. Note that non-shuffle-heavy jobs may be sometimes over-estimated as shuffle-heavy jobs. However, over-estimation is better than under-estimation, as under-estimation may inappropriately place some shuffle-heavy jobs, which generates unwanted demand matrices.

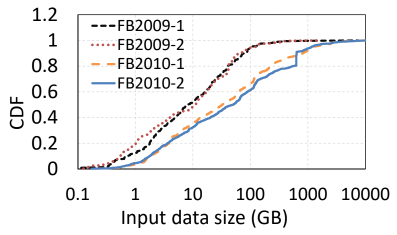


Figure 7: Workload analysis of shuffle-heavy jobs.

To determine the priorities of ad-hoc jobs, we use the default user-specified scheduler, such as Fair [1]. Recall that in the offline schedule, each rack is assigned with a sequence of recurring jobs. During scheduling, when a rack has a container available, the real-time scheduler tries to follow the offline schedule (which is only for recurring jobs). If there are recurring jobs, the real-time scheduler selects a map/reduce task of the first job in the sequence. Only when there is no recurring job assigned to this rack, an ad-hoc job will be scheduled to the container. In this step, the scheduler tries to schedule shuffle-heavy job first while avoiding scheduling shuffle-heavy jobs together in a rack in order to achieve the sparsity property. Specifically, the scheduler checks whether there are any tasks of shuffle-heavy jobs *currently running* in the rack. If yes, the scheduler selects a task from the ad-hoc non-shuffle-heavy job with the highest priority. Otherwise, the scheduler gives high priority to the task from the ad-hoc shuffle-heavy job with the highest priority if there are any in the queue. In the case of failure of a rack, JobPacker ignores the guidance from the offline scheduler and schedules the jobs assigned to this rack based on the default scheduler.

Overhead. In real-time scheduler, for each scheduling decision, JobPacker performs simple examinations (e.g., sequence and priority of jobs), which is quite similar to the Fair scheduler [1]. Hence, the computation overhead in real-time scheduler is no more than Fair, indicating the excellent scalability of JobPacker.

5 PERFORMANCE EVALUATION

5.1 Traces and Settings

5.1.1 Workload traces. We evaluated JobPacker assuming that all the jobs are recurring first, and then using the workload with a mix of both ad-hoc and recurring jobs. We also conducted the sensitivity analysis of different settings. The workload trace we used was from the SWIM Facebook workloads [6]. Since this workload trace misses important information such as task running time, we first replayed all the jobs in the trace (using the tools provided in the same project [6]) 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 trace for simulation and emulation.

5.1.2 Simulation. In order to evaluate the performance of JobPacker in a large scale, we built a flow-based event simulator to replay the workload trace. In the simulation, there are 600 servers, organized into 20 racks with 30 servers each. Each server can run up to 20 tasks and has 10Gbps network interface card (NIC). The Hybrid-DCN topology is the same as in Figure 1. The link rate between the ToR switch and core EPS is 30Gbps, which yields a 10:1 oversubscription ratio. The ToR switch and OCS are always connected with 100Gbps link. We ran 1000 jobs selected from the workload. The job characteristics of the 1000 jobs are listed in Table 1. The elephant flow threshold was set to 1.125GB, which is inferred empirically from previous studies [14, 21, 25] to achieve high OCS utilization. As in [14, 33], we used Edmonds' algorithm [12] to compute the optimal input-to-output configuration for OCS in every reconfiguration.

5.1.3 Emulation on GENI. We also conducted an emulation on GENI [2]. We built a testbed with 10 servers on GENI, each emulating a virtual rack (VR). We assumed that each VR can run up to 10 tasks. Due to the bandwidth availability, the link rate between VRs via OCS is 1Gbps, while the link rate between VR via EPS is 0.1Gbps. We limited the bandwidth for each task to 0.1Gbps. We ran 200 jobs chosen from the workload trace. We shrank the input and output data sizes of each task, and the elephant flow threshold by a factor of 100, which equals the network bandwidth shrinking factor in GENI. The emulation allows us to evaluate the performance of JobPacker under real network environment.

Table 1: Job characteristics.

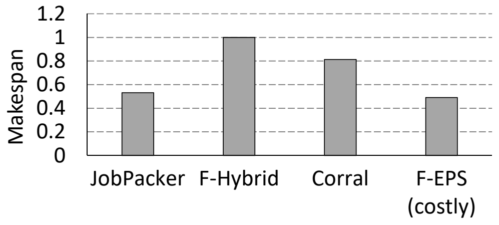
Percentile	5%-tile	Median	95%-tile
Shuffle data size (GB)	0.1	2.5	82

5.1.4 Baselines. We compared JobPacker with Hybrid-DCN with three other systems.

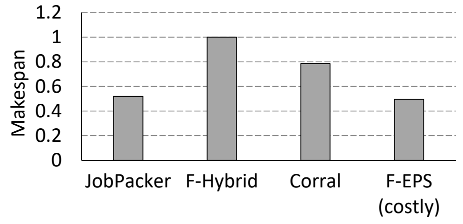
(1) Fair scheduler [1] with Hybrid-DCN (F-Hybrid). It uses the Fair scheduler to schedule the jobs in Hybrid-DCN. Fair is the most widely used scheduler in current production clusters [1], and it assigns resources to jobs so that each job roughly receives an equal share of resources (containers) over time.

(2) Corral [22] with Hybrid-DCN (Corral). Corral places the map and reduce tasks of the same job on the same set of racks to reduce the cross-rack shuffle data transfer.

(3) Fair scheduler with traditional packet-switched datacenter network (F-EPS). It uses the Fair scheduler to schedule the jobs in a packet-switched network. In this system, the ToR switches are connected through an EPS core switch and their link rate is 100Gbps (1Gbps on GENI). This link rate is as high as the link rate of OCS though the system does not have OCS. However, the high-bandwidth EPS in this architecture leads to up to a factor of 9 higher CapEx and OpEx [14, 33], compared to the Hybrid-DCN.



(a) Simulation



(b) GENI

Figure 8: Makespan results (batch).

5.2 Experimental Results

In this section, we considered that all the jobs (1000 jobs in simulation and 200 jobs in GENI) are recurring and can be predicted with zero error. All the experiments were run for 20 times and the average results are reported. The OCS was reconfigured every 1 second [33] and the reconfiguration delay was 10ms. The slowstart threshold was 0.7. The over-provisioning ratio was set to 0, as all the jobs are recurring.

5.2.1 Batch Scenario. In the batch scenario, the 1000 jobs were divided into 5 sub-batches. Figures 8(a) and 8(b) show the makespan of different methods in the simulation and GENI, respectively. All the results are normalized by the results of F-Hybrid. We see that JobPacker outperforms F-Hybrid by 47% and 49% in the simulation and GENI, respectively. Compared with Corral, JobPacker has 29% and 27% improvement of makespan in the simulation and GENI, respectively. We also see that in the simulation and GENI, JobPacker achieves a comparable performance as F-EPS (less than 5% difference). However, as mentioned above, F-EPS generates significantly higher CapEx and OpEx.

We then measured the percentage of total traffic sent via OCS and EPS, respectively. Figures 9(a) and 9(b) show the percentage of traffic sent by OCS and EPS in the simulation and GENI, respectively. Since F-EPS does not have OCS, we do not show its results. We see that the OCS has a much higher utilization in JobPacker in the batch scenario (>96%), compared with F-Hybrid (~0.8%) and Corral (<23%). The result demonstrates the outstanding performance of JobPacker on taking advantage of OCS.

5.2.2 Online Scenario. In this section, the jobs arrived uniformly at random in $[0, 90]min$ and in $[0, 20]min$ in the simulation and GENI experiment, respectively. Figures 10(a) and 10(b) show the

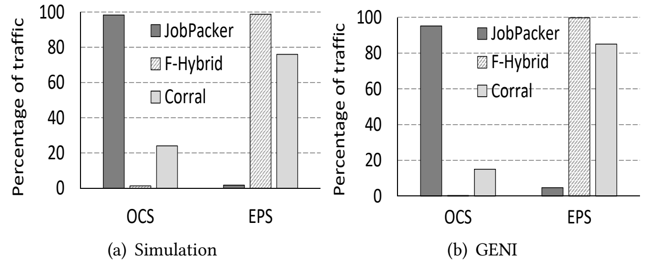


Figure 9: Percentage of traffic via OCS and EPS (batch).

CDF of job completion times in the simulation and GENI. We see that the JobPacker significantly shortens the job completion times, compared with F-Hybrid and Corral. Specifically, JobPacker outperforms F-Hybrid with 43% and 42% improvement at the median job completion time in the simulation and GENI, respectively. Compared with Corral, JobPacker reduces the median job completion time by 28% and 27% in the simulation and GENI, respectively. We also see that the CDF of JobPacker is very similar to the CDF of F-EPS which however is very costly, indicating that JobPacker can be a cost-efficient solution for the network bottleneck problem in data-parallel frameworks.

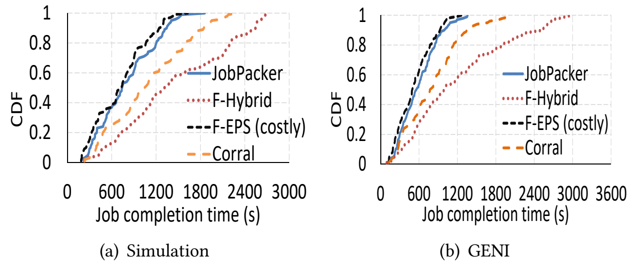


Figure 10: CDF of completion times (online).

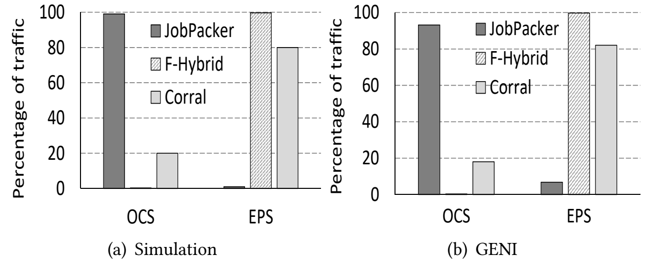


Figure 11: Percentage of traffic via OCS and EPS (online).

We also measured the traffic sent via OCS and via EPS in the online scenario, as shown in Figure 11. Similarly, compared with F-Hybrid and Corral, JobPacker has a much higher OCS utilization in the online scenario.

5.3 Mix of Ad-hoc and Recurring Jobs

In this section, as in [22], we evaluated JobPacker in an online scenario, where there are a mix of ad-hoc and recurring jobs. Previous studies [3, 15, 22] indicate that there are 40%-60% recurring jobs in the cluster. Therefore, we randomly selected half of the jobs as ad-hoc jobs and the rest are still recurring jobs. All the jobs

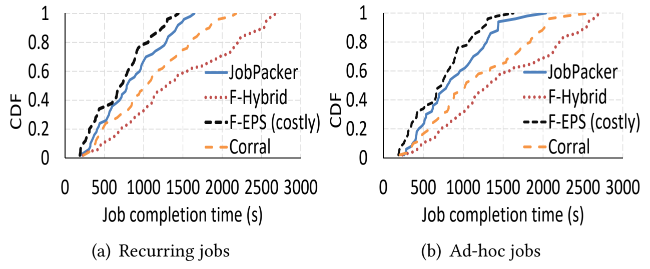


Figure 12: CDF of completion times with a mix of jobs (simulation).

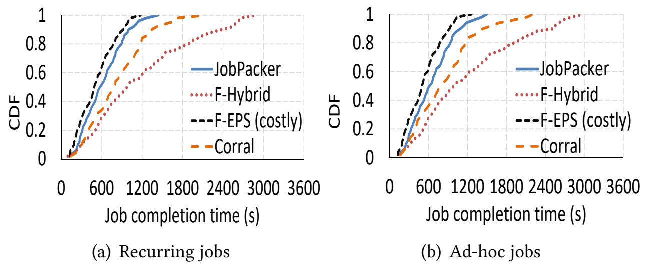


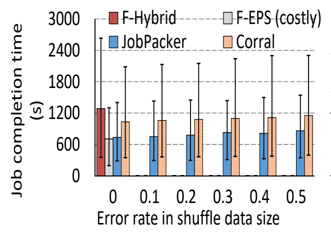
Figure 13: CDF of completion times with a mix of jobs (GENI).

arrived uniformly at random in $[0, 90]min$ and $[0, 15]min$ in simulation and GENI, respectively. In the offline scheduler, we set the over-provisioning ratio to 1.0.

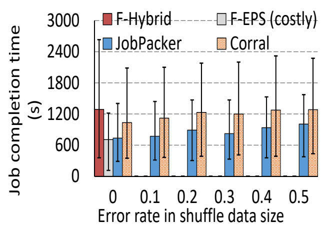
Figures 12(a) and 12(b) show the CDF of job completion times for recurring jobs and ad-hoc jobs in the simulation, respectively. Figures 13(a) and 13(b) show the CDF of job completion times for recurring jobs and ad-hoc jobs in GENI, respectively. Clearly, JobPacker generates shorter job completion times for both recurring and ad-hoc jobs, compared with F-Hybrid and Corral. In the simulation, we see that JobPacker reduces the median job completion time of recurring jobs and ad-hoc jobs by 40% and 38%, respectively, compared with F-Hybrid. In GENI, JobPacker reduces the median job completion time of recurring jobs and ad-hoc jobs by 39% and 37%, respectively, We also see that JobPacker outperforms Corral by 24% and 25% for recurring jobs and ad-hoc jobs in the simulation, and by 26% and 29% for recurring jobs and ad-hoc jobs in GENI. JobPacker achieves a comparable performance as F-EPS (which is very costly) in both simulation and GENI, demonstrating the superior performance of JobPacker. This is because in JobPacker, the recurring jobs can more effectively utilize the OCS to transfer their shuffle data, which significantly frees the network resource on EPS. Although we do not place the tasks of ad-hoc jobs on a few racks to aggregate the data transfers to use the OCS, they still benefit from the much lower utilization of network resource on EPS. Besides, as the recurring jobs finish faster, more computing resources are available for the ad-hoc jobs, which significantly increases the performance of ad-hoc jobs as well.

5.4 Sensitivity Analysis

In this section, we used our simulation environment to evaluate the robustness of JobPacker to several factors. Due to the space limit, we only show the results in online scenario, unless otherwise specified. We analyze the sensitivity when there are all recurring



(a) Error in shuffle data size



(b) Error in arrival time

Figure 14: Performance variation with error in prediction of job characteristics.

jobs, as in [22], unless otherwise specified. The experiment settings are the same as Section 5.2 unless otherwise specified.

5.4.1 Error in prediction of shuffle data size. We use the predicted shuffle data size to check if it is a shuffle-heavy job, determine all the feasible map-width and reduce-width, and compute the latency of shuffle stage. We define the prediction error rate of a job's shuffle data size as $\frac{|real-prediction|}{real}$. Though recent studies [3, 15, 22] show that the characteristics of recurring jobs can be predicted with a low error rate around 6.5%, we varied the error rate for the prediction of the shuffle data size of all the jobs by up to 50% to see how JobPacker performs. Figure 14(a) shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker versus different error rates. We do not measure the performance of F-Hybrid and F-EPS here since they do not predict the shuffle data size. As we see, the job completion times increase as the error rate increases. However, even with some prediction error, JobPacker still outperforms F-Hybrid by 33% and Corral by 25% at the median, and achieves similar performance as F-EPS, as JobPacker effectively utilizes the OCS.

5.4.2 Error in job arrival time. In the offline scheduler, we sort the jobs in the online scenario based on the predicted job arrival times. In practice, the job arrival time may vary from the predicted arrival time. In this experiment, we added a random time delay in the range of $[-200, 200]s$ to f portion of jobs, where f is varied in the range of [10%, 50%]. Figure 14(b) shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker with varying portion of delayed jobs. We see that although up to 50% of the jobs' arrival times are not accurate, JobPacker shortens the median job completion time by 23% and 22% compared with F-Hybrid and Corral, respectively, and achieves comparable performance to F-EPS.

5.4.3 Slowstart threshold. We varied the slowstart threshold in the range of [20%, 80%]. Figure 15 shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker with different slowstart thresholds. We see that when the slowstart threshold is in the range of [50%, 80%], the job completion times are almost the same, indicating that the slowstart threshold in JobPacker can be set to a sufficient large range to achieve the best performance. On the other hand, when the slowstart threshold is set to ≤ 40%, the job completion times slightly increase. In these cases, the reduce tasks of the jobs hog up the resources that can be allocated to the tasks of other jobs for too long because of low slowstart thresholds.

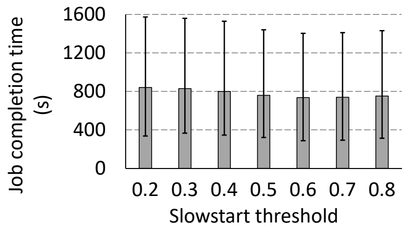


Figure 15: Varying slowstart threshold.

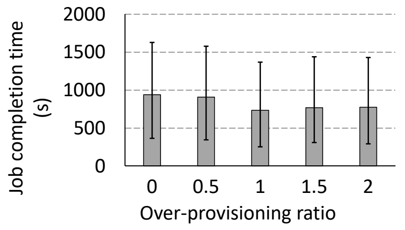


Figure 16: Varying over-provisioning ratios.

5.4.4 Over-provisioning ratio. In the previous experiments when there are both recurring and ad-hoc jobs, we set the over-provisioning ratio to 1.0. In this experiment, we varied the over-provisioning ratio in the range of [0, 2] and used the same other settings as in Section 5.3. Figure 16 shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker with different over-provisioning ratios. The figure indicates that if the over-provisioning ratio is too low, the job completion times are significantly affected, since the planned resources are not sufficient to complete the recurring and ad-hoc jobs. The performance is not affected much when the over-provisioning ratio becomes larger. This is because in real cluster running, the over-provisioned resources can be used to run the ad-hoc jobs and other recurring jobs planned on the same resources.

5.4.5 OCS reconfiguration interval. We varied the OCS reconfiguration interval in the range of $[0.1s, 3s]$. Figure 17 shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker with different OCS reconfiguration intervals. We see that varying the reconfiguration interval only has minimal impacts on the performance of JobPacker. Even setting the reconfiguration interval to $3s$ can achieve a good performance (38% reduction on median job completion time compared with F-Hybrid). This is because JobPacker aggregates the shuffle traffic to only a few racks, which generates sparse and skew demand matrices for OCS. This allows OCS to be reconfigured less frequently.

5.4.6 The number of sub-batches. In the batch scenario, we divide the entire batch into several sub-batches and sort each sub-batch individually. In this experiment, we varied the number of sub-batches B in the range of [1, 20]. Figure 18 shows the makespans in JobPacker with different number of sub-batches. All the results are normalized by the results of F-Hybrid. As B increases from 1 to 5, the makespan decreases, since placing the jobs into multiple sub-batches can prevent high competition of resource. However, when $B \geq 5$, increasing B only slightly impacts the performance. This is because dividing the workload into 5 sub-batches is sufficient to prevent the network contention from extremely shuffle-heavy jobs at the same time.

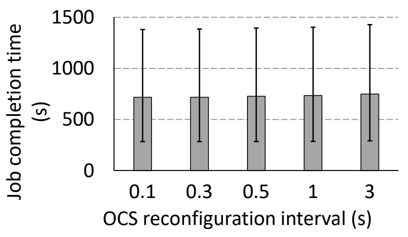


Figure 17: Varying OCS reconfiguration intervals.

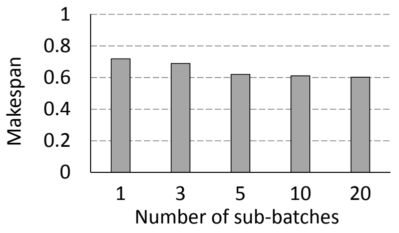


Figure 18: Varying the number of sub-batches.

6 CONCLUSION

We descibed JobPacker to fully takes advantage of OCS in Hybrid-DCN to improve job performance. JobPacker aggregates the data transfers of a job to use OCS effectively. Based on the predictable characteristics of recurring jobs, JobPacker has an offline scheduler to find out all feasible (map-width, reduce-width) pairs for every recurring job that can use OCS effectively while achieving sufficient parallelism, find out the best (map-width, reduce-width) pair with the shortest job completion time, and generate the global schedule including which racks and the sequence to run the recurring jobs that yields the best performance. The offline scheduler also has a new sorting method to prioritize the recurring jobs to prevent high resource contention while fully utilizing cluster resource. Based on the offline schedule, a real-time scheduler places input data and schedules the recurring jobs, and schedules non-recurring jobs to idle resources that are not assigned to recurring jobs. We evaluated JobPacker using large-scale simulation and small-scale emulation on GENI based on production workload, which demonstrates its higher performance in comparison with other schedulers. In the future work, we would like to consider dependency among jobs in the job scheduling.

ACKNOWLEDGMENT

This research was supported in part by U.S. NSF grants NSF-1827674, CCF-1822965, OAC-1724845, CNS-1733596 and ACI-1661378, and Microsoft Research Faculty Fellowship 8300751.

REFERENCES

[1] 2019. Fair Scheduler. https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[2] 2019. GENI. https://www.geni.net.

[3] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. 2012. Re-optimizing data-parallel computing. In *Proc. of USENIX ATC.*

[4] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. 2014. ShufWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *Proc. of ATC.*

[5] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen. 2012. OSA: An optical switching architecture for data center networks with unprecedented flexibility. In *Proc. of NSDI.*

[6] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. 2011. The Case for Evaluating MapReduce Performance Using Workload Suites.. In *Proc. of MASCOTS.*

[7] Z. Li and H. Shen. 2017. Job Scheduling for Data-parallel Frameworks with Hybrid Electrical/Optical Datacenter Networks. In *Proceedings of the Symposium on Cloud Computing, Poster.* 662–662.

[8] L. Cheng, J. Murphy, Q. Liu, C. Hao, and G. Theodoropoulos. 2018. Minimizing Network Traffic for Distributed Joins Using Lightweight Locality-Aware Scheduling. In *European Conference on Parallel Processing.*

[9] L. Cheng, Y. Wang, Y. Pei, and D. Epema. 2017. A coflow-based co-optimization framework for high-performance data analytics. In *Proc. of ICPP.* IEEE.

[10] J. Dean and S. Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of OSDI.*

[11] Z. Li and H. Shen. 2017. 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* 28, 11 (2017), 3201–3214.

[12] J. Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of mathematics* 17, 3 (1965), 449–467.

[13] Z. Li and H. Shen. 2015. Designing a hybrid scale-up/out hadoop architecture based on performance measurements for high application performance. In *Proceedings of International Conference on Parallel Processing.* 21–30.

[14] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. 2010. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. of SIGCOMM.*

[15] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proc. of EuroSys.*

[16] R. L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[17] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. 2014. Multi-resource packing for cluster schedulers. In *Proc. of SIGCOMM.*

[18] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *Proc. of OSDI.*

[19] Z. Li and H. Shen. 2016. Performance measurement on scale-up and scale-out hadoop with remote and local file systems. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD).* IEEE, 456–463.

[20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. 2009. VL2: a scalable and flexible data center network. In *Proc. of SIGCOMM.*

[21] X. S. Huang, X. S. Sun, and T. Ng. 2016. Sunflow: Efficient Optical Circuit Scheduling for Coflows. In *Proc. of CoNext.*

[22] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. 2015. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *Proc. of SIGCOMM.*

[23] Z. Li and H. Shen. 2019. Co-scheduler: Accelerating Data-Parallel Jobs in Datacenter Networks with Optical Circuit Switching. In *Proc. of ICDCS.* IEEE.

[24] Z. Li, H. Shen, W. Ligon, and J. Denton. 2017. An exploration of designing a hybrid scale-up/out hadoop architecture based on performance measurements. *IEEE Transactions on Parallel and Distributed Systems* 28, 2 (2017), 386–400.

[25] 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. 2015. Scheduling techniques for hybrid circuit/packet networks. In *Proc. of CoNext.*

[26] V. Nagarajan, J. Wolf, A. Balmin, and K. Hildrum. 2013. Flowflex: Malleable scheduling for flows of mapreduce jobs. In *Proc. of Middleware.*

[27] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. 2013. Integrating Microsecond Circuit Switching into the Data Center. In *Proc. of SIGCOMM.*

[28] Z. Li, H. Shen, and A. Sarker. 2018. A network-aware scheduler in data-parallel clusters for high performance. In *Proc. of CCGrid.* IEEE, 1–10.

[29] H. Shen and Z. Li. 2016. New bandwidth sharing and pricing policies to achieve a win-win situation for cloud provider and tenants. *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (2016), 2682–2697.

[30] H. Shen, L. Yu, L. Chen, and Z. Li. 2016. Goodbye to fixed bandwidth reservation: Job scheduling with elastic bandwidth reservation in clouds. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom).* IEEE, 1–8.

[31] J. Turek, J. L. Wolf, and P. S. Yu. 1992. Approximate algorithms scheduling parallelizable tasks. In *Proc. of SPAA.*

[32] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of SOCC.*

[33] G. Wang, D. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan. 2010. c-Through: Part-time optics in data centers. In *Proc. of SIGCOMM.*

[34] M. Zaharia, D. Borthakur, S. Sen, K. Elmeleegy, S. Shenker, and I. Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys.*

[35] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI.*