Transparent and Flexible Network Management for Big Data Processing in the Cloud

Anupam Das[¶], Cristian Lumezanu[♯], Yueping Zhang[♯], Vishal Singh[♯], Guofei Jiang[♯], Curtis Yu[♮]
[¶]*UIUC* [♯]*NEC Labs* [♮]*UC Riverside*

Abstract

We introduce FlowComb, a network management framework that helps Big Data processing applications, such as Hadoop, achieve high utilization and low data processing times. FlowComb predicts application network transfers, sometimes before they start, by using software agents installed on application servers and while remaining completely transparent to the application. A centralized decision engine collects data movement information from agents and schedules upcoming flows on paths such that the network does not become congested. Results on our lab testbed show that FlowComb is able to reduce the time to sort 10GB of randomly generated data by 35% while changing paths for only 6% of the transfers.

1 Introduction

Recent years have witnessed the emergence of applications and services (*e.g.*, social media, e-commerce, search) that generate massive collections of data (also known as *Big Data*). To analyze the data quickly and efficiently and extract value for customers, these services use distributed frameworks (*e.g.*, Map/Reduce, Dryad) deployed in cloud environments (*e.g.*, Amazon AWS, Microsoft Azure, Google Apps). The frameworks split the data across clusters of hundreds or thousands of computers, analyze each piece in parallel, and then transfer and merge the results across the cloud network. To reduce the running costs of the cloud provider (who manages the infrastructure) and the customer (who pays by the hour), it is important to improve cluster utilization and keep the duration of data processing jobs low.

Previous research has taken two directions to optimize utilization and keep running time low: *schedule computation* or *schedule communication*. Several works propose to improve job scheduling by preserving data locality [17, 24], maintaining fair allocation among multiple resource types [12] or culling time-consuming tasks [4]. Even with optimal computation scheduling, the cluster network can still become a bottleneck. A previous study on Facebook traces shows that on average, transferring data between successive stages accounts for 33% of the running time and that for many jobs the communication phase takes up more than half of the running time [8].

Consequently, recent proposals seek to schedule communication rather than, or in addition to computation. They optimize network transfers by improving the flow bandwidth allocation [8, 19] or by dynamically changing paths in response to demand [3, 21, 10]. These approaches need accurate and timely application demand information, obtained either from the application itself through instrumentation [8], which is quick and accurate but intrusive, or from the network through monitoring [3], which does not require application involvement, but can be expensive, slow, and detects changes in demand only *after* they have occurred.

We propose FlowComb, a network management middleware for Big Data processing frameworks that is both *transparent* to the applications and *quick and accurate* in detecting their demand. FlowComb detects network transfers, sometimes before they start, and adapts the network by changing the paths in response of these transfers. We present a proof of concept for FlowComb using the Hadoop Map/Reduce framework.

Three questions lie at the foundation of FlowComb's design: (1) how to anticipate the network demand of the application?, (2) how to schedule detected transfers? and (3) how to enforce the schedule in the network?

First, accurately inferring Hadoop network demand without application involvement is difficult. Relying on past demands is not an option because different jobs may have different network footprints [7]. Monitoring the network is expensive and detects demand changes only *after* they have occurred [3]. Instead, we rely on ap-

This work was done while Mr. Das and Mr. Yu were interns at NEC Labs Princeton.

plication domain knowledge to detect network transfers. To alleviate the load on the network and avoid the incast problem [18], Hadoop randomly delays the network transfers of data that becomes available [16]. To detect when this happens, we install agents on each server in the cluster and continually monitor the local tasks and logs.

Second, adapting the network in time after detecting a network transfer is challenging, especially when the transfer is short. A centralized decision engine collects data from each agent and maintains network topology and utilization information. If the pending or current transfer creates congestion, the decision engine finds an alternative path with sufficient available bandwidth. Finally, FlowComb uses OpenFlow to enforce the path and install forwarding rules into switches.

FlowComb balances the load in the network by redirecting flows along paths with sufficient available bandwidth, similarly to Hedera [3] or ECMP [13]. However, FlowComb uses application domain knowledge to detect network transfers that lead to congestion, rather than rely on the network to detect and reschedule only large-volume flows, such as Hedera, or choose paths by hashing selected fields in the packet header, such as ECMP. As argued by others as well, network scheduling with application input may lead to better allocations [22, 5].

FlowComb is effective when at least one network link is fully utilized, *i.e.*, when the application may not be able to improve transfer time by increasing the flow rate. In such situations, shifting part of the traffic on alternate paths is necessary. FlowComb is complementary to recent systems, such as Orchestra [8] or Seawall [19], that perform rate allocation, rather than route allocation, of flows on their default saturated paths.

We deployed FlowComb on a 14 node lab Hadoop cluster connected by a network consisting of two hardware and six software OpenFlow switches. FlowComb is able to reduce the average running time of sorting 10GB of randomly generated data by 35%. While few (6%) of all transfers are rescheduled on alternate paths, 60% of path changes are enforced before the midpoint of a transfer and 10% before the transfer even starts.

To summarize, we propose a Big Data network management framework that detects application demand without application involvement. FlowComb uses software-defined networking to adapt the network paths to the demands. Our work shows that *it is possible to make the network more responsive to application demands* by combining the power of software-defined networking with lightweight end-host monitoring.

2 Motivation

We use Hadoop's MapReduce framework to motivate the design of our system. MapReduce provides a divide-and-conquer data processing model, where large workloads are split into smaller tasks, each processed by a single server in a cluster (the *map* phase). The results of each task are sent over the cluster network (the *shuffle* phase) and merged to obtain the final result (the *reduce* phase). In this section, we describe the network footprint of a MapReduce job, evaluate its impact on the overall data processing, and outline our vision to alleviate it.

2.1 Network footprint

The network footprint of a MapReduce job consists predominantly of traffic sent during the shuffle phase. In some cases, map tasks do not have the required data on the local server and must request it from another node, thus generating additional network traffic. As such scenarios are rare, we do not consider them in our study [1]. We describe the network footprint of MapReduce shuffle from three perspectives: time, volume, burstiness.

Time: The shuffle phase consumes a significant time of the job processing. Chowdhury *et al.* analyzed a week-long trace from Facebook's Hadoop cluster, containing 188,000 MapReduce jobs, and discovered that, on average, the shuffle phase accounted for 33% of the running time [8]. In addition, in 26% of jobs the shuffle takes up more than half of the running time. This shows that attempting to optimize network communication can yield big gains in processing time.

Volume: How much traffic is exchanged during a typical shuffle phase depends on the type of job and the cluster setup. Jobs with small map input-to-output ratio generate less network traffic. Similarly, Hadoop configurations with many tasks running on a server generate less traffic since it is more likely for a mapper and reducer to run on the same server. Chen *et al.* performed a study on seven industrial MapReduce workloads and found that, while the shuffle size varies widely, there are workloads whose processing generates more than 1GB of network traffic among the nodes of the cluster for each job [7].

Burstiness: Previous studies show that MapReduce shuffle traffic is bursty [1]. We set up a Hadoop cluster (see Section 4) and performed several operations, while varying Hadoop configuration parameters, such as replication factor or block size. In all experiments, we observed significant traffic spikes that can introduce network congestion and delay job processing.

2.2 Network impact

Intuitively, having more network capacity reduces the communication time and decreases job processing duration. We use our Hadoop cluster (Section 4 to repeatedly sort the same 10GB workload while varying the capacity of each link in the network from 10 to 100 Mbps. The average sort time (computed over 10 runs) increases from

Link capacity (Mbps)	Avg. processing time (min)
100	39
50	53 (x1.3)
25	67 (x1.7)
10	146 (x3.7)

Table 1: Average job processing time increases as the network capacity decreases. The results represent averages over 10 runs of sorting a 10GB workload on a 14 node Hadoop cluster. The network topology is presented in Figure 2. The numbers within parentheses represent increases from the baseline 100Mbps network.

39 to 146 min (almost four times) when we reduce the link capacity from 100Mbps to 10 Mbps. The results, summarized in Table 1, match our intuition and indicate that finding paths with unused capacity in the network and redirecting congested flows along these paths could improve performance.

2.3 Our goal

As the network plays an important role in the performance of distributed data processing, it is crucial to tune it to the demands of applications. Obtaining accurate demand information is difficult [5]. Requiring users to specify the demand is unrealistic because changes in demands may be unknown to users. Instrumenting applications to give the instant demand is better but is intrusive and deters deployment because it requires modifications to applications [11]. Finally, inferring demand from switch counters [3] does not place any burden on the user or application but gives only current and past statistics without revealing future demand. In addition, polling switch counters must be carefully scheduled to maintain scalability, which may lead to stale information.

Our goal is to build a network management platform for distributed data processing applications that is both *transparent* to the applications and *quick and accurate* in detecting their demand. We propose to use application domain knowledge to detect network transfers (possibly before they start) and software-defined networking to update the network paths to support these transfers without creating congestion. Our vision bridges the gap between the network and the application by introducing a middle layer that collects demand information transparently and scalably from both the application (data transfers) and the network (current network utilization) and adapts the network to the needs of the application.

3 Design

FlowComb improves job processing times and averts network congestion in Hadoop MapReduce clusters by predicting network transfers and scheduling them dynamically on paths with sufficient available bandwidth. Figure 1 highlights the three main components of FlowComb, *flow prediction*, *flow scheduling*, and *flow control*.

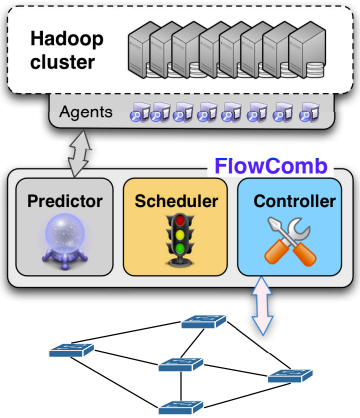

Figure 1: FlowComb consists of three modules: flow prediction, flow scheduling, and flow control.

3.1 Prediction

FlowComb detects data transfers between nodes in a Hadoop cluster using domain knowledge about the interaction between Hadoop components.

Hadoop operation. When a map task finishes, it writes its output to disk and notifies the job tracker, which in turn notifies the reduce tasks. Each reduce task then retrieves from the mapper the data corresponding to its own key space. However, not all transfers start immediately. To avoid overloading the same mapper with many simultaneous requests and burdening themselves with concurrent transfers from many mappers, reducers start a limited number of transfers (5 by default). When a transfer ends, the reducer starts retrieving data from another mapper chosen at random. Hadoop makes available information about the transfer (*e.g.*, source, destination, volume) using its logs or through a web-based API.

Agents. To obtain information about data transfers without modifying Hadoop, we install software agents on each server in the cluster. An agent performs two simple tasks: 1) periodically scans Hadoop logs and queries Hadoop nodes to find which map tasks have finished and which transfers have begun (or already finished), and 2) sends this information to FlowComb's flow scheduling module. To detect the size of a map output, an agent learns the ID of the local mappers from the job tracker and queries each mapper using the web API. Essentially, our agent performs the same sequence of calls as a reducer that tries to obtain information about where to retrieve data. In addition, the agent scans the local Hadoop logs to learn whether a transfer has already started.

3.2 Scheduling

The scheduler receives periodically a list of current or pending data transfers (*i.e.*, source and destination IPs and volume), detects if any of them creates congestion on their default path and if it does, schedules them on a new path. The scheduler maintains a current map of the network with the scheduled flows and available capacity on each link. Three important decisions underline the functioning of the flow scheduler: 1) choose a flow to schedule, 2) decide whether the flow needs another path, and 3) find a good alternate path for it.

Choosing flows. At any moment, the scheduler may have several flows that are waiting to be scheduled. We use FIFO scheduling, where the decision engine schedules flows as it learns about them, because it introduces the least delay in processing a flow. We plan to experiment with other policies, such as prioritizing flows with larger volume or larger bandwidth.

Detecting congestion. Once we have selected a flow to schedule, we must detect whether leaving it on the default path leads to congestion. For this, we compute the flow's natural demand, *i.e.* its max-min fair bandwidth if it was limited only by the sender and the receiver. The natural demand estimates the rate that the flow will have, given the current network allocation. We use the algorithm developed by Al-Fares *et al.* [3] to compute the natural demand. If the natural demands for all active flows together with that of the current flow create congestion then we must choose a new path for our flow.

Choosing a new path. We schedule each flow whose natural demand creates congestion on the first path with enough available bandwidth [3] that we find between the source and the destination of the flow.

3.3 Control

To exploit the full potential of FlowComb, the switches in the network must be programmable from a centralized controller. FlowComb maintains a map of the network with all switches and the flows mapped to paths. It can derive the utilization of each link in two ways: from the server agents (if Hadoop is the only application running in the network) or by polling switch counters (if the network is shared). We leave the description for future work but note that there exist scalable methods for utilization monitoring [20, 23].

4 Preliminary Evaluation

We implemented FlowComb as an extension to the Nox OpenFlow controller [14] and deployed it on a Hadoop 14-server cluster. Figure 2 shows the network topology of the cluster. Our preliminary results focus on the performance of FlowComb. We also discuss the overhead involved in running it and the scalability implications.

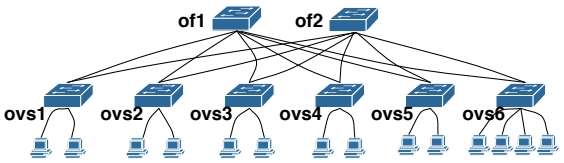

Figure 2: Network topology used for experiments: *of1* and *of2* are NEC PF5240 OpenFlow switches; *ovs1-6* are software switches, running Open Vswitch. All links have 100Mbps capacity.

4.1 Performance

We seek to understand (1) how effective FlowComb is in detecting network transfers and scheduling them on alternative paths, and (2) how much it can reduce job processing time. We initially set the default paths between all servers not connected to the same switch to traverse *of1* and sort 10GB of randomly generated data. Each Hadoop task tracker has two slots for map or reduce tasks; the block size is 64MB and the replication factor is 3. The agent polling period is 1s. We repeat the experiment 10 times.

Prediction. FlowComb detects around 28% of all shuffle phase flows before they start, and 56% before they end. Each flow transfers, on the average 6.5MB of data, and each host is the source of roughly 200 flows during each run. Even though most transfers are short, FlowComb is able to detect them in time for positive gains in running time, as we show below.

Scheduling. FlowComb reroutes few flows (6%) to avert congestion. We compute when FlowComb change the route of a flow relative to the start of the flow. Figure 3 presents the distribution for the *normalized time of path change* (the difference between the route change time and the flow start time divided by the length of the flow) for one run. FlowComb sets up the new path before the flow starts for about 10% of flows, and before the flow ends for 80%. For 60% of the flows, the new path is enacted before the midpoint of the flow.

Processing time. We computed the average job processing time with FlowComb and without FlowComb. FlowComb is able to reduce the time to sort 10GB of data from 39 min to 25 min (by 36%) just by detecting application demand and rerouting a small percentage of flows. We also ran ECMP to randomly assign paths to flows. With ECMP, the sorting took 35 min, better than the baseline but 40% longer than with FlowComb.

4.2 Discussion

4.2.1 Scalability

While we did not have the resources to experiment with FlowComb at scale, we try to identify and discuss the elements that could impact the system's scalability:

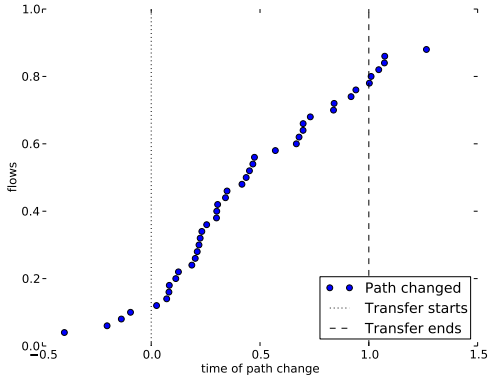

Figure 3: Normalized time of path change for each rescheduled flow, relative to the start and end of the flow. The x value of each point represents the time when the flow's path is changed as a fraction of the flow duration. Points with negative x values correspond to flows rescheduled before they start. Points with x values between 0 and 1 are for flows scheduled before they finish.

Agents. Agents scan log files and query the local task tracker. We measured the increase in CPU utilization that an agent introduces and found that on the average, running an agent adds less than 5% of CPU utilization.

Network. Agents send periodically demand information to the centralized decision engine. Each message containing demand information for one transfer has a payload of 25B. The amount of network traffic increases linearly with the number of transfers detected, which depends linearly on the number of map and reduce tasks. However, because not all agents see the same demand or learn of new demands at the same time, it is unlikely that the decision engine communicate with all agents at the same time and be overloaded by their packets.

Scheduling. The time taken by the decision engine to compute a new path and install it in the network depends on the size of the network and on the number of flows. Previous work analyzed this control loop and found it takes on the average 100 ms, even in deployments with thousands of hosts and tens of flows per host [3].

4.2.2 Limitations

The requirement that FlowComb be transparent to applications yet flexible to adapt the network to new application demands introduces a few limitations.

Transfer start. Although we know that a transfer is about to start, we do not know when, since the reducer starts transfers at random. Thus, we may setup a path that will not be used for some time. However, as shown above, the number of flows for which we change the path is relatively small compared to the total number of flows.

Polling. Polling task trackers or scanning logs introduces computation overhead. Choosing the polling period requires careful consideration. Large periods yield lower overhead but may not detect transfers in time; small periods may prove too demanding on the system.

Applicability. FlowComb is effective when the network is congested. This is more likely to occur for network-heavy MapReduce jobs, where the ratio of map input to output is close to 1, and for large Hadoop clusters, with little data locality.

4.2.3 Extensions

Multiple jobs. Hadoop frameworks frequently execute multiple, unrelated jobs at the same time. While the operation of FlowComb should largely be unchanged under such scenarios, we underline two aspects that may introduce additional overhead. First, monitoring application demand and network utilization may introduce more traffic. Second, the decision engine must be careful in changing the congested path shared by multiple jobs: simply switching all transfers on the same new path would just transfer the congestion to that path.

Other applications. Extending FlowComb to other Big Data processing applications (*e.g.*, Cassandra, HBase) requires domain knowledge of how the application components interact with each other. Whether FlowComb would exhibit the same performance on other platforms depends on the dynamics between application components (*e.g.*, whether data becomes available long before it is sent over the network). We are currently investigating the applicability of FlowComb to the Cassandra key-value storage system.

5 Related Work

Prior work has tackled improving network communications in MapReduce jobs in several ways.

Communication scheduling. Systems such as Orchestra [8] and Seawall [19] propose to improve the performance of the shuffle phase by scheduling flows using a weighted fair sharing scheme rather than the default fair sharing mechanism of TCP. This has the effect of making transfers proportional with data sizes. However, as Chowdhury *et al.* observe [8], when at least one link on a transfer path is fully utilized, there is little to be gained from using a weighted scheme. FlowComb performs route allocation, rather than rate allocation, and is complementary with Orchestra and Seawall

Data aggregation. Rather than modify the way data transfers are scheduled, other systems propose to reduce the amount of data being transferred. Camdoop performs in-network aggregation of the shuffle data by building aggregation trees with the sources of the intermediate

MapReduce data as children and the server executing the final reduction as root [9]. The service, however, is designed specifically for CamCube [2], a cluster built from commodity hardware where servers are connected directly with each other, and does not work in a traditional Hadoop cluster.

Full bisection bandwidth. Many data center topologies are designed to achieve full bisection bandwidth between any two parts of the topology [13, 15]. To take advantage of the multiple paths between any two hosts, operators use ECMP forwarding, where the path selected for a packet is selected by hashing selected fields in the packet's header. However, if flows collide on their hash and follow the same path, it can lead to congestion and reduced performance. FlowComb assigns flows to paths using both application and network demand information, rather than bits in the packet header.

Malleable topologies. c-Through [21], Helios [10], and OSA [6], propose to dynamically allocate optical circuits in response to traffic demand. Recently, Wang *et al.* proposed to make such architectures application-aware using software-defined networking [22]. In their design, the job tracker requests the SDN controller to setup the network for the shuffles with the greatest estimated volume. However, the granularity of their approach is too coarse in that it may impact other traffic traversing the network and which has different requirements.

6 Conclusions

We presented a network management platform for Big Data processing applications that is transparent to the applications yet is able to quickly and accurately detect changes in their demand. FlowComb relies on application domain knowledge to detect network transfers between the application components, sometimes before they even start, and on software-defined networking to change the network path to support these transfers. Experiments on a lab testbed show that FlowComb can improve MapReduce sort times by an average of 35%.

References

[1] Big Data and the Enterprise: Network Design and Considerations. Cisco white paper, 2011.

[2] ABU-LIBDEH, H., COSTA, P., ROWSTRON, A., O'SHEA, G., AND DONNELLY, A. Symbiotic Routing in Future Data Centers. In *ACM Sigcomm* (2010).

[3] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *USENIX NSDI* (2010).

[4] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the outliers in Map-Reduce clusters using Mantri. In *USENIX OSDI* (2010).

[5] BAZZAZ, H. H., TEWARI, M., WANG, G., PORTER, G., NG, T. E., ANDERSEN, D. G., KAMINSKY, M., KOZUCH, M. A., AND VAHDAT, A. Switching the optical divide: Fundamental challenges for hybrid electrical/optical datacenter networks. In *SOCC* (2011).

[6] CHEN, K., SINGLA, A., SINGH, A., RAMACHANDRAN, K., XU, L., ZHANG, Y., WEN, X., AND CHEN, Y. OSA: An optical switching architecture for data center networks with unprecedented flexibility. In *USENIX NSDI* (2012).

[7] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. In *VLDB* (2012).

[8] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M. I., AND STOICA, I. Managing data transfers in computer clusters with orchestra. In *ACM Sigcomm* (2011).

[9] COSTA, P., DONNELLY, A., ROWSTRON, A., AND O'SHEA, G. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *USENIX NSDI* (2012).

[10] FARRINGTON, N., PORTER, G., RADHAKRISHNAN, S., BAZZAZ, H. H., SUBRAMANYA, V., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *ACM Sigcomm* (2010).

[11] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *USENIX NSDI* (2007).

[12] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI* (2011).

[13] GREENBERG, A., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *ACM Sigcomm* (2009).

[14] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev. 38*, 3 (2008), 105–110.

[15] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *ACM Sigcomm* (2009).

[16] Hadoop. `http://hadoop.apache.org/`.

[17] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *SOSP* (2009).

[18] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST* (2008).

[19] SHIEH, A., KANDULA, S., GREENBERG, A., KIM, C., AND SAHA, B. Sharing the data center network. In *USENIX NSDI* (2011).

[20] TOOTOONCHIAN, A., GHOBADI, M., AND GANJALI, Y. OpenTM: Traffic Matrix Estimator for OpenFlow Networks. In *PAM* (2010).

[21] WANG, G., ANDERSEN, D. G., KAMINSKY, M., PAPAGIANNAKI, K., NG, T. E., KOZUCH, M., AND RYAN, M. c-Through: Part-time Optics in Data Centers. In *ACM Sigcomm* (2010).

[22] WANG, G., NG, T. E., AND SHAIKH, A. Programming Your Network at Run-time for Big Data Applications. In *HotSDN* (2012).

[23] YU, C., LUMEZANU, C., SINGH, V., ZHANG, Y., JIANG, G., AND MADHYASTHA, H. V. Monitoring network utilization with zero measurement cost. In *PAM* (2013).

[24] ZAHARIA, M., BORTHAKUR, D., SARMA, J. S., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Eurosys* (2010).