2018 IEEE 43rd Conference on Local Computer Networks (LCN)

CLF: An Online Coflow-Aware Packet Scheduling Algorithm

Jie Xiao, Kwan L. Yeung
Dept. of Electrical and Electronic Engineering
The University of Hong Kong
Hong Kong, PRC
{jiexiao, kyeung}@eee.hku.hk

Sugih Jamin
Dept. of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI, USA
jamin@eecs.umich.edu

***Abstract* — Literature on coflow-aware packet scheduling for input-queued switches is limited. Yet most of them are offline algorithms, requiring (unrealistic) a priori knowledge of all coflows and solving (time-consuming) linear programming (LP) problems for determining their expected coflow completion times (CCTs). In this paper, we propose an efficient online packet scheduling algorithm called Critical Line First (CLF). In CLF, coflows are ordered based on their easy-to-find ideal CCTs, or *would-be-CCTs*. In scheduling, coflows with the smallest *would-be-CCTs* are considered first; for each coflow chosen, packets on most heavily loaded rows/columns, i.e., critical lines, of the coflow traffic matrix are scheduled first. To avoid starvation, we propose to limit the number of times a coflow can be preempted by other coflows. Extensive simulation results show that our CLF outperforms all existing algorithms.**

Index Terms*—*Coflow; datacenter network; input-queued switch; online scheduling; packet scheduling.

I. Introduction

Nowadays datacenter involves a plenty of data-parallel computing applications, such as MapReduce [1], Dryad [2], Spark [3], Hadoop [4] and Google Dataflow [5]. In such applications, the computation process is divided into multiple successive stages. Each stage is executed on a group of servers in parallel and produces a large collection of intermediate data pieces for further processing. A stage often cannot start until having completely received all the required data pieces from the previous stage. For example, in the shuffle operation of MapReduce, each reducer needs to pull intermediate data from all mappers. The shuffle phase cannot complete until all the data transmissions from the mappers to the reducers are finished. It is reported [6] that such intermediate communication stages can account for more than 50% of the entire job completion time. Therefore, optimizing the parallel communication/transmission process is important for improving application performance in datacenter.

However, the traditional flow-based resource allocation strategies ignore the fact that typical datacenter application tasks depend on multiple parallel flows. For example, DCTCP [7] considers all flows independently, while targeting at the coexistence of flows requiring small predictable latency and flows requiring large sustained throughput. The recently proposed pFabric [8] focuses on minimizing individual flow completion times (FCTs). With such flow-based strategies, the performance measured at flow level can be ensured/enhanced, but the performance perceived by applications is still limited. To address the issue, the resource allocation process in datacenters should keep job performance in mind, and schedule flows systematically rather than in isolation.

To expose the application-level communication objectives to the datacenter network (DCN), the coflow abstraction is proposed by Chowdhury and Stoica [9]. A coflow is defined as a collection of parallel flows that share the same application semantic and collective performance goal, between two groups of servers. It has the *all-or-nothing* property: a coflow is not completed unless all its component flows are finished. So the coflow completion time (CCT) is determined by its last finished flow, and minimizing CCT speeds up the completion of the corresponding job.

To minimize the average CCT, initial efforts (see Section II.A) abstract the DCN as a giant, non-blocking input-queued switch and scheduling is done by allocating flow rates between servers; we call it *coflow-aware flow scheduling*. Allocated flow rates are enforced by individual servers and thus no modifications to switches/DCN are needed. Although this approach is simple to implement, it is limited in several ways. First, the input-queued *flow* switch abstracted from a fat-tree topology is not nonblocking. Second, flow scheduling may result in unrealistic fractional packets. Third, packet-level performance such as queueing delay cannot be captured. In this paper, we assume switches inside DCN are coflow-aware input-queued switches and we focus on designing efficient *coflow-aware packet scheduling* algorithms. Indeed, the associated scheduling problem belongs to a generalized concurrent open shop problem [10], and has been proved to be strongly NP-hard [11-12]. Although several heuristics (see Section II. B), including a few deterministic approximation algorithms [12-14] with provable performance bound, have been proposed, most of them are offline. They require a priori knowledge of all coflows (including arrival time/size/width/length) and coflows are then ordered by their expected CCTs found using linear programming (LP). Packets are then scheduled from the coflow with the smallest expected CCT first.

Let each coflow be represented by a coflow matrix and entry (i, j) of the matrix denotes the number of packets to be sent from input *i* to output *j*. In each matrix, let the rows/columns with the largest row/column sum be its *critical lines*. In this paper, an efficient online coflow-aware packet scheduling algorithm, called Critical Line First (CLF), is proposed. The key idea is that for a given set of coflows, the one that has the smallest *would-be-CCT* is scheduled first. The *would-be-CCT* of a coflow is the number of packets on the critical line, which is much easier to find than the expected CCT [12-14]. Like [12-14], CLF consists of two parts: ordering and scheduling. In ordering, coflows are ranked in the nondecreasing order of their (current) *would-be-CCTs*. In scheduling, the critical lines of all coflows are scheduled first; then the *noncritical* lines of all coflows are scheduled for maximizing the match size. To avoid starvation, we set a *preemption threshold* to limit the number of times a coflow can be preempted by coflows with smaller *would-be-CCTs*. To improve algorithm efficiency, we reorder the coflows only when a coflow arrives or finishes, and reuse a schedule until a component flow is completed or a coflow arrives.

Our contribution can be summarized as follows: 1) we consider the coflow-aware *packet* scheduling, which is more realistic than *flow* scheduling; 2) our proposed online algorithm is much simpler than all existing algorithms, which require a priori knowledge of all coflows, as well as solving large-scale LPs; and 3) extensive simulations under both model-generated traffic patterns and real Facebook trace show that our CLF always provides the smallest average CCT.

The rest of the paper is organized as follows. Section II reviews the related work. Section III introduces the system model. Section IV describes our CLF algorithm. In Section V, we compare the performance of CLF with other algorithms by simulations. In Section VI, we conclude the paper.

II. Related Work

According to the scheduling granularity, existing work on coflow-aware scheduling can be classified into two types, coflow-aware *flow* scheduling and coflow-aware *packet* scheduling.

A. Coflow-aware flow scheduling algorithms

The initial efforts on coflow-aware scheduling in DCN are conducted via flow rate allocation. They assume that the bottleneck in coflow scheduling is only at the edge of DCN and the entire DCN can be abstracted as a giant nonblocking input-queued switch [11-12]. Under this assumption, the scheduler only needs to allocate/control flow rates at the uplinks and downlinks of servers. As no changes inside the DCN are needed, this approach is fast to deploy. Specifically, Varys [11] adopts a Smallest Effective Bottleneck First (SEBF) heuristic. It sorts coflows in ascending order of their *would-be-CCTs*,[1] and then assigns rates to each coflow to guarantee all its component flows be finished at the same time. If there is any bandwidth left, backfilling will be carried out. A similar policy called Minimum Remaining Time First (MRTF) is developed in D-CAS [15]. When a coflow arrives/finishes, in order to minimize average CCT, all coflows will be reordered based on their remaining *would-be-CCTs*. Due to reordering, ongoing coflows/flows may be interrupted/preempted by coflows with shorter (remaining) *would-be-CCTs*. To eliminate such disturbance, a *non-preemptive* scheduler is proposed in [16] to keep all ongoing flows intact. The non-preemptive scheduler in Barrat [17] is different because it allows flow rate adjustment through *limited multiplexing* -- a local switch mechanism for serving coflows in the FIFO order, yet being able to reduce the head-of-line blocking. In [18], when a new coflow arrives or a coflow completes, rerouting of on-going coflows is not allowed, but rate adjustment is permitted. In [19], max-min fairness among coflows is studied. To facilitate deployment, MinCOF [20] focuses on migrating coflow features to OpenFlow switches. When a priori knowledge of coflows is not available, decentralized coflow-aware schedulers, such as Barrat [17], D-CAS [15] and Aalo [21] can be used. Barrat makes a scheduling decision based on the local/switch view of coflows. D-CAS obtains a global/DCN view through periodic negotiation between senders and receivers. Aalo enhances Barat by having a central controller periodically collect the amount of data sent from each flow sender, combine them and send it back. This gives each sender a global view. Recently, machine learning is studied in CODA [22] to identify coflows. To fully utilize the path diversity in DCN, joint coflow scheduling and routing algorithms, such as RAPIER [23], OMColfow [18], MinCOF [20] and the scheduler in [16], are studied.

Due to the use of flow rate allocation, the coflow-aware flow scheduling algorithms above are limited in several ways. First, abstracting a DCN as a nonblocking input-queued *flow* switch, by assuming bottleneck only occurs at the uplink/downlink of a server, is problematic. Assume a fat-tree based DCN is adopted, which is equivalent to a multi-stage Clos network [24]. But the rearrangeably nonblocking condition [25] for a multirate Clos network cannot be satisfied by fat-tree[2]. That means the input-queued *flow* switch based on fat-tree is inherently blocking. Second, preemptive schedulers can allocate a very small rate to a server for a very short duration, which may require sending of unrealistic, fractional packets. Third, packet-level performance such as queueing delay inside DCN cannot be modeled if flow scheduling is used. Note that when the traffic load is heavy, queueing delay dominates the packet delay-throughput performance, and it becomes more significant under the coflow context. While it is tempting to adapt a coflow-aware flow scheduling algorithm to function in packet mode, coflow-aware scheduling algorithms are generally too complicated for slot-based operation/scheduling.

B. Coflow-aware packet scheduling algorithms

To address the issues with flow scheduling, several coflow-aware *packet* schedulers [12-14, 26] are designed by directly taking packets/slots into account. They assume that each input/output can send/receive at most one packet per time slot. Among them, [12-14] try to minimize the total weighted CCT.

[1] In this case, *would-be-CCT* is obtained by jointly considering the coflow traffic amount and the residual bandwidth.

[2] For a multirate Clos C(*n*, *m*, *r*), it is rearrangeably nonblocking if $m \geq \min\left\{\left\lfloor \frac{7}{4}n + \frac{5r}{8} + \frac{15}{8} \right\rfloor, \left\lfloor \frac{7}{4}n + \frac{r+1}{2} + \frac{1}{4}\min\{n, r\} \right\rfloor\right\}$. But fat-tree is constructed with *m*=*n*.

They share the same approach of ordering coflows first, and then scheduling them one by one. Specifically, Zhen et al. [12] propose the first polynomial-time deterministic approximation algorithm. In [12], an interval-indexed LP (linear programming) is used to find the expected CCTs of all coflows, which are then used for coflow ordering. The coflows whose *would-be-CCTs* fall into the same time interval are combined, treating as a single augmented coflow. This augmented coflow will be scheduled by Brikhoff-von Neumann (BvN) decomposition to optimize its completion time. Both an offline and an online algorithms are designed. And the offline algorithm [12] is proved to have an approximation ratio of 67/3. In [13], Mehrnoosh and Javad improve/reduce this offline approximation ratio to 5 by formulating another LP and scheduling coflows according to a simple *list scheduling* policy. Saba et al. [14] propose a scheduler with the same deterministic approximation ratio as [13]. But coflows are ordered by a primal-dual algorithm and scheduling is based on a modified Hall's theorem. Algorithms in [12-14] are mainly designed for offline operation. As a result, they need to have a priori knowledge of all coflows. When the number of coflows is large, solving the associated LP for expected CCTs becomes the performance bottleneck. Besides, scheduling coflows by either BvN decomposition or Hall's theorem is too complicated.

The work in [26] is perhaps closest to ours. It proposes a coflow-aware batching (CAB) scheduler to achieve the best scaling of CCT when the switch size is large. However, its CCT performance is shown to be even worse than MWM (Maximum Weight Matching) [27]. MWM is a well-known coflow-agnostic scheduler that provides the best delay-throughput performance. (In Section V, MWM is implemented as a benchmark for performance comparison instead of CAB.)

III. System Model

A. Switch model

We consider an $N \times N$ input-queued switch with crossbar switch fabric. Each input/output port can send/receive one packet in each time slot. To avoid head-of-line blocking, each input port maintains N virtual output queues (VOQ) for the N possible destinations/outputs. When a coflow arrives, its

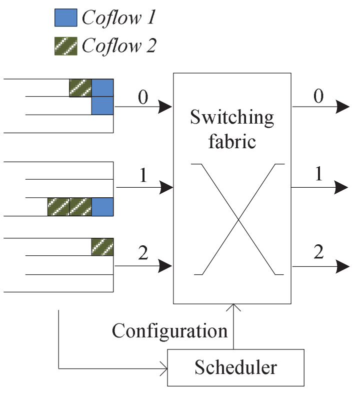


Fig. 1. An 3 × 3 input-queued switch with 2 coflows.

packets will join the corresponding VOQs based on their destinations. As shown in Fig. 1, there are two coflows (*coflow 1* and *coflow 2*) in the switch. *Coflow 1* is destined to outputs 0, 1 and 2. *Coflow 2* is destined to outputs 0 and 2. *Coflow 2* arrives later than *coflow 1*. To support preemptive scheduling, we assume packets in VOQs can be removed in any arbitrary order. A central packet scheduler is adopted to select a subset of packets from different inputs for sending, as well as properly configuring the switch fabric for packet transmission in each time slot.

B. Coflow abstraction

A coflow is a collection of parallel flows working for the same job. Each coflow can be denoted by an integer traffic matrix $C_{(k,t)} = \left[c_{ij}^{(k,t)}\right]$, where k is the coflow ID, t is its arrival time, and entry $c_{ij}^{(k,t)}$ is the number of packets to be sent from input i to output j. When $c_{ij}^{(k,t)} > 0$, $C_{(k,t)}$ has a component flow from i to j. If the switch size is N, each coflow can have at most $N \times N$ parallel component flows. For the two coflows in Fig. 1, if they arrive in t=1 and t=2 separately, they can be denoted as:

$$C_{(1,1)} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \quad C_{(2,2)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 2 \\ 1 & 0 & 0 \end{bmatrix}. \tag{1}$$

Next we define some characteristics [11] of a coflow. The *width* of a coflow is the number of component flows. The *length* of a coflow is the number of packets of the largest component flow. The *size* of a coflow is the total number of packets of all component flows. The three characteristics can be expressed by a 3-tuple <*width*, *length*, *size*>. So for $C_{(1,1)}$ and $C_{(2,2)}$, we have <3, 1, 3> and <3, 2, 4> respectively.

For simplicity, we assume that all packets of a coflow are released simultaneously upon the arrival of this coflow (as in [12-13]). However, our design still holds for the case that the packets of a coflow arrive in different time slots (which may happen in practice [21]).

C. Schedule matrix

At each time slot, the switch fabric is configured according to the schedule generated by a coflow-aware packet scheduler. The schedule indicates which pairs of input and output ports should be connected, and which coflow(s) can send a packet. Let the current slot be t'. For each coflow $C_{(k,t)}$, we define an $N \times N$ schedule matrix $S_{(k,t')} = \left[s_{ij}^{(k,t')}\right]$, where

$$s_{ij}^{(k,t')} = \begin{cases} c_{ij}^{(k,t)} & \text{if } C_{(k,t)} \text{ is scheduled to send} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

If $c_{ij}^{(k,t)} > 0$, coflow $C_{(k,t)}$ will send one packet from input i to output j in the current slot. For a conflict-free transmission, each row/column of $S_{(k,t')}$ can contain at most one nonzero entry. Let $S_{t'} = \left[s_{ij}^{(t')}\right] = \sum_k S_{(k,t')}$ be the *combined* schedule matrix of *all* coflows in slot t', where $s_{ij}^{(t')} = \sum_k s_{ij}^{(k,t')}$. For a conflict-free transmission, each row/column of $S_{t'}$ can contain at most one nonzero entry.

Combining the conflict-free transmission requirements for individual schedule matrices $S_{(k,t')}$ and the combined schedule matrix $S_{t'}$, each nonzero entry in $S_{t'}$ corresponds to one coflow. Without loss of generality, let nonzero entry $s_{ij}^{(t')} = s_{ij}^{(k,t')} = c_{ij}^{(k,t)}$. At slot t', the switch fabric is configured according to $S_{t'}$. For each nonzero entry $s_{ij}^{(t')}$, connect input i to output j, send one packet from $c_{ij}^{(k,t)}$, and update $s_{ij}^{(t')} = c_{ij}^{(k,t)} - 1$.

IV. Critical Line First (CLF) Algorithm

A. Key ideas

Recall that the CCT of a coflow is determined by its last finished flow. Focusing on a single coflow, when all switch ports are of the same speed, the bottleneck of getting it sent is at the most heavily loaded rows/columns of the coflow matrix.

Critical line: the row/column with the largest line sum in the coflow traffic matrix. A coflow can have multiple *critical lines*. The remaining rows/columns in the traffic matrix are *noncritical lines*.

Would-be-CCT: the largest line sum in the traffic matrix, or the total number of packets in a *critical line*. For coflow $C_{(k,t)}$, its *would-be-CCT* is denoted by $m_{(k,t)}$. If a coflow is partially scheduled/sent, its *would-be-CCT* will be updated based on its residual traffic matrix.

According to the following theorem [28], $m_{(k,t)}$ is the minimum time required to schedule $C_{(k,t)}$.

Theorem: The necessary and sufficient number of time slots required to schedule $C_{(k,t)}$ is $m_{(k,t)}$.

In other words, if one packet can be sent from each *critical line* in each time slot, $C_{(k,t)}$ will be completed after $m_{(k,t)}$ slots. Optimal algorithms for scheduling $C_{(k,t)}$ in $m_{(k,t)}$ time slots, such as BvN decomposition, exist but are of high complexity. We thus propose a greedy approach to focus on the *critical lines* only.

<u>Criterion 1:</u> for a coflow, its *critical lines* should be scheduled first.

When a critical line (which can be a row/column) is scheduled, it means one of the nonzero entries in this line is selected for sending. This nonzero entry should be located at the intersection of the critical line and a perpendicular line that has the largest line sum. Let us schedule the *critical line* of $C_{(1,1)}$ as an example. From Fig. 2, $C_{(1,1)}$ has a single *critical line*, or row 0, which has two nonzero entries, $c_{00}^{(1,1)}$ and $c_{01}^{(1,1)}$. As the line

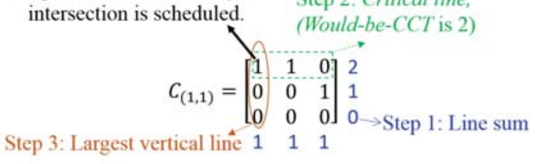

Step 4: The nonzero entry at intersection is scheduled.
Step 2: Critical line, (Would-be-CCT is 2)
$C_{(1,1)} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$
2
1
0
Step 1: Line sum
Step 3: Largest vertical line
1 1 1

Fig. 2. Schedule the critical line of $C_{(1,1)}$.

sums of both columns 0 and 1 are 1, we (sequentially) choose column 0. As a result, packets from entry $c_{00}^{(1,1)}$ are scheduled (i.e., schedule matrix $S_{(1,1)}$ is updated by setting $s_{00}^{(1,1)} = c_{00}^{(1,1)}$); row/input 0 and column/output 0 are marked as occupied. If the scheduling of a *critical line* fails, e.g., the corresponding input/output ports are occupied, this line is marked as considered. If all *critical lines* are occupied/considered, scheduling based on Criterion 1 ends.

<u>Criterion 2:</u> for a coflow already scheduled by Criterion 1, all of its *noncritical lines* should be scheduled based on the principle of largest line sum first.

Scheduling based on Criterion 2 is similar to Criterion 1.

When there is a set of concurrent coflows in the system, how to schedule them to minimize the average CCT is NP-hard [11-12]. But scheduling the coflows with smaller *would-be-CCTs* first can always improve the average CCT. Therefore, we adopt the *smallest would-be-CCT first* (SWF) criterion.

<u>Criterion 3:</u> for a set of coflows, they should be scheduled based on the principle of smallest *would-be-CCT* first.

This criterion is similar to MRTF [15] and SEBF [11]. But our SWF is designed for packet-level scheduling and our *would-be-CCTs* are easier to obtain.

B. Design details

Based on the three scheduling criteria, our CLF algorithm is detailed in Algorithm 1. It has three parts: lines 1~6 detail the coflow ordering process (Criterion 3); lines 7~23 generate a schedule (Criteria 1&2); and lines 24~26 *reuse* the schedule until a new coflow arrives or an ongoing component flow ends. At the end of each time slot, packets from the scheduled coflows will be sent according to the schedule, and the corresponding coflow and schedule matrices will be updated.

According to Criterion 3, coflows are sorted in nondescresing order of their *would-be-CCTs* (line 4). Ties are broken by ranking the oldest coflow first. Since an old coflow with a large *would-be-CCT* may be frequently preempted by new coflows with smaller *would-be-CCTs*, coflow starvation may happen. To avoid starvation, we set a *preemption threshold* *W*. For an ongoing coflow $C_{(k,t)}$, if more than *W* coflows who arrived later than $C_{(k,t)}$ but completed earlier, $C_{(k,t)}$ will be moved to the front of the ordered coflow list (line 5). When *W*=0, coflows are scheduled according to their arrival times. When *W*=∞, coflows are scheduled purely based on their *would-be-CCTs* (and coflow starvation may occur).

If there are no coflow arrivals/departures, the ranking of all existing coflows will remain more or less the same – this is because CLF focuses on making coflows with smaller *would-be-CCTs* even smaller; so the relative order among coflows should remain about the same. As a result, CLF conducts sorting only when a new coflow arrives or an existing coflow is completed (line 1). This modification makes our CLF run much faster, yet without sacrificing performance.

The CCT of a coflow is determined by its *critical lines*. For each coflow, we only need to schedule its *critical lines* as soon as possible, while reserving the remaining switch capacity for the *critical lines* of other coflows. (This is similar to the Minimum-Allocation-for-Desired-Duration (MADD) in [11]). Accordingly, our scheduling is a two-step process: a) schedule all *critical lines* of each coflow based on Criterion 1 (lines 8~15), and b) schedule all *noncritical lines* of each coflow based on Criterion 2 (lines 16~23). In essence, step a) aims more at coflow-level performance (by scheduling *critical lines* only), and step b) aims more at packet-level performance (by sending as many packets as possible). In our implementation, if there is a tie in selecting a line for scheduling, rows will be considered (sequentially) before columns. When an entry is scheduled, the corresponding input and output ports will be occupied. All occupied ports will not be considered in the subsequent iterations (lines 13 and 21) of the current slot.

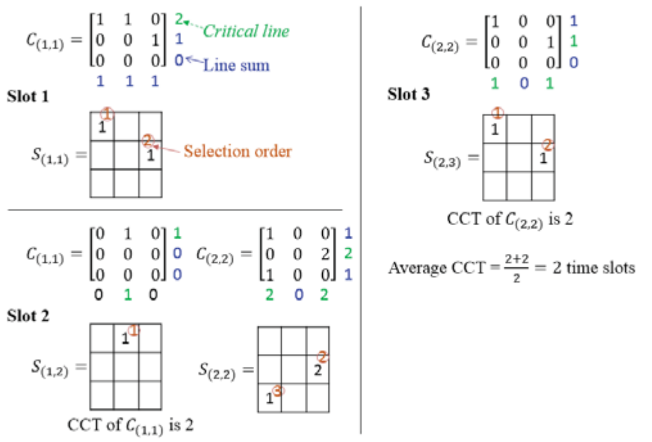
Algorithm 1 Critical Line First (CLF)

```
1:  if a new coflow arrives or an existing coflow is completed: /*Criterion 3*/
2:      Update the coflow list $\mathbb{C} = \{C_{(k,t)}\}$.
3:      Calculate the would-be-CCT for each coflow.
4:      Sort $\mathbb{C}$ based on would-be-CCTs in nondecreasing order.
5:      If a coflow has been preempted by $\geq W$ coflows, move it to the
        front of the coflow list.
6:  end if
7:  initialize schedule matrix $S_{t'} = [\sum_k s_{ij}^{(k,t')}]$ and $t'$ is the current slot.
8:  for $C_{(k,t)}$ in $\mathbb{C}$:    /*Criterion 1*/
9:      do
10:         Find a not-yet-considered critical line. If fail, break.
11:         Schedule an entry in the critical line, say $c_{ij}^{(k,t)}$. If fail, mark
            the line as considered.
12:         Set $s_{ij}^{(k,t')} = c_{ij}^{(k,t)}$, and update $S_{t'}$.
13:         $\forall C_{(k'',t'')} \in \mathbb{C}$, mark its row $i$ and column $j$ as occupied.
14:     while less than $N$ times or $S_{t'}$ is changed
15: end for
16: for $C_{(k,t)}$ in $\mathbb{C}$:    /*Criterion 2*/
17:     do
18:         Find the largest not-yet-considered line. If fail, break.
19:         Schedule an entry in the largest line, say $c_{ij}^{(k,t)}$. If fail, mark the
            line as considered.
20:         Set $s_{ij}^{(k,t')} = c_{ij}^{(k,t)}$, and update $S_{t'}$.
21:         $\forall C_{(k'',t'')} \in \mathbb{C}$, mark its row $i$ and column $j$ as occupied.
22:     while less than $N$ times or $S_{t'}$ is changed
23: end for
24: Do
25:     Reuse $S_{t'}$ in the subsequent time slots (i.e., $t'$, $t'$+1, $t'$+2, …).
26: while a component flow is competed or a new coflow arrives
```

To cut down the scheduling overhead, CLF reuses the schedule for sending in the subsequent time slots until a new coflow arrives or a component flow is completed (lines 24~26). A component flow is completed if the corresponding traffic matrix entry becomes 0. When there are no new coflow arrival, a schedule can be reused for $\min_{i,j,k}\{s_{ij}^{(k,t')} \mid s_{ij}^{(k,t')} > 0\}$ consecutive time slots. When the coflow *length* is long, reusing a schedule can significantly cut down the algorithm execution time.

C. An example

To better understand the scheduling process, we apply CLF to schedule the two coflows, $C_{(1,1)}$ and $C_{(2,2)}$, in Section III.B.



$C_{(1,1)} = \begin{bmatrix}1&1&0\\0&0&1\\0&0&0\end{bmatrix}$ 2 1 0
1 1 1
Critical line
Line sum
Slot 1
$S_{(1,1)}$
Selection order
$C_{(1,1)} = \begin{bmatrix}0&1&0\\0&0&0\\0&0&0\end{bmatrix}$ 1 0 0
0 1 0
$C_{(2,2)} = \begin{bmatrix}1&0&0\\0&0&2\\1&0&0\end{bmatrix}$ 1 2 1
2 0 2
Slot 2
$S_{(1,2)}$
$S_{(2,2)}$
CCT of $C_{(1,1)}$ is 2
$C_{(2,2)} = \begin{bmatrix}1&0&0\\0&0&1\\0&0&0\end{bmatrix}$ 1 1 0
1 0 1
Slot 3
$S_{(2,3)}$
CCT of $C_{(2,2)}$ is 2
Average CCT = $\frac{2+2}{2}$ = 2 time slots

Fig.3. An example of using CLF.

The result is shown in Fig. 3 and is explained below. Assume the system is empty when $C_{(1,1)}$ arrives at slot 1. The row and column sums of $C_{(1,1)}$ are shown in the right/bottom of the matrix. One *critical line* is found and is highlighted in green. According to CLF, this critical line is scheduled first. And we get a schedule entry $s_{00}^{(1,1)} = c_{00}^{(1,1)} = 1$ (zero entries are not shown). Both row 0 and column 0 are marked as occupied. Then we schedule *noncritical lines*. Three *noncritical lines* have the largest line sum of 1: row 1, column 1 and column 2. We first schedule row 1 and get $s_{12}^{(1,1)} = c_{12}^{(1,1)} = 1$. Since no other schedule entry could be added, the final schedule is $S_1 = S_{(1,1)}$ for slot 1. S_1 is then immediately used for sending packets. (S_1 will be used in slot 1 only because its smallest entry is 1.) At slot 2, $C_{(2,2)}$ arrives. $C_{(2,2)}$ and the *updated* $C_{(1,1)}$ are ordered based on their *would-be-CCTs*, $m_{(2,2)} = 2$ and $m_{(1,1)} = 1$. $C_{(1,1)}$ is considered first. Its *critical lines* are scheduled to get $s_{01}^{(1,2)} = c_{01}^{(1,1)} = 1$. Then the three *critical lines* of $C_{(2,2)}$ are considered. We first get $s_{12}^{(2,2)} = c_{12}^{(2,2)} = 2$, and then $s_{20}^{(2,2)} = c_{20}^{(2,2)} = 1$. Since no more schedule entry can be added, the final schedule for slot 2 is $S_2 = S_{(1,2)} + S_{(2,2)}$. S_2 is then immediately used in slot 2 for sending packets. By the end of slot 2, $C_{(1,1)}$ is completed, and its CCT is 2 slots. At slot 3, only $C_{(2,2)}$ is left. By scheduling its *critical lines*, we get the final schedule $S_3 = S_{(2,3)}$. With it, $C_{(2,2)}$ is completed at the end of slot 3 with a CCT of 2 slots. Finally, the average CCT of the two coflows is (2+2)/2=2 slots.

D. Time complexity

If there are M coflows, sorting their *would-be-CCTs* requires $O(M\log M)$. To generate a new schedule, N iterations (for scheduling up to N packets) are required, and each iteration needs at most $2MN$ lines to be compared. So in the worst case, the time complexity of CLF is $O(M\log M + MN^2)$. In online scheduling, M is the number of *concurrent* coflows in the system and is usually small. Since we sort coflows only when a coflow arrives/finishes, the time complexity of CLF without sorting is reduced to $O(MN^2)$.

Table. 1. Comparison of time complexity.

Name	Algorithm overview	Complexity
LP+BvN [12]	Interval-indexed LP & BvN decomposition	Need to solve LP periodically & complexity of BvN is $O(N^3)$.
LP+LS [13]	LP with ordering variables & List scheduling	Need to solve a LP with $\bar{M}^2$ variables, $\bar{M}$ is the number of *all* coflows in the system.
MWM [27]	Maximum weight matching; coflow-agnostic	$O(N^3)$
CLF	Critical line first	$O(MN^2)$, M is the number of *concurrent* coflows.

Table. 1 compares the time complexity of CLF with three other scheduling algorithms. The LP+BvN algorithm [12] needs to solve a LP upon each new coflow arrival. And the BvN decomposition alone has a time complexity of $O(N^3)$, which is often larger than $O(MN^2)$. The LP in the LP+LS algorithm [13] has $\bar{M}^2$ variables, where $\bar{M}$ is the number of *all* coflows in the system (because it is an offline algorithm). Take the Facebook trace [11] as an example, it has 526 coflows. The corresponding LP requires $526^2 = 276676$ variables, which is very time-consuming to solve. With the same Facebook trace, the largest number of concurrent coflows with our CLF (i.e., the largest *M*) is only 10. Note that MWM [27] is a coflow-agnostic algorithm (with a much worse coflow performance than CLF). When the number of concurrent coflows in the system is small, CLF is more efficient than MWM.

V. Performance Evaluations

In this section, we evaluate the CCT performance of CLF under both model-generated traffic patterns and real Facebook trace. Three existing representative algorithm are implemented for performance comparison.

- LP+BvN [12]: It is an *online* algorithm. It divides time horizon into geometrically increasing intervals. A set of binary decision variables are introduced to indicate whether a particular coflow is scheduled to complete within a particular interval. A linear programming (LP) is then formulated based on these variables and some additional scheduling constraints. When a new coflow arrives, the LP is solved to estimate the CCTs of all coflows. Based on them, coflows are ordered. In scheduling, all coflows whose *would-be-CCTs* fall into the same interval are grouped together and treated as a single augmented coflow. This augmented coflow are then scheduled by BvN decomposition. Each schedule holds until a component flow is completed or a new coflow arrives.
- LP+LS [13]: It is an *offline* algorithm. It sheds light on how well an offline algorithm can perform. At *t*=0, the knowledge of all coflows (including future) are assumed to be known. For each pair of coflows, a binary variable is defined to denote which coflow completes first. Based on these variables and some additional scheduling constraints, a LP is formulated to estimate their CCTs. Then all coflows are ranked to form a global list. In the scheduling stage, when a coflow arrives, it becomes active in the global list. Then according to this list, coflows are scheduled. Each schedule holds until a component flow is completed or a new coflow arrives.
- MWM [27]: It is an *online*, *coflow-agnostic* algorithm. It is a well-known, throughput-optimal scheduling algorithm for input-queued packet switches. In our implementation, we set VOQ queue length as the weight. We use the performance of MWM as a benchmark.

We implement the above four algorithms by Python under the Linux system, and solve LPs by CPLEX 12.7. A factor of improvement (FOI) is defined to evaluate the performance gap between a specific algorithm and MWM:

$$\mathrm{FOI} = \frac{\text{Average CCT of MWM}}{\text{Average CCT of a specific algorithm}} \tag{3}$$

According to the simulation results under model-generated traffic (detailed below), on average our CLF improves the average CCT by about 1.13× over LP+BvN, 1.21× over LP+LS and 2.18× over MWM.

A. Simulations under model-generated traffic

In our model-generated traffic, we set switch size *N*=32 and *W*=∞. Coflow arrival follows a Poisson process with average rate of λ coflows/slot. The *width* of each coflow is uniformly distributed between $[1, N^2]$. The sources and destinations of each coflow are chosen randomly. The number of packets of each component flow follows the same geometric distribution, with mean $\bar{\mathrm{L}}$=100 packets. To avoid overloaded input/output ports, the input load at each input port should satisfy the following constraint.

$$\lambda \times \frac{1+N^2}{2N} \times \bar{\mathrm{L}} \leq 1 \tag{4}$$

For each scheduling algorithm at each input load (obtained by varying λ), we run the simulation for 1 million time slots to get the average CCT of all completed coflows. To get the best average CCT performance for CLF, the *preemption threshold W* is set to infinity. Fig. 4 plots the average CCT versus input load. We can see that CLF consistently outperforms the other three algorithms. When the input load is smaller than 0.68, LP+LS

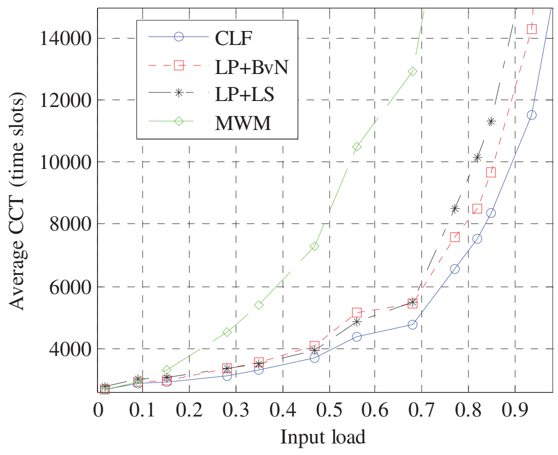


Fig. 4. Average CCT under model-generated traffic.

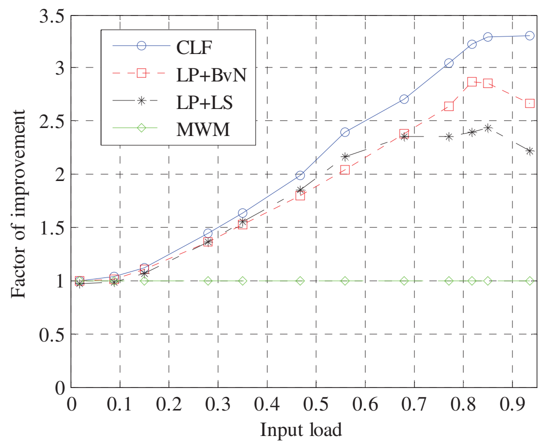


Fig. 5. Factor of improvement under model-generated traffic.

performs close to or slightly better than LP+BvN. However, when the input load exceeds 0.68, LP+BvN outperforms LP+LS. This is because the LP in LP+LS for estimating CCTs becomes less accurate with more coflows. The average CCTs of the four algorithms are comparable when the input load is smaller than 0.1. This is because most of the time there is only one active coflow in the system. As the input load increases, the performance gap between MWM and the other three algorithms grows. This shows that a coflow-aware scheduler is essential.

Based on the average CCTs presented in Fig. 4, we obtain the FOIs for the four algorithms according to (3) and plot them in Fig. 5. We can see that the FOIs of the three coflow-aware scheduling algorithms increases with the load. When the input load is larger than 0.82, the average CCT of the three algorithms increases faster than that of MWM, resulting in a slight drop in FOIs. From Fig. 5, it is clear that CLF consistently gives the best/highest FOI.

To study the starvation-free mechanism in CLF, we compare the performance of CLF using two *preemption thresholds*, or CLF(W=∞) and CLF(W=10). In Fig. 6, the y-axis on the left is the average CCT and that on the right is the *largest preemptive number*, or the number of preemptions experienced by the *most unlucky* coflow. From Fig. 6, we can see that when the input load is smaller than 0.56, CLF(W=∞) and CLF(W=10) have comparable performance of average CCT and the largest preemptive number. But as the input load approaches 1, the largest preemptive number with CLF(W=∞) increases sharply, implying that some unlucky coflows will be starved. On the other hand, the largest preemptive number of CLF(W=10) remains around our target value 10. This is because coflows that have been preempted 10 times will be served with the highest priority. This addresses the coflow starvation problem, though at the cost of a slightly higher average CCT.

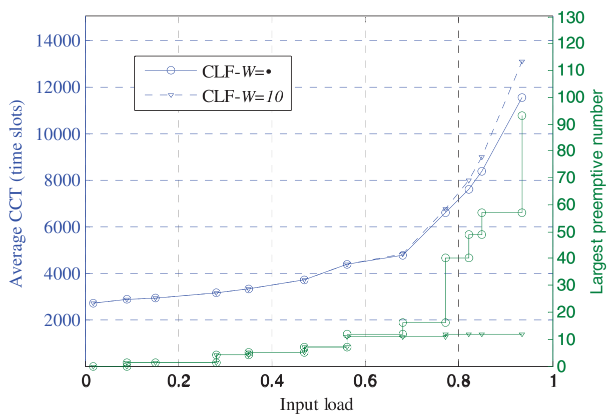


Fig. 6. Starvation-free verification under model-generated traffic.

B. Trace-driven simulation

To gain more insights into CLF's performance, we run a trace-driven simulation based on the Hive/MapReduce trace, collected from a 3000-machine Facebook production MapReduce cluster [11]. The trace contains 526 coflows between 150 racks. Each coflow provides its arrival time, locations of mappers and reducers, and the aggregated shuffle load at each reducer. We assume the shuffle data at each reducer of each coflow is evenly generated from its source mappers with a ±5% perturbation. Since the original cluster has a 10:1 core-to-rack oversubscription ratio and a total bisection bandwidth of 300Gbps, each link/port has a capacity of 1Gbps. In our simulations, we set each time slot to 7.8125ms and W to ∞. Like [12, 13], we filter the coflows based on their *width*. Two cases are considered, with all coflows (*width* ≥ 0) and with only dense coflows (*width* ≥ 50). Note that with *width* ≥ 50, there are only 128 coflows in the system.

Fig. 7 compares the average CCTs of using algorithms CLF, LP+BvN, LP+LS and MWM, with (*width*≥0) and (*width* ≥50), respectively. We can see that CLF always gives the smallest average CCT. Notably, with *width* ≥ 50, the average CCT obtained by each algorithm is higher than its *width* ≥ 0 counterpart. This is because with *width*≥50, we only focus on dense coflows, whose CCTs are often larger. As expected, the coflow-agnostic MWM gives the worst performance in both cases, and CLF outperforms MWM by 17.3× and 9.4×, respectively. Although CLF still outperforms LP+BvN, the performance gap is much smaller than that under model-generated traffic in Fig. 4. This is because the majority (>80%) of coflows in the Facebook trace are narrow/short [11]. For this type of coflows, when they are ranked by CLF using *would-be-*

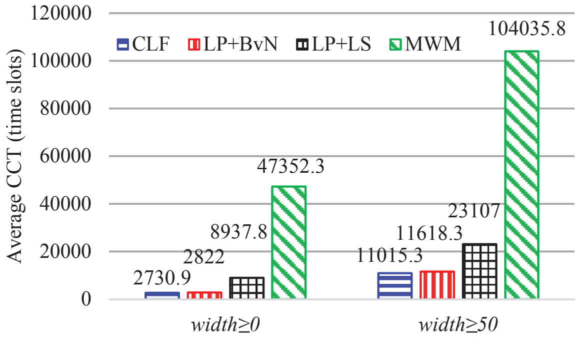


Fig. 7. Average CCT under Facebook trace.

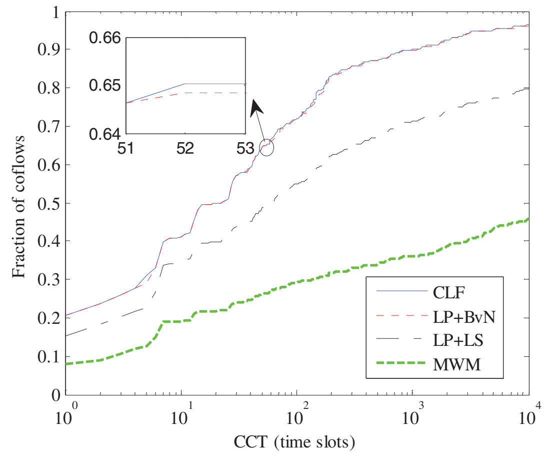


Fig. 8. CCT distribution under *width* ≥ 0.

CCTs or LP+BvN using expected CCTs, similar results will be obtained. This is mainly because both algorithms are online, focusing only on currently active coflows. That said, our CLF runs much faster because ranking in LP+BvN needs to solve a time-consuming LP, and its scheduling is also based on a more complicated Brikhoff-von Neumann decomposition.

Fig. 8 and Fig. 9 present the CDFs of CCTs under *width* ≥ 0 to *width* ≥ 50, respectively. We find that the CDF of CLF is always slightly higher/better than LP+BvN. The CDF of LP+LS is always worse than both CLF and LP+BvN because it ranks coflows only once. MWM gives the worst CDF performance because it is coflow-agnostic. When we set the coflow *width* ≥ 0, we found that more than 20% of coflows have the CCTs of 1, and about 90% of coflows have CCTs smaller than 10000. (Note that the average CCT under this case is 2730.9, as shown in Fig.7.) When we set *width*>=50, there are only 128 dense coflows. The smallest CCT is larger than 10. In this case, more than 80% of coflows have CCTs smaller than 10000 (and the average CCT is 11015.3).

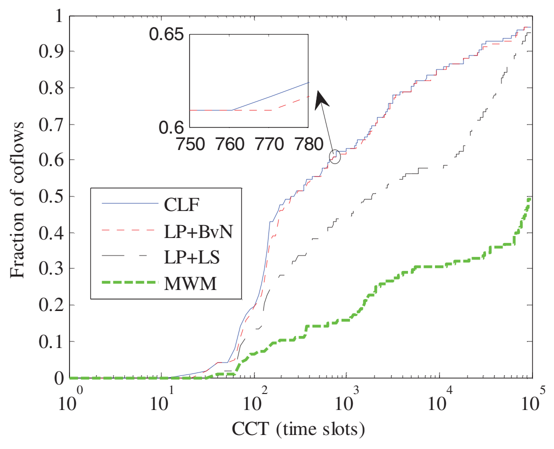


Fig. 9. CCT distribution under *width* ≥ 50.

C. *Impact of coflow types*

The Facebook trace driven simulation above motivates us to study the impact of different coflow types. Based on their *length* and *width*, coflows can be classified into four types: short&narrow (SN), long&narrow (LN), short&wide (SW) and long&wide (LW) [11, 15]. If a coflow is wide, its *width* is uniformly distributed between [N+1, N^2]; otherwise, its *width* is uniformly distributed between [1, N]. If a coflow is long, its *length* is uniformly distributed between [11, 40]; otherwise, its *length* is uniformly distributed between [1, 10]. We consider a switch with size N=64 and based on the same traffic model in Section V.A. In addition to the four pure types of coflows above, we also consider a mixed type, with 52%, 16%, 15% and 17% of all coflows [11] are of types SN, LN, SW and LW. For each case, 500 coflows are generated and the input load is set to 1 (by tuning the coflow arrival rate λ).

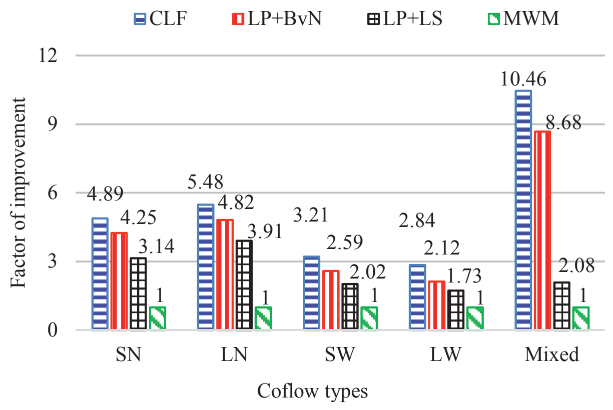


Fig. 10. FOI comparison under various types of coflows

Fig. 10 presents the FOI comparison under 5 types of coflows. We can see that the FOI order is always CLF > LP+BvN > LP+LS > MWM. CLF outperforms LP+BvN because the ordering of coflows (using *would-be-CCTs*) in CLF is more accurate than LP+BvN. We also notice that under the mixed type of traffic, CLF and LP-BvN perform noticeably better than under the four pure types of traffic. This is because on-line algorithms are more responsive to dynamic/mixed traffic. (Note that the result above is consistent with the mixed Facebook trace simulation in Section V.B, where > 80% coflows are SN.) Besides, CLF, LP+BvN and LP+LS outperform MWM under mixed type by 10.46×, 8.68× and 2.08×, respectively. When comparing the four pure types, we find that the FOIs are more sensitive to the *width* of coflows. This is because wide coflows are more likely to be blocked by others.

VI. Conclusions

We proposed an online coflow-aware packet scheduling algorithm, called Critical Line First (CLF), for input-queued switches. Different from the state-of-the-art algorithms, who are designed for offline or needs to solve large-scale LPs, CLF only needs simple operations on the coflow traffic matrix. It sorts coflows based on their *would-be-CCTs* (i.e., largest line sums), and schedules ordered coflows by considering their *critical lines* (i.e., most heavily loaded rows/columns) first. To avoid coflow starvation, a *preemptive threshold* is set. Extensive simulation results show that CLF outperforms all existing algorithms.

REFERENCES

[1] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Commun. ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” SIGOPS Oper. Syst. Rev., vol. 41, no. 3, pp. 59–72, Mar. 2007.

[3] M. Zaharia, et al, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation, pp. 15–28, 2012.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop distributed file system,” in 2010 IEEE 26th symposium on mass storage systems and technologies (MSST), pp. 1–10, 2010.

[5] Google dataflow. https://www.google.com/events/io.

[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, “Managing data transfers in computer clusters with orchestra,” in Proceedings of the ACM SIGCOMM, pp. 98–109, 2011.

[7] M. Alizadeh, et al. "Data center tcp (dctcp)," ACM SIGCOMM computer communication review, vol. 40, no. 4. 2010.

[8] M. Alizadeh et al., “pFabric: Minimal Near-Optimal Datacenter Transport,” Proc. of ACM SIGCOMM, 2013.

[9] M. Chowdhury and I. Stoica, “Coflow: A Networking Abstraction for Cluster Applications,” in ACM Hotnets, 2012.

[10] M. Mastrolilli, M. Queyranne, A. S. Schulz, O. Svensson, and N. A. Uhan, “Minimizing the sum of weighted completion times in a concurrent open shop,” Operations Research Letters, vol. 38, no. 5, pp. 390-395, 2010.

[11] M. Chowdhury, Y. Zhong, and I. Stoica, “Efficient coflow scheduling with varys,” in ACM SIGCOMM Computer Communication Review, vol. 44, no. 4. ACM, pp. 443–454, 2014.

[12] Z. Qiu, C. Stein, and Y. Zhong, “Minimizing the total weighted completion time of coflows in datacenter networks,” in Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures. ACM, pp. 294–303, 2015.

[13] M. Shafiee and J. Ghaderi, "An Improved Bound for Minimizing the Total Weighted Completion Time of Coflows in Datacenters." arXiv preprint arXiv:1704.08357,2017.

[14] S. Ahmadi, S. Khuller, M. Purohit and S. Yang, “On scheduling coflows. In International Conference on Integer Programming and Combinatorial Optimization,” Springer, 2017.

[15] S. Luo, H. Yu, Y. Zhao, B. Wu, and S. Wang, “Minimizing average coflow completion time with decentralized scheduling,” ICC, pp. 307-312, 2015.

[16] R. Yu, G. Xue, X. Zhang and J. Tang, “Non-preemptive coflow scheduling and routing,” Globecom, 2016.

[17] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, “Decentralized task-aware scheduling for data center networks,” ACM SIGCOMM Computer Communication Review, vol. 44, no. 4, pp. 431–442, 2014.

[18] Y. Li, S. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou and F. M, Lau, “Efficient online coflow routing and scheduling,” MobiHoc, 2016.

[19] L. Chen, W. Cui, B.C. Li and B. Li, “Optimizing coflow completion time with utility max-min fairness,” INFOCOMM, 2016.

[20] C.H. Chiu, D.K. Singh, Q. Wang, K. Lee and S.J. Park, “Minimal Coflow Routing and Scheduling in OpenFlow-based Cloud Storage Area Networks,” IEEE 10th International Conference on Cloud Computing (CLOUD), June 2017.

[21] M. Chowdhury and I. Stoica, “Efficient coflow scheduling without prior knowledge,” SIGCOMM, 2015.

[22] H. Zhang, L. Chen, B. Yi, M. Chowdhury and Y. Geng, “CODA: Toward automatically identifying and scheduling coflows in the dark,” SIGCOMM, 2016.

[23] Y. Zhao, et al, “Rapier: Integrating routing and scheduling for coflow-aware data center networks,” INFOCOM, 2015.

[24] C. Clos, “A study of non-blocking switching networks,” The Bell System Technical Journal, vol. 32, no.2, March 1953.

[25] W. Dou, E. Yao, “On rearrangeable multirate three-stage Clos networks,” Theoretical Computer Science, vol. 372, no. 1, pp. 103-107, 2007.

[26] Q. Liang and E. Modiano, “Coflow Scheduling in Input-Queued Switches: Optimal Delay Scaling and Algorithms,” INFOCOM, 2017.

[27] N. McKeown, V. Anantharam, and J. Walrand, “Achieving 100% throughput in an input queued switch,” IEEE INFOCOM, 1996.

[28] Y. Ito, Y. Urano, T. Muratani, and M. Yamaguchi, “Analysis of a switch matrix for an SS/TDMA system,” IEEE Trans. Commun., vol. 63, no. 3, pp. 411–419, 1977.