A Network-aware Scheduler in Data-parallel Clusters for High Performance

Zhuozhao Li, Haiying Shen, and Ankur Sarker
Department of Computer Science
University of Virginia, Charlottesville, VA 22903
Email: {zl5uq, hs6ms, as4mz}@virginia.edu

***Abstract*—In spite of many shuffle-heavy jobs in current commercial data-parallel clusters, few previous studies have considered the network traffic in the shuffle phase, which contains a large amount of data transfers and may adversely affect the cluster performance. In this paper, we propose a network-aware scheduler (NAS) that handles two main challenges associated with the shuffle phase for high performance: i) balancing cross-node network load, and ii) avoiding and reducing cross-rack network congestion. NAS consists of three main mechanisms: i) map task scheduling (MTS), ii) congestion-avoidance reduce task scheduling (CA-RTS) and iii) congestion-reduction reduce task scheduling (CR-RTS). MTS constrains the shuffle data on each node when scheduling the map tasks to balance the cross-node network load. CA-RTS distributes the reduce tasks for each job based on the distribution of its shuffle data among the racks in order to minimize cross-rack traffic. When the network is congested, CR-RTS schedules reduce tasks that generate negligible shuffle traffic to reduce the congestion. We implemented NAS in Hadoop on a cluster. Our trace-driven simulation and real cluster experiment demonstrate the superior performance of NAS on improving the throughput (up to 62%), reducing the average job execution time (up to 44%) and reducing the cross-rack traffic (up to 40%) compared with state-of-the-art schedulers.**

I. Introduction

Over the past decade, data-parallel frameworks such as MapReduce [14], Cosmos [9] and Spark [2] become increasingly common for big data analysis [38]. These frameworks need to process petabytes of data every day. Network is often identified as the bottleneck of the data-parallel frameworks because of the network-intensive job running stages such as *shuffle* [22]. Thus, we focus on addressing the network congestion problem in this paper. We use MapReduce as a study case though our proposed methods can be applied to other data-parallel frameworks.

A job in MapReduce consists of the map and reduce stages, each of which consists of multiple map and reduce tasks. When each of these tasks has all its input data ready, it is assigned to *container* on a node to execute, as shown in Figure 1. Each container contains certain amount of CPU and memory resources [22]. Each map task processes one input data block and generates the intermediate key-value pairs (called map output data or shuffle data). Each reduce task consists of two phases: shuffle and reduce phases. In the shuffle phase, all data with the same key from different map tasks is assigned to the same reduce task, and the reduce task fetches the data from the corresponding map tasks. Finally, each reduce task processes the input data and generates the final output. In Hadoop, when a certain percent (called map completion 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 phase can start. Overlapping the map and shuffle phases (i.e., intra-job concurrency) improves the performance in terms of throughput and execution time. Usually, the nodes that process reduce tasks (i.e., reducers) for a job are different from the nodes that process the map tasks (i.e., mappers) for the same job. Thus, the map output data usually is transferred through network to different computing nodes that run the reduce tasks. As a result, almost all the shuffle data is transmitted to different nodes, generating a large amount of cross-node and even cross-rack network traffic.

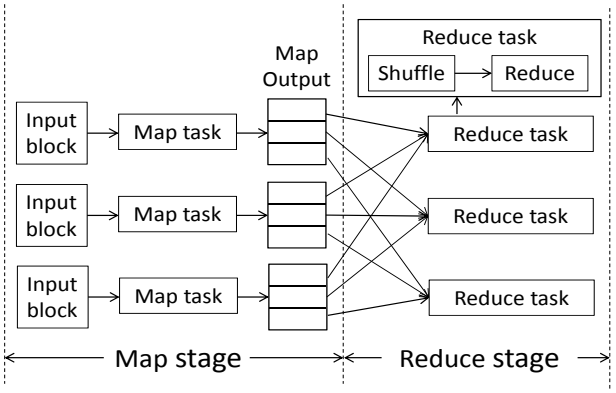


Fig. 1: Map, shuffle and reduce phases in MapReduce [14].

It has been shown that in a modern commercial MapReduce cluster, there are a large amount of shuffle-heavy jobs (i.e., jobs that generate a large amount of shuffle data). For example, 60% and 20% of the jobs are shuffle-heavy jobs on the Yahoo! [10] and Facebook MapReduce clusters [39], respectively. The transfer of shuffle data is the dominant source of cross-node/rack network traffic [5], which greatly affects the performance of MapReduce clusters. In fact, network utilization has been identified as a key factor in increasing the performance of MapReduce clusters [13]. Therefore, it is critical to consider the shuffle phase in task scheduling to reduce the traffic and improve performance.

However, many schedulers [3], [4], [17], [39] only focus

on the map stage. For example, Capacity [3], Fair [4] and Dominant Resource Fairness [17] schedulers aim to achieve the fairness of resource allocation (e.g., CPU, memory, storage and bandwidth) among users or jobs in the map stage. Delay scheduler in [39] aims to improve the map input data locality (i.e., the node running a map task has its required data) in the map stage. ShuffleWatcher [5] has been proposed to reduce cross-rack shuffle data traffic and avoid network congestion. When the cross-rack network is congested, ShuffleWatcher delays scheduling reduce tasks (and hence the shuffle data transfer) by assigning map tasks instead, and assigns a job's reduce tasks to different racks based on the amount of shuffle data each rack contains. Although ShuffleWatcher reduces the cross-rack traffic of the reduce tasks, it increases the cross-rack traffic to read map input data, as mentioned in [22]. Therefore, it is important to consider the shuffle phase in task scheduling to avoid and reduce network congestion without compromising cluster performance. There are three main challenges in designing such a scheduler as listed below.

1. **Balancing cross-node network load.** The output data sizes of map tasks are different from jobs to jobs. Then, the distribution of the sizes of shuffle data transferred among different nodes may be skewed. Since each node has a fixed amount of bisection bandwidth, the map task scheduling may lead to imbalanced shuffle data transfer load among different nodes. That is, the network is congested on some nodes, while remaining idle on some other nodes. Also, the total network load in a cluster at a time must be constrained by controlling cross-node network load.
2. **Avoiding cross-rack network congestion.** Shuffle-heavy jobs require to transfer a large amount of shuffle data across racks, resulting in high requirement of network bandwidth. Many jobs executed by different users simultaneously exacerbate the pressure of the need of network bandwidth. Unlike the CPU, memory, and disk resources that are easy to scale-up by deploying more hardware, network bandwidth is hard to scale-up with current hardware technology [5]. Current datacenter network architectures [6] typically provide cross-rack bandwidth and within-rack bandwidth per node with a ratio of 5:1 to 20:1 [14], [18], [39]. Poor scheduling of reduce tasks on different nodes may lead to cross-rack network congestion, which degrades the performance.
3. **Reducing cross-rack network congestion.** The overlap between the map and shuffle phases improves the performance. ShuffleWatcher sacrifices such overlap to avoid network congestion [5], which degrades the performance. Therefore, it is important to reduce congestion while reducing the sacrifices of intra-job concurrency to achieve better performance.

In this paper, we propose network-aware scheduler (NAS) that incorporates three mechanisms to handle these three challenges respectively. The three mechanisms are map task scheduling (MTS), congestion-avoidance reduce task scheduling (CA-RTS) and congestion-reduction reduce task scheduling (CR-RTS). NAS considers the shuffle phase in both the map and reduce task scheduling.

1. **Map task scheduling (MTS).** MTS aims to balance the cross-node network traffic generated in the shuffle phase by constraining the size of the shuffle data transmitted from each node under a pre-determined threshold. More importantly, through constraining the shuffle data size on each node, MTS constrains the maximum total shuffle data being processed in the cluster and avoids the cross-rack network congestion. Specifically, based on the predicted shuffle data size [36] of each map task, MTS calculates the total shuffle data size of all the map tasks in every node. Once a worker node requests for a map task, MTS checks the user list in the top-down manner until finding a map task with an output data size that can keep the updated total shuffle data size in the node no higher than the threshold, while still providing a certain degree of data-locality and fairness.
2. **Congestion-avoidance reduce task scheduling (CA-RTS).** CA-RTS aims to avoid the cross-rack network congestion while improving cluster performance. For each job, it distributes its reduce tasks based on the distribution of its shuffle data among the racks in order to minimize cross-rack traffic. It also gives higher priority to the reduce tasks of jobs with larger shuffle data sizes and completed map tasks in scheduling in order to fully utilize available bandwidth when the cross-rack network is not congested and start the jobs with all completed map tasks as earlier as possible.
3. **Congestion-reduction reduce task scheduling (CR-RTS).** Usually, both shuffle-heavy and shuffle-light jobs (i.e., jobs that generate very little shuffle data size) run at the same time. Considering that the shuffle-light jobs consume negligible cross-rack network bandwidth [5], rather than delaying the shuffle phase of all jobs as in ShuffleWatcher, CR-RTS does not delay the shuffle phases of shuffle-light jobs. As a result, CR-RTS reduces network congestion while reducing the sacrifices of overlap between the shuffle and map phases.

We implemented NAS in Hadoop on a cluster. Our trace-driven simulation and real cluster experiment demonstrate the superior performance of NAS on improving the throughput (up to 62%), reducing the average job execution time (up to 44%) and reducing the cross-rack traffic (up to 40%) compared with state-of-the-art schedulers.

The rest of the paper is organized as follows. In Section II, we provide an overview of the related work. We describe the main design of our scheduler in Section III and present our experiment evaluation in Section IV. Section V concludes this paper with remarks on our future work.

II. Related Work

Several efforts [3], [4], [17] aim to achieve fairness among jobs or users for the map tasks. Fair scheduler [4] is most widely used in real clusters to achieve fairness among jobs,

i.e., each job occupies approximately the same amount of resources. Dominant Resource Fairness scheduler [17] achieves a max-min fairness for multiple resources (e.g., CPU, memory and I/O). Delay scheduler [39] reduces network traffic by solving the tradeoff between fairness and map input data locality. When the job selected based on fairness cannot launch a local task, Delay scheduler delays the job a small amount of time and launches a local task instead to maintain high data locality. Quincy [21] calculates the cost of each assignment of map tasks and nodes based on locality and fairness, and uses a min-cost flow algorithm to find the optimal scheduling assignment. However, the above schedulers mainly focus on the scheduling of map tasks but do not consider the shuffle phase, which is the major network traffic source in MapReduce clusters [5]. We focus on reducing the shuffle traffic and avoiding cross-rack network congestion to improve the cluster performance within certain data locality and fairness constraints.

Some previous studies (e.g., [19], [32], [35]) consider the scheduling of reduce tasks to improve the cluster performance. Guo *et al.* [19] presented *ishuffle* that actively pushes map output data to nodes and flexibly schedules reduce tasks considering workload balance. Coupling scheduler [35] gradually launches reduce tasks based on the progress of map tasks rather than using a greedy algorithm to launch reduce tasks like Fair scheduler [4]. However, the above works do not reduce the cross-rack network traffic or avoid cross-rack network congestion. Tan *et al.* [34] formulated the reduce task scheduling that minimizes the shuffle data transfer cost to a classic stochastic assignment problem to find out the optimal reduce task placement. Jiang *et al.* [23] designed Symbiosis, which identifies and corrects unbalanced utilization of multiple resources during runtime to improve the resource utilization such as computing and network resources. Purlieus [30] aims to improve locality of MapReduce in a cloud by carefully placing virtual machine and data. These works can avoid cross-rack network congestion by reducing cross-rack traffic but do not handle the cross-rack congestion. Our work not only reduces the cross-rack traffic to avoid congestion but also handles cross-rack network congestion, which greatly improves the performance of MapReduce clusters.

ShuffleWatcher [5] reduces the cross-rack congestion by delaying all the reduce tasks and tries to place the map tasks into one or a fewer racks. However, it sacrifices the intra-job concurrency to achieve higher shuffle locality. Moreover, ShuffleWatcher increases the cross-rack traffic introduced by reading map input data. Compared to ShuffleWatcher, NAS improves both data-locality and intra-job concurrency with its three mechanisms.

Several flow-level techniques [11], [12], [20], [29], [31], [33], [37] have been proposed to decrease the communication time of data-parallel frameworks. The flow-level techniques leverage the Coflow abstraction (i.e., a collection of flows that follow the same objectives such as shuffling for the same job) in data-parallel framework and design flow-level techniques to optimally schedule the flows to improve the Coflow completion time (i.e., the completion time of the last flow in the shuffle phase). These studies are orthogonal to our work and can be combined with our work for better performance.

III. The Design of NAS

In this section, we first introduce the shuffle data size predictor and then introduce each mechanism in NAS.

A. Shuffle Data Size Predictor

Our mechanisms need to learn the shuffle data size beforehand to schedule the map and reduce tasks and distinguish shuffle-heavy and shuffle-light jobs. We utilize a predictor [36] to estimate the map output data size of each map task to be shuffled. The predictor leverages the fact that the map tasks from the same job have similar map output/input ratios. The map output/input ratio of a job is obtained from the completed map tasks for the same job. Then, the predictor extrapolates the map output data size of a map task in the job by:

$$\text{MapOutput} = (\text{map output/input ratio}) * \text{MapInput} \quad (1)$$

where *MapOutput* and *MapInput* are the output and input data size of the map task, respectively.

It is worth mentioning that the shuffle data size can be provided by the users, if it is known in advance, or obtained from previous runs. Many previous studies [7], [16], [22], [24]–[28] indicate that most of the jobs in production clusters are recurring, whose characteristics can be estimated with a low error. For a newly submitted job without knowing its map output/input ratio, the map output/input ratio of the job can be initialized to 1 [5]. We call this kind of jobs as *unpredicted jobs*, otherwise, *predicted jobs*. Once a map task of the job is completed, this task's map output/input ratio can be calculated by $MapOutput/MapInput$. Then, the ratio of this job is updated by calculating the average of all the completed map tasks.

Based on the predicted shuffle data size of a job, we can classify the jobs to shuffle-heavy jobs, shuffle-medium jobs and shuffle-light jobs. For example, in the experiment in Section IV, we define shuffle-light, shuffle-medium and shuffle-heavy jobs as the jobs with shuffle data size smaller than 1MB, in the range of (1-100)MB and larger than 100MB, respectively.

Algorithm 1 Pseudocode of NAS scheduler, which is called when a worker node requests for a task.

Inputs: Current network condition of the rack of this worker node $NetState$

```
1: if request for a map task then
2:     call MTS
3: else if request for a reduce task then
4:     if NetState < CogestionThreshold then
5:         call CA-RTS
6:     else
7:         call CR-RTS
```

B. Overview of NAS

The overall procedure of NAS is illustrated in Algorithm 1. When a node requests for a map task, MTS is invoked to schedule the map task (lines 1-2). The scheduler keeps monitoring the network conditions of the cluster and returns the network condition periodically (e.g., each heartbeat). When a node requests for a reduce task, the network condition is checked (lines 4 and 6). If the network is not congested (lines 4-5), CA-RTS is called to schedule the reduce task. Otherwise (lines 6-7), CR-RTS is called to schedule the reduce task.

C. Map Task Scheduling (MTS)

In this section, we introduce the map task placement (MTS) mechanism. It balances the cross-node network traffic generated by the shuffle phase.

MTS aims to balance the cross-node network load and shape the total cluster network traffic and hence the cross-rack traffic. Specifically, it constrains the shuffle data size generated on each node under its pre-determined threshold. We will explain how to determine the threshold later.

We build MTS upon Delay scheduler [39], which attempts to achieve high data locality while maintaining fairness among users in resource sharing. Accordingly, MTS creates a user list based on fairness, where the users with less resource have higher priority to be allocated with resources. MTS first predicts map output data sizes of all the map tasks running on a worker node, and calculates its available space for map output based on the threshold. Next, from the user list, MTS finds a map task that has output size no larger than the available space (namely shuffle-qualified map task) and also meets the data-locality requirement (i.e., the data block of a task is stored on the same node where the task runs). MTS skips a user if the user does not have a qualified map task.

To achieve fairness between users to a certain degree, as in Delay scheduler [39], MTS sets a maximum skip count D^m. Once a user has been skipped for D^m times, the user's task can be scheduled without satisfying the data-locality or shuffle-qualified requirement. Skipping users will not greatly deviate the fairness requirement. This is because in a large cluster, thousands of tasks run in the cluster, and the containers that enable the tasks of the skipped user to meet the shuffle-qualified and data-locality requirements will be freed in a few seconds [39]. We will present this analysis later.

Algorithm 2 shows the pseudocode of the MTS mechanism. First, MTS searches the tasks of the first user in the user list and tries to find a map task that meets the shuffle-qualified and data-locality requirements. If the user has such a map task, MTS selects this map task (lines 3-4). If the user has several such map tasks, the map task for an unpredicted job has higher priority so that the job can become predicted earlier later on. Then, the map task whose map output data size is the closest to the available space is preferred so that the available shuffle data space can be fully utilized. Once the user's task is scheduled, its map skip counter is set back to 0. When MTS cannot find such a map task from the first user, if the map skip counter equals D^m, MTS identifies a shuffle-qualified map task without the data-locality in the first user (lines 7-12); otherwise, MTS skips the first user, increases its map skip counter by 1 (lines 17-18), and checks the second user in the same manner.

Without data-locality, we give a higher priority to the map tasks from small-input jobs than large-input jobs considering that large-input jobs have more input data blocks throughout the cluster and hence have a higher possibility to launch a local map task later on. Accordingly, we classify the jobs to different categories based on the input data size (first priority) and whether a job is a predicted job (second priority). Take two levels as an example, we categorize the jobs into four categories to select map tasks from as shown in lines 9-12. Note that the categorization of small-input and large-input jobs can be different from clusters to clusters. The cluster operators can define their own thresholds (the same as many other parameters in current Hadoop) to categorize the jobs. For example, in the experiment in Section IV, we classify the jobs with input data size smaller and larger than 10MB as small-input jobs and large-input jobs, respectively. In each category, we further evaluate the data transfer cost of each map task to determine the priority to select a map task. The data transfer cost is calculated by: $MapCost = \gamma * MapInputSize$, where γ is equal to 0 if the map input data is on the local node (data locality); equal to 1 if the map input data is on the local rack (rack locality); and equal to 2 if the map input data is on a remote rack (rack remote). If there are several map tasks with the same *MapCost*, we select the map task whose map output data size is the closest to the available space in order to fully utilize the available shuffle data space.

Algorithm 2 Pseudocode for MTS.

Inputs: Initialize skip count of the i^{th} user $D_i^m = 0$
maximum number of skips D^m

```
1: Calculate the available map output data size on the worker node.
2: for user i in the user list do
3:     if the user has data-local and shuffle-qualified map task then
4:         launch this map task on this node, set D_i^m = 0
5:     else
6:         if D_i^m == D^m then
7:             if we can find shuffle-qualified map tasks of this user
               then
8:                 launch a map task in the following order:
9:                     (1) map task from small-input unpredicted job
10:                    (2) map task from small-input predicted job
11:                    (3) map task from large-input unpredicted job
12:                    (4) map task from large-input predicted job
13:            else
14:                launch a map task in the following order:
15:                    (1) data-local map task
16:                   (2) map task with the smallest map output data size
17:        else
18:            D_i^m++
```

When there is no map task that is shuffle-qualified from all the users (lines 13-16), if there exist data-locality map tasks, MTS selects the one with the smallest shuffle data size since it exceeds *TrafficThreshold* the least; otherwise, MTS just selects

the map task with the smallest shuffle data size among all map tasks (lines 15-16) in order to reduce map output data.

Node traffic threshold determination. Now, we explain how the threshold for the shuffle data size of each node is determined, denoted by *TrafficThreshold*. When all the containers in the cluster are assigned and freed one time, it is called one *wave* of map (reduce) tasks. All submitted map (reduce) tasks cannot be scheduled to the clusters simultaneously and hence they are scheduled through several continuous waves. The shuffle data transfers of the tasks in one wave are conducted in approximately the same time. We set a threshold on each node for two purposes.

- First, it avoids scheduling many map tasks that generate large shuffle data on each node, which balances the cross-node network load.
- Second, it avoids scheduling many map tasks that generate large shuffle data simultaneously in the cluster (i.e., in one wave), which potentially constrains the network traffic generated in the cluster at a time and hence avoids cross-rack network congestion.

We assume that $\{J_1, J_2, ..., J_n\}$ are the n submitted jobs currently in the cluster. Job J_i has S_i shuffle data size and contains K_i map tasks. We assume that in the cluster, there are N nodes, each of which has m containers and hence there are Nm containers in total. The map tasks generate $\sum_{i=1}^{n} S_i$ shuffle data size in total in the cluster. Then, $\sum_{i=1}^{n} K_i$ map tasks are processed in $\sum_{i=1}^{n} K_i/Nm$ waves. We divide the total shuffle data of all the jobs evenly into several waves. The average size of shuffle data generated in each wave is:

$$\frac{\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} K_i/Nm} = \frac{Nm \sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} K_i} \tag{2}$$

Keeping approximately the same amount of shuffle data in each wave in the cluster prevents scheduling many map tasks with large shuffle data sizes at the same time and hence avoids cross-rack congestion.

To achieve a balanced cross-node traffic, we set the threshold for the shuffle data size on each node *TrafficThreshold* as the average size of shuffle data generated on each node in each wave:

$$\frac{Nm \sum_{i=1}^{n} S_i}{N \sum_{i=1}^{n} K_i} = \frac{m \sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} K_i} \tag{3}$$

TrafficThreshold is updated periodically. The cluster operators can change *TrafficThreshold* dynamically (the same as many other parameters in current Hadoop) that serves their own clusters more accurately. For example, in some clusters, there are fewer shuffle-heavy jobs and then *TrafficThreshold* can be set to a smaller value.

Analysis of the map skip counter strategy. We analyze the probability of launching a map task with the data-locality and shuffle-qualified constraints. When a worker node requests for a map task, we assume that user i is the first one in the user list and it has m_i submitted jobs denoted by $\{J_1^i, J_2^i, ..., J_{m_i}^i\}$. Let p_J be the fraction of nodes that have job J's required data and q_J be the probability that the map tasks of job J are shuffle-qualified. Note that q_J is easy to adjust by the cluster operators by setting an appropriate *TrafficThreshold*. Then, the probability that user i cannot launch a map task that meets the data-locality and shuffle-qualified requirements after skipping D^m times is $\prod_{k=1}^{m_i}(1-p_{J_k^i} q_{J_k^i})^{D^m}$. This probability decreases exponentially as D^m decreases. For example, assume that a user has 3 jobs, 10% of nodes have the jobs' input data (i.e., $p_j = 0.1$) and the probability that the map tasks of the jobs are shuffle-qualified is $q_J = 0.5$. Then, this user has a 78.5% probability to launch a map task within 10 skips and a 99.8% probability to launch a map task within 40 skips. In Facebook cluster [39], 27 containers are freed every second on average, which means that there is 99.8% probability to take less than 2 seconds for the user to launch a data-locality and shuffle-qualified map task.

D. Congestion-avoidance Reduce Task Scheduling (CA-RTS)

In this section, we introduce the CA-RTS mechanism, which aims to avoid traffic congestion and reduce the cross-rack network traffic in reduce task scheduling.

We define a threshold of desired upper bound of network utilization *CogestionThreshold* (e.g., 90% of cross-rack bandwidth is used). As in [5], we utilize some network monitor tools (e.g., NetHogs) to monitor the cross-rack network load in the cluster. When a worker node requests for the next reduce task to process, if *CogestionThreshold* is not yet reached, it means that the cross-rack network is not congested and CA-RTS is used for reduce task scheduling. Otherwise, CR-RTS (Section III-E) is used for reduce task scheduling.

Cross-rack traffic reduction method. It is indicated in [8] that for a job that has evenly distributed map output data on several racks, the best placement of reduce tasks to avoid cross-rack congestion on one rack is to evenly distribute the reduce tasks among these racks. Therefore, for each job, keeping the distribution of its reduce tasks the same as the distribution of its shuffle data among the racks can minimize cross-rack traffic and hence avoid cross-rack congestion because placing more reduce tasks of a job on a rack may congest its downlink, while placing fewer reduce tasks of this job on a rack may congest its uplink. That is, for a job, if $x\%$ (called *MapOuputPortion*) of its total map output data is generated in rack R_i, scheduling $x\%$ of its total reduce tasks (denoted by *TotalReduceNum*) in rack R_i can minimize the cross-track network traffic for shuffle data transfer of the job. We define:

$$\text{ReduceNum} = \text{TotalReduceNum} * \text{MapOutputPortion}, \tag{4}$$

where *ReduceNum* denotes the preferred number of reduce tasks on rack R_i in order to reduce the cross-rack traffic in transferring shuffle data.

When CA-RTS handles a reduce task request from a worker node on a rack R_i, it first predicts the shuffle data size (*ShuffleSize*) (as introduced in Section III-A), and then calculates *MapOuputPortion* and *ReduceNum* of each job on rack R_i.

Algorithm 3 shows the pseudocode of CA-RTS. From the first user in the user list (line 1), CA-RTS selects the reduce tasks from the jobs that run fewer reduce tasks than *ReduceNum* on rack R_i (lines 3-5). In addition, CA-RTS also considers i) whether it is delayed in CR-RTS, ii) whether the percentage of completed map tasks of a job, *MapProgressRate*=100%, and iii) *ShuffleSize* to achieve high performance. CA-RTS gives a higher priority to the reduce tasks marked as "delayed" by CR-RTS in order not to delay some reduce tasks for too long. Next, CA-RTS gives a higher priority to the reduce tasks of the jobs with fully completed map tasks (i.e., *MapProgressRate*=100%) in order to start them as early as possible. Finally, CA-RTS prefers the reduce tasks from the jobs with larger shuffle data sizes in order to fully utilize available bandwidth when the cross-rack network is not congested.

Algorithm 3 Pseudocode for CA-RTS.

1: Select a user from the user list based on fairness.
2: Launch reduce task from a job that satisfies map completion threshold in the following order (a job with $delayed$ or $MapProgressRate = 100\%$ has higher priority in the same category):
3: (1) Shuffle-heavy jobs whose $ReduceNum$ is not reached
4: (2) Shuffle-medium jobs whose $ReduceNum$ is not reached
5: (3) Shuffle-light jobs whose $ReduceNum$ not reached
6: (4) Shuffle-light jobs whose $ReduceNum$ is reached
7: (5) Shuffle-medium jobs whose $ReduceNum$ is reached
8: (6) Shuffle-heavy jobs whose $ReduceNum$ is reached

If CA-RTS cannot find the reduce tasks from the jobs that have the number of reduce tasks less than *ReduceNum* on rack R_i, CA-RTS then gives a higher priority to the reduce tasks from the jobs with smaller shuffle data size because such tasks causes a smaller amount of cross-rack traffic (lines 6-8). As a result, CA-RTS reduces the cross-rack traffic generated from shuffle data transfer.

E. Congestion-reduction Reduce Task Scheduling (CR-RTS)

In this section, we introduce the CR-RTS mechanism, which aims to mitigate the cross-rack network congestion caused by shuffle data transfer.

If *CogestionThreshold* is reached, the bandwidth is highly utilized. Delaying scheduling all reduce tasks to reduce the congestion sacrifices intra-job concurrency and compromises performance. To reduce the network congestion while maintaining the overlap between the map and shuffle phases, CR-RTS schedules reduce tasks and map tasks that will not generate a large amount of shuffle data traffic. Specifically, CR-RTS has three strategies. First, it selects the shuffle-light jobs to schedule and delays scheduling the reduce tasks of shuffle-heavy and shuffle-medium jobs until the network is not congested. Second, CR-RTS stops scheduling the map tasks of shuffle-heavy and shuffle-medium jobs. Then, the map completion threshold cannot be reached and the shuffle data of these jobs will not be transferred.

Algorithm 4 shows the pseudocode of CR-RTS. Again, there is a sorted user list created based on Delay scheduler. CR-RTS checks the users in the user list in the top-down manner. From the first user, CR-RTS tries to find a reduce task of shuffle-light job to schedule and delays the reduce tasks of shuffle-medium and shuffle-heavy jobs (lines 1-8). If the first user does not have a reduce task from shuffle-light jobs, CR-RTS searches the next user until it finds a matched reduce task. The reduce skip counter is handled in the same manner as the map skip counter. For the reduce tasks of shuffle-heavy and shuffle-medium jobs, each reduce task has a delay tag. CR-RTS changes the delay flag to "delayed". These delayed tasks will have a higher priority to be scheduled when the network is not congested as explained in Section III-D. Further, CR-RTS notifies MTS not to schedule shuffle-heavy and shuffle-medium map tasks until the network is not congested.

Algorithm 4 Pseudocode for CR-RTS.

Inputs: Initialize skip count of the i^{th} user $D_i^r = 0$
maximum number of skips D^r
1: **for** user i in the user list **do**
2: **if** $D_i^r < D^r$ **then**
3: **if** this user has shuffle-light jobs **then**
4: Select a reduce task from shuffle-light jobs, set $D_i^r = 0$
5: **else**
6: D_i^r++ and skip this user
7: **else**
8: Select a reduce task from any jobs

Analysis of the reduce skip counter strategy. We assume that user i has m_i submitted jobs. Let f_J denote the probability that job J is a shuffle-light job. Therefore, the probability that user i does not have a shuffle-light job is $(1 - f_J)^{m_i}$. Thus, the probability that top u users in the user list do not have a shuffle-light job is $(1 - f_J)^{m_i u}$. Take the Facebook trace [10] as an example. According to our definition of shuffle-light jobs in Section IV, we find that 68.7% of the jobs (f_J=0.687) are shuffle-light jobs. Assume that each user has only one job ($m_i = 1$). Therefore, skipping 3 users ($u = 3$) has a 97% probability of launching a shuffle-light job and skipping 5 users ($u = 5$) has a 99.7% probability of launching a shuffle-light job. When the cross-track network is congested, there should be a large amount of users and jobs in the cluster. Hence, it is very likely to launch a shuffle-light job. Then, CR-RTS needs to skip only a few users or even no users if a user has several jobs, which maintains a high fairness.

F. Complexity of NAS

Similar to current schedulers [4], [39], NAS has very simple computations such as finding shuffle-qualified and data-local map tasks, which are in $\mathcal{O}(n)$ complexity (n is the number of jobs in the list). These computations are quite simple and generate negligible overheads. Also, the network monitor is low-overhead. Therefore, NAS has the same scalability as the state-of-the-art schedulers [4], [39].

IV. Performance Evaluation

In this section, we evaluate NAS in comparison with other schedulers through trace-driven simulation. We also

implemented our scheduler in Hadoop on a real cluster for performance evaluation.

A. Facebook Trace and Experimental Environment

Trace-driven simulation. We used the Facebook day-long workload FB-2010 trace [10] in our simulation. The trace provides detailed information of 24442 jobs. We considered the small-input jobs and large-input jobs as the jobs with input data size smaller and larger than 10MB, respectively. We considered the shuffle-light jobs, shuffle-medium jobs and shuffle-heavy jobs as the jobs with shuffle data size smaller than 1MB, in the range of (1-100)MB, and larger than 100MB, respectively. Figure 2 shows the percentage of jobs of each type in the workload.

Job type	Percentage
Small-input	50.02%
Large-input	49.98%
Shuffle-light	68.70%
Shuffle-medium	12.58%
Shuffle-heavy	18.82%

Fig. 2: Distribution of each job type.

In the simulation, we set the number of users to 200 and the number of nodes to 600 in the cluster, which are consistent with the Facebook cluster reported in [10], [39]. The job arrival time strictly follows the trace. Since there is no user information in the trace, we assigned each job randomly to a user. In the 600-node cluster, we assume that there are 30 racks, each of which has 20 nodes. Each node has 6 containers [39]. The block size was set to 128MB [39]. The replication factor was set to 3. In a commercial cluster, it is common that the cross-rack bandwidth for each node is much lower than the within-rack bandwidth for each node [1], [14], [21], [39]. Like [5], we set the cross-rack bandwidth to 1Gbps (i.e., 50Mbps per-node, which is a typical per-node bisection bandwidth [13], [14]). We set the within-rack bandwidth for each node to 250Mbps, so that the ratio of within-rack and cross-rack bandwidth for each node follows 5:1. Typical oversubscription ratio ranges from 5:1 to 20:1 [1], [14], [39] and with a higher oversubscription ratio than the setting, NAS can achieve more performance improvement than our reported experimental results.

We built an event-based simulator as in [15] to evaluate the performance. The FB-2010 trace does not provide any information about the execution time of each map and reduce task, which is important in our simulation. In order to obtain the trace information about the execution time of each map and reduce task, we used the synthesized execution framework [10] to generate 24442 corresponding jobs based on the trace and ran the jobs on a single node cluster. We carried out experiments on a single-node because it allows us to collect the execution time of each map and reduce task without considering the network. We collected the execution time of each map and reduce task from Hadoop logs. We compared NAS with Fair scheduler (Fair) [4], Delay scheduler (Delay) [39] and ShuffleWatcher based on the Delay scheduler (SW-delay) [5]. Fair is the default and the state-of-the-art scheduler for Hadoop. It achieves fairness among jobs, i.e., each job occupies approximately the same amount of resources. Delay is built upon the Fair scheduler and is another default scheduler in Hadoop. It achieves high data locality of map tasks by delaying the jobs that cannot launch a local map task. ShuffleWatcher can be based on either Fair or Delay. We simulated ShuffleWatcher on top of Delay since it achieves the best throughput in [5]. For both ShuffleWatcher and NAS, we set the network congestion threshold to 80%. We set the maximum map (reduce) skip count $D^m = D^r = 135$.

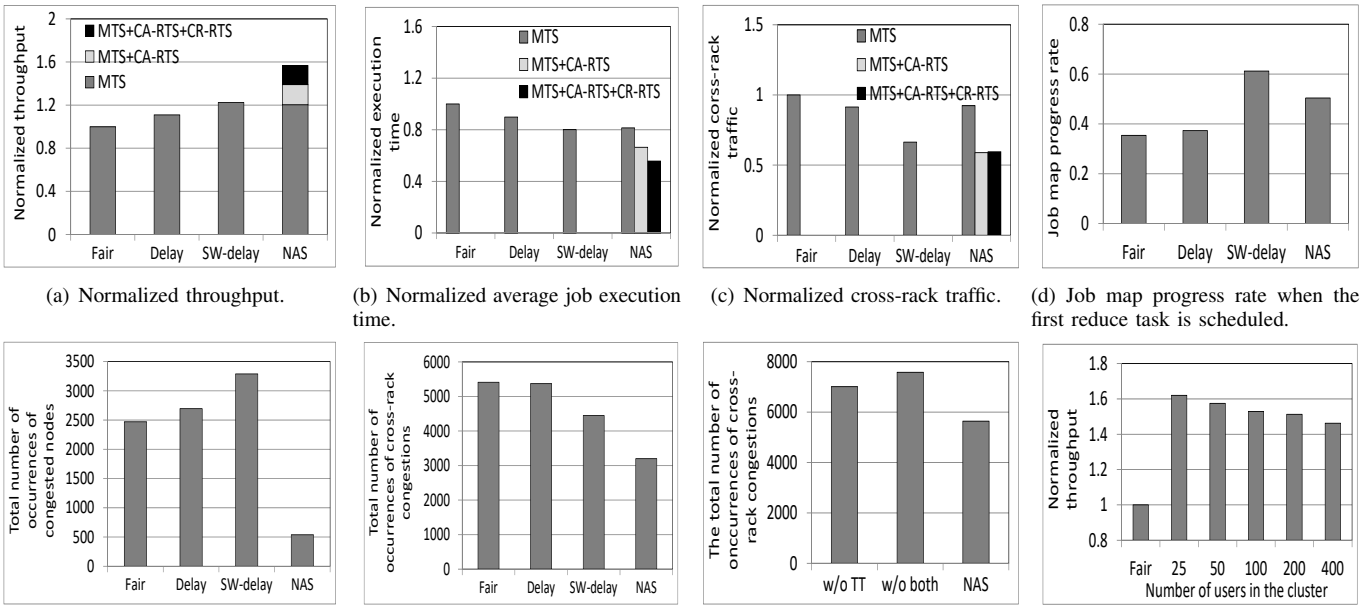
Real cluster experiment. We generated a workload consisting of 200 jobs using the Facebook workload synthesized execution framework [10] and the distribution of job types is the same as shown in Figure 2. As in [39], the job arrival time of these 200 jobs follows an exponential distribution with a mean of 14 seconds, which makes the process of all submissions 45 minutes long. We implemented NAS in Hadoop and conducted the evaluation on a 40-node cluster. The 40 nodes were organized in 8 racks with interconnection of 1Gbps Ethernet. Each rack contains 5 nodes and each node has 1Gbps Ethernet interconnect, resulting in a 5:1 oversubscription ratio. The number of containers on each node was set to 16. The replication factor was set to 3 as default. Other settings are the same as the simulation.

In order to implement NAS in Hadoop, we modified the source codes including ResourceManager, ApplicationMaster, RMAppManager, AMLauncher, and etc. We compared NAS with Fair and Delay schedulers, which are open-source in Hadoop. As in [39], rather than using the maximum skip count D^m and D^r, we set a maximum wait time of 5 seconds, i.e., a user cannot be skipped more than 5 seconds to launch tasks. Other settings are the same as the simulation and other configuration parameters of Hadoop are the same for all the methods.

B. Experimental Results

In order to show the results more clearly, *we normalize the experimental results by the Fair scheduler*. In the figures of simulation, we also show the performance of MTS, MTS+CA-RTS and MTS+CA-RTS+CR-RTS (i.e., NAS) in order to show the effectiveness of different mechanisms in NAS.

We first compared the throughput of different schedulers, which is calculated by the number of jobs (i.e., 24442) divided by the total time to run all the jobs. Figure 3(a) shows the normalized throughput of different schedulers in the simulation. We see that NAS achieves improvement over Fair, Delay, and SW-delay with 56.9%, 45.8%, 34.8% higher throughput. Figure 4(a) shows the normalized throughput of different schedulers in the real cluster experiment. We see that NAS achieves improvement over Fair and Delay with 62.5% and 52.1% higher throughput. We then measured the average job execution time of all the jobs, which is calculated by the sum of job execution time of all the jobs divided by the total number of jobs. Figure 3(b) shows the normalized average job



(a) Normalized throughput.

(b) Normalized average job execution time.

(c) Normalized cross-rack traffic.

(d) Job map progress rate when the first reduce task is scheduled.

(e) Percentage of congested nodes.

(f) The number of occurrences of cross-rack congestions.

(g) The total number of occurrences of cross-rack congestions.

(h) Normalized throughput for different number of users.

Fig. 3: Simulation results comparing NAS with other schedulers.

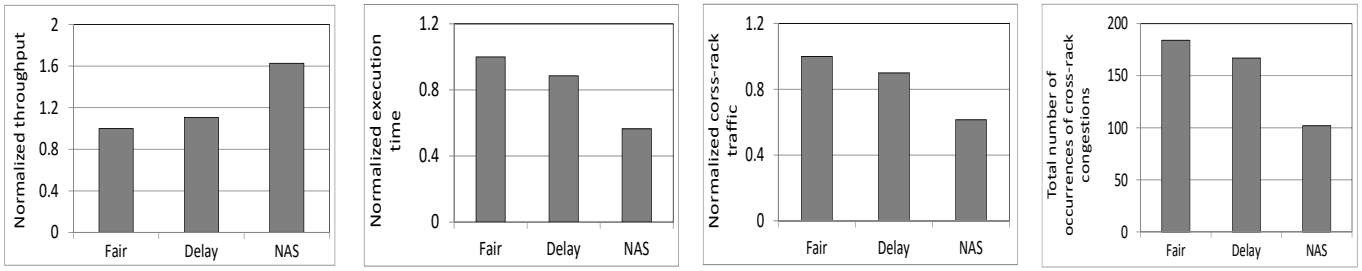
execution time of different schedulers. We see that the average job execution time of NAS is lower than Fair, Delay and SW-delay by 44.3%, 33.9%, and 25.4%, respectively. Figure 4(b) shows the normalized average job execution time of different schedulers in the real cluster experiment. We see that the average job execution time of NAS is 44.6% and 32.1% lower than Fair and Delay, respectively.

SW-delay and NAS outperform the Fair and Delay scheduler because they reduce the cross-rack shuffle network traffic, which greatly expedites shuffle data transfer. NAS produces higher throughput and lower average job execution time than SW-delay. This is because i) NAS balances the shuffle data transfer load on each node, while SW-delay does not, and ii) SW-delay sacrifices the map and reduce phase overlap to achieve higher shuffle locality (i.e., most shuffle data of a reduce task is located on the same rack where this reduce task is run). When the network is saturated, SW-delay delays all the reduce tasks including the shuffle-light jobs, while NAS does not delay the shuffle-light jobs, which are the majority of the jobs in the workload (i.e., 60%). From Figures 3(a) and 3(b), we see that MTS, CA-RTS, and CR-RTS all show great impact on the improvement of throughput and execution time in NAS. The experimental results indicate that NAS outperforms other schedulers on improving the throughput and average job execution time, which demonstrates the effectiveness of the mechanisms in NAS.

Figure 3(c) shows the normalized cross-rack traffic in the simulation, which is measured by the total amount of data transferred cross-rack in the cluster. Figure 4(c) shows the normalized cross-rack shuffle data traffic in the real cluster experiment. We see that SW-delay and NAS produce less cross-rack traffic than Fair and Delay. SW-delay and NAS try to reduce cross-rack shuffle data traffic upon the cross-rack network congestion while Delay and Fair do not address cross-rack network congestion caused by shuffle data transfer.

Figure 3(c) shows that MTS achieves similar cross-rack traffic as Delay, since both MTS and Delay attempt to guarantee data locality for the map tasks. Using CA-RTS with MTS, the cross-rack traffic is decreased, since CA-RTS places the reduce tasks proportional to the map output distribution to minimize the cross-rack traffic. The additional use of CR-RTS does not further decrease the cross-rack traffic. This is because CR-RTS does not reduce cross-rack traffic in the system and it actually shapes the cross-rack traffic (i.e., delays the transfer of heavy shuffle data until the network is uncongested) to reduce cross-rack congestion, which improves the throughput and average job execution time, as shown in Figures 3(a) and 3(b).

In order to show the degree of the overlap sacrifice because of the delay scheduling for the reduce tasks, we draw Figure 3(d), which shows the average *MapProgressRate* for all the jobs when the first reduce tasks of these jobs are scheduled. The *MapProgressRate* of a job equals the fraction of completed map tasks of the job. A lower *MapProgressRate* means more overlap between the map and shuffle phases. We see that the results follows SW-delay>NAS>Delay≈Fair. Since both NAS and SW-delay delay the reduce task scheduling when the network is congested, their overlaps between map and reduce phases are smaller than those of Fair and Delay. Moreover, NAS has a lower average *MapProgressRate* than SW-delay. This is because when the network is congested, SW-delay delays all the reduce tasks, while NAS keeps assigning shuffle-light jobs rather than delaying them, resulting in a



(a) Normalized throughput.

(b) Normalized average job execution time.

(c) Normalized cross-rack traffic.

(d) The number of occurrences of cross-rack congestions.

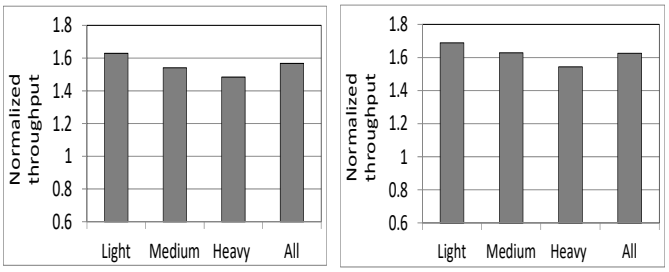
Fig. 4: Real cluster experimental results comparing NAS with other schedulers.

lower average *MapProgressRate*.

Figure 3(e) shows the percentage of nodes that have map output data size higher than *TrafficThreshold* (i.e., congested nodes). We see that NAS has a very small percentage of congested nodes, while Fair, Delay and SW-delay have relatively higher percentage of congested nodes. In NAS, MTS tries to constrain each node's map output data below *TrafficThreshold*. Some nodes become congested because their map skip counter reaches the maximum value and then their map tasks are scheduled without the shuffle-qualified or data-locality constraint. The figure shows that this situation happens only a few times, which means that the cross-node traffic are constrained below *TrafficThreshold* most of the time. In Fair and Delay schedulers, the scheduling of map tasks is only based on fairness without considering cross-node traffic, leading to a large number of congested nodes. SW-delay generates even more congested nodes than Fair and Delay because SW-delay schedules map tasks on the containers when the network is congested, which may result in reading remote input data and hence more congested network. This figure demonstrates the effectiveness of MTS in NAS on constraining cross-node traffic.

Figure 3(f) shows the total number of occurrences of cross-rack congestions in the simulation. Figure 4(d) shows the total number of occurrences of cross-rack congestions in the real cluster experiment. We see that NAS and SW-delay generate fewer cross-rack congestions than Fair and Delay. This is because when the network is close to saturation, SW-delay delays the scheduling of all reduce tasks and NAS delays the scheduling of reduce tasks from shuffle-medium and shuffle-heavy jobs to reduce congestion, while Fair and Delay do not have mechanisms to deal with the network congestion. This figure indicates the effectiveness of NAS on reducing cross-rack congestion.

Recall that setting of *TrafficThreshold* in MTS helps avoid the cross-rack network congestion. To verify this, we tested the performance of NAS without the setting of *TrafficThreshold* in MTS (*w/o TT*), and NAS without the both methods (*w/o both*). Figure 3(g) shows the total number of occurrences of cross-rack congestions during the entire experiment time of these methods compared with NAS. The total numbers of occurrences of *w/o TT*, *w/o both*, and NAS are 3901, 4197, and 3198, respectively. This result demonstrates that the setting of *TrafficThreshold* can help avoid cross-rack network congestion.



(a) Simulation results.

(b) Real cluster results.

Fig. 5: Throughput improvement for different jobs.

The performance of NAS may vary if the number of jobs per user changes. Figure 3(h) shows the normalized throughput of NAS with different number of users in the cluster comparing with Fair with 200 users. Note that when the number of users decreases, the number of jobs per user increases. We varied the number of users for NAS as 25, 50, 100, 200, and 400 in the cluster. We see that NAS achieves higher throughput with fewer users. This is because each user has more jobs when the number of users decreases, which provides more choices for scheduling the appropriate task to the container and hence decreases the needs to skip a user in task searching in MTS and CR-RTS.

We further evaluate how NAS improves the performance of jobs with different shuffle data size. Figures 5(a) and 5(b) show the throughput improvement (i.e., $\frac{with\ NAS}{without\ NAS}$) for shuffle-light, shuffle-medium, and shuffle-heavy jobs in simulation and real cluster experiment, respectively. We see that all kinds of jobs achieve significant improvement in both simulation and real cluster experiments. Specifically, shuffle-light jobs achieve the highest throughput improvement, followed by shuffle-medium and then shuffle-heavy jobs. This is because the shuffle-light jobs tend to have shorter execution time than shuffle-heavy jobs. Without NAS, the short execution time of shuffle-light jobs are severely degraded by shuffle-heavy jobs when the network is congested, since the shuffle-heavy jobs need to wait for the shuffle data transfer and do not release the container resources for a long time. NAS significantly reduces

the network congestion and hence reduces the impact from shuffle-heavy jobs on shuffle-light jobs, which results in the most throughput improvement for shuffle-light jobs.

V. Conclusion

Shuffle data transfer is the dominant source of cross-node/rack network traffic, which greatly affects the performance of MapReduce clusters. However, few previous schedulers handle the network traffic caused in the shuffle phase. Therefore, we proposed a new network-aware MapReduce scheduler (NAS). NAS consists of three mechanisms: Map Task Scheduling (MTS), Congestion-reduction Reduce Task Scheduling (CR-RTS) and Congestion-avoidance Reduce Task Scheduling (CA-RTS). These three mechanisms jointly work to constrain the cross-node network traffic and reduce cross-rack network traffic. We implemented NAS in Hadoop on a supercomputing cluster. Through large-scale trace-driven simulation based on the Facebook workload and real Hadoop cluster experiment, we showed that NAS greatly improves cluster throughput and reduces average job completion time compared with the Fair, Delay and ShuffleWatcher schedulers. In the future, we will extend NAS to schedule the jobs considering the dependency between jobs in which, a job's output is the input of another job. For example, we will consider the placement of dependent jobs to reduce the network traffic generated from input data reading among the dependent jobs.

Acknowledgment

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

References

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Apache Spark. https://spark.apache.org/.

[3] Capacity Scheduler. http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html.

[4] Fair Scheduler. http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *Proc. of ATC*, 2014.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of SIGCOMM*, pages 63–74, 2008.

[7] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. of NSDI*, 2017.

[8] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of OSDI*, pages 1–16, 2010.

[9] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

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

[11] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *Proc. of SIGCOMM*, 2015.

[12] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *Proc. of SIGCOMM*, 2014.

[13] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proc. of NSDI*, 2012.

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

[15] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of USENIX ATC*, 2015.

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

[17] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. of NSDI*, 2011.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proc. of SIGCOMM*, pages 51–62, 2009.

[19] Y. Guo, J. Rao, and X. Zhou. ishuffle: Improving hadoop performance with shuffle-on-write. In *Proc. of ICAC*, 2013.

[20] X. S. Huang, X. S. Sun, and T. Ng. Sunflow: Efficient optical circuit scheduling for coflows. In *Proc. of CoNEXT*, pages 297–311, 2016.

[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proc. of SOSP*, pages 261–276, 2009.

[22] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proc. of SIGCOMM*, pages 407–420, 2015.

[23] J. Jiang, S. Ma, B. Li, and B. Li. Symbiosis: Network-aware task scheduling in data-parallel frameworks. In *Proc. of INFOCOM*, 2016.

[24] Z. Li and H. Shen. Designing a hybrid scale-up/out hadoop architecture based on performance measurements for high application performance. In *Proc. of ICPP*, 2015.

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

[26] Z. Li and H. Shen. Measuring scale-up and scale-out hadoop with remote and local file systems and selecting the best platform. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3201–3214, 2017.

[27] Z. Li, H. Shen, J. Denton, and W. Ligon. Comparing application performance on HPC-based Hadoop platforms with local storage and dedicated storage. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 233–242. IEEE, 2016.

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

[29] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu. Network scheduling aware task placement in datacenters. In *Proc. of CoNEXT*, 2016.

[30] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *Proc. of SC*, 2011.

[31] H. Shen and Z. Li. New bandwidth sharing and pricing policies to achieve a win-win situation for cloud provider and tenants. In *Proc. of INFOCOM*, 2014.

[32] H. Shen, A. Sarker, L. Yu, and F. Deng. Probabilistic network-aware task placement for mapreduce scheduling. In *Proc. of IEEE Cluster*, 2016.

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

[34] J. Tan, S. Meng, X. Meng, and L. Zhang. Improving reducetask data locality for sequential mapreduce jobs. In *Proc. of INFOCOM*, 2013.

[35] J. Tan, X. Meng, and L. Zhang. Coupling task progress for mapreduce resource-aware scheduling. In *Proc. of INFOCOM*, 2013.

[36] A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for mapreduce jobs with performance goals. In *Proc. of Middleware*, 2011.

[37] H. Wang, H. Shen, and G. Liu. Swarm-based incast congestion control in datacenters serving web applications. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 217–226. ACM, 2017.

[38] L. Yan, H. Shen, Z. Li, A. Sarker, J. A. Stankovic, C. Qiu, J. Zhao, and C. Xu. Employing opportunistic charging for electric taxicabs to reduce idle time. In *Proc. of UbiComp*, 2018.

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