Siphon: Expediting Inter-Datacenter Coflows in Wide-Area Data Analytics

Shuhao Liu, Li Chen, and Baochun Li, *University of Toronto*

https://www.usenix.org/conference/atc18/presentation/liu

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC ’18).
July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Siphon: Expediting Inter-Datacenter Coflows in Wide-Area Data Analytics

Shuhao Liu, Li Chen and Baochun Li
Department of Electrical and Computer Engineering, University of Toronto

Abstract

It is increasingly common that large volumes of production data originate from geographically distributed datacenters. Processing such datasets with existing data parallel frameworks may suffer from significant slowdowns due to the much lower availability of inter-datacenter bandwidth. Thus, it is critical to optimize the delivery of inter-datacenter traffic, especially *coflows* that imply application-level semantics, to improve the performance of such geo-distributed applications.

In this paper, we present *Siphon*, a building block integrated in existing data parallel frameworks (*e.g.,* Apache Spark) to expedite their generated inter-datacenter coflows at runtime. Specifically, Siphon serves as a transport service that accelerates and schedules the inter-datacenter traffic with the awareness of workload-level dependencies and performance, while being completely transparent to analytics applications. Novel intra-coflow and inter-coflow scheduling and routing strategies have been designed and implemented in Siphon, based on a software-defined networking architecture.

On our cloud-based testbeds, we have extensively evaluated Siphon's performance in accelerating coflows generated by a broad range of workloads. With a variety of Spark jobs, Siphon can reduce the completion time of a single coflow by up to 76%. With respect to the average coflow completion time, Siphon outperforms the state-of-the-art scheme by 10%.

1 Introduction

Big data analytics applications are typically developed with modern data parallel frameworks, such as Apache Hadoop [1] and Spark [25], taking advantage of their out-of-the-box features of scalability. With the trend of further scaling out, it has been reported that these applications are deployed across the globe, with their raw input data generated from different locations and stored in geographically distributed datacenters [19, 24]. When processing such geo-distributed data, computation tasks in different datacenters would transfer their intermediate results across the inter-datacenter network, which has much lower bandwidth, typically by an order of magnitude [12], than that within a datacenter. As such, applications that involve heavy inter-datacenter traffic easily suffer from significantly degraded performance, known as the *wide-area data analytics* [23].

To alleviate such performance degradation, existing work in the literature has largely focused on rearranging the pattern of inter-datacenter traffic, with the hope of relieving network bottlenecks. Specifically, one category of such efforts [19, 23, 24] attempted to design optimal mechanisms of assigning input data and computation tasks across datacenters, to reduce or balance the network loads. Another category of the existing work [12, 22] tried to adjust the application workloads towards reducing demands on inter-datacenter communications.

However, given particular traffic from an application, improving its performance by directly accelerating the completion of its inter-datacenter data transfers has been largely neglected. To fill this gap, we propose a deliberate design of a fast delivery service for data transfers across datacenters, with the goal of improving application performance from an orthogonal and complementary perspective to the existing efforts. Moreover, it has been observed that an application cannot proceed until all its flows complete [7], which indicates that its performance is determined by the collective behavior of all these flows, rather than any individual ones. We incorporate the awareness of such an important application semantic, abstracted as *coflows* [8], into our design, to better satisfy application requirements and further improve application-level performance.

Existing efforts have investigated the scheduling of coflows within a single datacenter [6, 8, 17, 27], where the network is assumed to be congestion free and abstracted as a giant switch. Unfortunately, such an assumption no longer holds in the wide area inter-datacenter network, yet the requirement for optimal coflow scheduling to improve application performance becomes even more critical.

In this paper, we present three inter-datacenter coflow scheduling strategies that can significantly improve application-level performance. *First*, we have designed a novel and practical inter-coflow scheduling algorithm to minimize the average coflow completion time, despite the unpredictable available bandwidth in wide-area networks. The algorithm is based on Monte Carlo simulations to handle the uncertainty, with several optimizations to ensure its timely completion and enforcement. *Second*, we have proposed a simple yet effective intra-coflow scheduling policy. It tries to prioritize a subset of flows such that the potential straggler tasks can be accelerated. *Finally*, we have designed a greedy multi-path routing algorithm, which detours a subset of the traffic on a bottlenecked link to an alternate idle path, such that the slowest flow in a shuffle can be finished earlier.

Further, to enforce these scheduling strategies, we have designed and implemented *Siphon*, a new building block for data parallel frameworks that is designed to provide a transparent and unified platform to expedite inter-datacenter coflows.

From the perspective of data parallel frameworks, Siphon decouples *inter-datacenter* transfers from *intra-datacenter* traffic, serving as a transport with full coflow awareness. It can be easily integrated to existing frameworks with minimal changes in source code, while being completely transparent to the analytics applications atop. We have integrated Siphon to Apache Spark [25].

With Siphon, the aforementioned coflow scheduling strategies become feasible thanks to its software-defined networking architecture. For the datapath, it employs *aggregator* daemons on all (or a subset of) workers, forming a virtual overlay network atop the inter-datacenter WAN, aggregating and forwarding inter-datacenter traffic efficiently. At the same time, a *controller* can make centralized routing and scheduling decisions on the aggregated traffic and enforce them on aggregators. Also, the controller can work closely with the resource scheduler of the data parallel framework, to maintain a global and up-to-date knowledge about ongoing inter-datacenter coflows at runtime.

We have evaluated our proposed coflow scheduling strategies with Siphon. Across five geographical regions on Google Cloud, we have evaluated the performance of Siphon from a variety of aspects, and the effectiveness of intra-coflow scheduling in accelerating several real Spark jobs. Our experimental results have shown an up to 76% reduction in the shuffle read time. Further experiments with the Facebook coflow benchmark [8] have shown an $\sim 10\%$ reduction on the average coflow completion time as compared to the state-of-the-art schemes.

We make three original contributions in this paper:

• We have proposed a novel and practical inter-coflow scheduling algorithm for wide-area data analytics. Starting from analyzing the network model, new challenges in inter-datacenter coflow scheduling have been identified and addressed.

• We have designed an intra-coflow scheduling policy and a multi-path routing algorithm that improve WAN utilization in wide-area data analytics.

• We have built Siphon, a transparent and unified building block that can easily extend existing data parallel frameworks with out-of-box capability of expediting inter-datacenter coflows.

2 Motivation and Background

In modern big data analytics, the network stack traditionally serves to deliver *individual flows* in a timely fashion [3, 4, 26], while being oblivious to the application workload. Recent work argues that, by leveraging workload-level knowledge of flow interdependence, the proper scheduling of coflows can improve the performance of applications in datacenter networks [8].

As an application is deployed at an inter-datacenter scale, the network is more likely to be a system bottleneck [19]. Existing efforts in wide-area data analytics [12, 19, 22] all seek to avoid this bottleneck, rather than mitigating it. Therefore, it is necessary to enforce a systematic way of scheduling inter-datacenter coflows for better link utilization, given the fact that the timely completion of coflows can play an even more significant role in application performance.

However, new challenges arise in inter-datacenter networks, which have quite different characteristics as compared to datacenter networks [11, 14]. Such unique characteristics can invalidate the assumptions made by existing coflow scheduling algorithms.

First, inter-datacenter networks have a different network model. Networks are usually modeled as a big switch [8] or a fat tree [20] in the recent coflow scheduling literature, where the ingress and egress ports at the workers are identified as the bottleneck. This is no longer true in wide area data analytics, as the available bandwidth on inter-datacenter links are magnitudes lower than the edge capacity (see Table 1).

Second, the available inter-datacenter bandwidth fluctuates over time. Unlike in datacenter networks, the completion time of a given flow can hardly be predictable, which makes the effectiveness of existing deterministic scheduling strategies (*e.g.*, [8, 27]) questionable. The reason is easily understandable: though the aggregated link bandwidth between a pair of datacenters might be abundant, it is shared among tons of users and their

	Oregon	Carolina	Tokyo	Belgium	Taiwan
Oregon	**3000**	236	250	152.0	194
Carolina	237	**3000**	83.8	251	45.1
Tokyo	83.8	81.7	**3000**	89.2	586
Belgium	249	242	86.6	**3000**	76.0
Taiwan	182	35.8	508	68.0	**3000**

Table 1: Peak TCP throughput (Mbps) achieved across different regions on the Google Cloud Platform, measured with `iperf3` in TCP mode on standard 2-core instances. Rows and columns represent source and destination datacenters, respectively. These statistics match the reports in [12].

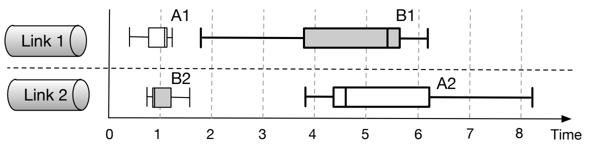


Figure 1: An example with two coflows, *A* and *B*, being sent through two inter-datacenter links. Based on link bandwidth measurements and flow sizes, the duration distributions of four flows are depicted with box plots. Note that the expected duration of *A*1 and *B*2 are the same.

launched applications, with varied, unsynchronized and unpredictable networking patterns.

Third, our ability to properly schedule and route inter-datacenter flows is limited. We may gain full control via software-defined networking within a datacenter [28], but such a technology is not readily available in inter-datacenter WANs. Flows through inter-datacenter links are typically delivered with best effort on direct paths, without the intervention of application developers.

To summarize, it calls for a redesigned coflow scheduling and routing strategy for wide-area data analytics, as well as a new platform to realize in existing data analytics frameworks. In this paper, Siphon is thus designed from the ground up for this purpose. It is an application-layer, pluggable building block that is readily deployable. It can support a better WAN transport mechanism and transparently enforce a flexible set of coflow scheduling disciplines, by closely interacting with the data parallel frameworks. A Spark job with tasks across multiple datacenters, for example, can take advantage of Siphon to improve its performance by reducing its inter-datacenter coflow completion times.

3 Scheduling Inter-Datacenter Coflows

3.1 Inter-Coflow Scheduling

Inter-coflow scheduling is the primary focus of the literature [6, 8, 21, 27]. In this section, we first analyze the practical network model of wide-area data analytics. Based on the new observations, we propose the details of a Monte Carlo simulation-based scheduling algorithm.

3.1.1 Goals and Non-Goals

Our major objective is to *minimize the average coflow completion time*, in alignment with the existing literature. However, we focus on inter-datacenter coflows, which are constrained by a different network model. In particular, based on the measurement in Table 1, we conclude that inter-datacenter links are the only bottlenecked resources, and congestion can hardly happen at the ingress or egress port. For convenience, we call it a *dumb bell* network structure. In addition, we consider the availability of inter-datacenter bandwidth as a dynamic resource. Scheduling across coflows should take runtime variations into account, making a scheduling decision that has a higher probability of completing coflows faster.

Similar to [8, 28], we assume the complete knowledge of ongoing coflows, *i.e.,* the source, the destination and the size of each flow are known as soon as the coflow arrives. Despite recent work [6, 27] which deals with zero or partial prior knowledge, we argue that this assumption is practical in modern data parallel frameworks. It is conceivable that the task scheduler is fully aware the potential cross-worker traffic before launching the tasks in the next stage and triggering the communication stage [1, 7, 25]. We will elaborate further on its feasibility in Sec. 4.3.

3.1.2 Schedule with Bandwidth Uncertainty

Coflow scheduling in a big switch network model has been proven to be NP-hard, as it can be reduced to an instance of the concurrent open shop scheduling with coupled resources problem [8]. With a dumb bell network structure, as contention is removed from the edge, each inter-datacenter link can be considered an independent resource that is used to service the coflows (jobs). Therefore, it makes sense to perform fully preemptive coflow scheduling, as resource sharing always results in an increased average [10].

The problem may seem simpler with this network model. However, it is the sharing nature of inter-datacenter links that complicates the scheduling. The real challenge is, being shared among tons of unknown users, the available bandwidth on a certain link is not predictable. In fact, the available bandwidth a random variable whose distribution can be inferred from history measurements. Thus, the flow durations are also random variables. The coflow scheduling problem in wide-area data analytics can be reduced to the *independent probabilistic job shop scheduling problem* [5], which is also NP-hard.

We seek a heuristic algorithm to solve this online scheduling problem. An intuitive approach is to make an estimation of the flow completion times, *e.g.,* based on the expectation of recent measurements, such that we can solve the problem by adopting a deterministic scheduling policy such as Minimum-Remaining Time First (MRTF) [8, 27].

Unfortunately, this naive approach fails to model the

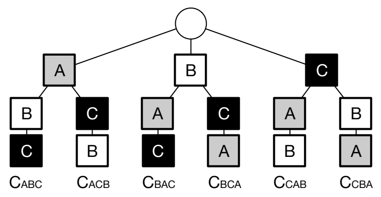


Figure 2: The complete execution graph of Monte Carlo simulation, given 3 ongoing coflows, A, B and C. The coflow scheduling order is determined by the distributions at the end of all branches.

probabilistic distribution of flow durations. Fig. 1 shows a simple example in which deterministic scheduling does not work. In this example, the available bandwidth on Link 1 and 2 have distinct distributions because users sharing the link have distinct networking behaviors. With Coflow A and B competing, the box plots depict the skewed distributions of flow durations if the corresponding coflow gets all the available bandwidth.

With a naive, deterministic approach that considers average only, scheduling either A or B will result in a minimum average coflow completion time. However, it is an easy observation that, with a higher probability, the duration of flow $A1$, will be shorter than $B2$. Thus, prioritizing Coflow A over B should yield an optimum schedule.

3.1.3 Monte Carlo Simulation-based Scheduling

To incorporate such uncertainty, we propose an online Monte Carlo simulation-based inter-coflow scheduling algorithm, which is greatly inspired by the offline algorithm proposed in [5].

The basic idea of Monte Carlo simulation is simple and intuitive: For every candidate scheduling order, we repeatedly simulate its execution and calculate its *cost*, *i.e.,* the simulated average coflow completion time. With enough rounds of simulations, the cost distribution will approximate the actual distribution of average coflow completion time. Based on this simulated cost distribution, we can choose among all candidate scheduling orders at a certain confidence level.

As an example, Fig. 2 illustrates an algorithm execution graph with 3 ongoing coflows. There are 6 potential scheduling orders, corresponding to the 6 branches in the graph. To perform one round of simulation, the scheduler generates a flow duration for each of the node in the graph, by randomly drawing from their estimated distributions. By summing up the cost for each branch, it yields a best scheduling decision instance, which results in a counter increment. After plenty of rounds, the best scheduling order will converge to the branch with the maximum counter value.

One major concern of this algorithm is its high complexity. With n ongoing coflows, there will be up to $n!$ branches in the graph of simulation. Luckily, thanks to the nature of coflow scheduling, we can apply the following techniques to limit the simulation search space.

Bounded search depth. In online coflow scheduling, all we care about is *the* coflow that should be scheduled *next*. This property makes a full simulation towards all leaf nodes unnecessary. Therefore, we set an upper bound, d, to the search depth, and simulate the rest of branches using MRTF heuristic and the median flow durations. This way, the search space is limited to a polynomial time $\Theta(n^d)$.

Early termination. Some "bad" scheduling decisions can be identified easily. For example, scheduling an elephant coflow first will always result in a longer average. Based on this observation, after several rounds of full simulation, we cut down some branches where performances are always significantly worse. This technique limit the search breath, resulting in a $O(n^d)$ complexity.

Online incremental simulation. As an online simulation, the scheduling algorithm should quickly react to recent events, such as coflow arrivals and completions. Whenever a new event comes, the previous job execution graph will be updated accordingly, by pruning or creating branches. Luckily, the existing useful simulation results (or partial results) can be preserved to avoid repetitive computation.

These optimizations are inspired by similar techniques adopted in Monte Carlo Tree Search (MCTS), but our algorithm differs from MCTS conceptually. In every simulation, MCTS tends to reach the leave of a single branch in the decision tree, where the outcome can be revealed. As a comparison, our algorithm has to go though all branches at a certain depth, otherwise we cannot figure out the optimal scheduling for the particular instance of available bandwidth.

3.1.4 Scalability

In wide-area data analytics, a centralized Monte Carlo simulation-based scheduling algorithm may be questioned with respect to its scalability, as making and enforcing a scheduling decision may experience seconds of delays.

We can exploit the parallelism and staleness tolerance of our algorithm. The beauty of Monte Carlo simulation is that, by nature, the algorithm is infinitely parallelizable and completely agnostic to staled synchronization. Thus, we can potentially scale out the implementation to a great number of scheduler instances placed in all worker datacenters, to minimize the running time of the scheduling algorithm and the propagation delays in enforcing scheduling decisions.

3.2 Intra-Coflow Scheduling

To schedule flows belonging to the same coflow, we have designed a preemptive scheduling policy to help flows share the limited link bandwidth efficiently. Our schedul-

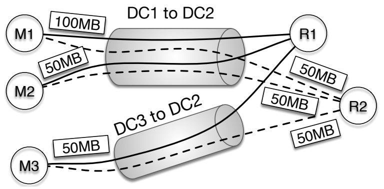


Figure 3: Network flows across datacenters in the shuffle phase of a simple job.

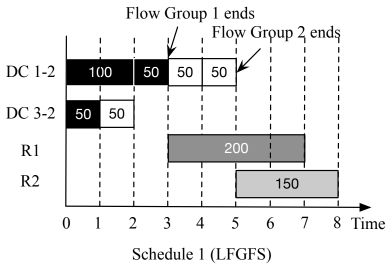


Figure 4: Job timeline with LFGF scheduling.

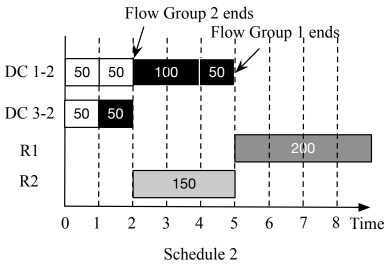


Figure 5: Job timeline with naive scheduling.

ing policy is called *Largest Flow Group First (LFGF)*, whose goal is to minimize *job* completion times. A *Flow Group* is defined as a group of all the flows that are destined to the same reduce task. The size of a flow group is the total size of all the flows within, representing the total amount of data received in the shuffle phase by the corresponding reduce task. As suggested by its name, LFGF preemptively prioritizes the flow group of the largest size.

The rationale of LFGF is to coordinate the scheduling order of flow groups so that the task requiring more computation can start earlier, by receiving their flows earlier. Here we assume that the task execution time is proportional to the total amount of data it received for processing. It is an intuitive assumption given no prior knowledge about the job.

As an example, we consider a simple Spark job that consists of two reduce tasks launched in datacenter 2, both requiring to fetch data from two mappers in datacenter 1 and one mapper in datacenter 3, as shown in Fig. 3. Corresponding to the two reducers *R1* and *R2*, two flow groups are sharing both inter-datacenter links, with the size of 200 MB and 150 MB, respectively. For simplicity, we assume the two links have the same bandwidth, and the calculation time per unit of data is the same as the network transfer time.

With LFGF, Flow Group 1, corresponding to *R1*, has a larger size and thus will be scheduled first. As is illustrated in Fig. 4, the two flows (*M1-R1*, *M2-R1*) in Flow Group 1 are scheduled first through the link between datacenter 1 and 2. The same applies to another flow (*M3-R1*) of Flow Group 1 on the link between datacenter 3 and 2. When Flow Group 1 completes at time 3, *i.e.*, all its flows complete, *R1* starts processing the 200 MB data received, and finishes within 4 time units. The other reduce task *R2* starts at time 5, processes the 150 MB data with 3 units of time, and completes at time 8, which becomes the job completion time.

If the scheduling order is reversed as shown in Fig. 5, Flow Group 2 will complete first, and thus *R2* finishes at time 5. Although *R1* starts at the same time as *R2* in Fig. 4, its execution time is longer due to its larger flow group size, which results in a longer job completion time. This example intuitively justifies the essence of LFGF — for a task that takes longer to finish, it is better to start it earlier by scheduling its flow group earlier.

3.3 Multi-Path Routing

Beyond ordering the coflows, we design a simple and efficient multi-path routing algorithm to utilize available link bandwidth better and to balance network load. The idea is similar to water-filling — it identifies the bottleneck link, and shifts some traffic to the alternative path with the lightest network load in an iterative fashion.

The bottleneck link is identified based on the time it takes to finish all the passing flows. In the first iteration, we calculate all the link load and the time it takes to finish all the passing flows, given that all the flows go through their direct links. To be particular, for each link l, the link load is represented as $D_l = d_i$, where d_i represents the total amount of data of the fetch i whose direct path is link l. The completion time is thus calculated as $t_l = D_l / B_l$, where B_l represents the bandwidth of link l. We identify the most heavily loaded link l^*, which has the largest t_{l^*}, and choose one of its alternative paths which has the lightest load for traffic re-routing. In order to compute the percentage of traffic to be rerouted from l^*, represented by α, we solve the equation $D_{l^*}(1-\alpha)/B_{l^*} = (D_{l^*}\alpha + D_{l'})/B_{l'}$, where l' is the link with the heaviest load on the selected detour path.

4 Siphon: Design and Implementation

4.1 Overview

To realize any coflow scheduling strategies in wide-area data analytics, we need a system that can flexibly enforce the scheduling decisions. Traditional traffic engineering [11, 14] techniques can certainly be applied, but they are not yet available to common cloud users. As is concluded in Sec. 2, Siphon is designed and implemented as a host-based building block to achieve this goal.

Fig. 6 shows a high-level overview of Siphon's architecture. Processes, called *aggregator daemons*, are deployed on all (or a subset of) workers, interacting with the worker processes of the data parallel framework directly. Conceptually, all these aggregators will form

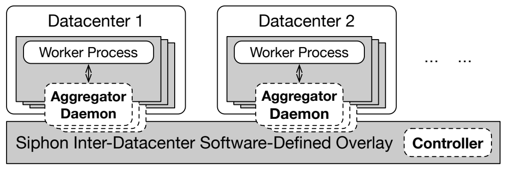


Figure 6: An architectural overview of *Siphon*.

an overlay network, which is built atop inter-datacenter WANs and supports the data parallel frameworks.

In order to ease the development and deployment of potential optimizations for inter-datacenter transfers, the Siphon overlay is managed with the software-defined networking principle. Specifically, aggregators operate as application-layer switches at the data plane, being responsible for efficiently aggregating, forwarding and scheduling traffic within the overlay. Network and flow statistics are also collected by the aggregators actively. Meanwhile, all routing and scheduling decisions are made by the central Siphon *controller*. With a flexible design to accommodate a wide variety of flow scheduling disciplines, the centralized controller can make fine-grained control decisions, based on coflow information provided by the resource scheduler of data parallel frameworks.

4.2 Data Plane

Siphon's data plane consists of a group of aggregator daemons, collectively forming an overlay that handles inter-datacenter transfers requested by the data parallel frameworks. Working as application-layer switches, the aggregators are designed with two objectives: it should be simple for data parallel frameworks to use, and supports high switching performance.

4.2.1 Software Message Switch

The main functionality of an aggregator is to work as a software switch, which takes care of fragmentizing, forwarding, aggregating and prioritizing the data flows generated by data parallel frameworks.

After receiving data from a worker in the data parallel framework, an aggregator will first divide the data into fragments such that they can be easily addressable and schedulable. These data fragments are called *messages*. Each data flow will be split into a sequence of messages to be forwarded within Siphon. A minimal header, with a flow identifier and a sequence number, will be attached to each message. Upon reaching the desired destination aggregator, they will be again reassembled and delivered to the final destination worker.

The aggregators can *forward* the messages to any peer aggregators as an intermediate nexthop or the final destination, depending on the forwarding decisions made by the controller. Inheriting the design in traditional OpenFlow switches, the aggregator looks up a forwarding table that stores all the forwarding rules in a hash table, to ensure high performance. Fortunately, wildcards in forwarding rule matching are also available, thanks to the hierarchical organizations of the flow identifiers. If neither the flow identifier nor the wildcard matches, the aggregator will consult the controller. A forwarding rule includes a nexthop to enforce routing, and a flow weight to enforce flow scheduling decisions.

Since messages forwarded to the same nexthop share the same link, we use a priority queue to buffer all pending outbound messages to support *scheduling* decisions. Priorities are allowed to be assigned to individual flows sharing a queue, when it is backlogged with a fully saturated outbound link. The control plane will be responsible for assigning priorities to each flows.

4.2.2 Performance-Centric Implementation

Since an aggregator is I/O-bounded, it is designed and implemented with performance in mind. It has been implemented in C++ from scratch with the event-driven asynchronous programming paradigm. Several optimizations are adopted to maximize its efficiency.

Event-driven design. events are raised and handled asynchronously, including all network I/O events. All the components are loosely coupled with one another, as each function in these components is only triggered when specific events it is designed to handle are raised. As examples of such an event-driven design, the switch will start forwarding messages in an input queue as soon as the queue raises a `PacketIn` event, and the output queue will be consumed as soon as a corresponding worker TCP connection raises a `DataSent` event, indicating that the outbound link is ready.

Coroutine-based pipeline design pattern. Because an aggregator may communicate with a number of peers at the same time, work conservation must be preserved. In particular, it should avoid head-of-line blocking, where one congested flow may take all resources and slow down other non-congested flows. An intuitive implementation based on input and output queues cannot achieve this goal. To solve this problem, our implementation takes advantage of a utility called "stackful coroutine," which can be considered as a procedure that can be paused and resumed freely, just like a thread whose context switch is controlled explicitly. In an aggregator, each received message is associated with a coroutine, and the total number of active coroutines is bounded for the same flow. This way, we can guarantee that non-congested flows can be served promptly, even coexisting with resource "hogs."

Minimized memory copying. Excessive memory copying is often an important design flaw that affects performance negatively. We used smart pointers and reference counting in our implementation to avoid memory

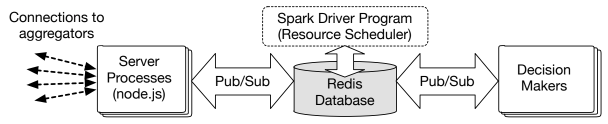

Connections to aggregators
Server Processes (node.js)
Pub/Sub
Spark Driver Program (Resource Scheduler)
Redis Database
Pub/Sub
Decision Makers

Figure 7: The architecture of the Siphon Controller.

copying as messages are forwarded. In the lifetime of a message through an aggregator, it is only copied between the kernel socket buffers for TCP connections and the aggregator's virtual address space. Within the aggregator, a message is always accessed using a smart pointer, and passed between different components by copying the pointer, rather than the data in the message itself.

4.3 Control Plane

The controller in Siphon is designed to make flexible control plane decisions, including flow scheduling and routing.

Although the controller is a logically centralized entity, our design objective is to make it highly scalable, so that it is easy to be deployed on a cluster of machines or VMs when needed. As shown in Fig. 7, the architectural design of the controller completely decouples the decision making processes from the server processes that directly respond to requests from Siphon aggregators, connecting them only with a Redis database server. Should the need arises, the decision making processes, server processes, and the Redis database can be easily distributed across multiple servers or VMs, without incurring additional configuration or management cost.

The Redis database server provides a reliable and high-performance key-value store and a publish/subscribe interface for inter-process communication. It is used to keep all the states within the Siphon datapath, including all the live statistics reported by the aggregators. The publish/subscribe interface allows server processes to communicate with decision-making processes via the Redis database.

The server processes, implemented in `node.js`, directly handle the connections from all Siphon aggregators. These server processes are responsible for parsing all the reports or requests sent from the aggregators, storing the parsed information into the Redis database, and responding to requests with control decisions made by the decision-making processes. It is flexible how the decision-making processes are implemented, depending on requirements of the scheduling algorithm.

In inter-coflow scheduling, the controller requires the full knowledge of a coflow before it starts. This is achieved by integrating the resource scheduler of the data parallel framework to the controller's Pub/Sub interface. Particularly in Spark, the task scheduler running in the driver program have such knowledge as soon as the reduce tasks are scheduled and placed on workers. We have modified the driver program, such that whenever there are new tasks being scheduled, the generated traffic information will be published to the controller. The incremental Monte Carlo simulations will then be triggered on the corresponding parallel decision makers.

5 Performance Evaluation

In this section, we present our results from a comprehensive set of experimental evaluations with Siphon, organized into three parts. First, we provide a coarse-grained comparison to show the application-level performance improvements by using Siphon. A comprehensive set of machine learning workloads is used to evaluate our framework compared with the baseline Spark. Then, we try to answer the question how Siphon expedite a single coflow by putting a simple shuffle under the microscope. Finally, we evaluate our inter-coflow scheduling algorithm, by using the state-of-the-art heuristic as a baseline.

5.1 Macro-Benchmark Tests

Experimental Setup. In this experiment, we run 6 different machine learning workloads on a 160-core cluster, which spans across 5 geographical regions. Performance metrics such as application runtime, stage completion time and shuffle read time are to be evaluated. The shuffle read time is defined as the completion time of the slowest data fetch in a shuffle. It reflects the time needed for the last task to start computing, and it determines the stage completion time to some extent.

The Spark-Siphon cluster. We set up a 160-core, 520 GB-memory Spark cluster. Specifically, 40 `n1-highmem-2` instances are evenly disseminated in 5 Google Cloud datacenter (N. Carolina, Oregon, Belgium, Taiwan, and Tokyo). Each instance provides 2 vCPUs, 13 GB of memory, and a 20 GB SSD of disk storage. Except for one instance in the N. Carolina region works as both Spark master and driver, all instances serve as Spark standalone executors. All instances in use are running Apache Spark 2.1.0.

The Siphon aggregators run on 10 of the executors, 2 in each datacenter. An aggregator is responsible for handling Pub/Sub requests from 4 executors in the same datacenter. The Siphon controller runs on the same instance as the Spark master, in order to minimize the communication overhead between them.

Note that we do not launch extra resources for Siphon aggregators to make the comparison fair. Even though they occupy some computation resource and system I/Os with their co-located Spark executors, the consumption is minimal.

Workload specifications. 6 machine learning workloads, with multiple jobs and multiple stages, are used

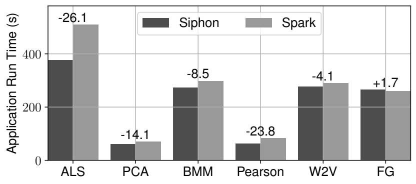
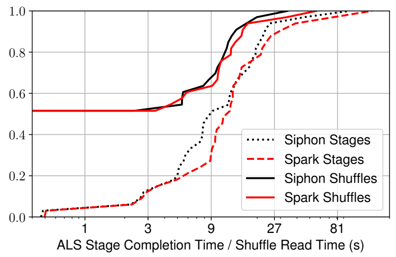


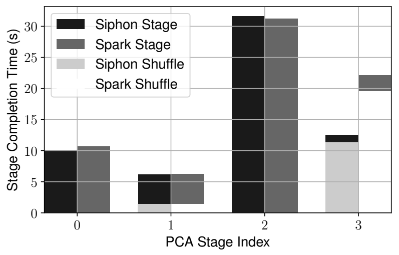
Figure 8: Average application run time.

Workload	# Shuffles	Total Bandwidth Usage(GB)	Extra Bandwidth Usage(MB)	Siphon Shuffle Read Time (s)	Spark Shuffle Read Time (s)	Runtime Reduction (%)	Cost Difference (¢)
ALS	18	40.47	2186.3	46.8	90.5	**48.3**	**-26.56**
PCA	2	0.51	37.6	3.3	13.7	**76.1**	**-6.80**
BMM	1	42.3	2911.1	48.9	97.8	**50.0**	**-29.26**
Pearson	2	0.57	23.8	3.6	13.1	**72.6**	**-6.23**
W2V	5	0.45	10.2	5.8	9.6	**39.9**	**-2.49**
FG	2	0.57	20.5	1.77	1.87	**5.4**	**-0.05**

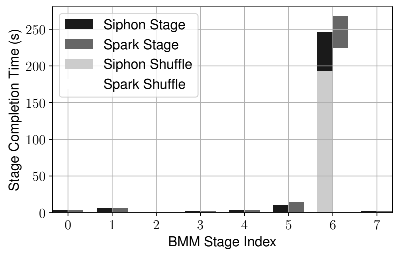
Table 2: Summary of shuffles in different workloads (present the run with median application run time).



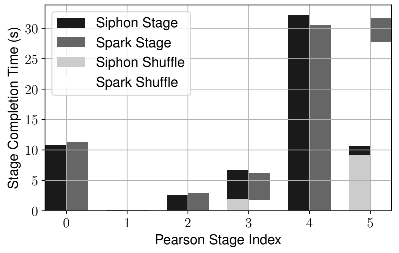
(a) Alternating Least Squares (in CDF).



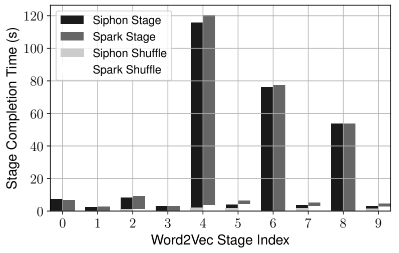
(b) Principal Component Analysis.



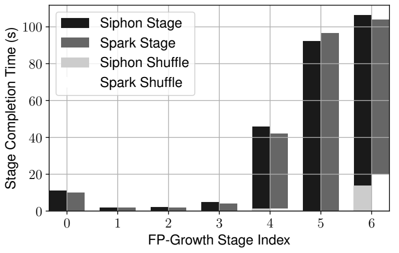
(c) Block Matrix Multiplication.



(d) Pearson's Correlation.



(e) Word2Vec distributed presentation of words.



(f) FP-growth frequent item sets.

Figure 9: Shuffle completion time and stage completion time comparison (present the run with media application run time).

for evaluation.

- *ALS:* Alternating Least Squares.
- *PCA:* Principle Component Analysis.
- *BMM:* Block Matrix Multiplication.
- *Pearson:* Pearson's correlation.
- *W2V:* Word2Vec distributed presentation of words.
- *FG:* FP-Growth frequent item sets.

These workloads are the representative ones from Spark-Perf Benchmark[1], the official Spark performance test suite created by Databricks[2]. The workloads that are not evaluated in this paper share the same characteristics with one or more selected ones, in terms of the network traffic pattern and computation intensiveness. We set the scale factor to 2.0, which is designed for a 160-core, 600 GB-memory cluster.

Methodologies. With different workloads, we compare the performance of job executions, with or without Siphon integrated as its cross-datacenter data transfer service.

Note that, without Siphon, Spark works in the same way as the out-of-box, vanilla Spark, except one slight change on the `TaskScheduler`. Our modification eliminates the randomness in the Spark task placement decisions. In other words, each task in a given workload will be placed on a *fixed* executor across different runs. This way, we can guarantee that the impact of task placement on the performance has been eliminated.

Performance. We run each workload on the same input dataset for 5 times. The average application run time comparisons across 5 runs are shown in Fig. 8. Later we focus on job execution details, taking the run with median application run time for example. Table 2 summarizes the total shuffle size and shuffle read time of each workload. Further, Fig. 9 breaks down the time for network transfers and computation in each stage, providing more insight.

Among the 6 workloads, BMM, the most network-intensive workload, benefits most from Siphon. It enjoys a 23.6% reduction in average application run time. The reason is that it has one huge shuffle — sending more than 40 GB of data in one shot — and Siphon can help significantly. The observation can be proved by Fig. 9(c), which shows that Siphon manages to reduce around 50 seconds of shuffle read time.

Another network-intensive workload is ALS, an iterative workload. The average run time has been reduced by 13.2%. The reason can be easily seen with the information provided in Table 2. During a single run of the application, 40.47 GB of data is shuffled through the network, in 18 stages. Siphon collectively reduces the shuffle time by more than 30 seconds. Fig. 9(a) shows the CDFs of shuffle completion times and stage comple-

[1]`https://github.com/databricks/spark-perf`
[2]`https://databricks.com/`.

tion times, using Spark and Siphon respectively (note the *x*-axis is in log scale). As we observe, the long tail of the stage completion time distribution is reduced because Siphon has significantly improved the performance of all shuffle phases.

The rest of the workloads generate much less shuffled traffic, but their shuffle read time have also been reduced (5.4%$\sim$76.1%).

PCA and Pearson are two workloads that have network-intensive stages. Their shuffle read time constitutes a significant share in some of the stages, but they also have computation intensive stages that dominate the application run time. For these workloads, Siphon greatly impacts the job-level performance, by minimizing the time used for shuffle (Table 2).

W2V and FG are two representative workloads whose computation time dominates the application execution. With these workloads, Siphon can hardly make a difference in terms of application run time, which is mostly decided by the computation stragglers. An extreme example is shown in Fig. 9(e). Even though the shuffle read time has been reduced by 4 seconds (Table 2), the computation stragglers in Stage 4 and Stage 6 will still slow down the application by 0.7% (Fig. 8). Siphon is not designed to accelerate these computation-intensive data analytic applications.

Cost Analysis. As the acceleration of Spark shuffle reads in Siphon is partially due to the relay of traffic through intermediate datacenters, it is concerned how it affects the overall cost for running the data analytics jobs. On the one hand, the relay of traffic increases the total WAN bandwidth usage, which is charged by public cloud providers. On the other hand, the acceleration of jobs reduces the cost for computation resources.

We present the total cost of running the machine learning jobs in Table 2, based on Google Cloud pricing[3]. Each instance used in our experiment costs \$1.184 per hour, and our cluster costs ¢ 0.6578 per second. As a comparison, the inter-datacenter bandwidth only costs 1 cent per GB.

As a result, Siphon actually reduced the total cost of running all workloads (Table 2). On the one hand, a small portion of inter-datacenter traffic has been relayed. On the other hand, the idle time of computing resources has been reduced significantly, which exceeds the extra bandwidth cost.

5.2 Single Coflow Tests

Experimental Setup. In the previous experiment, Siphon works well in terms of speeding up the coflows in complex machine learning workloads. However, one question remains unanswered: how does each component of Siphon contribute to the overall reduction on the coflow completion time? In this experiment, we use a smaller cluster to answer this question by examining a single coflow more closely.

[3]`https://cloud.google.com/products/calculator/`

The cross-datacenter Spark cluster consists of 19 workers and 1 master, spanning 5 datacenters. The Spark master and driver is on a dedicated node in Oregon. The geographical location of worker nodes is shown in Fig. 13, in which the number of executors in different datacenters is shown in the black squares. The same instance type (`n1-highmem-2`) is used.

Most software configurations are the same as the settings used in Sec. 5.1, including the Spark patch. In other words, the cluster still offers a fixed task placement for a given workload.

In order to study the system performance that generates a single coflow, we decided to use the `Sort` application from the HiBench benchmark suite [13]. `Sort` has only two stages, one map stage of sorting input data locally and a reduce stage of sorting after a full shuffle. The only coflow will be triggered at the start of the reduce stage, which is easier to analyze. We prepare the benchmark by generating 2.73 GB of raw input in HDFS. Every datacenter in the experiment stores an arbitrary fraction of the input data without replication, but the distribution of data sizes is skewed.

We compare the shuffle-level performance achieved by the following 4 schemes, with the hope of providing a comprehensive analysis of the contribution of each component of Siphon:

- *Spark:* The vanilla Spark framework, with fixed task placement decisions, as the baseline for comparison.
- *Naive:* Spark using Siphon as its data transfer service, without any flow scheduling or routing decision makers. In this case, messages are scheduled in a round-robin manner, and the inter-datacenter flows are sent directly through the link between the source to the destination aggregators.
- *Multi-path:* The *Naive* scheme with the multi-path routing decision maker enabled in the controller.
- *Siphon:* The complete Siphon evaluated in Sec. 5.1. Both LFGF intra-coflow scheduling and multi-path routing decision makers are enabled.

Job and stage level performance. Fig. 10 illustrates the performance of sort jobs achieved by the 4 schemes aforementioned across 5 runs, with respect to their job completion times, as well as their stage completion times for both map and reduce stages. As we expected, all 3 schemes using Siphon have improved job performance by accelerating the reduce stage, as compared to *Spark*. With *Naive*, the performance improvement is due to a higher throughput achieved by pre-established parallel TCP connections between Siphon aggregators.The im-

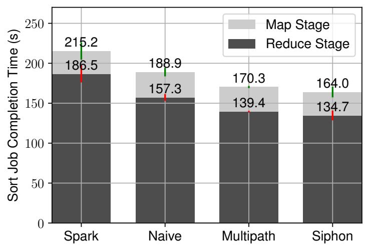


Figure 10: Average job completion time across 5 runs.

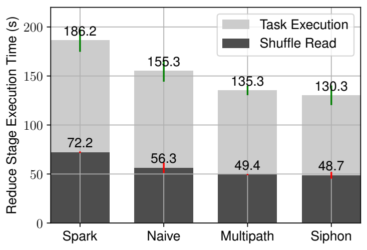


Figure 11: Breakdowns of the reduce stage execution across 5 runs.

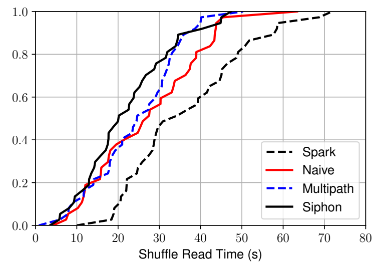


Figure 12: CDF of shuffle read time (present the run with median job completion time).

provement of *Multi-path* over *Naive* is attributed to a further reduction of reduce stage completion times — with multi-path routing, the network load can be better balanced across links to achieve a higher throughput and faster network transfer times. Finally, it is not surprising that *Siphon*, benefiting the advantages of both intra-coflow scheduling and *Multi-path* routing, achieves the best job performance.

To obtain fine-grained insights on the performance improvement, we break down the reduce completion time further into two parts: the shuffle read time (*i.e.,* coflow completion time) and the task execution time. As is shown in Fig. 11, the improvement of *Naive* over *Spark* is mainly attributed to a reduction of the shuffle read time.*Multi-path* achieves a substantial improvement of shuffle read time over *Naive*, since the network transfer completes faster by mitigating the bottleneck through multi-path routing. *Siphon* achieves a similar shuffle read time with *Multi-path*, with a slight reduction in the task execution time. This implies that multi-path routing is the main contributing factor for performance improvement, while intra-coflow scheduling helps marginally on the straggler mitigation as expected.

Shuffle: Spark v.s. Naive. To allow a more in-depth analysis of the performance improvement achieved by the baseline Siphon (*Naive*), we present the CDFs of shuffle read times achieved by *Spark* and *Naive*, respectively, in Fig. 12. Compared with the CDF of *Spark* that exhibits a long tail, all the shuffle read times are reduced by ∼10 s with *Naive*, thanks to the improved throughput achieved by persistent, parallel TCP connections between aggregators.

Shuffle: intra-coflow scheduling and multi-path routing. We further study the effectiveness of the decision makers, with *Multi-path* and *Siphon*'s CDFs presented in Fig. 12.

With multi-path routing enabled, both *Multi-path* and *Siphon* achieve shorter completion times (∼50 s) for their slowest flows respectively, compared to *Naive* (>60 s) with direct routing. Such an improvement is contributed by the improved throughput with a better balanced load across multiple paths. It is also worth noting that the percentage of short completion times achieved with *Multi-path* is smaller than *Naive* — 22% of shuffle reads complete within 18 s with *Multi-path*, while 35% complete with *Naive*. The reason is that by rerouting flows from bottleneck links to lightly loaded ones via their alternative paths, the network load, as well as shuffle read times, will be better balanced.

It is also clearly shown that with LFGF scheduling, the completion time of the slowest shuffle read is almost the same with that achieved by *Multi-path*. This meets our expectation, since the slowest flow will always finish at the same time in spite of the scheduling order, given a fixed amount of network capacity.

We further illustrate the inter-datacenter traffic during the sort job run time in Fig. 13, to intuitively show the advantage of multi-path routing. The sizes of the traffic between each pair of datacenters are shown around the bidirectional arrow line, the thickness of which is proportional to the amount of available bandwidth shown in Table 1.

The narrow link from Taiwan to S. Carolina becomes the bottleneck, which needs to transfer the largest amount of data. With our multi-path routing algorithm, part of the traffic will be rerouted through Oregon. We can observe that the original traffic load along this path is not heavy (only 149 MB from Taiwan to Oregon and 170 MB from Oregon to S. Carolina), and both alternate links have more available bandwidth. This demonstrates that our routing algorithm works effectively in selecting optimal paths to balance loads and alleviate bottlenecks.

5.3 Inter-Coflow Scheduling

In this section, we evaluate the effectiveness of Monte Carlo simulation-based inter-coflow scheduling algorithm, by comparing the average and the 90th-percentile Coflow Completion Time (CCT) with existing heuristics.

Testbed. To make the comparison fair, we set up a testbed on a private cloud, with 3 datacenters located in **V**ictoria, **T**oronto, and **M**ontreal, respectively. We have conducted a long-term bandwidth measurement among them, with more than 1000 samples collected for each link. Their distributions are depicted in Fig. 14, which are further used in the online Monte Carlo simulation.

Benchmark. We use the Facebook benchmark [8] workload, which is a 1-hour coflow trace from 150 work-

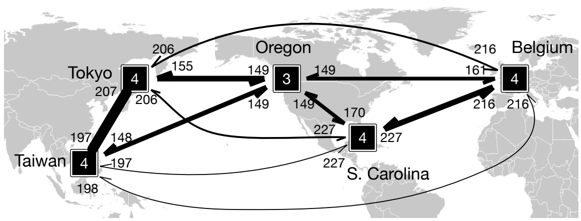


Figure 13: The summary of inter-datacenter traffic in the shuffle phase of the sort application.

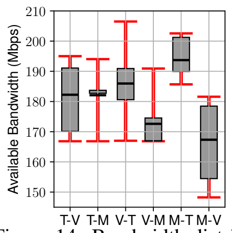


Figure 14: Bandwidth distribution among datacenters.

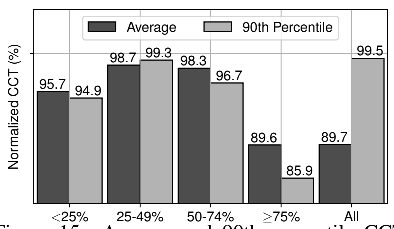


Figure 15: Average and 90th percentile CCT comparison.

ers. We assume workers are evenly distributed in the 3 datacenters, and generate aggregated flows on inter-datacenter links. To avoid overflow, the flow sizes are scaled down, with the average load on inter-datacenter links reduced by 30%.

Methodology. A coflow generator, together with a Siphon aggregator, is deployed in each datacenter. All generated traffic goes through Siphon, which can enforce proper inter-coflow scheduling decisions on inter-datacenter links. As a baseline, we experiment with the Minimum Remaining Time First (MRTF) policy, which is the state-of-the-art heuristic with full coflow knowledge [27]. The metrics CCTs are then normalized to the performance of the baseline algorithm.

Performance. Fig. 15 shows that Monte Carlo simulation-based inter-coflow scheduling outperforms MRTF in terms of both average and tail CCTs. Considering all coflows, the average CCT is reduced by $\sim$10%. Since the coflow size in the workload follows a long-tail distribution, we further categorize coflows in 4 bins, based on the total coflow size. Apparently, the performance gain mostly stems from expediting the largest bin – elephant coflows that can easily overlap with each other. Beyond MRTF, Monte Carlo simulations can carefully study all possible near-term coflow ordering with respect to the unpredictable flow completion times, and enforce a decision that is statistically optimal.

6 Related Work

Wide-Area Data Analytics. For data analytics spanning across datacenters, wide area network links easily become the performance bottleneck. To reduce the usage of inter-datacenter bandwidth, existing works either tweak applications to generate different workloads [12, 16, 23, 24], or assign input datasets and tasks to datacenters optimally [19, 22]. However, all these efforts focus on adding wide-area network awareness to the computation framework, without tackling the lower-level inter-datacenter data transfers directly. Orthogonal and complementary to these efforts, Siphon is designed for the inter-datacenter network optimization — it delivers the inter-datacenter traffic with better efficiency, regardless of the upper-layer decisions on task placement or execution plan.

Software-Defined Networking (SDN). The concept of SDN has been proposed to facilitate the innovation in network control plane [2, 18].In the inter-datacenter wide-area network, SDN has been recently adopted to provide centralized control with elegantly designed traffic engineering strategies [11, 14, 15]. Different from these efforts, our work realizes the SDN principle in the application layer, without requiring hardware support. Moreover, our work focuses on improving performance for data analytics jobs with more complex communication patterns, controlling flows at a finer granularity.

Network Optimization for Data Analytics. Accounting for the job-level semantics, coflow scheduling algorithms (*e.g.*, [6, 8, 9]) are proposed to minimize the average coflow completion time within a datacenter network, which is assumed to be free of congestion. Without such assumptions, joint coflow scheduling and routing strategies [17, 28] are proposed in the datacenter network, where both the core and the edge are congested. Different from these models, the network in the wide area has congested core and congestion-free edge, since the inter-datacenter links have much lower bandwidth than the access links of each datacenter. Apart from the different network model, our coflow scheduling handles the uncertainty of the fluctuating bandwidth in the wide area, while the existing efforts assume that the bandwidth capacities remain unchanged.

7 Concluding Remarks

To address the performance degradation of data analytics deployed across geographically distributed datacenters, we have designed and implemented Siphon — a building block that can be seamlessly integrated with existing data parallel frameworks — to expedite coflow transfers. Following the principles of software-defined networking, a controller implements and enforces several novel coflow scheduling strategies.

To evaluate the effectiveness of Siphon in expediting coflows as well as analytics jobs, we have conducted extensive experiments on real testbeds, with Siphon deployed across geo-distributed datacenters. The results have demonstrated that Siphon can effectively reduce the completion time of a single coflow by up to 76% and improve the average coflow completion time.

References

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

[2] Open Network Foundation Official Website. `https://www.opennetworking.org/`.

[3] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM* (2010).

[4] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal Near-Optimal Datacenter Transport. In *Proc. ACM SIGCOMM* (2013).

[5] BECK, J. C., AND WILSON, N. Proactive Algorithms for Job Shop Scheduling with Probabilistic Durations. *Journal of Artificial Intelligence Research 28* (2007), 183–232.

[6] CHOWDHURY, M., AND STOICA, I. Efficient Coflow Scheduling Without Prior Knowledge. In *Proc. ACM SIGCOMM* (2015).

[7] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M., AND STOICA, I. Managing Data Transfers in Computer Clusters with Orchestra. In *Proc. ACM SIGCOMM* (2011).

[8] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient Coflow Scheduling with Varys. In *Proc. ACM SIGCOMM* (2014).

[9] DOGAR, F. R., KARAGIANNIS, T., BALLANI, H., AND ROWSTRON, A. Decentralized Task-Aware Scheduling for Data Center Networks. In *Proc. ACM SIGCOMM* (2014).

[10] HONG, C. Y., CAESAR, M., AND GODFREY, P. B. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. of ACM SIGCOMM* (2012).

[11] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving High Utilization with Software-Driven WAN. In *Proc. of ACM SIGCOMM* (2013).

[12] HSIEH, K., HARLAP, A., VIJAYKUMAR, N., KONOMIS, D., GANGER, G. R., GIBBONS, P. B., AND MUTLU, O. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[13] HUANG, S., HUANG, J., DAI, J., XIE, T., AND HUANG, B. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *Proc. International Conference on Data Engineering Workshops (ICDEW)* (2010).

[14] 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 *Proc. of ACM SIGCOMM* (2013).

[15] JIN, X., LI, Y., WEI, D., LI, S., GAO, J., XU, L., LI, G., XU, W., AND REXFORD, J. Optimizing Bulk Transfers with Software-Defined Optical WAN. In *Proc. of ACM SIGCOMM* (2016).

[16] KLOUDAS, K., MAMEDE, M., PREGUIÇA, N., AND RODRIGUES, R. Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics. *VLDB Endowment 9*, 2 (2015), 72–83.

[17] LI, Y., JIANG, S. H.-C., TAN, H., ZHANG, C., CHEN, G., ZHOU, J., AND LAU, F. Efficient Online Coflow Routing and Scheduling. In *Proc. ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)* (2016).

[18] 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.

[19] PU, Q., ANANTHANARAYANAN, G., BODIK, P., KANDULA, S., AKELLA, A., BAHL, P., AND STOICA, I. Low Latency Geo-Distributed Data Analytics. In *Proc. ACM SIGCOMM* (2015).

[20] RASLEY, J., STEPHENS, B., DIXON, C., ROZNER, E., FELTER, W., AGARWAL, K., CARTER, J., AND FONSECA, R. Planck: Millisecond-Scale Monitoring and Control for Commodity Networks. In *Proc. of ACM SIGCOMM* (2014).

[21] SUSANTO, H., JIN, H., AND CHEN, K. Stream: Decentralized Opportunistic Inter-Coflow Scheduling for Datacenter Networks. In *Proc. IEEE International Conference on Network Protocols (ICNP)* (2016).

[22] VISWANATHAN, R., ANANTHANARAYANAN, G., AND AKELLA, A. Clarinet: Wan-Aware Optimization for Analytics Queries. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[23] VULIMIRI, A., CURINO, C., GODFREY, P., JUNGBLUT, T., PADHYE, J., AND VARGHESE, G. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[24] VULIMIRI, A., CURINO, C., GODFREY, P., KARANASOS, K., AND VARGHESE, G. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. In *Proc. Conference on Innovative Data Systems Research (CIDR)* (2015).

[25] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

[26] ZATS, D., DAS, T., MOHAN, P., BORTHAKUR, D., AND KATZ, R. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proc. ACM SIGCOMM* (2012).

[27] ZHANG, H., CHEN, L., YI, B., CHEN, K., CHOWDHURY, M., AND GENG, Y. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *Proc. ACM SIGCOMM* (2016).

[28] ZHAO, Y., CHEN, K., BAI, W., YU, M., TIAN, C., GENG, Y., ZHANG, Y., LI, D., AND WANG, S. Rapier: Integrating Routing and Scheduling for Coflow-Aware Data Center Networks. In *Proc. IEEE INFOCOM* (2015).