2018 IEEE International Parallel and Distributed Processing Symposium

Swallow: Joint Online Scheduling and Coflow Compression in Datacenter Networks

Qihua Zhou[†], Peng Li[‡], Kun Wang[†], Deze Zeng[§], Song Guo[*], and Minyi Guo[♮]
[†]*Nanjing University of Posts and Telecommunications, Nanjing, China*
[‡]*The University of Aizu, Japan*
[§]*China University of Geoscience, Wuhan, China*
[*]*The Hong Kong Polytechnic University, Hong Kong, China*
[♮]*Shanghai Jiao Tong University, Shanghai, China*
Email: [†]*kimizqh@foxmail.com,* [‡]*pengli@u-aizu.ac.jp,* [†]*kwang@njupt.edu.cn,*
[§]*deze@cug.edu.cn,* [*]*song.guo@polyu.edu.hk,* [♮]*guo-my@cs.sjtu.edu.cn*

***Abstract*—Big data analytics in datacenters often involves scheduling of data-parallel job, which are bottlenecked by limited bandwidth of datacenter networks. To alleviate the shortage of bandwidth, some existing work has proposed traffic compression to reduce the amount of data transmitted over the network. However, their proposed traffic compression works in a coarse-grained manner at job level, leaving a large optimization space unexplored for further performance improvement. In this paper, we propose a flow-level traffic compression and scheduling system, called Swallow, to accelerate data-intensive applications. Specifically, we target on coflows, which is an elegant abstraction of parallel flows generated by big data jobs. With the objective of minimizing coflow completion time (CCT), we propose a heuristic algorithm called *Fastest-Volume-Disposal-First* (FVDV) and implement Swallow based on Spark. The results of both trace-driven simulations and real experiments show the superiority of our system, over existing algorithms. Swallow can reduce CCT and job completion time (JCT) by up to $1.47\times$ and $1.66\times$ on average, respectively, over the SEBF in Varys, one of the most efficient coflow scheduling algorithms so far. Moreover, with coflow compression, Swallow reduces data traffic by up to 48.41% on average.**

***Keywords*-Big Data; Coflow Scheduling; Traffic Compression; Datacenter Networks.**

I. Introduction

Datacenters run various applications, including query, web service and big data analytics [1, 2]. Although some cluster frameworks (e.g., Hadoop [3] and Spark [4]) have been developed to handle the growing data traffic in data centers, their performance is constrained by network bandwidth that has been identified as the main bottleneck [5–7] of datacenter applications. To conquer this challenge, many network traffic scheduling techniques have been proposed [8–13], but they have achieved only limited success due to the ignorance of job-specific traffic patterns [14]. For example, a MapReduce job is usually divided into multiple tasks, whose generated intermediate data in current a stage are to be shuffled to the ones in the next stage.

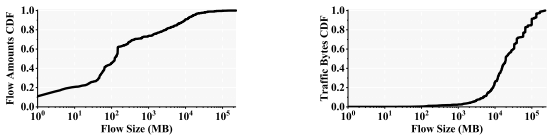
The abstraction of *coflow* [15] has been proposed to improve network performance by exploiting the traffic patterns of big data jobs. A coflow is defined as a set of flows associated with the same computing stage, e.g. the shuffling stage in MapReduce jobs. A coflow completes only when the transmission of all associated flows finish. Coflow has been extensively studied with different objectives under various scenarios [14, 16–20]. Although coflow scheduling is promising, its performance improvement is still bounded by given network bandwidth. Unfortunately, most existing efforts [21–25], which have been paid to increase network bandwidth, either need significant changes of datacenter network topology or incur high hardware cost. In this paper, instead of struggling to bandwidth enhancement, we adopt an alternative approach to reducing network traffic by using idle CPU resources to make data compression.

The idea of compressing network traffic is not new. Many distributed data processing frameworks (e.g., Hadoop and Spark) can compress intermediate data to reduce network traffic. Packet compression functions [26] are also implemented in some switches [27–29]. However, they conduct traffic compression in a coarse-grained manner, i.e., compress all data associated with a job once the compression function is enabled, unaware of flow characteristics and available network bandwidth.

In this paper, we propose Swallow, a system that implements flow-level traffic compression. Specifically, we target on coflows and study how to minimize coflow completion time (CCT) by jointly scheduling traffic compression on CPU and transmission over networks. Swallow's design follows a master-slaver structure. A slaver resides at each computing node, responsible for collecting system information (e.g., coflow sizes, CPU utilization and network bandwidth) and controlling compression and transmission after receiving messages from the master. A master node makes scheduling decisions, i.e., which flow should be compressed and when should it be transmitted. To minimize CCT, we design a heuristic algorithm, called *Fastest-Volume-Disposal-First* (FVDF), with superior performance and low-complexity. We implement Swallow based on Spark-2.2.0 and provide a programming library, so that other frameworks (e.g., Hadoop [30] and Dryad [31]) can also use it through library APIs.

We evaluate the performance of Swallow by both trace-driven simulation and real deployment. We use Hibench [32] as the benchmark. Experimental results show that (1) Swal-

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00060



(a) CDF of flow amounts (b) CDF of traffic bytes

Figure 1: Flow properties.

low outperforms existing scheduling algorithms in Spark by up to $4.33\times$ in terms of minimizing average flow completion time (FCT) [9, 14], (2) Swallow speeds up coflow completion time (CCT) [14, 16] and job completion time (JCT) [5] by up to $1.47\times$ and $1.66\times$ on average, respectively, over the Smallest Effective Bottleneck First (SEBF) algorithm [14] in Varys, respectively, and (3) Swallow reduces data traffic by up to 48.41% on average. The main contributions of this work are summarized as follows:

- To conquer the network bottleneck in running big data jobs, we jointly consider traffic compression and flow scheduling to improve the performance of average CCT.
- We design a fast heuristic algorithm called *Fastest-Volume-Disposal-First* (FVDF) to minimize average CCT while avoiding starvation. It controls compression in a fine granularity without relying on knowledge about future coflow arrivals.
- We implement the proposed algorithm in a system called Swallow based on Spark-2.2.0. Both trace-driven simulation and prototype deployment show that Swallow performs well in both coflow-level and job-level metrics.

The rest of paper is organized as follows. We discuss the background and motivation in Section II. An architectural overview of Swallow is given in Section III. Then, we analyze the coflow scheduling algorithm in Section IV and present the implementation details in Section V. Moreover, the evaluation of Swallow is demonstrated in Section VI. Finally, we draw the conclusion in Section VII.

II. Background and Motivation

A. Flow Properties

Existing works [14, 33, 34] have shown that network flows in datacenters follow a heavy-tailed distribution, in terms of flow amounts and traffic bytes. Our experimental results also confirm this fact. We observe that flows in small size make up major proportion of total amounts and a small number of large flows generate most traffic in datacenter networks. As the Cumulative Distribution Function (CDF) shown in Fig. 1(a), 89.49% flows is smaller than 10 GB and most flows are scattered in length of $[10MB, 10GB]$. Meanwhile, Fig. 1(b) shows that more than 93.03% traffic bytes are created by the flows larger than 10 GB.

Application	Compressed	Uncompressed	Ratio
Wordcount	246,497	440,872	55.91%
Sort	757,621,572	3,034,919,593	24.96%
Terasort	8,713,992,886	31,200,010,752	27.93%
Enhanced DFSIO	354,606	1,868,846	18.97%
Logistic Regression	5,077,091	6,757,608	75.13%
Latent Dirichlet Allocation	515,454	754,677	68.30%
Support Vector Machine	3,368	7,023	47.96%
Bayes	2,153,182	8,176,706	26.33%
Random Forest	815,832	1,194,464	68.30%
Pagerank	27,741,768	65,413,648	42.41%
NWeight	3,814,494	13,168,667	28.97%

Table I: Intermediate data (bytes) of one block in shuffles.

B. Feasibility of Compression

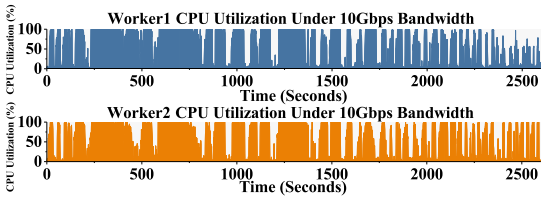
1) Limited bandwidth: Available bandwidth on a machine in realistic datacenters usually ranges from 100 Mbps to 10 Gbps [8–10, 21, 35], which we also adopt in our experiments and motivation example illustrated in this section. Existing works [10–13] show that bandwidth is often a bottleneck for optimizing coflow scheduling, which means there is an improvement bound [6] when the network resource is limited. However, most previous works concentrate on optimizing scheduling priority and enhancing network resource utility, which are subject to the restriction above. Network-level optimization may only reduce JCT at most 2% in median to SQL applications [6].

2) Idle CPU Resources: We measure CPU utilization and show the results in Fig. 2. When the bandwidth is 10 Gbps (Fig. 2(a)), we observe that the CPU is often in low utilization and more than 30.77% of CPU time is wasted. Moreover, when the bandwidth decreases to 100 Mbps (Fig. 2(b)), this phenomenon becomes more prominent — over 69.23% wasted CPU time. When processes are shifting from CPU-bound to I/O-bound, plenty of CPU resources will be left unused. This gap causes the ubiquity of idle CPU periods. Therefore, it is reasonable to make full use of these idle resources and introduce compression techniques to reduce transmission consumption [6].

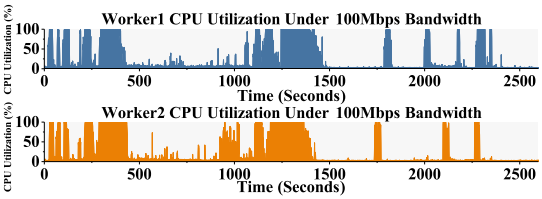
3) Compression in Shuffling: As shown in TABLE I, we collect information of intermediate data generated in the shuffling stages of 11 typical big data jobs. The ratio of compressed and uncompressed data evidences potential compression feasibility in flows.

C. Coflow Scheduling with Compression

As shown in Fig. 3, we abstract the datacenter fabric as a large logic switch that interconnects all machines. This model has been widely adopted by existing works [14, 19]. We consider a number of coflows, each consisting of several flows that are generated during job execution. An motivation example is shown in Fig. 3, where there are two coflows C_1 and C_2, coflow C_1 contains three single flows in size of 4, 4 and 2 data units, respectively, and C_2 has two flows transferring 2 and 3 data units, respectively. Fig. 4 shows the scheduling results of 6 algorithms. Per-Flow Fairness (PFF) [36] guarantees the max-min fairness principle among



(a) Abundant bandwidth with gigabit Ethernet



(b) Limited bandwidth with megabit Ethernet

Figure 2: CPU utilization records show the ubiquity of idle periods displayed in blank areas. Note that idle periods occur frequently in poor network condition.

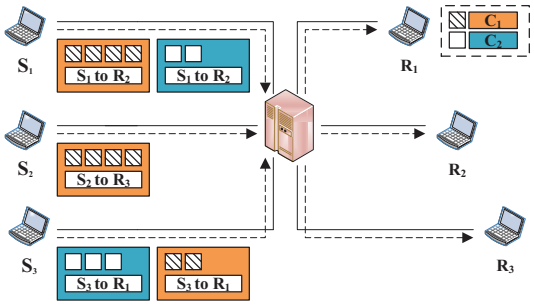
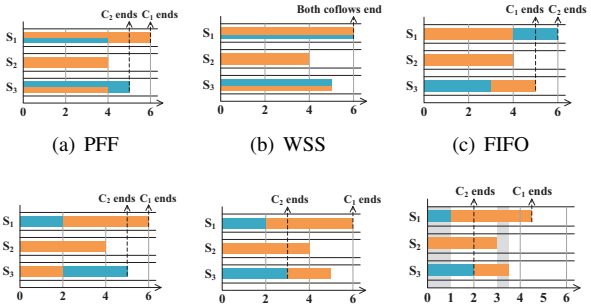


Figure 3: Schedule coflows in a 3×3 datacenter fabric. Five flows are interleaved for network transmission in different channels and two coflows are represented in two graphic patterns — C_1 in orange/light with slashed squares and C_2 in blue/dark with blank squares.



(a) PFF (b) WSS (c) FIFO

(d) PFP (e) SEBF (f) Proposed scheduling

Figure 4: Average FCT and CCT (in time units) achieved by different mechanisms: (a) 4.6 and 5.5 by PFF; (b) 5.2 and 6 by WSS; (c) 4.4 and 5.5 by FIFO; (d) 3.8 and 5.5 by PFP; (e) 4 and 4.5 by SEBF; and (f) 2.8 and 3.25 by our proposed algorithm.

flows, with average CCT of 5.5 time units. Weighted Shuffle Scheduling (WSS) [36] has the same average CCT with PFF but the average FCT is increased. First-In First-Out (FIFO) [8] may lead to *head-of-line* blocking. Per-Flow Prioritization (PFP) — the optimal scheduling algorithm [9, 37] for minimizing average FCT in a single link may not perform well in terms of minimizing average CCT. SEBF is

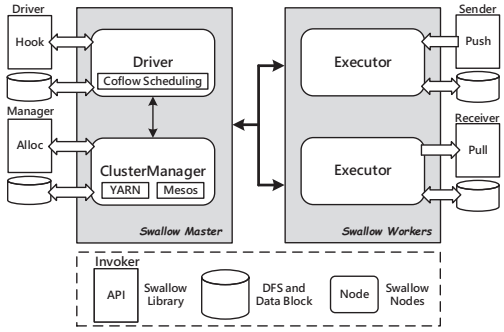


Figure 5: System architecture.

an effective coflow scheduling algorithm [14], with average CCT of 4.5 time units. When traffic compression is adopted, we can further reduce CCT as shown in Fig. 4(f). The shadow areas indicate two durations (time 0–1 and 3–3.5) where CPU is idle. In these durations, there are enough CPU resources for traffic compression, leading to reduced flow size. Suppose the compression ratio of 47.59%, both C_1 and C_2 reduce 2 data units. The average CCT in this case is 3.25 time units.

III. Swallow Architecture

Swallow targets on minimizing CCT by jointly considering coflow scheduling and compression. It is implemented in a *master-worker* structure.

A. Problem Statement

Coflow compression and scheduling is determined by the system environment of compression speed, compression ratio and algorithm selection, which are strongly associated with CPU resources and coflow data. Swallow should decide *whether to compress a coflow, when to start the transmission, and how much bandwidth to be allocated*. Scheduler can compress parts or the entire coflow, enabling or disabling compression process at an arbitrary time point. Additionally, a preemptive scheduling is essential for avoiding *head-of-line* blocking without starvation.

B. Architecture Overview

A Swallow enabled cluster consists of a master node and multiple worker nodes. All workers collaborate with master using Swallow APIs and functionality details of each component can be found in Section V. As shown in Fig. 5, Swallow master aggregates coflow information of flow amounts, flow size, arrival time and required resources for determining compression strategy, scheduling flows and allocating bandwidth. Swallow workers capture the intermediate data when network transmission service is invoked (e.g., shuffle process that Spark transfers RDD stage from *map* to *reduce*). Each executor tackles coflows asynchronously, which means senders and receivers are *time-decoupled*. To avoid thread blocking and polling detection, senders send messages about coflows information to Swallow master when receivers (e.g., reducers) are ready for fetching data. Additionally, Swallow daemons on each machine periodically send measurement messages to the Swallow master, including node status, CPU utilization, bandwidth usage and job situation. The master aggregates this information and makes scheduling decisions.

IV. Algorithm Design

In this section, we first study an offline problem given full knowledge of coflow information, CPU status and compression parameters (speed and ratio). Motivated by its insights, we propose an online scheduling algorithm without prior knowledge.

A. Fastest-Volume-Disposal-First Heuristic

Coflow scheduling can be formulated as a *concurrent open shop problem* [38–40], which is NP-hard [41]. Traditional linear programming (LP) techniques [42, 43] for deriving approximation results do not perform well when coflow compression is taken into consideration. To conquer this challenge, we abstract a coflow scheduling into two stages: *compression* (CPU-bound process) and *transmission* (I/O-bound process). We assume that prior knowledge of coflow information, CPU status and compression parameters (speed and ratio) is known in advance. Coflows can be splitted and it is possible to compress or transmit a part of a flow. Moreover, compression speed can be estimated according to current CPU utilization. We consider $|\mathbb{C}|$ coflows ($\mathbb{C} = \{C_1, C_2, ...C_i, ..., C_{|\mathbb{C}|}\}$) arrive and each coflow C_i consists of $|C_i|$ flows ($C_i = \{f_{i,1}, f_{i,2}, ...f_{i,j}, ..., f_{i,|C_i|}\}$). As the network and CPU utilization dynamically changes, we need to update the resource information and adjust the compression strategy. We divide the entire scheduling process into a series of small basic time units called *slices*. For each flow, there are two stages: *data compression* and *network transmission*. Therefore, the FCT can be calculated as: $FCT = T_c + T_t$, where T_c and T_t are the time consumption of compression and transmission, respectively.

Algorithm	Compression	Decompression	Ratio
LZ4	785 MB/s	2,601 MB/s	62.15%
LZO	424 MB/s	560 MB/s	50.30%
Snappy	327 MB/s	1,075 MB/s	48.19%
LZF	251 MB/s	565 MB/s	48.14%
Zstandard	330 MB/s	930 MB/s	34.77%

Table II: Compression parameters of flows.

1) Volume Disposal: With data compression and network transmission, the remaining size of a flow gradually reduces until it is marked as completed. Scheduling each flow can be regarded as a process about reducing the flow volume. Thus, we propose the abstraction of *volume disposal* to simplify the problem analysis. The volume represents the size of remaining flow, which needs subsequent scheduling. A flow can reduce its volume in two ways: transmission (line 31 in Pseudocode 2) and compression (line 27 in Pseudocode 2). Traditional techniques for optimizing the former meet an improvement bound when the network bandwidth is limited. Fortunately, the latter provides a possibility of further minimizing average FCT.

Our basic idea is to efficiently "slim" the flow volume. Given a time slice δ, more volume should be reduced when compression is enabled. The volume decrement with compression can be defined as:

$$\Delta_c = R \cdot \delta \cdot (1 - \xi), \tag{1}$$

where R represents the compression speed and ξ is the compression ratio. Note that we omit the time consumption of decompression because the decompression is much faster than compression. Compression and decompression speed in some typical algorithms are summarized in TABLE II. The volume decrement without compression is given as:

$$\Delta_t = \min\{B_s, B_r\} \cdot \delta, \tag{2}$$

where B_s and B_r represent the remaining bandwidth of sender and receiver, respectively. Thus, we use the minimum value to describe the essential available bandwidth for transmission. To benefit from compression, condition $\Delta_c > \Delta_t$ should be satisfied. By defining $B = \min\{B_s, B_r\}$, it can be rewritten as:

$$R \cdot (1 - \xi) > B. \tag{3}$$

2) Compression Strategy: Pseudocode 1 describes how to make compression decisions. It returns a binary variable β, indicating the strategy: $\beta = 1$ when compression is enabled and $\beta = 0$ otherwise.

3) Time Calculation: Given the prior knowledge of flow information, CPU status and compression parameters, we can calculate the total completion time $\Gamma_{\mathcal{F}}$ of a specific flow:

$$\min: \ \Gamma_{\mathcal{F}} = n \cdot \delta, \tag{4}$$

$$s.t., \ \sum_{i=1}^{n} (\beta_i \cdot \Delta_c + (1 - \beta_i) \cdot \Delta_t) \geq \mathbb{V}, \tag{5}$$

$$n \in N^+, \tag{6}$$

Pseudocode 1 Compression Strategy

```
1: procedure CompressionStrategy(Flow f)
2:     β = 0;
3:     if f is compatible with compression
4:         if CPU resources are enough for compression
5:             if Eq. (3) is satisfied
6:                 β = 1;
7:     return β;
8: end procedure
```

where $\mathbb{V}$ is the flow volume containing compressed data D and uncompressed raw part d, i.e., $\mathbb{V} = d + D$. Note that i is the index of a time slice, from flow arrival ($i = 1$) to completion ($i = n$).

4) Scheduling Policy: The provable optimal *Shortest-Remaining-Time-First* (SRTF) principle is a natural guideline to minimize average FCT [9, 37]. Therefore, we expect to schedule flows in the *Shortest-*$\Gamma_{\mathcal{F}}$*-First* order. Unfortunately, in online scenarios, it is difficult to obtain CPU utilization in advance. Therefore, we have to figure out expected FCT based on local prediction. The compression will be enabled when it can accelerate the volume disposal more prominently. Scheduling a flow with compression is never slower than the case without compression. Thus, to estimate the expected FCT, we adopt the worst case that compression is perpetually disabled after current time slice is used up. With the compression strategy β (line 15 in Pseudocode 2), an expected FCT is revised as:

$$\Gamma_{\mathcal{F}} = \delta + \frac{(\mathbb{V} - (\beta \cdot \Delta_c + (1 - \beta) \cdot \Delta_t))}{B}. \tag{7}$$

The scheduling preemption only occurs when new flows arrive or existing flows complete. We recalculate Eq. (7) of each flow (line 17 in Pseudocode 2) to select the one with smallest $\Gamma_{\mathcal{F}}$ and schedule it first.

5) From Flow to Coflow: Shifting the scenario from flow to coflow, the minimum CCT ($\Gamma_{\mathcal{C}}$) of a coflow (C) is determined by the slowest flow (f) [14] and can be defined as:

$$\Gamma_{\mathcal{C}} = \max_{f \in C}(\Gamma_{\mathcal{F}}(f)). \tag{8}$$

Based on above analysis, we propose a *Fastest-Volume-Disposal-First* heuristic that schedules coflows in the *Shortest-*$\Gamma_{\mathcal{C}}$*-First* order (line 9 in Pseudocode 2). Moreover, minimum bandwidth is allocated to transmit all flows in time $\Gamma_{\mathcal{C}}$. We can guarantee it by assigning the transmission rate (r) of each flow at $\frac{f.\mathbb{V}}{C.\Gamma_{\mathcal{C}}}$ (lines 29–30 in Pseudocode 2). Note that the expression in format of $obj.var$ represents the operation that getting a variable or invoking a method of an object, similar to the *syntactic sugar* of *getter/setter* in Scala and Java.

Pseudocode 2 Fastest-Volume-Disposal-First

```
1: procedure FVDF(Coflows ℂ, Boolean online)
2:     for each C ∈ ℂ do
3:         Γ_C = TimeCalculation(C);
4:         if online is true then
5:             Γ_C = Γ_C / C.P;
6:         end if
7:         C.Γ_C = Γ_C;
8:     end for
9:     ℂ' = Sort ℂ using Shortest-Γ_C-First;
10:    return The first element of ℂ';
11: end procedure
12: procedure TimeCalculation(Coflow C)
13:    Γ_C = 0;
14:    for each f ∈ C do
15:        β = CompressionStrategy(f) ;
16:        f.β = β;
17:        Calculate Γ_F using Eq. (7);
18:        if Γ_F > Γ_C then
19:            Γ_C = Γ_F;
20:        end if
21:    end for
22:    return Γ_C;
23: end procedure
24: procedure VolumeDisposal(Coflow C)
25:    for each f ∈ C do
26:        if f.β is 1 then
27:            Compress f;
28:        else
29:            r = f.𝕍 / C.Γ_C;
30:            Allocate minimum bandwidth with r;
31:            Transmit f;
32:        end if
33:    end for
34:    Update coflow and resource information;
35: end procedure
```

B. Online Scheduling

1) Compression Parameters: In practice, complete priori knowledge about *network utilization*, *coflow information* and *compression parameters* is unknown. The first two can be collected by *master* node. The last one dynamically changes and is strongly associated with current CPU utilization. In each time slice, we consider available CPU and bandwidth are nearly stable because time slice is a well-designed short interval. It should not be too long, otherwise the predictability of resource information will be lost. Meanwhile, it cannot be too short due to the heavy calculation pressure incurred by frequent rescheduling. Detailed discussion about the length of time slice can be found in Section VI-B3. Additionally, as shown in TABLE III, we observe that the compression ratio

Application Sort Input		10 KB	50 KB	100 KB	1 MB	10 MB	100 MB	1 GB	10 GB
Flow Size	Compressed	8.70 KB	31.85 KB	57.84 KB	416.54 KB	2.76 MB	25.48 MB	253.61 MB	2.51 GB
	Uncompressed	13.09 KB	54.26 KB	102.75 KB	1.01 MB	10.06 MB	100.59 MB	1.01 GB	10.01 GB
Compression Ratio		66.46%	58.70%	56.29%	41.24%	27.44%	25.33%	25.11%	25.07%

Table III: Property of flow compression.

Pseudocode 3 Online Scheduling

```
1:  procedure OnlineScheduling(Coflows ℂ)
2:      while ℂ ≠ ∅ do
3:          Upgrade(ℂ);
4:          C = FVDF(ℂ, true);
5:          for each time slice δ do
6:              if at coflow arrivals or completions then
7:                  ℂ = Get coflows waiting for scheduling;
8:                  Upgrade(ℂ);
9:                  C = FVDF(ℂ, true);
10:             end if
11:             VolumeDisposal(C);
12:         end for
13:     end while
14: end procedure
15: procedure Upgrade(Coflows ℂ)
16:     logbase = 1.2;
17:     for each C ∈ ℂ do
18:         if C.𝒫 = null then
19:             C.𝒫 = 1;
20:         end if
21:         C.𝒫 = C.𝒫 * logbase;
22:     end for
23: end procedure
```

gradually increases as the growth of flow size. Moreover, when the size increases to a certain scale, the compression ratio tends to be a constant. Therefore, we can calculate compression parameters through plenty of experiments and summarize them in TABLE II. Our algorithm conducts the volume disposal of a coflow in each time slice δ (line 11 in Pseudocode 3).

2) Starvation Freedom: Our algorithm is designed in a preemptive form to avoid *head-of-line* blocking [8, 16]. Unfortunately, preemption scheduling may bring starvation in the worst case. As the small coflows keep arriving, scheduler always gives priority to the smallest one and large coflows have to wait persistently. A common solution for avoiding starvation is to employ the principle of fairness (e.g., *lottery* scheduling [44]). However, resources may not be fully used in this approach. Therefore, our solution is to set up priority classes for achieving service fairness in online scenarios. We assign each arrived coflow with a priority class $\mathcal{P}$ to adjust scheduling sequence to meet a tradeoff between *minimizing average CCT* and *maintaining starvation freedom* (lines 4–6 in Pseudocode 2). Moreover,

***SwallowContext* Methods**	**Invoker**
$hook(executor) \Rightarrow Array[flowInfo]$	Driver
$aggregate(Array[flowInfo]) \Rightarrow coflowInfo$	Driver
$add(coflowInfo) \Rightarrow coflowRef$	Driver
$remove(coflowRef)$	Driver
$scheduling(Array[coflowRef]) \Rightarrow schResult$	Driver
$alloc(schResult)$	ClusterManager
$push(coflowRef, blockId, blockData)$	Sender
$pull(coflowRef, blockId) \Rightarrow blockData$	Receiver

Table IV: Swallow API.

we upgrade $\mathcal{P}$ at arrivals and completions of coflows (line 8 in Pseudocode 3) in an exponential-growth speed (line 17–22 in Pseudocode 3). A large coflow which is blocked by the continuously arriving small coflows will gradually increase its priority class $\mathcal{P}$. At first, $\mathcal{P}$ is still approximate to 1 and does not significantly impact $\Gamma_{\mathcal{C}}$. However, when a large coflow has been preempted by small coflows exceeding a certain number of times, the priority class $\mathcal{P}$ will become large rapidly. In this situation, the adjusted $\Gamma_{\mathcal{C}}$ of even a GB-level coflow is in a same magnitude compared with the small coflows. Therefore, large coflows will get service after limited waiting and perpetual starvation is eliminated.

V. Implementation

A. *Swallow* Library: Coflow Scheduling API

Swallow[1] has been implemented in Scala. We employ Akka [45] for message passing and Kryo [46] for object serialization. Swallow provides an option `swallow.smartCompress` to control the coflow compression. Users can enable compression by configuring it to *true*. Three compression algorithms are available: *Sanppy*, *LZ4* and *LZF*. We choose *LZ4* as the default compression algorithm. Cluster frameworks can interact with Swallow by creating a `SwallowContext` object.

We list the main methods of Swallow in TABLE IV. The master node captures the information of each flow on different executors through `hook()`, which returns an array of *flowInfo*. These flow information is merged into *coflowInfo* using `aggregate()`. With *coflowInfo*, we can add a coflow to Swallow with `add()`, which returns a reference handler *coflowRef*. The corresponding `remove()` will be invoked when a coflow is completed. The method `scheduling()` returns scheduling results *schResult*, including the scheduling sequence, compression strategy and resource requirements. Then, the cluster manager (e.g., YARN and Mesos [47]) allocates bandwidth to each executor through `alloc()` and notifies senders to make preparation

[1]https://github.com/kimihe/Swallow.

for data compression and network transmission. A sender initiates *blockData* when it receives the notification from cluster manager and generates a unique *blockId* to represent each block in network transmission. The *blockData* may be a serialized object stored in memory or a temporary file spilled to disks. After above preparation, a sender invokes `push()` to transfer data. If the corresponding receiver has detected the transmission requests, it executes the `pull()` method for data receiving. When the transmission of a flow finishes, the receiver reports *callBack* information to the master. Once all the flows belonging to a coflow finish, this coflow is marked as completed and is removed from the scheduler.

B. Usage Example

The following code snippet written in Scala shows a usage example of Swallow for shuffling — a typical communication pattern in Spark. These codes illustrate how Swallow organizes coflows and tackles the shuffling process.

First, we creates a new object or gets the singleton of existing *SwallowContext*.

```
val sc = new SwallowContext()
val sc = SwallowContext.getInstance()
```

With the handler of *SwallowContext*, we merges all flow information and forwards the aggregated coflow information to scheduler.

```
val flowInfo = sc.hook(executor)
val coflowInfo = sc.aggregate(flowInfo)
val coflowRef = sc.add(coflowInfo)
```

During the shuffling, a task (sender) calls `push()` method for sending data to another task (receiver).

```
for (receiver <- reduceExecutors){
  sc.push(coflowRef, blockId, blockData)}
```

The receiver uses `pull()` method for data fetching.

```
for (sender <- mapExecutors){
  sc.pull(coflowRef, blockId)}
```

Once transmission is done, we terminate the coflow by removing it from scheduler and releasing relevant resources.

```
sc.remove(coflowRef)
```

VI. Evaluation

Swallow is evaluated through both trace-driven simulations and real deployment. The highlights of experimental results are:

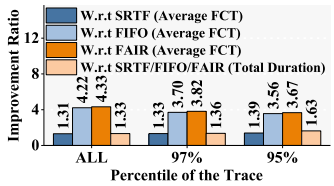
- In terms of minimizing average FCT, Swallow outperforms original algorithms FIFO and FAIR in Spark by up to $4.22\times$ and $4.33\times$, respectively.
- Swallow speeds up CCT and JCT by up to $1.47\times$ and $1.66\times$ on average, respectively, over the SEBF in Varys.
- With coflow compression, Swallow reduces data traffic by up to 48.41% on average.
- Swallow also reduces the garbage collection time and economizes more hardware resources.

A. Trace-Driven Simulation

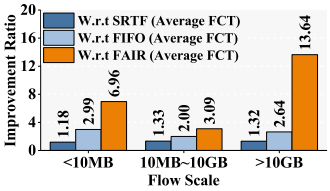
Workloads for trace-driven simulation are generated from the shuffling process in Spark. We collect the trace information of coflow width, coflow size, arrival time, transmission channels, and hardware status to replay the properties of coflows.

1) Flow-level Performance: We compare FVDF with SRTF, FIFO and FAIR in terms of average FCT and JCT. Note that FIFO and FAIR are two primary scheduling algorithms used in Spark. The SRTF is an effective algorithm employed in many task schedulers for minimizing average completion time.

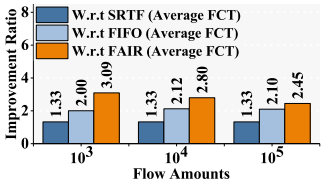
As shown in Fig. 6(a), we consider three sets of traces settings: all flows, 97% flows and 95% flows. The latter two are obtained by filtering out some small flows (e.g., size in kilobyte). FVDF accelerates the scheduling process by up to 1.31, 4.22 and 4.33 times over SRTF, FIFO and FAIR, respectively. With the elimination of small flows, the improvements over FIFO and FAIR decrease lightly. This is caused by the nature that FIFO and FAIR give benefits to large flows, while SRTF prefers to small ones. In Fig. 6(b), results are classified into three categories according to flow sizes. It shows that FVDF brings significant improvements, especially compared to FIFO and FAIR. Note that the improvement over SRTF in large flows is better than that in small ones. This is because FVDF and SRTF schedule the smallest flow first. Moreover, we set the time slice as 10 milliseconds in simulation, which is short enough for most scenarios in realistic production. However, flows from our traces may still in a smaller scale — dozens of kilobytes or several megabytes. When a flow is quite small, the procedures of data compression and network transmission will be completed immediately, e.g., much less than 10 milliseconds. This may produce waste of time slices. Fortunately, most flows are in middle scale and this phenomenon can be alleviated in the real deployment. Fig. 6(c) shows the results when the number of parallel flows is changed. In three different magnitudes, FVDF always outperforms the other three algorithms. Fig. 6(d) compares the CDFs of flow completion time. At the beginning, small flows are scheduled first in SRTF and FVDF. It shows that SRTF outperforms FVDF slightly due to the time-slice waste in FVDF. However, with the completion of small ones, larger flows get the opportunity for dispatching. FVDF gradually exceeds SRTF with the advantages from enabling coflow compression. The superiority of FVDF is outstanding when flows become larger. Finally, over 24.67% accumulated time is saved. Although the average FCT in SRTF is reduced by $3.22\times$ over FIFO and $3.31\times$ over FAIR, the completion time of all flows still gets no improvement. However, FVDF improves this metric by up to $1.33\times$ against SRTF, FIFO and FAIR.



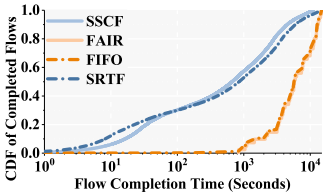
(a) Improvements in average FCT under different percentile of traces



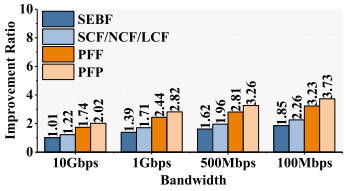
(b) Improvements in average FCT under various flow size



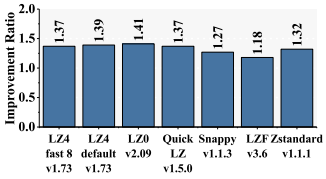
(c) Improvements in average FCT under different numbers of parallel flows



(d) CDF of FCT for a number of algorithms



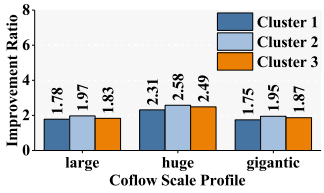
(e) Improvements in CCT under different bandwidth



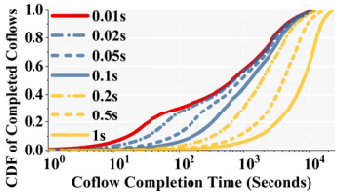
(f) Improvements in SEBF using different compression formats

Figure 6: Comparisons of average FCT and CCT.

(a) Improvements in JCT over different stages



(b) Improvements in traffic reduction in three clusters



(c) CDF of CCT under different lengths of time slice

Figure 7: Results of realistic deployment.

Algorithm	Time Unit 1	Time Unit 2	Time Unit 3	Time Unit 4	Time Unit 5	Time Unit 6	MAX	MIN	AVG
FVDF	5,808	7,203	7,734	7,966	8,141	8,224	2.91	0.04	0.74
FAIR	800	2,600	4,400	5,400	5,800	6,200	0.90	0.2	0.55
FIFO	886	2,636	4,581	5,423	5,894	6,264	0.97	0.19	0.56
SRTF	5,347	6,727	7,410	7,742	7,921	8,007	2.67	0.04	0.71

Table V: Comparison of job throughput.

Algorithm	FVDF	SEBF	SCF/NCF/LCF	PFF/FAIR	PFP
AVG CCT (ms)	79,913	111,809	136,629	195,064	225,296
Job Duration (ms)	639,304	894,472	1,093,032	1,560,512	1,802,368

Table VI: Time consumption of different algorithms.

2) Coflow-level Performance: To evaluate the performance in minimizing CCT, we compare FVDF with different coflow scheduling algorithms, including SEBF proposed in Varys. Moreover, improvements against SEBF under different compression formats are also included. In Fig. 6(e), we show the improvements over six coflow scheduling algorithms under different bandwidths, ranging from 100 Mbps to 10 Gbps. It shows that FVDF outperforms SEBF by up to 1.62× in megabit Ethernet and 1.39× in gigabit Ethernet, respectively. The detailed time consumption of average CCT and job duration are listed in TABLE VI. Note that Swallow will disable compression when bandwidth is suffiecnt because enabling compression will merely aggravate CPU pressure with little improvement in transmission. Therefore, the performance of FVDF is close to that of SEBF under 10 Gbps bandwidth. However, when network condition is poor, the improvements can be up to 1.85×. Moreover, we observe that FVDF outperforms SEBF (Fig. 6(f)) under different compression algorithms [48–53], including LZ4 and Snappy that have been widely used in cluster frameworks. Despite the fact that their compression speed and ratio differ and affect the scheduling performance, FVDF still exceeds SEBF.

3) Job-level Performance: To evaluate the number of job completed per unit time, we measure the job throughput and summarize the results in TABLE V. We assume that each job contains 10 flows. A job is marked as completed when all associated flows finish. Note that the length of a time unit is 2,000 seconds. Six records are under comparison

Workload Scale	With Swallow	Without Swallow	Reduction
large	1,278.6 MB	2.4 GB	46.73%
huge	12.9 GB	25.7 GB	49.81%
gigantic	1.36 TB	2.65 TB	48.68%

Table VII: Data traffic.

Workload Scale	25%	50%	75%	100%
large-c	0.3s/1s	0.4s/2s	0.5s/2s	0.5s/2s
large	0.3s/1s	0.6s/2s	0.6s/2s	0.6s/3s
huge-c	0.2s/10s	0.7s/16s	0.8s/16s	1s/16s
huge	0.5s/17s	0.6s/19s	0.9s/19s	2s/19s
gigantic-c	0.1s/1.4min	0.3s/1.6min	0.6s/1.6min	5s/1.6min
gigantic	0.2s/1.5min	0.3s/1.9min	0.8s/1.9min	8s/19min

Table VIII: Garbage collection time (Map/Reduce).

to show the job throughput in each time unit (indexed from 1 to 6) under different algorithms. Moreover, we summarize the MAX, MIN and AVG values to show the number of completed jobs per seconds. From TABLE V, we can observe that FVDF outperforms other algorithms significantly in terms of extremum and average.

B. Real Deployment

For performance evaluation, we setup a cluster consisting of 100 virtual machines, which are virtualized from 20 physical machines. Each physical machine is equipped with 4×3.1 GHz intel-Xeon CPUs, 28 GB memory, and 8 gigabit Ethernet interfaces. We use Hibench [32] as a benchmark, which contains applications of machine learning, SQL, websearch, graph and streaming. Data storage is based on HDFS/Hadoop. We deploy Swallow in a virtual machine cluster, using Hibench as the benchmark. As shown in TABLE VII, workloads are divided into three categories: large, huge and gigantic based on the input size of jobs, ranging from MB-level to TB-level.

1) Shuffling Performance: We observe that the map and reduce stages in shuffles constitute the major portion of job processing, which strongly impacts the entire JCT. Fig. 7(a) shows that Swallow reduces the completion time of shuffling stage (e.g., shuffle in sort) and result stage (e.g., save output as Hadoop files) by up to $1.90\times$ and $2.12\times$, respectively. The improvement of reducing JCT is up to $1.66\times$ on average. Additionally, Swallow also improves traffic reduction (Fig. 7(b)) in different scales, which means fewer costs of hardware resources.

2) Overheads: To overheads, we mainly focus on data traffic (TABLE VII) and garbage collection time (TABLE VIII).

Traffic Reduction: As shown in TABLE VII, we list three typical workloads and compare the network traffic amount with and without Swallow. Thanks to coflow compression, Swallow can effectively reduce data traffic by 48.41% on average. Specifically, the traffic reduction is up to 49.81% at most in our experiments.

Garbage Collection Time: The application-level garbage collection time also impacts scheduling performance, which reflects the speed of resource release (e.g., buffer free and object deallocation). A machine may lack resources if the garbage collection process is inefficient. In this scenario, an operation system has to frequently execute switching operations (e.g., page replacement in memory swap), which will aggravate the scheduling pressure and degrade the system performance. As shown in TABLE VIII, Swallow performs well in garbage collection. Note that the workloads with suffix *-c* represent the cases enabling coflow compression. Correspondingly workloads without this suffix are those not using compression. And the value in the left of slash is the garbage collection time of a map stage and the value in the right is that of reduce. It shows that the garbage collection time in map and reduce stages both reduces with coflow compression.

3) Impact of Time Slice: The length of a time slice impacts system performance in terms of prediction accuracy and rescheduling pressure. We measure the time slice influence (Fig. 7(c)) on Swallow from $O(10)$ milliseconds to $O(1)$ seconds. With the growth of slice length, the average CCT increases and the corresponding CDF curves gradually evolve into the form of exponential functions, changing from the red solid line (slice in $O(10)$ milliseconds) on left to the yellow solid line (slice in $O(1)$ seconds) on right. The gradient of a CDF curve represents the growth rate of completed coflows at an arbitrary time point. When time slice is in the length of $O(10)$ milliseconds, over 48.63% of coflows are completed after 1000 seconds. However, given the same processing duration, only a few coflows finish in the scale of $O(1)$ seconds. And in the end stage of scheduling, the gradient of the yellow/light solid line rapidly increases, which means most coflows are serviced after a long time waiting. This waiting time leads to the increment of average CCT. In Swallow implementation, we set the time slice in a length of 0.01 seconds by default.

VII. Conclusion

This paper takes a first step to explore the idea of traffic compression in coflow scheduling and proposes a *Fastest-Volume-Disposal-First* heuristic algorithm. We implement our algorithm in a system called Swallow and embed it into Spark-2.2.0. Cluster frameworks can rapidly interact with our system through the API provided by programming library. Both trace-driven simulation and realistic deployment demonstrate the performance superiority of Swallow. The proposed FVDF outperforms native FIFO and FAIR in Spark by up to $4.22\times$ and $4.33\times$ in minimizing average FCT, respectively. Moreover, Swallow speeds up CCT and JCT by up to $1.47\times$ and $1.66\times$ on average, respectively, over the SEBF in Varys. Furthermore, Swallow reduces data traffic by up to 48.41% on average and decreases the garbage collection time in scheduling.

ACKNOWLEDGMENT

This work is supported by NSFC (61572262, 61772480); National China 973 Project (2015CB352401); China Postdoctoral Science Foundation (2017M610252); China Postdoctoral Science Special Foundation (2017T100297); JSPS KAKENHI (16K16038); and Strategic Information and Communications R&D Promotion Programme (SCOPE No.162302008), MIC, Japan.

REFERENCES

[1] X. Zhou, K. Wang, W. Jia, and M. Guo, "Reinforcement learning-based adaptive resource management of differentiated services in geo-distributed data centers," in *Proc. IWQoS*, Vilanova, 2017.
[2] C. Xu, K. Wang, and M. Guo, "Intelligent resource management in blockchain-based cloud datacenters," *IEEE Cloud Computing*, vol. 4, no. 6, pp. 50–59, 2017.
[3] "Apache hadoop." http://hadoop.apache.org.
[4] "Apache spark." https://spark.apache.org.
[5] P. Li, S. Guo, T. Miyazaki, X. Liao, H. Jin, A. Y. Zomaya, and K. Wang, "Traffic-aware geo-distributed big data analytics with predictable job completion time," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1785–1796, 2017.
[6] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun, "Making sense of performance in data analytics frameworks," in *Proc. NSDI*, Oakland, 2015.
[7] H. Ke, P. Li, S. Guo, and M. Guo, "On traffic-aware partition and aggregation in mapreduce for big data applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 818–828, March 2016.
[8] F. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. SIGCOMM*, Chicago, 2014.
[9] C. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proc. SIGCOMM*, New York, 2012.
[10] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. SIGCOMM*, Toronto, 2011.
[11] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. SIGCOMM*, Hong Kong, 2013.
[12] D. Xie, N. Ding, Y. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *Proc. SIGCOMM*, Helsinki, 2012.
[13] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Li, W. Lin, J. Zhou, and L. Zhou, "Optimizing data shuffling in data-parallel computation by understanding user-defined functions," in *Proc. NSDI*, San Jose, 2012.
[14] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proc. SIGCOMM*, Chicago, 2014.
[15] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. HotNets*, Redmond, 2012.
[16] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. SIGCOMM*, London, 2015.
[17] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proc. SIGCOMM*, Florianopolis, 2016.
[18] W. Wang, S. Ma, B. Li, and B. Li, "Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling," in *Proc. INFOCOM*, Atlanta, 2017.
[19] J. Jiang, S. Ma, B. Li, and B. Li, "Adia: Achieving high link utilization with coflow-aware scheduling in data center networks," *IEEE Transactions on Cloud Computing*, pp(99):1-1, 2016.
[20] C. Li, C. Wei, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *Proc. INFOCOM*, San Francisco, 2016.
[21] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. NSDI*, San Jose, 2010.
[22] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proc. SIGCOMM*, Los Angeles, 2017.
[23] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *Proc. SIGCOMM*, Los Angeles, 2017.
[24] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proc. SIGCOMM*, Los Angeles, 2017.
[25] J. Cao, G. Kerr, K. Arya, and G. Cooperman, "Transparent checkpoint-restart over infiniband," in *Proc. HPDC*, Vancouver, 2014.
[26] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker, "Monotasks: Architecting for performance clarity in data analytics frameworks," in *Proc. SOSP*, Shanghai, 2017.
[27] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. IMC*, Chicago, 2009.
[28] S. Ravindra, M. Dayarathna, and S. Jayasena, "Latency aware elastic switching-based stream processing over compressed data streams," in *Proc. ICPE*, L'Aquila, 2017.
[29] R. Park, H. Lee, H. Kim, and W. Lee, "The previewable switch: A light switch with feedforward," in *Proc. DIS*, Vancouver, 2014.
[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. MSST*, Incline Villiage, 2010.
[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. EuroSys*, Lisbon, 2007.
[32] "Intel hibench." https://github.com/intel-hadoop/HiBench.
[33] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. IMC*, Melbourne, 2010.
[34] G. Ananthanarayanan, A. Wang, D. Borthakur, S. Kandula, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *Proc. NSDI*, San Jose, 2012.
[35] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *Proc. INFOCOM*, San Francisco, 2016.
[36] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. SIGCOMM*, Toronto, 2011.
[37] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *Proc. SIGCOMM*, Hong Kong, 2013.
[38] T. Roemer, "A note on the complexity of the concurrent open shop problem," *Journal of Scheduling*, 9(4):389-396, 2006.
[39] J. Leung, H. Li, and M. Pinedo, "Order scheduling in an environment with dedicated resources in parallel," *Journal of Scheduling*, 8(5):355386, 2005.
[40] M. Mastrolilli, M. Queyranne, A. Schulz, O. Svensson, and N. Uhan, "Minimizing the sum of weighted completion times in a concurrent open shop," *Operations Research Letters*, vol. 38, no. 5, pp. 390–395, 2010.
[41] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, 1990.
[42] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc, 1986.
[43] R. Meyer, "A class of nonlinear integer programs solvable by a single linear program," *SIAM Journal on Control and Optimization*, vol. 15, no. 6, pp. 935–946, 1977.
[44] C. Waldspurger and W. Weihl, "Lottery scheduling: Flexible proportional share resource management," in *Proc. OSDI*, Monterey, 1994.
[45] "Akka." http://akka.io.
[46] "Kryo serialization library." https://github.com/EsotericSoftware/kryo.
[47] "Apache mesos." http://mesos.apache.org.
[48] "Lz4 compression library." http://lz4.github.io/lz4.
[49] "Lzo compression library." http://www.oberhumer.com/opensource/lzo.
[50] "Quicklz compression library." http://www.quicklz.com.
[51] "Snappy compression library." https://github.com/google/snappy.
[52] "Lzf compression library." https://github.com/ning/compress.
[53] "Facebook zstandard." http://facebook.github.io/zstd.