Phurti: Application and Network-Aware Flow Scheduling for Multi-Tenant MapReduce Clusters

Chris X. Cai*, Shayan Saeed*, Indranil Gupta*, Roy H. Campbell*, Franck Le†
*Department of Computer Science
University of Illinois at Urbana-Champaign
{xiaocai2,saeed4,indy,rhc}@illinois.edu
†IBM Research T.J.Watson
{fle}@us.ibm.com

***Abstract*—Traffic for a typical MapReduce job in a data center consists of multiple network flows. Traditionally, network resources have been allocated to optimize network-level metrics such as flow completion time or throughput. Some recent schemes propose using application-aware scheduling which can shorten the average job completion time. However, most of them treat the core network as a black box with sufficient capacity. Even if only one network link in the core network becomes a bottleneck, it can hurt application performance.**

We design and implement a centralized flow-scheduling framework called Phurti with the goal of improving the completion time for jobs in a cluster shared among multiple Hadoop jobs (multi-tenant). Phurti communicates both with the Hadoop framework to retrieve job-level network traffic information and the OpenFlow-based switches to learn about the network topology. Phurti implements a novel heuristic called Smallest Maximum Sequential-traffic First (SMSF) that uses collected application and network information to perform traffic scheduling for MapReduce jobs. Our evaluation with real Hadoop workloads shows that compared to application and network-agnostic scheduling strategies, Phurti improves job completion time for 95% of the jobs, decreases average job completion time by 20%, tail job completion time by 13% and scales well with the cluster size and number of jobs.

I. Introduction

The shuffling phase (data transfer from map tasks to reduce tasks) in Hadoop [1] can account for 33% of the running time of a MapReduce job [14]. The shuffling traffic for a job contains multiple flows between host pairs. The reduce phase of the job cannot start until all the flows have finished. In a multi-tenant cluster with multiple jobs running, a job flow might be throttled by traffic belonging to other jobs and can become a straggler. Flow-based scheduling policies [7], [8], [21] decrease the average completion time of the flows. However, they can starve the large flows, thereby increasing the completion time of the job. Consequently, it is important to have application-awareness while scheduling network flows.

In modern datacenters, it is common for multiple MapReduce jobs to share cluster resources i.e. these clusters are multi-tenant.[1] While CPU and memory can be allocated efficiently, it is very hard to control network usage since it is a distributed resource. This means that in addition to application-awareness, it is desirable to have network-awareness during flow scheduling for better application performance. Current network-aware traffic scheduling schemes [6], [23], [22] focus on improving network utilization instead of application performance.

While other application-aware traffic scheduling techniques [14], [15], [17] have been proposed, our goal is to use both application and network topology information for allocation of network resources. Our approach can work in conjunction with the approach by Alkaff *et.al.* [9]. They utilize the application and topology information for task placement and choosing the network route while we perform flow scheduling and bandwidth allocation along predetermined network routes.

We design a centralized scheduling framework called Phurti that provides APIs to dynamically collect the shuffling phase traffic information from Hadoop jobs, as well as network topology and flow routing path information from the OpenFlow-based [25] Software Defined Network (SDN) switches. Phurti provides the option to suspend or throttle the traffic of any job at any time. Unlike a decentralized architecture [17], our approach does not require any change in the network switches, thus making deployment easier. The information and functionality provided by Phurti in turn can be used by any flow scheduling algorithm. To the best of our knowledge, Phurti is the first framework that collects and uses both the application and network information for scheduling the traffic for MapReduce jobs.

We also design and evaluate a new flow scheduling heuristic called Smallest Maximum Sequential-traffic First (SMSF) on top of Phurti to minimize the job completion time of MapReduce jobs. Our algorithm can preempt the flows based on job priority, achieve high network utilization and protect against starvation. Our approach works well when a majority of jobs are small and the datacenter network is congested. Both of these facts have been found to be true in real Hadoop clusters. Facebook traces for Hadoop workloads [11] show that more than 70% are small jobs less than 1 MB in size while [19] shows that network congestion is one of the main reasons for poor job completion times in MapReduce framework.

We deployed and evaluated Phurti on a cluster of 6 machines interconnected by 2 switches. We evaluate it using both simulations and realistic workload generated by the SWIM Facebook workload [11]. Phurti improves job completion time for 95% of the jobs (which in turn also increases cluster

This work was supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, AFOSR/AFRL FA8750-11-2-0084, and generous gifts from Microsoft and Google.

[1]We assume all jobs in the cluster are Hadoop jobs.

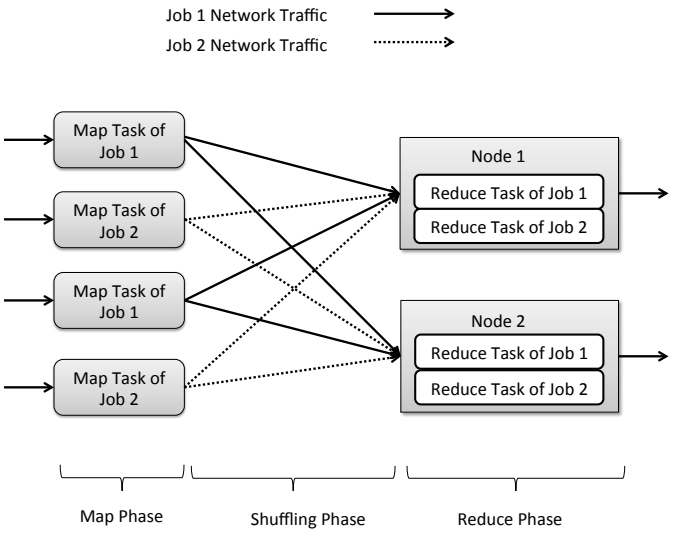


Fig. 1: Traffic pattern for two Hadoop MapReduce jobs in a cluster.

throughput), decreases average job completion time by 20% and tail job completion time by 13%. Via simulations, we also demonstrate that Phurti can scale well to work with a large number of concurrent jobs and clusters involving up to a thousand hosts.

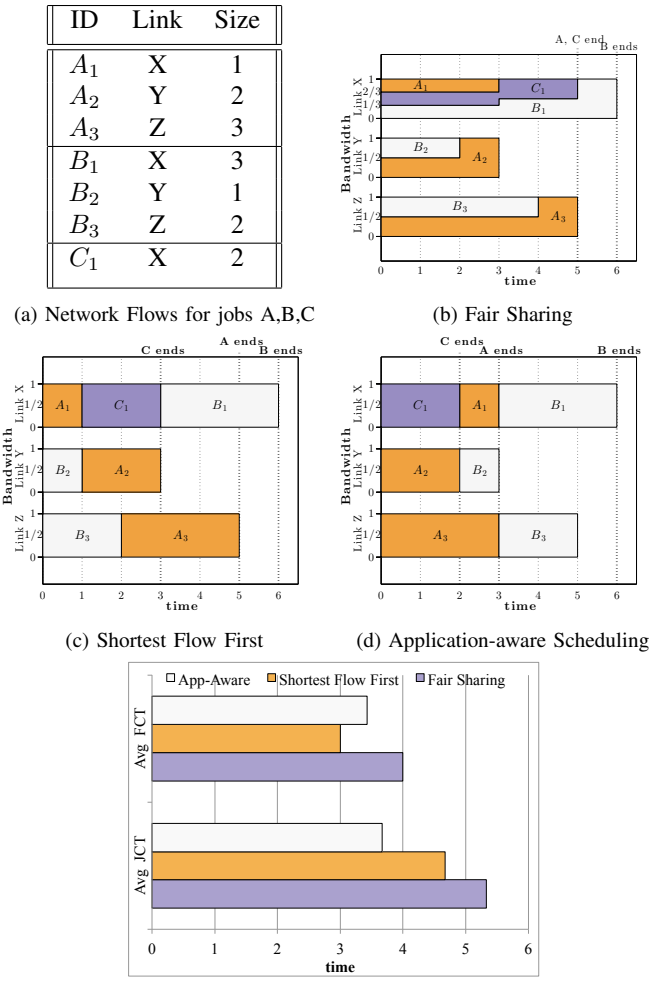
II. Motivation

A. Application-Awareness

The shuffling phase in a typical MapReduce job generates several flows in the network. A flow consists of all the traffic in a transport (e.g. TCP) connection. Since the computation for reduce function cannot start before all the flows in the shuffling phase complete, the overall job completion time depends upon the successful completion of all of the constituent flows of that job. If two large MapReduce jobs happen to send data on shared network links simultaneously, as shown in Figure 1, they may slow each other down due to network contention. This kind of network contention between jobs can be very common in a multi-tenant MapReduce cluster.

Flow-based scheduling policies such as shortest flow first (SFF) concerned with optimizing flow level metrics such as flow completion time etc. have grown popular for datacenter networks [8], [21]. However, since flows of many simultaneous jobs are scheduled independently by such policies, they only perform well for network flow metrics and may not improve application performance. An *application-aware* scheduling strategy would take into account the workload characteristics and schedule all the flows of a job together. This would be more suitable for improving the average job completion time.

We demonstrate this using an example in Figure 2. This shows three concurrent jobs A, B and C running on a shared cluster. A and B are larger jobs with three flows each while C is a small job with only one flow. A fair sharing (FS) strategy such as DCTCP [7] (Figure 2b) divides the bandwidth equally between the flows on shared links. For the example, all the jobs transmit concurrently on link X, so it becomes the bottleneck and increases the average job completion time



(a) Network Flows for jobs A,B,C

(b) Fair Sharing

(c) Shortest Flow First

(d) Application-aware Scheduling

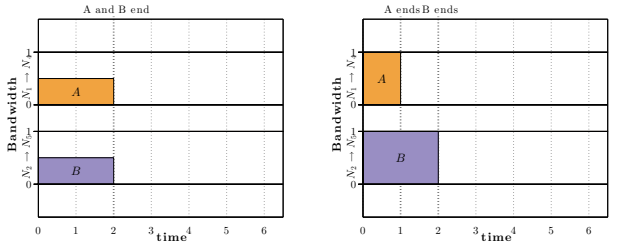
(e) JCT and FCT for Scheduling Strategies

Fig. 2: Application-Aware vs. Application-Agnostic scheduling strategies for three concurrent jobs. Shortest Flow-first has the minimum average flow completion time (FCT) but an application-aware scheduler performs best in terms of average job completion time (JCT).

to 5.33s. A flow based scheduling strategy, shortest-flow-first (SFF) [8], [21] as shown in Figure 2c, serializes the flows on each link and prioritizes the shorter ones on interfering links. This optimizes the average flow completion time but leads to an increase in completion time for both jobs A and B because each job has straggler flows. A simple application-aware scheduling strategy (Figure 2d) serializes the jobs and schedules all their flows together on different links. While this increases the average flow completion time from 3s to 3.43s, it improves the job completion time from 4.67s to 3.67s compared to SFF as shown in Figure 2e. This behavior was also recognized by [15], [17].

B. Network-Awareness

Application-awareness alone is not sufficient for a scheduler to prevent concurrent jobs from slowing each other down. Even if different jobs are scheduled on different nodes, their shuffling traffic might still interfere on a common link inside the network. A network-agnostic scheduler treats the network as a black box and assumes sufficient capacity at the core.



(a) Network-Agnostic Scheduling (b) Network-Aware Scheduling

Fig. 3: Network-Aware vs. Network-Agnostic Scheduling for two concurrent jobs in the network in Fig. 4. Network awareness can reduce conflict in the network and improve the job completion time.

It is unaware of any conflict between interfering flows, so it treats them as independent and schedules them concurrently. This can lead to a slowdown in data transfer if the link does not have sufficient capacity. A *network-aware* scheduler can use the network topology information to help prevent potential network congestion.

Figure 3 considers two jobs, each consisting of one flow of size 1. The topology of the network is shown in Figure 4. All the links in the network have the same capacity. The flows do not share the end hosts, but interfere in the network on the link between the switches S1 and S2. The network-agnostic scheduler (Figure 3a) lets both the jobs send traffic at the same time. As a result, they split the bandwidth of the bottleneck link S1→S2 between each other. This leads to a slowdown and both the jobs complete in 2s. A network-aware scheduler (Figure 3b) would predict the flow *interference* and serialize the jobs. We define interference as the overlap of the paths of network flows from different jobs on at least one link. Initially, job A fully utilizes the link, completes and then job B can utilize the link fully. This reduces the average job completion time from 2s to 1.5s.

III. System Architecture

To achieve application-aware and topology-aware network resource allocation, we design a flow scheduling framework called Phurti. The key idea of Phurti is to enable the applications and OpenFlow switches to pass the information about the system through APIs to enable global network traffic coordination. As shown in Figure 4, we propose a centralized architecture that communicates with the traffic-generating applications as well as with the OpenFlow switches. Phurti receives information about the underlying network topology, host placement and the path taken by each flow from the OpenFlow switches via its Southbound API. Phurti also gathers information about the application generated network traffic by communicating with them via the Northbound API. We now discuss the design goals of these APIs. We will describe more implementation details in Section V.

A. Northbound API

As shown in Figure 4, the Northbound API enables Hadoop to pass information about the shuffling phase traffic of each MapReduce job to Phurti. Whenever a MapReduce job

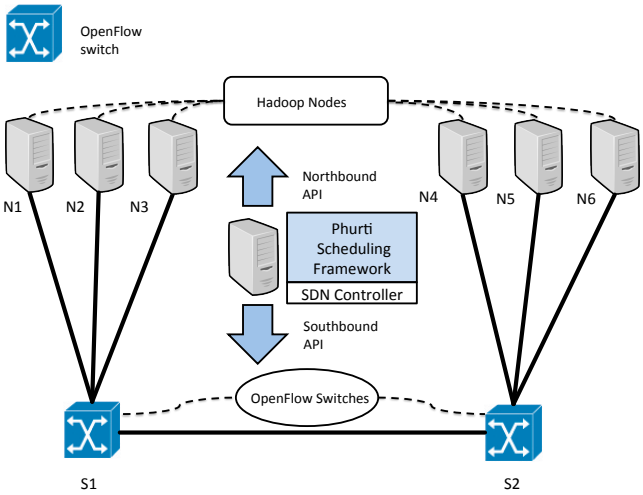


Fig. 4: System Architecture of Phurti. Phurti communicates with the network switches through the southbound API and the Hadoop application through the northbound API.

launches, it contacts Phurti to register the job. It also sends notifications whenever it starts or stops sending the traffic into the network on a per-task basis. It can provide additional information to help in the scheduling decision, e.g., the size of network traffic a job needs to send between any pair of hosts during the shuffling phase, the number of concurrent flows in a job, etc. Phurti implements a rate-limiting module described in Section V that enables flow preemption. Depending on the flow scheduling algorithm, Phurti can choose to suspend, rate limit, or transfer a flow of any given job.

B. Southbound API

For topology-awareness, Phurti leverages the Southbound API to gain knowledge from the OpenFlow switches about the network topology of the cluster, including current hosts in the cluster network and how they are connected. It can also identify the complete path a flow traverses in the network. Acquiring this information allows Phurti to predict where the interference of flows can happen to help avoid congestion.

IV. Scheduling Algorithm

In this section, we describe Phurti's scheduling heuristic with the goal of optimizing end-to-end job completion time. The end-to-end job completion time is the time interval between when a MapReduce job is submitted and when it finishes. We consider heuristics because scheduling the data transfers to minimize average completion time of the shuffling phase has been shown to be NP-hard, even without considering link-level capacity in core network [15].

A. Smallest Maximum Sequential-traffic First Heuristic

Scheduling data transfers of MapReduce jobs purely based on the size of the network traffic of the job can be inefficient. The completion time of the shuffling phase is affected more by the size of the largest amount of network transmission among all pair of hosts, rather than the size of the total amount of traffic. This is because the former is the bottleneck of the shuffling phase.

We define the *sequential-traffic* T_{ij} of a MapReduce job M as the traffic it needs to transmit between host i and host j. Note that the sequential-traffic might consist of multiple flows being transmitted sequentially. For a MapReduce job M, we calculate the *Maximum Sequential-traffic* as $\max(T_{ij}^M)$ across all host pairs (i,j). In Figure 5, the maximum sequential-traffic of job J1 is 1GB, while for job J2 it is 2GB. In Section V, we describe how MapReduce jobs notify Phurti about their maximum sequential-traffic size.

Using this, Phurti's flow scheduling strategy allocates network bandwidth to the flows of MapReduce jobs in increasing order of maximum sequential-traffic of jobs. We call this heuristic Smallest Maximum Sequential-traffic First (SMSF). We further discuss Phurti's mechanism for enforcing SMSF on flows in Section IV-B and Phurti's bandwidth allocation strategy in Section IV-C.

Phurti maintains a priority queue for the jobs. A job with smaller maximum sequential-traffic has higher priority. Phurti updates the priority queue continuously as it receives new information from the Northbound API. When the size of maximum sequential-traffic of a MapReduce job changes, Phurti adjusts its priority accordingly.

The intuition behind SMSF is the well-known observation that smallest-first scheduling policies minimize the completion times [8][20]. Similar to SRTF [20], SMSF sorts the jobs based on the metric that determines the job completion time for MapReduce jobs — the size of its maximum sequential traffic. SMSF allows for preemption and minimizes the average job completion time. We describe how Phurti avoids starvation for the jobs with large maximum sequential-traffic in Section IV-C4.

B. Flows & States

In this section, we describe how Phurti enforces the priority defined by SMSF on the network traffic generated by the MapReduce jobs.

1) Flow States: A flow can belong to one of the two possible states: TRANSMIT and SLOW. For a given link, only flows from one job can be in the TRANSMIT state. All the flows in the SLOW state share a small portion of the bandwidth of the links they traverse while the majority of the bandwidth of the links is used by the flows in the TRANSMIT state.

2) Flow Entry: When a flow arrives, Phurti retrieves the network path for the incoming flow. Phurti checks if there are any higher priority flows in the TRANSMIT state on any of the links along the network path of the incoming flow. If there are, the incoming flow is assigned a SLOW state, so that it does not interfere with the higher priority flows along its path. If not, it starts in the TRANSMIT state and asks Phurti to preempt other flows belonging to lower priority jobs along its path to the SLOW state. Phurti acquires all of these conflicting lower priority flows and rate-limits them to prevent interference. We summarize *Flow Entry* as pseudocode in Algorithm 1.

3) Flow Exit: Phurti keeps track of the state of all the flows and examines these states as the flows finish. Firstly, if a flow finishes in the TRANSMIT state, Phurti retrieves all the links along its path and collects all the flows in SLOW state that are traversing on those links. The collected SLOW flows are sorted in decreasing order of job priority. Each of the sorted SLOW flows is examined to check if it can be switched to TRANSMIT state, by the same procedure used in Section IV-B2. Phurti preempts any flows if necessary. Secondly, if the finished flow was in SLOW state, Phurti takes no action since no readjustment of bandwidth allocation is needed. *Flow Exit* is summarized as pseudocode in Algorithm 2.

Algorithm 1 Flow Entry

```
1:  function FLOWENTRY(flow)
2:      if canTransmit(flow, flow.path) then
3:          flow.state = TRANSMIT
4:          PreemptFlows(flow,flow.path)
5:      else
6:          flow.state = SLOW
7:      end if
8:  end function
9:  function CANTRANSMIT(flow,path)
10:     for Flow f along path do
11:         if f.state == TRANSMIT AND flow.prio ≤ f.prio then
    ▷ interfere with a higher priority flow in TRANSMIT state
12:             return False
13:         end if
14:     end for
15:     return True
16: end function
17: function PREEMPTFLOWS(flow,path)
18:     for Flow f along path do
19:         if f.prio < flow.prio AND f.state == TRANSMIT then
    ▷ interfere with a lower priority flow in TRANSMIT state
20:             f.State = SLOW
21:         end if
22:     end for
23: end function
```

Algorithm 2 Flow Exit

```
1:  function FLOWEXIT(flow)
2:      if flow.state == TRANSMIT then
3:          S_FLOWS = {}
4:          for Flow f along flow.path do
5:              if f.state == SLOW then
6:                  S_FLOWS = S_FLOWS ∪ {f}
7:              end if
8:          end for
9:          for Flow sf ∈ S_FLOWS do
10:             if canTransmit(sf, sf.path) then
11:                 PreemptFlows(sf,sf.path)
12:                 sf.state = TRANSMIT
13:             end if
14:         end for
15:     end if
16: end function
```

C. Flow Management Discussion

1) Avoiding Flow Contention: Since SMSF sorts job in a strict order, on any given link only the flows belonging to the highest priority job are allowed to transmit using the majority of the link capacity. This ensures that the flows of different jobs are not slowing each other down on the same link.

2) Allowing Preemption: Flows of a job can be preempted at any time due to the arrival of flows of higher priority jobs (with the priority defined by SMSF). Without preemption, ongoing flows of low priority jobs can potentially hog network resources and increase average job completion time.

3) Maximal Network Utilization: If there are two concurrent jobs in the cluster, our algorithm serializes them to let the higher priority job transfer first. However, some of the flows of the lower priority job might not interfere with the high priority job. If we use a strict policy to let only one job transfer at a time, the majority of network resources might be idle, which would be undesirable.

Phurti aims for a congestion-free maximal network utilization approach. Network flows of a MapReduce job can start with TRANSMIT state when it arrives, as long as it does not interfere with network flows from higher priority jobs. When determining flow interference, Phurti detects if a higher priority flow is not using a given link fully because of being throttled on another link, i.e. in the SLOW state. The spare capacity is then allocated to the lower priority flows, allowing for maximal network utilization.

4) Starvation Protection: If there is a continuous stream of high priority jobs arriving into the cluster, SMSF can lead to perpetual starvation for low priority jobs. We present a twofold solution to protect the jobs from getting starved as described below.

First, all SLOW flows with same source host s share together a small fraction β of B, where B is the capacity of the link that connects s to the core network. This approach is better than blocking the interfering flows of lower priority jobs. It allows the queued low priority jobs to make some progress, albeit small, even if they remain queued for a long time.

We also keep track of time elapsed since each job is submitted. Every T seconds, we check if a job has been submitted for more than *threshold* seconds and any of its flows are in the SLOW state. For those flows, we switch them to the TRANSMIT state for τ seconds. This ensures that all jobs can make steady progress towards completion without getting stuck behind short jobs perpetually. We mention the default values we use for these system parameters in Section VI-A.

V. Implementation

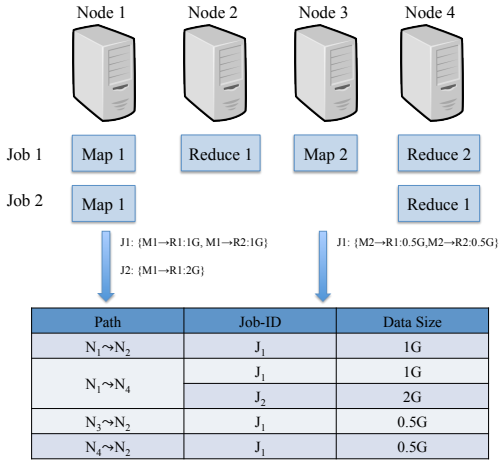
In this section, we present implementation details about how Phurti interacts with Hadoop and OpenFlow switches.

A. Northbound API

The Northbound API of Phurti provides a push-based notify function that can accept different types of messages from Hadoop for communicating the traffic information of different jobs.

1) Job Registration and Unregistration: When a MapReduce job starts, it registers itself by calling notify(JOB_START, jobID). Phurti adds it to the list of active jobs. When a job finishes, it unregisters itself by calling notify(JOB_COMPLETE, jobID).

2) Task Host Notification: When a map or reduce task launches, it calls notify(TASK_HOST, jobID, taskID, host) to notify Phurti of the host this task is running on.



Path	Job-ID	Data Size
$N_1 \rightsquigarrow N_2$	J_1	1G
$N_1 \rightsquigarrow N_4$	J_1	1G
	J_2	2G
$N_3 \rightsquigarrow N_2$	J_1	0.5G
$N_4 \rightsquigarrow N_2$	J_1	0.5G

Fig. 5: An Example of Constructing Traffic Pattern by Phurti. These traffic patterns are later used for assigning priorities to the jobs.

3) Partition Size Notification: When a Map task completes, it notifies the size of intermediate data to Phurti by calling notify(SIZE, jobID, taskID, sizeInformation[]). sizeInformation contains information about the amount of data this map task needs to send to each reduce task.

In Figure 5, we show an example of how Phurti uses the SIZE notifications. There are two MapReduce jobs J1 and J2. J1 has two map tasks: M1 at node 1 and M2 at node 3, and two reduce tasks: R1 at node 2 and R2 at node 4. J2 has one map task M1 at node 1 and one reduce task R1 at node 4. When the map tasks finish, M1 of J1 notifies Phurti it has generated 1GB data for each of the reduce tasks while M2 of J1 notifies it has generated 0.5GB data for each of the reduce tasks. Based on this traffic data and the host information of tasks learned through TASK_HOST notification, Phurti constructs the flows for all ongoing jobs.

4) Flow Registration and Unregistration: When the data transfer from a map task to a reduce task starts, Hadoop notifies Phurti of the source and destination as well as the size via notify(FLOW_REQUEST, jobID, flowID, flowInformation). Phurti keeps track of the state of ongoing flows, including the network paths they traverse, in order to predict where flow interference can happen and make corresponding scheduling decisions. When the data transfer completes, Hadoop notifies Phurti by calling notify(FLOW_COMPLETE, jobID, flowID). This results in actions in Section IV-B3.

B. Southbound API

Phurti uses the Southbound API to discover the underlying network topology and predict flow interference.

1) Path Retrieval: Phurti uses the network topology information provided by OpenFlow to query the network path of a flow. When Phurti needs to identify the path a flow traverses through the network, it calls our implemented function query(GET_PATH, source, destination) where source and destination are the endpoints of the flow. The path returned consists of all the links this flow traverses in the network.

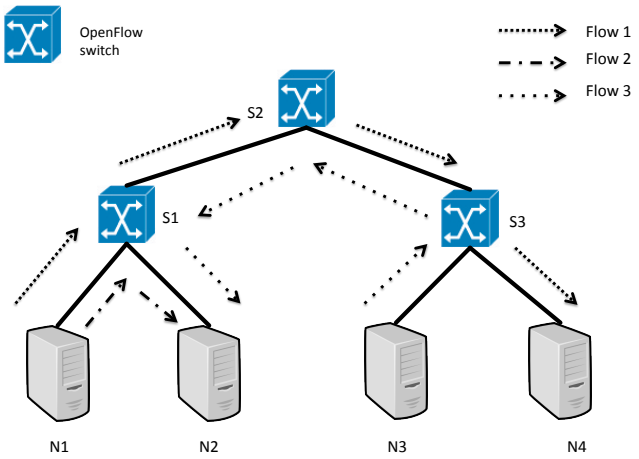


Fig. 6: An example of predicting Flow Interference by Phurti. Phurti can calculate the network path of a flow proactively and predict network interference before the flow starts.

Since SDN controller and OpenFlow switches are continually tracking the network topology, even if there are changes in the network topology caused by adding or removing switches or hosts, Phurti will be capable of detecting these changes.

2) Interference Avoidance: Phurti predicts flow interference by implementing query(CHECK_PATH, PATH1, PATH2). It returns true if two flows intersect at any link in the network. Phurti queries the paths for the flows by using the GET_PATH query and then uses the CHECK_PATH query to detect if those paths intersect on a certain link. These queries utilize native API calls provided by Openflow to get the topology.

An example is shown in Figure 6. Phurti uses GET_PATH query to retrieve the paths P1, P2 and P3 for Flow1, Flow2 and Flow3 respectively. query(CHECK_PATH, P1, P2) returns True since both of them traverse on the same link (N1→S1). query(CHECK_PATH, P1, P3) returns False since there is no overlap between P1 and P3, assuming all the switches are full-duplex. Phurti uses this information to predict flow interference and help avoid congestion.

VI. Evaluation

A. Experimental Setup

We evaluate Phurti on a local testbed running Hadoop YARN 2.3.0. We use default settings for Hadoop parameters including reduce.shuffle.parallelcopies (5) and reduce.slowstart.completedmaps (0.05). We use a realistic workload based on production Hadoop trace from Facebook on a local cluster. We also perform scalability tests using simulations.

Our cluster consists of 6 servers (nodes) divided into 2 racks, where each rack consists of 2 nodes with 6 GB and 1 node with 3 GB RAM configured for Hadoop YARN containers. There are two HP 3500 OpenFlow switches each connected to 3 nodes of the same rack. Both the switches are connected by a single link. The network topology is shown in Figure 4. All the ports of the switches are capable of supporting 100 Megabits/sec full-duplex bandwidth. We use POX [3] as the OpenFlow controller. We run Phurti on a separate server with 2.40 GHz CPU and 4 GB memory.

Choosing the System Parameters: T and *threshold* depend on the workload. *threshold* should be an order of magnitude higher than the average job completion time so we only service the jobs that are truly starved. T should be an order of magnitude less than that so we do not let a job starve for too long a period (i.e. beyond *threshold*+T). If β and τ are too large, then we are not really doing smallest first heuristic, but just approximating fair sharing. If they are too small, we are not protecting against starvation. τ should be an order of magnitude less than T. Based on our workload, we set β to be 1%, *threshold* to be 100 seconds, T to be 10 seconds and τ to be 1 second.

B. Deployment Experiment on Local Cluster

We generate MapReduce jobs using SWIM [11], based on real MapReduce trace from the Facebook cluster. The generated workload consists of 100 jobs. The original workload was collected on a 600-node cluster and we scaled it down proportionally according to our testbed. The scaled trace still maintains original workload characteristics, including job arrival time, job size distribution and time variant cluster utilization.

We divide the jobs in the workload into three categories, based on the size of intermediate data they generate: small, medium and large. Small jobs are jobs with 100MB or less intermediate data, medium jobs have data sizes between 100MB and 1GB, and large jobs have intermediate data sizes more than 1GB. Table I shows the percentage of the number of jobs belonging to each category along with the percentage of the intermediate data size. This shows that the majority of jobs in the workload are small jobs.

TABLE I: Categories of Jobs in Workload.

Job Size	% of Jobs	% of Bytes in Intermediate Data
Small	62%	5.5%
Medium	16%	10.3%
Large	22%	84.2%

Improvement of Job Completion Time: We first show the benefit of Phurti via the main metric of job completion time. For each job in the workload, we take the difference between its job completion time under Phurti and its job completion time under Fair Sharing to show performance improvement. A negative difference for a job shows its job completion time is better under Phurti. We average the results over 10 iterations.

In Figure 7, we plot the CDF of the difference in job completion time for all jobs. The plot shows that around 95% of the jobs have improved job completion time under Phurti compared to Fair Sharing [7]. Around 50% of the jobs have at least 100s improvement in job completion time. Around 5% of the jobs have higher job completion time under Phurti and the worst increase in job completion time is around 100s.

To further understand which of the jobs benefit the most under Phurti and how this improvement compares to Fair

Fig. 7: CDF of difference in Job Completion Time(sec): Phurti vs Fair Sharing. Negative values imply Phurti is better. Phurti improves job completion time for about 95% of jobs.

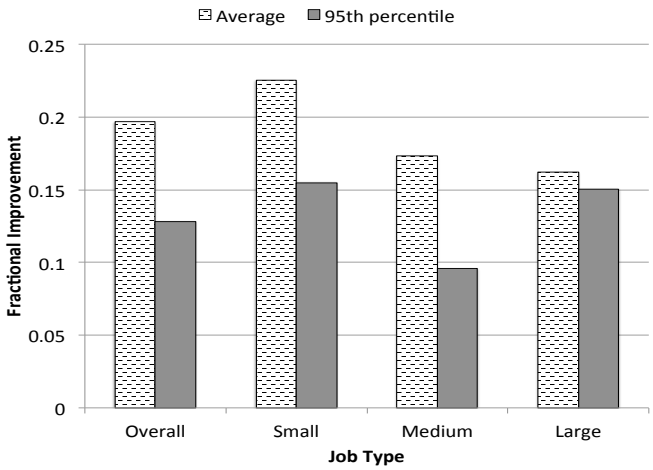


Fig. 8: Fractional Improvement of Job Completion Times (JCT) by Job Category compared to Fair Sharing. Phurti performs best for small jobs.

Sharing, we compute both the average and 95th percentile of the job completion time for each category of jobs. We show the results in Figure 8 as a form of fractional improvement over Fair Sharing. Overall, Phurti achieves an average fractional improvement close to 20% and a 95th percentile improvement of nearly 13%. As expected, small jobs have the highest average fractional improvement of nearly 23% and 95th percentile improvement of 16% among all job categories. This is expected since jobs with smaller size are likely to have smaller maximal sequential-traffic, and thus have a higher priority under Phurti. This is also significant since the majority of jobs are small jobs in the MapReduce workload used (62%).

Starvation Protection: Although our workload is dominated by small jobs which have a higher priority under Phurti, it should be pointed out that the large job category is still able to achieve an average fractional improvement of over 16% with 95th percentile improvement of 15%. This demonstrates that Phurti performs well for large jobs by avoiding perpetual starvation. This is also evident through the tail completion times. The 95th percentile completion time for Phurti shows significant improvements greater than 10% over Fair Sharing for all job categories.

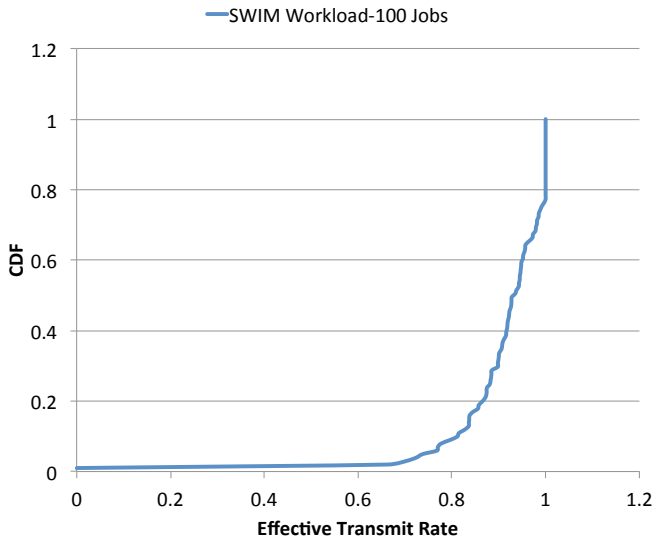


Fig. 9: CDF of Effective Transmit Rate. Higher values are better. The transmit rate of approx. 90% shows that Phurti achieves high utilization.

Effective Transmit Rate: To measure how often traffic is being throttled under Phurti, we compute the *effective transmit rate* for each job. Effective transmit rate for a job is the fraction of the time its flows spend in TRANSMIT state under Phurti. A higher value means that flow traffic of a job spends more time transmitting at its full potential without any congestion or throttling. We plot the CDF of effective transmit rate in Figure 9. Over 90% of the jobs have effective transmit rate larger than 0.8. The effective transmit rate for all the flows on average is greater than 0.9, which means that the flows are not being throttled 90% of the time. Based on this result, we conclude that Phurti succeeds in minimizing throttling.

In Figure 10, we show the average effective transmit rate for jobs in different size categories. The results show that the small jobs have highest average effective transmit rate of nearly 0.95 due to their higher priorities under Phurti. We observe that the average effective transmit rate even for large jobs is around 0.85, despite the fact that their flows have the lowest priorities and are likely to be preempted more frequently by other flows.

C. Simulation

1) Job Scheduling Overhead: We undertake simulation experiments to evaluate the job scheduling overhead for Phurti. In all the experiments, Phurti scheduler is running on a single core server with a 2.40 GHz processor and 8 GB RAM. We measure the effect of scale in terms of the cluster size as well as the number of simultaneous job arrivals. We choose to evaluate Phurti under fat-tree network topology, which is one of the most common network topologies in modern

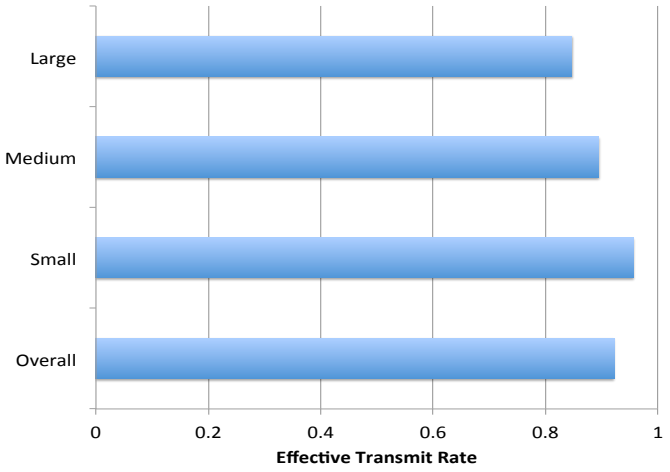


Fig. 10: Average Effective Transmit Rate by Job Category. Small jobs have the best effective transmit rates.

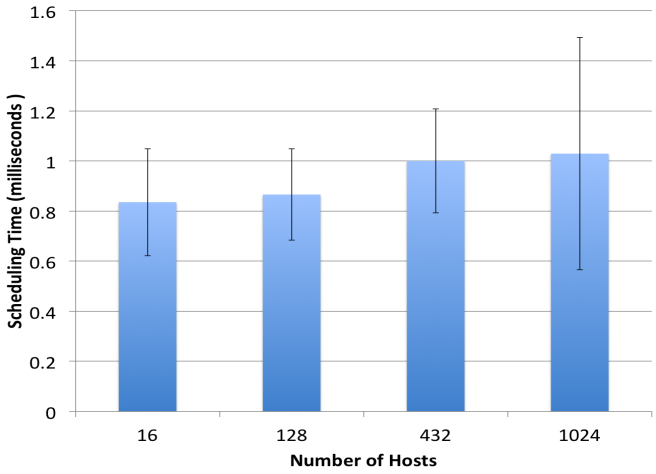


Fig. 11: Scheduling time for a new flow with different size of network topology. For a network topology with 1024 hosts, Phurti takes 6.96 milliseconds for scheduling decision. For a network topology with 1024 hosts, the scheduling time only increases to 1.03 milliseconds.

datacenters. We repeat all experiments for 10 iterations and describe the results in the following sections.

Scheduling Overhead with Increasing Number of Hosts: We simulate fat-tree topologies with increasing number of hosts to check whether Phurti would scale with the number of Hadoop nodes. We inject a new flow with 10 ongoing flows in the network. Figure 11 shows the scheduling overhead when we set the number of ports of each edge switch to be 4, 8, 12 and 16 ports, which corresponds to 16, 128, 432 and 1024 hosts respectively in the fat-tree topology. The graph shows that with 16 hosts, the average scheduling time for the newly arrived flow is around 0.84 milliseconds and for a network with 1024 hosts the average scheduling overhead is around 1.03 ms.

The scheduling overhead grows much slower than linear rate because Phurti is very efficient in making scheduling decisions. Instead of checking the entire topology, it only checks the links along the path a new flow traverses, to predict flow interference. This means that the scheduling overhead for each flow is proportional to the average path size, which grows much slower as the number of hosts increases.

Scheduling Overhead with Simultaneous Flow Arrivals: For this experiment, we simulate a fat-tree topology with 1024 hosts. We launch multiple flows at the same time and measure the time taken by the Phurti scheduler to schedule each flow.

Figure 12 shows that with 20 simultaneous flow arrivals, Phurti takes about 2.68 milliseconds on average to schedule each flow. Even when 100 flows arrive simultaneously, Phurti can still schedule each flow under 5 milliseconds. In the SWIM [11] workload, even when the cluster is most heavily loaded, the number of flows arriving simultaneously never exceeds 20. Compared to the overall job completion time of typical Hadoop MapReduce jobs, the scheduling overhead of Phurti is small under heavy workloads, even with bursty flow arrival patterns.

We evaluated the scalability of Phurti only with the number of simultaneous flow arrivals since the scheduling time does not vary much with the number of ongoing flows. The reason

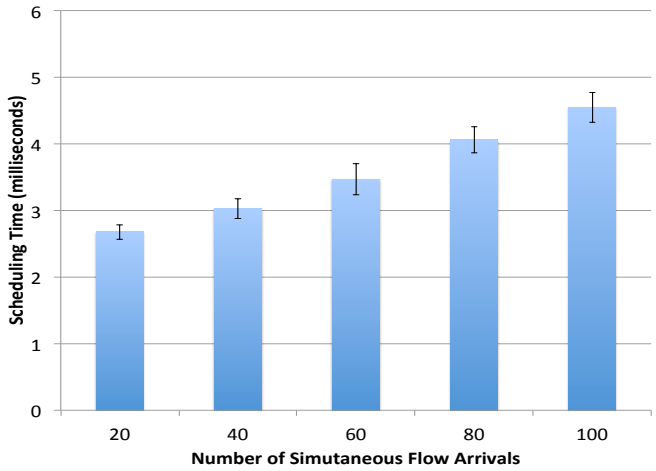


Fig. 12: Average flow scheduling time for simultaneous flow arrivals. With 1024 hosts, Phurti takes 4.55 milliseconds on average to schedule each flow when 100 flows arrive simultaneously.

is that in SMSF, each flow is scheduled based on the available capacity and priority along its network path. Neither of these metrics is influenced by the number of ongoing flows.

2) Comparison with Varys: In [15] the authors already demonstrated application-aware scheduling technique like Varys is more suitable for shuffling traffic than flow-level scheduling techniques including shortest-flow-first (SFF). For this experiment, we focus on comparing Varys and Phurti. We extended the trace-driven CoflowSim [2] simulator used by Varys [15] to test performances of Phurti and Varys under the fat-tree topology. We simulate a fat-tree network topology involving 128 servers. We replay the SWIM workload used in Section VI-B using the simulator and preserve the characteristics, including the number of jobs, shuffle size and number of mapper and reducers per job. We compare the shuffle completion time for each job under Phurti with Varys. Varys only considers the edge link capacity when making the scheduling decision and assumes there is enough network

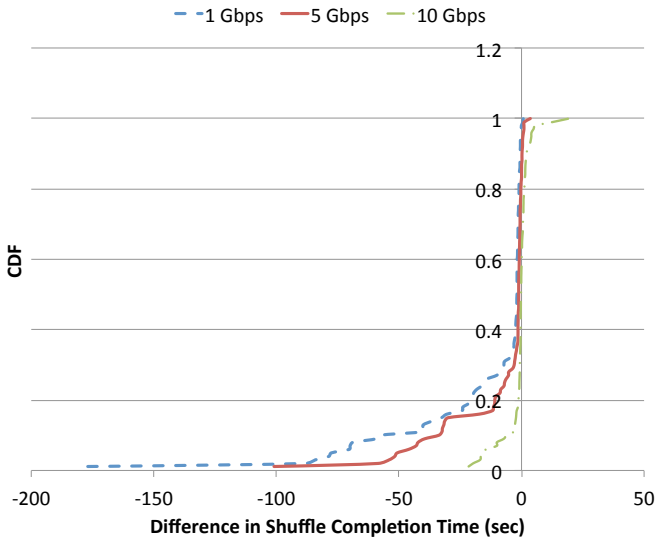


Fig. 13: CDF of difference in Shuffle Completion Time (sec): Phurti vs Varys. Negative values imply Phurti is better. Each of the lines represents the improvement corresponding to a given link capacity C_{ea} between edge switches and aggregation switches. The link capacity C_{se} between server and edge switches is fixed at 1Gbps. Phurti performs better if the oversubscription ratio in the core network increases.

bandwidth in the core network to accommodate the traffic demand.

We fix the link capacity C_{se} between each server and edge switches to be 1Gbps. To evaluate the impact of network topology and the oversubscribed core of datacenter network, we varied the link capacity C_{ea} between edge switches and aggregation switches from 10 Gbps to 1 Gbps. The link capacity between aggregation switches and core switches is always twice as much as C_{ea}.

We calculate the shuffle completion times of Phurti and Varys under different C_{ea} and show the evaluation result in Figure 13. We show the CDF of the difference of shuffle completion time for each job in the workload. A negative value means Phurti is better.

The results show that when C_{ea} has 10:1 ratio to C_{se}, Phurti performs similar to Varys. As the ratio between C_{ea} and C_{se} decreases, Phurti can outperform Varys due to its awareness of network topology. When the ratio between C_{ea} and C_{se} decreases to 1:1, Phurti can improve the shuffling time for nearly 40% of the jobs over Varys. This sort of oversubscription is fairly common in data centers to improve utilization [5].

We conclude that, compared to Varys, Phurti can show improvement of the communication time for a realistic workload when the core network is oversubscribed. The improvement provided by Phurti grows fast as the network becomes more congested. This is because Varys does not consider the core network capacity when scheduling the flows, which leads to increased contention in the core network.

VII. Related Work

Traditional Flow Scheduling and Traffic Engineering: There is a variety of literature about flow scheduling and traffic engineering techniques targeting only network-layer metrics. Both PDQ [21] and pFabric [8] can be used to approximate shortest-flow-first policy that is optimal for reducing the average flow completion time but may lead to increased job completion times. Hedera [6] performs dynamic flow scheduling in a data center network to optimize network capacity. FastPass [26] uses centralized control of both transmission time and path selection of packets to achieve higher throughputs. SWAN [22] and B4 [23] are software defined WANs that use a centralized controller to perform traffic engineering to improve network utilization. Unlike Phurti, all of them are concerned with improving network level metrics but not application performance.

Performance Optimization for Data-Parallel Computing: Task-level optimization for data-parallel computing has been widely studied by the research community. SUDO [28] is an optimization framework that analyzes user-defined functions to avoid unnecessary data-shuffling. RoPE [4] adapts execution plans based on estimates of user-defined code and data properties. Natjam [12] and WOHA [24] use job-level and task-level eviction policies as well as job deadlines to enforce the job priority constraints for Hadoop jobs. PACMan [10] is a distributed cache service that prioritizes jobs with a smaller number of parallel tasks. MapReduce Online [16] allows data to be pipelined between operators. We believe Phurti can work along with these performance optimization techniques to improve the overall job completion time. Phurti does not interfere with the computation processing, memory and storage part of MapReduce jobs that can be efficiently scheduled and improved upon independently.

Application-Aware and Network-Aware Task Schedulers There are frameworks that use the application and cluster information to allocate resources to tasks. Tetris [18] is a scheduler, that assigns tasks to machines based on their requirements for resources such as CPU, memory, storage and network, with priorities based on smallest remaining time first. Wang *et al.* [27] propose using application and network-awareness in schedulers for scheduling jobs and do run-time network configurations to jointly optimize application performance and network utilization. Alkaff *et al.* [9] propose a cross-layer scheduler between the application and the networking layer. It uses the application and network information to perform task placement and select network routes. These task schedulers can work in conjunction with Phurti since they focus on allocating the nodes and network routes to tasks while Phurti performs flow scheduling on precomputed paths.

Application-Aware Traffic Scheduling: Literature that is most closely related to Phurti includes Orchestra [14], Baraat [17] and Varys [15]. Orchestra was one of the first schedulers to advocate for job completion time as a metric. It uses Weighted Shuffle Scheduling to minimize the completion time of a shuffle. However, Orchestra relies on launching multiple

TCP connections to adjust flow transfer rate at a coarser level and the system has to give constant feedback to the hosts to adjust the rate. Instead, Phurti uses explicit rate limiting mechanism, which adjusts flow transfer rate faster and incurs lower traffic overhead.

Baraat utilizes a decentralized task-aware scheduling system to minimize the task completion time. It assigns flow priorities in a task-aware fashion and then uses a flow prioritization heuristic to schedule the tasks in a decentralized fashion. Baraat's approach is at the transport layer and requires modifications to both end-hosts and switches while Phurti is transparent to the underlying network.

Varys uses Coflow [13] abstraction to implement an inter-coflow scheduling policy for improved and predictable communication time. Varys serializes jobs and schedules and rate limits all the flows for the same job such that they finish at the same time. While Phurti prioritizes job transfers in a similar fashion as Varys, it differs from Varys in two important aspects: i) Phurti is network topology-aware, ii) Phurti can schedule a subset of flows of a MapReduce job as soon as they are ready and thus can achieve high network utilization.

VIII. Conclusion

In this paper, we have presented Phurti, which is an application and network topology-aware scheduling framework designed for multi-tenant MapReduce clusters. Phurti has interfaces both with the cluster applications to retrieve job-level traffic information and with the OpenFlow layer to learn the topology of the underlying network. We implemented and evaluated Phurti on a real testbed and demonstrated the advantage of Phurti compared to other traffic scheduling heuristics. Evaluation results on real-world workloads show that Phurti improves job completion time for 95% of the jobs. It decreases the average job completion time by 20% for all jobs and 23% for small jobs. It also prevents starvation by improving the tail job completion time by 13%. It can also scale to large cluster sizes and large number of jobs fairly easily with negligible overhead.

References

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

[2] CoflowSim. https://github.com/coflow/coflowsim/.

[3] POX. https://openflow.stanford.edu/display/ONL/POX+Wiki.

[4] Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I., and Zhou, J. Reoptimizing Data Parallel Computing. In *USENIX NSDI* (2012), pp. 281–294.

[5] Al-Fares, M., Loukissas, A., and Vahdat, A. A Scalable, Commodity Data Center Network Architecture. *ACM SIGCOMM Computer Communication Review 38*, 4 (2008), 63–74.

[6] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *USENIX NSDI* (2010), vol. 10.

[7] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data Center TCP (DCTCP). *ACM SIGCOMM CCR 41*, 4 (2011), 63–74.

[8] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal Near-Optimal Datacenter Transport. In *ACM SIGCOMM* (2013), pp. 435–446.

[9] Alkaff, H., Gupta, I., and Leslie, L. Cross-layer scheduling in cloud systems. In *Cloud Engineering (IC2E)* (2015).

[10] Ananthanarayanan, G., Ghodsi, A., Warfield, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX NSDI* (2012), pp. 267–280.

[11] Chen, Y., Ganapathi, A., Griffith, R., and Katz, R. The Case for Evaluating MapReduce Performance using Workload Suites. In *IEEE MASCOTS* (2011), pp. 390–399.

[12] Cho, B., Rahman, M., Chajed, T., Gupta, I., Abad, C., Roberts, N., and Lin, P. Natjam: Design and Evaluation of Eviction Policies for Supporting Priorities and Deadlines in MapReduce Clusters. In *IEEE SoCC* (2013), pp. 6:1–6:17.

[13] Chowdhury, M., and Stoica, I. Coflow: A Networking Abstraction for Cluster Applications. In *ACM HotNets* (2012), pp. 31–36.

[14] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM CCR* (2011), vol. 41, pp. 98–109.

[15] Chowdhury, M., Zhong, Y., and Stoica, I. Efficient Coflow Scheduling with Varys. In *ACM SIGCOMM* (2014), pp. 93–103.

[16] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. Mapreduce online. In *NSDI'10* (2010).

[17] Dogar, F. R., Karagiannis, T., Ballani, H., and Rowstron, A. Decentralized Task-Aware Scheduling for Data Center Networks. In *ACM SIGCOMM* (2014), pp. 431–442.

[18] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-Resource Packing for Cluster Schedulers. In *ACM SIGCOMM* (2014), ACM, pp. 455–466.

[19] Hammoud, M., Rehman, M. S., and Sakr, M. F. Center-of-Gravity Reduce Task Scheduling to lower MapReduce Network Traffic. In *IEEE CLOUD* (2012), pp. 49–58.

[20] Harchol-Balter, M., Schroeder, B., Bansal, N., and Agrawal, M. Size-based Scheduling to Improve Web Performance. *ACM Trans. Comput. Syst. 21*, 2 (2003), 207–233.

[21] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing flows quickly with preemptive scheduling. *SIGCOMM Comput. Commun. Rev.* (2012), 127–138.

[22] Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. Achieving High Utilization with Software-Driven WAN. In *ACM SIGCOMM* (2013), pp. 15–26.

[23] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *ACM SIGCOMM* (2013), pp. 3–14.

[24] Li, S., Hu, S., Wang, S., Su, L., Abdelzaher, T., Gupta, I., and Pace, R. Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters. In *IEEE ICDCS* (2014).

[25] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR 38*, 2 (2008), 69–74.

[26] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., and Fugal, H. Fastpass: A centralized "zero-queue" datacenter network. In *ACM SIGCOMM* (2014), pp. 307–318.

[27] Wang, G., Ng, T., and Shaikh, A. Programming your Network at Run-time for Big Data Applications. In *ACM HotSDN* (2012), pp. 103–108.

[28] Zhang, J., Zhou, H., Chen, R., Fan, X., Guo, Z., Lin, H., Li, J. Y., Lin, W., Zhou, J., and Zhou, L. Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions. In *USENIX NSDI* (2012), vol. 12, pp. 295–308.