Distributed In-Network Coflow Scheduling

Paper #745 (1570746474)

***Abstract*—Recently, there has been a growing interest in coflow scheduling due to the rise of data-intensive applications. However, existing solutions rely on modifying hosts to obtain coflow information and cooperatively prioritize their packets. Such a *host-assisted approach* may not work for public data centers and could be problematic when the central controller becomes a bottleneck. In this work, we present PICO, an *in-network coflow scheduling* system allowing a programmable switch to prioritize coflows in a fully distributed way. In the absence of host cooperation, we develop a pairwise coflow detection scheme that clusters sequentially arrived flows. We further design a data plane pipeline to enable fast feature extraction and efficient coflow size tracking for real-time priority adaptation. The experiments show that our sequential coflow grouping achieves an accuracy of up to 99%. The coflows, on average, complete $1.28\times$ faster than per-flow fair sharing, showing the effectiveness of PICO's distributed in-network scheduling even with no hardware modification and software cooperation.**

I. Introduction

With the rise of big data, cluster computing has become widely popular because of its low cost and high scalability for processing massive and complex computing tasks. Distributed computing applications can be deployed in data centers formed by a group of independent machines linked together by networks. With such a flexible and adaptive architecture, computing power is no longer a bottleneck; instead, the efficiency of communications between distributed machines becomes increasingly crucial to various cloud applications. Several efforts [1]–[3] have been made to enhance communication efficiency for cluster computing. A recent work [4] further proposes a new concept, called *coflow scheduling*, to collectively optimize the transmission efficiency of a *coflow*, i.e., a group of dependent parallel flows.

A *coflow* typically shares a common performance goal since it finishes only after all the dependent flows have been completed. Several recent studies [5]–[11] have empirically proved that *smallest-first scheduling* is an effective algorithm that significantly reduces the average *coflow completion time* (CCT), i.e., the completion time of a collection of tasks belonging to the same job. However, existing solutions mostly rely on a centralized server that collects (or predicts) the coflow information and performs *global scheduling*. Such centralized scheduling not only incurs high communication overhead and computational cost but also may suffer from a single point of failure. More importantly, existing schedulers usually require modifications of the applications or end hosts such that they can follow the schedule and prioritize their flows accordingly.

Such *host-assisted centralized coflow scheduling* may be possible for a private cloud but is not applicable for a public cloud, where the end hosts are not fully controlled by the central server. To resolve this issue, this paper presents PICO (Packet-In Coflow-Out), an *in-network coflow scheduling* system that schedules coflows without involving the cooperation of hosts. We achieve this goal by enabling each programmable switch to automatically prioritize incoming coflows in a distributed manner. Hence, a host can transmit as usual without reporting any information and reordering its packets. In the absence of host cooperation, in-network scheduling, however, faces two difficulties. First, unlike a centralized scheduler that uses full information to cluster all the flows as coflows, in-network scheduling receives parallel flows of a coflow at different times and, hence, cannot perform global clustering. Second, a flow should be prioritized immediately after it arrives. Without coflow information, a switch should fast extract representative features required for prompt scheduling in an early stage.

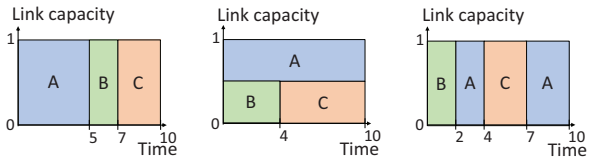
To cope with the above challenges, we propose a novel *sequential flow grouping* algorithm based on pairwise coflow detection. In particular, instead of clustering all the flows globally, PICO's switch predicts the likelihood of a flow being the same coflow as any other flow that has arrived by now. It then *sequentially* groups every new flow with existing flows that are most likely to belong to the same coflow. PICO then integrates such *sequential grouping* with a *sketch-based* fast feature extraction scheme to enable distributed coflow scheduling. We implement a prototype of PICO with a P4 Edgecore switch and perform large-scale trace-driven simulations. The experiments show that PICO's sequential flow grouping achieves an accuracy of up to 99%. On average, PICO's CCT can be improved by $1.28\times$ as compared to *per-flow fairness* and is only 14.76% longer than the host-assisted centralized approach, showing the effectiveness of in-network scheduling with no modification of host hardware and application software.

The rest of the paper is organized as follows. Section II summarizes previous studies on coflow scheduling. The background and motivation are discussed in Section III. We introduce the overview of PICO in Section IV and detail the system architecture in Section V. Section VI evaluates the performance of in-network scheduling. Finally, Section VII concludes this work.

II. Related Work

This section summarizes related work on flow scheduling.

Flow scheduling. Existing data center transport protocols are usually based on *processor sharing* (PS) and *shortest remaining processing time* (SRPT). The former, such as RCP [12], DCTCP [13], HULL [14], XCP [15] and CUBIC [16], divides bandwidth equally but cannot reach optimal flow completion



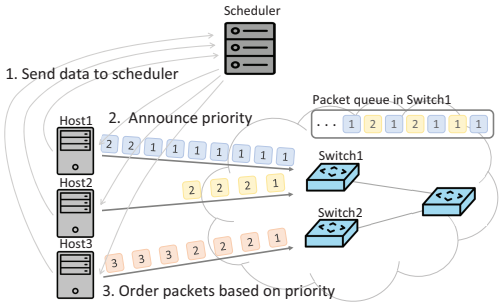
(a) FIFO (b) Fair-sharing (c) Shortest first

Fig. 1: Comparisons among various coflow scheduling algorithms

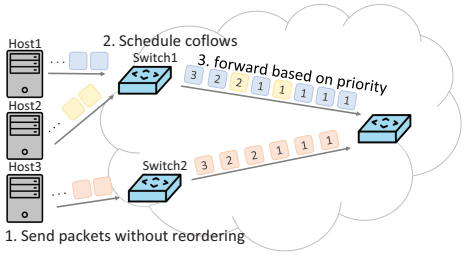
time. The latter, such as PDQ [1], pFabric [3] and L^{2}DCT [17], prioritizes flows in ascending order of remaining processing time. Though these mechanisms provide a better performance, they may starve long flows. The work [18] proposes the concept of *expansion ratio*, which is a metric making the tradeoff between SRPT and PS, to avoid starvation.

Information-aware coflow scheduling. Many recent proposals assume that the coflow information, such as flow identifiers and sizes, is known in priori. One of the early solutions, Orchestra [5], implements first-in-first-out (FIFO) coflow scheduling. Varys [6] is a smallest-effective-bottleneck-first (SEBF) heuristic that improves the work [5] to ensure guaranteed coflow completion time. RAPIER [8] and OMCoflow [9] leverage the information about coflows and the network topology to identify optimal routing and scheduling policies. RAPIER [8] formulates coflow scheduling as an optimization problem with the bandwidth constraints and proposes a shortest-job-first algorithm to minimize coflow completion time (CCT). It, however, may cause flows to be frequently rerouted. OMCoflow [9] addresses the rerouting problem by jointly solving coflow scheduling and coflow routing.

Information-agnostic coflow scheduling. Information-agnostic algorithms schedule coflows with no or only partial coflow information. Aalo [7] assumes that the coflow identifiers of each flow are given but the coflow size information is unknown. It then proposes a discretized coflow-aware least-attained-service (D-CLAS) algorithm to schedule coflows merely based on the currently received size of all the flows. CODA [10] does not require both the coflow identifier and size information. It also adopts D-CLAS but exploits machine learning techniques to cluster flows as coflows for scheduling. Zhang et al. [11] argue that different coflows or applications are not equally important and propose to take emergency levels into consideration. All the above centralized approaches still require a central server to collect *flow information* from the hosts or applications for global clustering and scheduling. Such a centralized scheme, however, incurs expensive data collection costs and may have to omit tiny coflows to reduce the complexity. Baraat [19] is a host-assisted decentralized protocol that allows switches to embed the coflow statistics in the packet metadata. Each destination host retrieves such information and feedbacks it to its source host for adjusting the sending rate accordingly. Different from the above *host-assisted schedulers*, we aim at developing *host-transparent fully distributed in-network coflow scheduling*.



(a) Host-assisted scheduling



(b) In-network scheduling

Fig. 2: Host-assisted scheduling vs. in-network scheduling

III. Background and Motivation

We start by giving background about coflow scheduling and then show some motivating examples for in-network scheduling.

Primer for Coflow Scheduling. Today's data center applications handle a large number of complex computing jobs, each of which is split into tasks processed in a parallel manner. The tasks belonging to the same job may be executed across a group of machines (workers), which cooperatively complete the job and feedback the results to a master for aggregation. The parallel flows to (and from) the cooperative workers typically share a common application-level performance objective. The concept of *coflow* [4], i.e., a collection of parallel flows, is recently proposed to capture their collective behavior. Since a coflow cannot finish until its last flow finishes, *Coflow Completion Time* (CCT) is defined as the completion time of the last flow of a coflow.

Several coflow scheduling algorithms [5]–[11], [19] have been proposed to prioritize parallel flows so as to minimize the CCT of competing coflows. Baraat [19] indicates that an ideal scheduling policy should be able to adapt to various coflow size distributions, which may change dynamically depending on the application requirements. Due to the collective behavior, conventional flow scheduling algorithms, such as *FIFO* (first-in-first-out) and *per-flow fair-sharing*, fail to minimize the CCT. *FIFO*, as shown in Fig. 1(a), is the optimal scheduling algorithm for the light-tailed distribution but suffers from head-of-line blocking if coflows follow a heavy-tailed distribution. *Per-flow fair-sharing*, as shown in Fig. 1(b), ensures max-min fairness among flows, which, however, leads to a long CCT. Recent studies [6], [7], [10] have indicated that, compared to conventional independent flow schedulers, the *shortest-first scheduling* algorithm in Fig. 1(c), which assigns the shortest

coflow the highest priority, can mostly reach the minimum average CCT.

Motivation. While prior efforts have shown promising results on CCT reduction, most solutions require a controller to collect the traffic information from workers (hosts) and assign coflow priority in a centralized manner. The server then requests the hosts to prioritize their packets based on the schedule, as shown in Fig. 2(a). Such *host-assisted centralized coflow scheduling*, though minimizing the CCT, has several limitations. First, it incurs expensive data collection overhead and is vulnerable to errors when the single controller becomes a bottleneck. Second, it needs to modify every host and application for explicit packet ordering, which may not be applicable for public data centers, where end-users do not cooperate with the cloud computing infrastructure, e.g., Amazon AWS or Microsoft Azure. Finally, the available bandwidth from each host to its ingress switch may differ due to heterogeneous and dynamic background traffic demands. In this case, the packet arrival order may violate the schedule, as shown in Fig. 2(a). To overcome those limitations, this paper aims at developing a distributed *in-network coflow scheduling* system, as demonstrated in Fig. 2(b), without the assistance of a controller and any hosts.

Owing to the emergence of software-defined networking, various programmable switching architectures, e.g., P4 (Programming protocol packet processors) [20], have been developed to enable flexible and custom match-action pipelines. The local controller of a programmable switch can also perform simple computation and learning tasks, which make *in-network intelligence at run-time* available. In this work, we ask whether it is possible to exploit *programmable ingress switches* to *schedule coflows* and *prioritize the packet forwarding order* without involving any controller and modifying hosts. By enabling in-network scheduling, each ingress switch should be able to identify parallel flows and promptly predict the coflow sizes for timely scheduling. While ingress switches operate independently in a distributed manner, in-network scheduling prioritizes coflows without expensive coordination overhead.

Challenges. In the absence of host cooperation, in-network scheduling faces some practical challenges.

i) Local coflow grouping: First, how can an ingress switch distinct whether arrival flows belong to the same coflow? Existing host-assisted schedulers usually collect the flow information, e.g., coflow identifiers and (or) sizes, from the hosts and cluster the flows globally as coflows. This is, however, not applicable for in-network scheduling because no cooperation is allowed. Note that a coflow can only be scheduled until its flows are identified by a switch. Hence, the efficacy of a local scheduler relies on how fast an ingress switch properly groups unknown parallel flows as a coflow. Moreover, the computing power of a programmable switch is fairly limited. It is impractical to perform frequent global clustering for every new flow. The problem is even more challenging as a switch has to identify the coflow identifier of every flow in a very early stage based on little information.

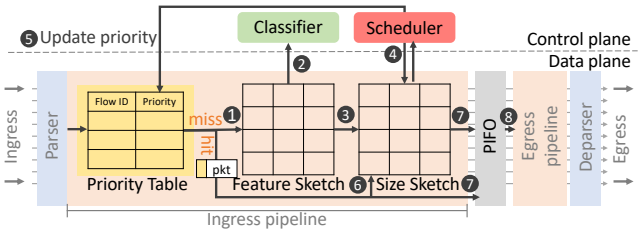


Fig. 3: Architecture of PICO

ii) Real-time coflow feature extraction and scheduling: Without coflow information given by the hosts, each ingress switch should extract representative features from the initial packets of a flow to enable fast coflow identification. In addition, without the coflow size information, it is not possible for a switch to perform *shortest-first scheduling* real-time under random flow arrivals. Instead, PICO adopts the D-CLAS algorithm [7], which schedules ongoing coflows simply based on the number of bits per coflow that have been received so far. Implementing D-CLAS in a switch with limited resources is however not an easy task. PICO needs to track the coflow sizes *on the fly* so as to adapt the coflow priority dynamically to their real-time workload.

IV. Overview

To resolve the challenges mentioned above, we present PICO, a fully decentralized in-network coflow scheduling system that enables each ingress switch to schedule coflows in a distributed manner. Since PICO does not modify hosts and applications, packets are delivered from a host to its ingress switch based on standard fair share scheduling or any other default algorithm. To enable in-network scheduling, each PICO switch runs a sequential flow grouping algorithm based on *pairwise coflow detection* and enables real-time feature extraction and coflow monitoring. Fig. 3 illustrates the architecture of PICO. In each ingress switch, the control plane runs a coflow classifier and priority scheduler, while the data plane deploys a pipeline architecture with four stages: *priority matching table*, *feature sketch*, *size sketch*, and *PIFO* [21]. The four stages, respectively, are used to match the priority of each flow, collect flow features, online track the accumulative coflow sizes, and order packets.

Each entry of the priority table records the identifier (ID) of each flow and its assigned priority. When a new flow fails to match its flow ID ①, it means that the classifier has not detected the coflow ID of this flow yet. For those unknown flows, we temporarily assign its packets the highest priority and tag this initial priority in the packet metadata. Then, the packets are redirected to the feature sketch, which extracts from the first few packets of a flow the information required for coflow classification. When the features are collected, the switch actively forwards them to the local controller ②, which performs *pairwise coflow detection* (see Section V-A). The packets finally go through the size sketch that keeps tracking the flow size ③. PICO's scheduler periodically queries the size sketch to estimate the coflow sizes and updates the coflow priority accordingly ④ (see Section V-B). When the priority

of a flow is updated, the scheduler writes the assigned priority into the priority matching table ⑤.

The later packets of a flow that successfully match the priority table are tagged with the assigned priority in their metadata. Those packets bypass the feature sketch ⑥ but should still go through the size sketch for continuous monitoring ⑦. Finally, all the packets are inserted into PIFO (Push-In First-Out) [21], a priority queue data structure proposed for a programmable switch to insert packets into priority queues in an order specified in the metadata ⑧. PIFO dequeues the packets in order, allowing the switch to forward the prioritized coflows in a distributed way.

V. PICO Design

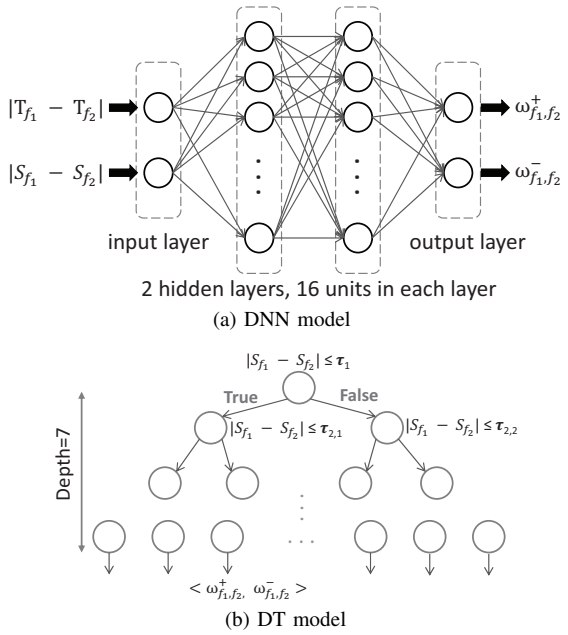
PICO integrates a data plane pipeline with a local controller that performs *sequential coflow classification* and *adaptive scheduling*.

A. Sequential Coflow Classification

Conventional coflow classification algorithms usually classify a set of flows into coflows, called *global clustering* hereafter. This is, however, not a feasible solution for distributed in-network scheduling. The main reason is that parallel flows usually arrive at different times sequentially. Frequent global clustering for every new flow would incur unacceptable latency and overhead. To enable fast incremental classification for each flow arrival, we propose *pairwise coflow detection*. The core idea is to predict whether any two flows belong to the same coflow. By estimating the correlation between a newly-arrived flow with any existing flow, PICO can quantify the confidence (score) of adding this new flow into a group of existing flows that are more likely to belong to the same coflow. In addition, to simplify the system architecture, we keep a flow in the same coflow group without migration even if the remaining parallel flows of the same coflow arrive later. By such *pairwise coflow detection* and *iterative score-based coflow classification*, parallel flows belonging to the same coflow could naturally gather together in the same queue and obtain the same priority. By doing this, PICO avoids the need for complex global reclustering and schedules each new flow promptly after it arrives.

Pairwise coflow detection. PICO's switch classifies each flow on the fly. Hence, unlike previous approaches that adopt global clustering algorithms similar to K-mean, which cluster the flows all together before they have been sent to a switch, PICO has to determine whether a newly-arrived flow belongs to the same coflow of other flows that have arrived earlier. To enable real-time classification, we propose *pairwise coflow detection*, which predicts the likelihood of two flows belonging to the same coflow. The predicted likelihood is then used as the *score* of the two flows for score-based coflow grouping.

We train two different models, DNN and decision tree (DT), to predict the coflow likelihood of two input flows. For both the models, we try various widely used flow attributes as the input features, including the source/destination ports, the protocols (UDP, TCP, ICMP, etc.), flow start time, the mean packet size, the variance of packet sizes, the average packet inter-arrival time, and the variance of packet inter-arrival time [22]. After extensive experiments and testing, we identify the two most meaningful attributes, *flow start time* and *mean packet size of the first few packets*, to be the input features of our detection model. The rationale behind such feature selection is that parallel flows of a coflow are usually generated by the same application and, hence, arriving (though not in bulk) at similar times. That is, their packet sizes and start time are often highly correlated.



(a) DNN model

(b) DT model

Fig. 4: Model architecture

The model is trained to output a 2×1 coflow likelihood vector of two input flows f and f', denoted by $[\omega^+_{f,f'}, \omega^-_{f,f'}]$, where $\omega^+_{f,f'}$ is the *positive likelihood* of two flows belonging to the same coflow and $\omega^-_{f,f'}$ is the *negative likelihood* of two flows *not* belonging to the same coflow. During training, we sample two random flows from a dataset. The positive label is marked as 1 if the two flows belong to the same coflow and 0, otherwise. Similarly, the negative label is 1 for two independent flows and 0, otherwise.

As illustrated in Fig. 4(a), we use a four-layer fully connected DNN model, which consists of an input layer, two hidden layers, and an output layer. The input layer feeds the input features to the network. The feature vector is defined as the absolute difference of the start time of two flows and the absolute difference of the mean size of two flows' first 20 packets. In the hidden layer, we adopt Rectified Linear Unit (ReLU) as the activation function. RMSProp is used as the optimizer with a learning rate of 0.001. The output layer is a sigmoid activation function that outputs two scores, $\omega^+_{f,f'}$ and $\omega^-_{f,f'}$, ranging between 0 to 1. We use categorical cross-entropy as the loss function, which can be expressed as

$$L = -[t_1 \log(\omega^+_{f,f'}) + t_2 \log(\omega^-_{f,f'})], \quad (1)$$

where $t_1, t_2 \in [0, 1]$ are the positive and negative labels,

Algorithm 1: Score-based classification Algorithm

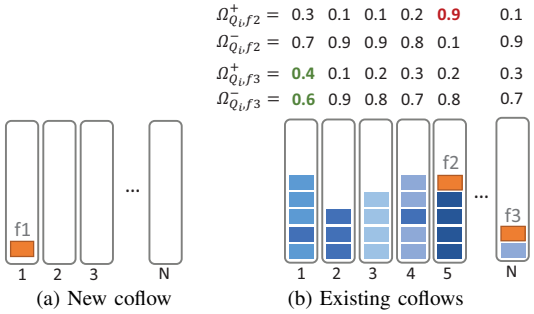
Input : f: a new flow; $\mathcal{Q}$: coflow queues

1 **if** *all* $Q \in \mathcal{Q}$ *are empty* **then**
2 insert f into a randomly selected coflow queue Q
3 **return** Q;
4 **end**
5 **for** $Q \in \mathcal{Q}$ **do**
6 $Q' \leftarrow$ randomly sample at most k flows from Q
7 **for** *each* $f_i \in Q'$ **do**
8 $[\omega^+_{f,f_i}, \omega^-_{f,f_i}] \leftarrow \text{PD}(f,\ f_i)$
9 **end**
10 $\Omega^+_{Q,f} \leftarrow \sum_{f_i \in Q'} \omega^+_{f,f_i}/|Q'|$
11 $\Omega^-_{Q,f} \leftarrow \sum_{f_i \in Q'} \omega^-_{f,f_i}/|Q'|$
12 **end**
13 $Q^*_f \leftarrow \arg\max_{Q \in \mathcal{Q}} \Omega^+_{Q,f}$
14 **if** $\Omega^+_{Q^*_f,f} \geq \Omega^-_{Q^*_f,f}$ **then**
15 insert f into Q^*_f
16 **else if** *there exists an empty queue* Q_ϕ **then**
17 insert f into Q_ϕ
18 **else**
19 insert f into the shortest coflow queue $Q_{\min}$
20 **end**
21 **return** Q;

respectively, and ω^+ and ω^- are the positive and negative likelihood of classifying two input flows as the same coflow.

Fig. 4(b) demonstrates the structure of the decision tree (DT) model with binary splitting (true or false). The DT model contains 197 nodes (99 leaves). We adopt mean squared error (MSE) as the criteria to determine locations for future splitting. The maximum depth is set to 7 to control the tree size for preventing overfitting. During inference, we input the features of two flows, one is a newly-arrived flow and the other is an existing flow, to the model. As the flow features are extracted from the first few packets of a flow, PICO can output the inference results in the early stage of flow arrival and enable fast classification.

Score-based sequential coflow classification. We propose a score-based coflow classification algorithm that leverages the predicted *pairwise coflow scores* to insert a newly-arrived flow into the *most possible coflow cluster*. To do so, PICO builds several coflow queues, each of which tries to collect the flows belonging to the same coflow. To make the system simple, after inserting a newly-arrived flow into a queue, we keep it in the same queue until it terminates, i.e., no migration across queues allowed. As flows may arrive at different times, our score-based classification aims at inserting all the flows of a coflow into the same queue *sequentially* whenever they arrive. To achieve this goal, we compare each newly-arrived flow with the flows that have been cached in any queue and estimate the likelihood (score) of a flow belonging to the queue. This new flow will be inserted into the queue with the highest coflow score. Note that, if a flow is incorrectly classified into a wrong queue, this classification error not only introduces



(a) New coflow (b) Existing coflows

Fig. 5: Example of coflow classification

a misclassification event but also increases the classification error probability of future flows since any future flow should be compared with the flows cached in the same queue.

Algorithm 1 summarizes our score-based sequential classification algorithm. Initially, we create N empty queues $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_N\}$. The goal is to insert a flow into a queue that caches its parallel flows. Ideally, if N is no smaller than the number of coflows, with perfect classification, each queue can represent a coflow. For each new flow f, we estimate the likelihood Ω^+_{f,Q_i} of this flow belonging to a queue Q_i based on our proposed *pairwise coflow detection* (PD) model. In particular, consider a queue caching a set of flows $Q = \{f_1, f_2, \cdots\}$. To estimate the likelihood of a flow belonging to Q, we adopt our DNN (or DT) PD model to estimate the similarity score of f and *any existing flow* $f_i \in Q$. However, when the queue size grows, it is time-consuming to calculate the pairwise score for all the flows cached in a queue. To keep constant time-complexity, we sample at most k flows from a queue and denote Q' as the collection of the sampled flows (line 6). The coflow likelihood of queue Q for the new flow f is defined as the average of the sampled pairwise similarity scores (line 10), i.e.,

$$\Omega^+_{Q,f} \equiv \sum_{f_i \in Q'} \omega^+_{f,f_i}/|Q'|. \tag{2}$$

By calculating the coflow likelihood of all the queues, we can find the queue with the highest similarity score (line 13), i.e.

$$Q^*_f = \arg\max_{Q \in \mathcal{Q}} \Omega^+_{Q,f},$$

as the *candidate queue (coflow)* of flow f.

To confirm that the identified candidate queue is an appropriate choice, we further check whether the coflow likelihood of the candidate queue is sufficiently high. Specifically, we compare the *positive coflow likelihood* of the candidate queue with its *negative coflow likelihood*, i.e.,

$$\Omega^-_{Q^*_f,f} \equiv \sum_{f_i \in Q'} \omega^-_{f,f_i}/|Q'|.$$

If $\Omega^+_{Q^*_f,f} \geq \Omega^-_{Q^*_f,f}$, we believe Q^*_f is the coflow of f and insert f into Q^*_f (line 15). Otherwise, Q^*_f may not be the coflow of f due to some possible reasons: *i)* This flow may not belong to any existing coflows, *ii)* the pairwise coflow detection model may not accurately estimate the coflow likelihood, and *iii)* the number of flows cached in the queue is still too small to gather

their parallel flows. In this case, we insert the new flow f into any empty queue (line 17). If none of the queues is empty, f will be inserted into the smallest queue (line 19). The intuition of this choice is that we have higher confidence for a larger queue that has collected more similar flows. Hence, it is better to put a flow into a small queue and avoid disturbing those converged large coflows.

Consider the example shown in Fig. 5(a). Assume that flow f_1 is the first arriving flow of coflow cf_1. Since all the queues are empty, the coflow scores of all the queues are the same when they are all empty. We put f_1 directly into a randomly selected queue, say Q_1. Next, say another flow f_2 belonging to coflow cf_2 arrives after all the queues have cached some flows. PICO calculates the coflow score of each queue and identifies the one with the highest score, i.e., Q_5 as illustrated in Fig. 5(b). Since $\Omega^+_{Q_5,f_2} > \Omega^-_{Q_5,f_2}$, we can insert f_2 into coflow Q_5 with high confidence. Note that, since the first flow could be inserted into any queue, the queue index may be different from the coflow identifier. Finally, assume the first flow f_3 of a new coflow cf_3 arrives. PICO detects that its coflow score is low for every queue, i.e., $\Omega^+_{Q,f_3} < \Omega^-_{Q,f_3}, \forall Q \in \mathcal{Q}$, we treat f_3 as the first flow of a new coflow and insert it to the shortest queue, i.e., Q_N in this example.

B. Adaptive Coflow Scheduling

Similar to Aalo [7], we assume that the coflow sizes are unknown and adopt D-CLAS [7], which assigns each coflow a priority that is decreasing in the number of bytes it has already sent. Different from Aalo [7], PICO performs in-network scheduling and does not rely on any host to report its traffic patterns or share flow features. To real-time monitor the required information, we adopt sketch-based network telemetry, which leverages hash techniques to track a large number of flows with constrained memory resources [23]. Since hashing takes $O(1)$ time complexity, sketch-based telemetry has been widely used to balance the tradeoff between tracking accuracy and time complexity. In PICO, we implement a *feature sketch* and a *size sketch*. The former logs the features required for coflow classification, while the latter real-time monitoring the cumulative coflow size for adaptive prioritization.

We design our feature sketch and size sketch based on count-min sketch (CM sketch) [23]. A CM sketch consists of a two-dimensional array with w columns and d rows. Each element is hashed to a bucket of d rows using d different hash functions. The values of the d hashed buckets are updated accordingly. Two different elements may be hashed to the same bucket in certain rows, leading to collisions. However, the probability of having collisions in all the d rows is fairly small. Hence, to query the statistics of an element, we can use the same hash functions to locate its d buckets and return the *smallest value* among the d buckets as the estimate. In such a CM sketch design, the queried value could be larger than the actual value of an element due to collisions but will never be smaller than the actual value.

Feature sketch. PICO's feature sketch tracks the average packet size of the first N packets. We use each bucket to track the packet count c and the cumulative size l of the first N packets. As the ALU of most existing programmable switches does not support multiplication and division, we let the local controller calculate the average packet size instead. The controller removes the entry when the information of N packets has been collected. Since the feature sketch only tracks the first few packets of a flow, collisions would rarely occur. We hence use only a single hash function, i.e., $d = 1$.

When a miss event of the priority table occurs, the packet will be directed to the feature sketch. We use the 5-tuple flow identifier as the key, which is hashed to its bucket. Hashing to an empty bucket implies that the packet is from a new flow. We then initialize the bucket to $(c = 1, l = l_{now})$, where l_{now} is the packet size, and clone this first packet, which is forwarded to the local controller. The controller maintains a master feature table to record the arrival time of each new flow by extracting the timestamp of the cloned (first) packet. As the controller will not insert the priority rule until collecting the features from the first N packets, the later packets are still forwarded to the feature sketch since no match can be found in the priority table.

However, since the later packets are hashed to a non-empty bucket, they do not need to be cloned as the controller is only interested in the start time of a flow. We hence only increase its packet count c by 1 and update the cumulate size by $l = l + l_{now}$. Finally, when the packet count reaches N, the data plane actively forwards the statistics to the local controller, which derives the average packet sizes of this flow, updates the master table, and cleans up the bucket in the data plane. The features recorded in the master table are used to perform pairwise coflow detection introduced in Section V-A. For some small flows with fewer than N packets, the controller periodically retrieves information from the feature sketch and cleans up the bucket when a flow gets timed out.

Size sketch. The size sketch is a CM sketch with three hash functions. The local controller periodically queries the size sketch for each flow recorded in the master flow table to estimate its cumulative size. Since the controller keeps tracking the flow identifiers of each coflow based on the score-based classification result, it then uses the estimated flow sizes to offline calculate the coflow size. PICO exploits D-CLAS [7] to recalculate the coflow priority based on the up-to-date coflow sizes. At a high level, a coflow with a higher cumulative size is assigned a lower priority. If the priority of a coflow changes or has not been inserted into the priority table yet, the controller installs the updated priority in the priority table such that the later packets of a flow can be forwarded based on the up-to-date priority.

VI. Performance Evaluation

We evaluate PICO over two platforms: a small-scale testbed implemented on an Edgecore Wedge100BF-32X programmable Tofino switch and a large-scale simulator implemented in Python. We conduct extensive simulations to evaluate the classification accuracy of PICO and compare our in-network scheduler with existing host-assisted coflow

TABLE I: Information about the dataset

Dataset	Num. of flows	Num. of coflows	Overhead (Bytes)
Facebook	109,288	100	874,304
Heavy tail	487,165	130	3,897,320
Light tail	71,797	300	574,376

schedulers. The throughput performance of PICO's pipeline design is measured on the Tofino testbed.

Simulation configurations: We use a realistic cluster trace [24] collected by Facebook from a 3,000 machines and 150-rack MapReduce cluster with 10:1 over-subscription as the workload of our simulations. The data-trace contains over 700,000 flows, which belong to over 500 coflows. However, the Facebook trace does not indicate packet-level information. We randomly select a number from 500 to 1,000 as the mean packet size of a coflow and let the packet sizes of its flows follow the normal distribution with the sampled mean and a random standard deviation ranging from 0 to 100.

To further check the impact of the coflow size distribution, we sample coflows from the Facebook trace to meet a specified distribution. We consider two different distributions: light-tailed distribution (i.e., coflows having similar sizes) and heavy-tailed distribution (i.e., most of the coflows being small). The light-tailed distribution characterizes the behavior of web search, while the heavy-tailed distribution characterizing the behavior of data analysis workflows [19]. The detailed information about each dataset is summarized in Table I. We further show the message overhead of each dataset if a host-assisted scheduler has to collect the required coflow ID and size information from the hosts. The heavy-tailed distribution incurs much higher overhead since it contains some large coflows, i.e., including a large number of parallel flows, and the overhead is proportional to the coflow size. The results show that, without PICO's distributed design, each round of schedule would incur around MB of signaling overhead, which is significant for frequent and adaptive scheduling.

The simulator implements a detailed task-level replay of the Facebook trace and preserves input-to-output ratios of the tasks, locality constraints, bandwidth constraints, and inter-arrival times between jobs. We simulate the network as events of 10s decision intervals. For simplicity, the job computational time is not considered in the simulator. That is, we only measure the transmission time of coflows. In the simulations, we implement PICO's coflow classification and sketch-based coflow tracking to verify the impact of classification errors and flow size estimation errors.

We compare the following coflow scheduling algorithms:

- *ideal:* Varys [6], which is a host-assisted scheduler with complete coflow size and ID information.
- *host:* Aalo [7], which is a host-assisted scheduler with coflow ID information but without size information.
- *host-PD:* information-agnostic host-assisted scheduler using our DNN-based pairwise coflow detection to perform sequential scheduling for all the flows.
- *PICO:* our in-network coflow scheduler with DNN-based pairwise coflow detection and local scheduling.

TABLE II: Comparison between different schedulers

Algorithm	Coflow Size	Coflow ID	Scheduling
ideal (Varys [6])	✓	✓	Global
host (Aalo [7])	✓	×	Global
host-PD	×	×	Global
PICO	×	×	In-network

TABLE III: Performance of coflow classification

Method	F1 score	Precision	Recall	Accuracy
PICO	98.9%	99.4%	98.4%	99.7%
PICO-DT	97.8%	99.3%	96.3%	99.4%
Distributed K-means	59.4%	86.8%	49.2%	52.6%
K-means	26.4%	33.3%	22.6%	68.6%
Host-PD	93.2%	99.6%	87.5%	99.2%

- *PICO-DT:* our in-network coflow scheduler with DT-based pairwise coflow detection and local scheduling.
- *fair-sharing:* a flow-based scheduler that allocates equal bandwidth to each flow.

Table II summarizes the differences among the comparison algorithms. Except for fair-sharing and Varys [25], the remaining schedulers are based on D-CLAS (a variation of smallest-first scheduling) proposed by Aalo [6].

Testbed. We implement our pipeline on a P4 Edgecore Wedge100BF-32X switch with 3.2 Tbps switching capacity, 4.7 Bpps forwarding rate, 22 MB integrated packet buffer and 32 x QSFP28 ports, each supports 1x40/50/100 GbE or 4 x 10/25 GbE using cable splitters. Our server includes a 6-core Intel i7-8700 3.2GHz CPU, 16G memory, a 128GB hard disk, and a Mellanox ConnectX-4 Lx EN Network Interface Card. Each server runs Ubuntu 20.04.2 64bit with Linux 5.8.0 kernel.

Metrics. Our simulations evaluate the accuracy of pairwise coflow detection and the scheduling performance based on the following metrics:

- *Precision:* ratio of the flows that are truly in coflow C_i among all the flows classified into C_i.
- *Recall*: ratio of the flows in C_i that are correctly classified into C_i.
- *F1 score:* $\frac{2*\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}}$, which considers both false positives and false negatives to measure the robustness of a classifier with 1 being the best and 0 being the worst.
- *Accuracy:* the ratio of the number of correctly classified flows to the total number of flows.
- *FI of CCT:* coflow completion time, which is illustrated as the factor of improvement (FI) over *per-flow fairness* (baseline scheme). That is, FI $= \frac{CCT_{fair}}{CCT}$, where CCT_{fair} is the CCT of *per-flow fairness* and CCT is the CCT of any algorithm. A larger FI implies a shorter CCT and, hence, a better scheduling efficiency.
- *ARA:* average relative accuracy $(1 - \frac{1}{n}\sum_{i=1}^{n}\frac{|f_i - \hat{f}_i|}{f_i})$, where n is the number of flows and f_i and $\hat{f}_i$, respectively, are the actual and estimated flow sizes (number of packets) used to evaluate the accuracy of flow size estimation.

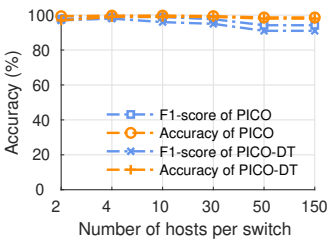


Fig. 6: Impact of host numbers

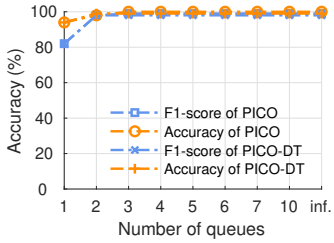
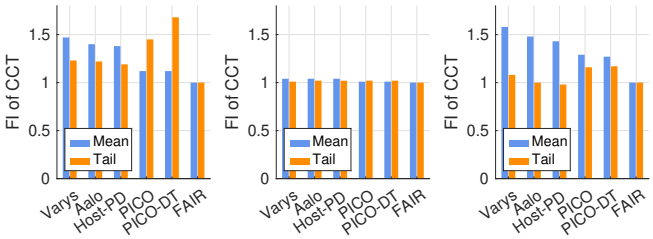


Fig. 7: Impact of queue numbers



(a) FB trace (b) Light-tailed (c) Heavy-tailed

Fig. 8: Factor of improvement of CCT

A. Accuracy of Sequential Coflow Classification

We first evaluate the performance of PICO's sequential classification. To exclude the impact of monitoring errors caused by sketch collisions, in this experiment, we extract the flow features directly from the dataset, instead of the features collected by PICO's pipeline. Table III compares the performance of four classifiers, including PICO, PICO-DT (PICO with decision tree), DK-means (Distributed K-means), K-means and Host-PD. The first three are distributed algorithms, while the last two are centralized algorithms. That is, K-means clusters all the flows at a time, while DK-means asks each switch to perform K-means locally for the flows traversing through it. To optimize the performance of K-means, we set K to the number of coflows.

The results show that our sequential classification achieves an accuracy of around 99%, whereas the K-means based algorithms produce much lower accuracy. It looks counter-intuitive that K-means uses global information to cluster all the flows as a whole but instead performs worse. This is because, when there exist a large number of flows being clustered at the same time, little noise added into distance estimation of any flow pair could disturb the clustering performance. Pairwise coflow detection leverages the arrival time information to only cluster flows arriving sequentially. Considering temporal proximity, hence, helps enhance classification accuracy.

Impact of traffic load. We next check the impact of the number of hosts connecting to a single ingress switch. We again exclude the effect of sketch errors and directly extract the flow features from the dataset. Fig. 6 plots the accuracy and F1 score of PICO and PICO-DT when the number of hosts varies from 2 to 150. The results show that the accuracy and F1 score slightly drop when more hosts connect to the same switch. The more hosts connecting to a single switch, the more flows being classified, increasing the probability of misclassifying a flow to a wrong group. However, the accuracy can still be higher than 95%. This small prediction error has a negligible impact on the packet scheduling performance. We also note that PICO performs similar to PICO-DT. We hence only show the performance of PICO with the DNN classifier in the following experiments.

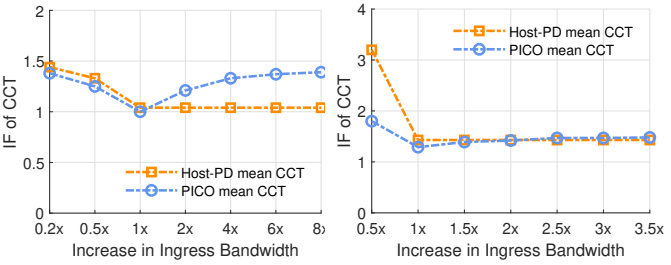
Impact of number of classes. We further examine the impact of the number of queues on classification accuracy in Fig. 7. We observe that, when the number of queues decreases, different coflows may be clustered together in the same queue. Hence, the switch may get confused when pairwise comparing a flow with the existing flows of a queue. In general, the accuracy (F1 score) can be very close to 100% when a switch deploys more than 4 queues for classification because the number of coflows simultaneously passing through a switch rarely exceeds 4. Since deploying too many queues would waste memory resources, we set the default number of queues to 7 in the following experiments.

B. Job Completion Time

We next examine the job completion time with consideration of classification errors and prioritization errors. In this experiment, we use the estimate of sketch-based tracking to schedule coflows. Fig. 8(a) plots the FI of the average CCT and the FI of the 95th percentile (tail) CCT, respectively, of the comparison algorithms. In general, both the host-assisted and in-network shortest-job first schedulers reduce the CCT significantly as compared to *per-flow fairness*. Recall that all the *ideal*, *host*, and *host-PD* schemes perform global scheduling. Hence, as PICO's pairwise coflow detection achieves fairly high accuracy, host-PD (without coflow ID information) produces the average CCT just slightly worse than Aalo (with full coflow ID information).

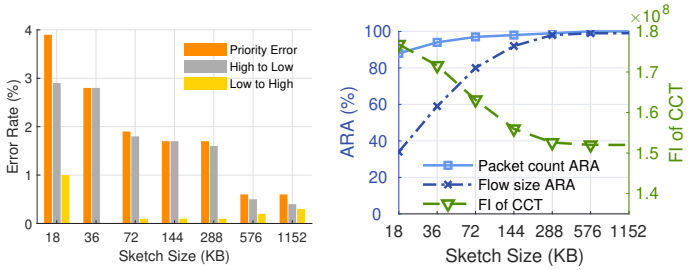
Compared to *host-PD*, PICO increases the average CCT slightly since our in-network scheduling cannot prioritize packets before they arrive in an ingress switch. Hence, the transmission efficiency from the hosts to the ingress switches may not be optimized. In addition, PICO leverages sketch-based telemetry, which leads to small estimation errors caused by hashing collisions and would introduce some prioritization errors. Even so, PICO's average CCT can still be 76.45% and 80.05% of those achieved by *ideal* and *host*, respectively, showing that fully distributed scheduling is quite efficient even without coordinating with the end hosts. Benefited by the slightly worse mean CCT, PICO, instead, achieves a much shorter tail CCT as some misclassified large coflows would be transmitted earlier than their ideal priority and, hence, reduce their CCT.

Impact of coflow size distribution. We next check how the coflow size distribution affects the performance of scheduling. Figs. 8(b) and 8(c) illustrate the FI of CCT for the light-tailed and heavy-tailed distribution, respectively. For the light-tailed distribution, our in-network scheduling performs almost the same as centralized scheduling. When most of the coflows are of roughly equal size, the scheduling performance does not change much when the traffic from the hosts to the ingress switches is not prioritized. As for the heavy-tailed distribution,



(a) Light-tailed distribution (b) Heavy-tailed distribution

Fig. 9: Impact of ingress bandwidth on CCT



(a) Prioritization errors (b) Correlation between ARA and CCT

Fig. 10: Impact of sketch sizes on estimation errors and CCT

PICO achieves the FI of 1.28×. Compared to the centralized schedulers, PICO cannot optimize the transmissions until the flows arrive in an ingress switch, as a result slightly increasing the mean CCT. However, even with the limited capability of local coflow scheduling, PICO enhances the CCT much without requiring host coordination.

Impact of ingress bandwidth. While our design prioritizes flows in the ingress switches, each incoming flow will be sent from the applications to their ingress switches simply based on naïve *per-flow fairness scheduling*. We now check how the end-to-end CCT is affected by neglecting coflow prioritization in the segment between a host to its ingress switch. To this end, we vary the ingress bandwidth and plot the results in Figs. 9(a) and 9(b) for the two distributions, respectively. For the light-tailed distribution, smallest-first scheduling (PICO and host-PD) performs similar to fair scheduling since most of the coflows are of similar sizes. For the heavy-tailed distribution, PICO's in-network scheduling performs worse than host-assisted global scheduling when the ingress bandwidth is small. Without proper prioritization, limited ingress bandwidth can not be utilized efficiently. However, in-network scheduling outperforms host-assisted scheduling when we increase the ingress bandwidth. This shows that, when the pre-ingress segment is not the bottleneck, in-network scheduling does not hurt CCT much even if it delays the scheduling task until packets arrive in an ingress switch. This also confirms that PICO effectively prioritizes coflows in a distributed manner after they arrive in an ingress switch.

Impact of coflow size estimation errors. We examine how large is the coflow size estimation error and how it affects the scheduling performance. Fig. 10(a) plots the priority error rate, i.e., the ratio of misclassified flows among all the flows, when the sketch size varies from 18 KB to 1,152 KB. To distinguish the types of errors, we also plot the rate of "high-to-low" errors and the rate of "low-to-high" errors, separately, in Fig. 10(a). The results show that most of the prioritization errors belong to "high-to-low" errors, i.e., small coflows assigned a priority lower than what they should get. Due to the possibility of hash collisions, sketch-based monitoring may overestimate the flow sizes but never underestimate the true size. When small coflows are collided by some large coflows, those overestimated small coflows would be scheduled a priority lower than what they should get, leading to a longer CCT. However, when the sketch size increases, the error rate can be lower than 2%. In addition, since the estimated size of a large coflow is never smaller than that of a small coflow, the "low-to-high" rate is mostly close to 0. It implies that low-priority large coflows rarely interrupt the transmissions of most small coflows. Fig. 10(b) plots the correlation between sketch ARA and coflow CCT. When the memory sizes of the feature table and size table are sufficiently large, the sketch errors are acceptably small to guarantee the convergence of CCT minimization.

TABLE IV: Forwarding time.

Method	P4 Target	Bmv2
without PICO	18.1 μs	22.3 μs
with PICO	20.5 μs	23.7 μs

C. Forwarding Performance

We finally measure the forwarding latency of a switch with and without PICO for the software version (bmv2) and hardware version (Tofino) implementation. We implement the priority match table as well as two sketches in the testbed. Table IV summarizes the average forwarding time of 2,000 packets with and without PICO. We tag the packet metadata with the timestamp in the ingress parser and egress parser and calculate the average forwarding time offline. The results show that the additional latency introduced by PICO's pipeline is fairly small in the software version and is slightly higher but still acceptable in the hardware version. It shows that PICO's pipeline can operate at a line rate.

VII. Conclusion

In this paper, we presented PICO, a host-transparent in-network coflow scheduling system that prioritizes coflows in switches in a fully distributed manner. Our system addresses practical issues of coflow classification and in-network scheduling. We leverage *pairwise coflow detection* to identify the correlation between any two flows and propose a *sequential coflow classification* algorithm to iteratively and fast gather parallel flows. We then implement a data-plane pipeline that enables early feature extraction and real-time coflow scheduling. Our trace-driven simulations confirm that PICO's sequential coflow grouping achieves an accuracy of up to 99% . With reliably and fast coflow classification, PICO's fully decentralized design performs comparable CCT as compared to host-assisted schedulers.

REFERENCES

[1] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *ACM SIGCOMM*, 2012.

[2] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM*, 2011.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," in *ACM SIGCOMM*, 2013.

[4] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *ACM HotNets*, 2012.

[5] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," in *ACM SIGCOMM*, 2011.

[6] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *ACM SIGCOMM*, 2014.

[7] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM*, 2015.

[8] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *IEEE INFOCOM*, 2015.

[9] Y. Li, S. H.-C. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou, and F. C. Lau, "Efficient online coflow routing and scheduling," in *ACM MobiHoc*, 2016.

[10] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *ACM SIGCOMM*, 2016.

[11] H. Zhang, X. Shi, X. Yin, and Z. Wang, "Yosemite: efficient scheduling of weighted coflows in data centers," in *IEEE ICNP*, 2017.

[12] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," in *ACM SIGCOMM*, 2006.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (DCTCP)," in *ACM SIGCOMM*, 2010.

[14] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *USENIX NSDI*, 2012.

[15] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *ACM SIGCOMM*, 2002.

[16] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[17] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, "Minimizing flow completion times in data centers," in *IEEE INFOCOM*, 2013.

[18] S. Zhang, Z. Qian, H. Wu, and S. Lu, "Efficient data center flow scheduling without starvation using expansion ratio," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, pp. 3157–3170, 2017.

[19] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM*, 2014.

[20] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," in *ACM SIGCOMM*, 2014.

[21] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *ACM SIGCOMM*, 2016.

[22] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE communications surveys & tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[23] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[24] "Coflow benchmark based on facebook traces." [Online]. Available: https://github.com/coflow/coflow-benchmark

[25] N. Chowdhury, "Coflow: A networking abstraction for distributed data-parallel applications," Ph.D. dissertation, UC Berkeley, 2015.