Providing In-network Support to Coflow Scheduling

Cristian Hernandez Benet*, Andreas J. Kassler*, Gianni Antichi †
Theophilus A. Benson‡ and Gergely Pongracz§

*Karlstad University, †Queen Mary University, ‡Brown University, §Ericsson Research

*cristian.hernandez-benet,andreas.kassler@kau.se, †g.antichi@qmul.ac.uk, ‡tab@cs.brown.edu, §Gergely.Pongracz@ericsson.com

arXiv:2007.02624v1 [cs.NI] 6 Jul 2020

***Abstract*—Many emerging distributed applications, including big data analytics, generate a number of flows that concurrently transport data across data center networks. To improve their performance, it is required to account for the behavior of a collection of flows, i.e., coflows, rather than individual. State-of-the-art solutions allow for a near-optimal completion time by continuously reordering the unfinished coflows at the end-host, using network priorities.**

This paper shows that dynamically changing flow priorities at the end host, without taking into account in-flight packets, can cause high-degrees of packet re-ordering, thus imposing pressure on the congestion control and potentially harming network performance in the presence of switches with shallow buffers. We present *pCoflow*, a new solution that integrates end-host based coflow ordering with in-network scheduling based on packet history. Our evaluation shows that *pCoflow* improves in CCT upon state-of-the-art solutions by up to 34% for varying load.

***Index Terms*—Coflow, Datacenter Networks, P4, Dataplane Programming**

I. Introduction

Emerging big data processing frameworks such as MapReduce [1], Spark [2] or Dyan [3] are based on a partition/aggregate programming model that allow them to distribute and parallelize the processing across different machines. Such big data analytics frameworks are also becoming an important enabler for future mobile networks with use cases ranging from incident detection at Network Operations Centers [4], Traffic Classification and Network Slicing [5] to IoT data processing [6]. A common property of such big data frameworks is that each processing stage cannot complete until all data have been transferred. As a consequence of this property, the performance of these frameworks is function of the behavior of the collection of flows used to transfer data in each stage [3], [7], i.e., coflows [8]. More formally, coflows are collection of flows of varying sizes with different communication endpoints.

Most of the growing work on improving performance within data centers build upon the advances in data center load balancing techniques, e.g., Hula [9], and data center centric transports, e.g., DCTCP, which improve low level details of the data center's network. By abstracting details about load balancing, routing, and transport, these emerging techniques can focus on the crucial aspect of the network which impact individual flow performance, i.e., controlling queuing, priorities, or rate-limits. However, existing approaches for queuing, priorities, rate-limits within the data center do not provide the levels of dynamicity required to support recent coflow proposals, e.g., Sincronia [10].

In this work, we ask the following fundamental question: *"Does the network provide sufficient primitives to faithfully support dynamic modification of coflow priorities and queues?"*. To answer this question, we use a large scale trace-driven simulation, which allows us to methodologically analyze a broad range of existing techniques and scenarios. Our initial observations are that current network primitives do not effectively support arbitrary modifications and changes to a coflow's queuing priorities. In particular, we observe that while the network provides the illusion that all packets in a flow can be atomically moved between queues. In practice, once a packet has been queued, it does not dynamically change queues, which leads to packets from a flow ending up in different queues. The end result of this phenomenon is that packets from a flow are spread across queues resulting in a high degree of packet reordering when they arrive at the destination end-hosts which lead to reduced performance due to TCP's design.

In particular, due to TCP's behavior high-degrees of packet re-ordering can in some cases cause the congestion control window to shrink with negative effect on performance.

In this paper, we argue that existing data center networks lack in-network support for dynamically changing coflow queuing priority: specifically, with existing network primitives changing a flow's priority does not consider packets already traversing the network, thus causing inconsistencies between the in-flights and newly generated packets of the flow at switches with multiple priority queues. Motivated by these observations, we propose an in-network primitive, called *pCoFlow*, which allows flows to temporarily maintain queue affinity until already enqueued packets are drained when flows are being reprioritized due to coflow order update. A key challenge in preserving flow affinity under re-prioritization lies in maintaining network state and using this state to dynamically override packet priorities and alter queuing behavior. Our work builds on emerging techniques for programmable data planes [11], [12] and uses them to maintain minimal per-coflow state, i.e., optimized packet histories, and dynamically manage flow priorities and queue assignments based on this state.

To demonstrate the strength of our approach, we propose

the design (§III) of a system that integrates state-of-the-art ordering mechanisms at the end-host, such as Sincronia, with in-network scheduling based on packet history (i.e., *pCoFlow*). We then show that the latter can be implemented in P4 starting from the PIFO abstraction [12]. Finally, we demonstrate that our approach reduces the average CCT by 15% up to 18% for 10% load and by 27 up to 34% for 90% load with respect to state-of-the-art solutions.

In summary, the contributions of this paper are:

- We make the case for in-network support in the context of coflow scheduling.
- We propose *pCoflow*, which is an architecture that integrates state-of-the-art techniques performing ordering at the end-host with advanced in-network packet scheduling.
- We design a coflow aware in-network priority scheduler that avoids reordering and can be implemented using the PIFO abstraction [12].

II. Motivation

Recently, there has been a tremendous effort on network designs for coflows [13], [14], [15], [16], [17], [18]. Some of the works advocate for adopting a distributed approach [17], [18], where coflows are scheduled and managed locally at the end hosts; others propose a centralized scheme [13], [14], [15], [16], where a single entity with global knowledge is in charge of managing coflows. Centralized solutions that rely on global knowledge have proved to guarantee better performances. However, the need to centrally calculate complex per-flow rate allocations has hindered the possibility to realize them in practice [10].

Recently, Sincronia [10] has demonstrated that the key ingredient to obtain near-optimal performances is to provide convenient coflow prioritization. Specifically, if the *right* ordering of coflows is given, any per-flow rate allocation mechanism would lead to *good* results provable close to the optimum if co(flow) scheduling preserves the order. In other words, if coflow C_i is ordered higher than coflow C_j, all flows and packets in C_i must be prioritized over C_j. Such an important finding has opened up the possibility of a new network design where a central controller is just in charge of ordering the coflows, while leaving any per-flow prioritization to the end-hosts, thus striking the right balance between centralized and distributed schemes. In this regard, Sincronia assumes that individual flow scheduling and rate allocation is provided by a priority-enabled transport layer at the end-host, obeying to a centrally managed coflow ordering controller [10]. In practice, given a coflow ordering, the end-host will continuously (re)assign the priority of a flow coherently with the coflow it belongs to using, for example, DiffServ markings.

The need for in-network support. Decoupling scheduling decisions, centrally managed, from the flow-rate allocation problem, controlled at the end-hosts through a priority-enabled transport layer, might generate an excessive amount of out-of-order packets thus affecting network performances. To illustrate this problem, we used the NS2 simulator. We assumed a non-blocking big-switch as network topology [19], [14], [10] and we used Data center TCP (DCTCP) [20] as state-of-the-art congestion control for data center networks. We generated traffic according to the Sincronia workload generator [10] which is based on Facebook traffic characteristics and used an increasing number of coflows from 20 to 200. We let Sincronia calculate the coflow ordering and we enforced the corresponding flow priority using DSCP marking. Furthermore, to properly enforce the correct ordering, we enhanced the big-switch abstraction with eight queues per port. Figure 2 shows the total number of timeout events obtained for different coflow sizes. The reason lies in the Sincronia behavior that dynamically change flow priorities enforcing new policies from the end hosts without considering packets that are already traversing the network. Duplicated ACKs might trigger a flow to shrink its congestion window, impacting directly on network performance. The effect on the CCT is shown in Figure 1. Here, we compare the CCT we obtained with Sincronia against an optimal scenario where a change in coflow priority does not cause any packet reordering. The ideal case performs up to 1.5x better than Sincronia.

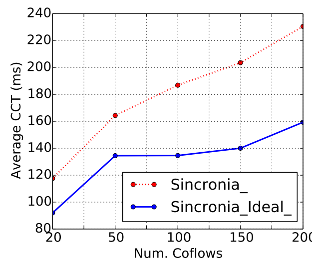

Fig. 1. Average coflow completion time

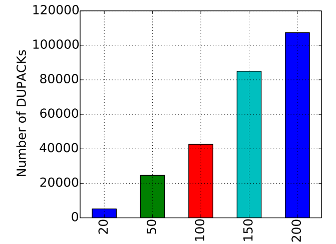

Fig. 2. DupAcks for different number of coflows

To better understand the cause of packet reordering, we illustrate in Figure 3 what happens when the Sincronia controller issues a change in coflow priority. Let us assume that at some point coflow 1 finishes and Sincronia increases the priority level of each remaining active coflow. This change affects new packets to be generated from end hosts, while the one already in-flight will be served with the old configuration, i.e., priority. This clearly creates packet reordering if packets of the same flow are still enqueued at a lower priority queue at the same switch and newer packets having higher priority arrive due to strict priority queue. Reordering may trigger congestion control, which reduces the rate of the flow.

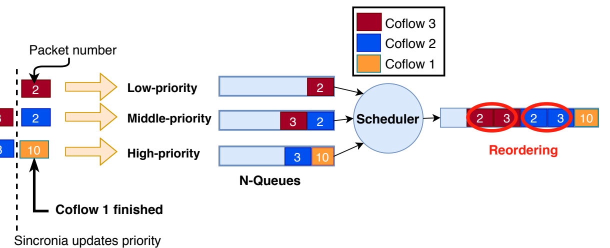

Fig. 3. Packet reordering after priority updates

This simple example illustrates a wider problem: in real data center topologies where multiple paths from source to destination are available, the amount of reordering as well as its effect on network performances can be even bigger. Indeed, to select the *best path*, the research community has shown the effectiveness of congestion aware flowlet-based load-balancing approaches [21], [9]. Those schemes split flows into smaller *flowlets*, exploiting the burstiness of TCP. The idea is to route each flowlet over the least congested path. However, when using Sincronia in combination with the mentioned solutions might trigger a reordering of not *just few packets*, but instead *entire flowlets*.

III. Design

Given the insights from the previous section, we ask the question *is it possible to minimise packet reordering due to priority changes by allowing switches to participate in scheduling decisions?* We answer this positively by describing *pCoflow*, a solution which provides in-network support for coflow scheduling. *pCoflow* integrates state-of-the-art ordering techniques at the end-host, e.g., Sincronia, with scheduling decisions taken *in-network*. The main idea is to leverage programmable switches to temporarily maintain queue affinity for newly arriving packets until already enqueued packets are drained when flows are being reprioritized due to coflow order update. We show that this can be realized with the PIFO abstraction [12] and by taking into account the priorities of packets *before* and *after* an update, i.e., their history.

A. Design Objectives

Avoiding In-Network Reordering:

Coflow schedulers such as Sincronia ensure coflow isolation and preserve the order of coflows by delegating prioritization to a priority-enabled transport mechanism. When run together with TCP, this requires a prioritization of IP packets according to the coflow priority, which is typically implemented using a multi-level queuing system with strict priorities. Such systems schedule packets waiting at higher priority queue first. Ideally, this would allow higher priority coflows to finish earlier and thus improving overall transmission time. However, the arrival, termination or changes in the remaining transmission time of coflows can lead to shifting in priority levels at end-hots that might lead to packet reordering, triggering congestion control and reducing the rate of the newly prioritized flow, achieving exactly the opposite effect. When using multi-level queuing systems, priority updates at end-hosts should therefore not result in packet reordering. Therefore, we aim to implement a single queue that manages coflow priorities during insert using packet histories that track priorities of enqueued packets.

Avoiding Coflow Starvation: Whether coflows are prioritized using per-flow rate allocation or transport layer priorities, coflow starvation must be avoided. This is necessary to ensure that large coflows do not starve short coflows. By leveraging network feedback in the form of Early Congestion Notification (ECN), congested network elements can signal end-host transport layer such as TCP [22] or DCTCP [20] that congestion is building up forcing them to scale back in rate. However, when using a single queue that manages different priorities, care must be taken to how ECN marking is applied.

B. pCoflow Design Overview

pCoflow uses state-of-the-art centralized coflow controllers such as Sincronia that orders coflows and derives their priority and combines it with transport layer that enforces priority scheduling (see Figure 4). The core component of *pCoflow* is a novel coflow aware strict-priority packet scheduler inside the data plane that is aware of priority levels of coflow packets waiting in the queues.

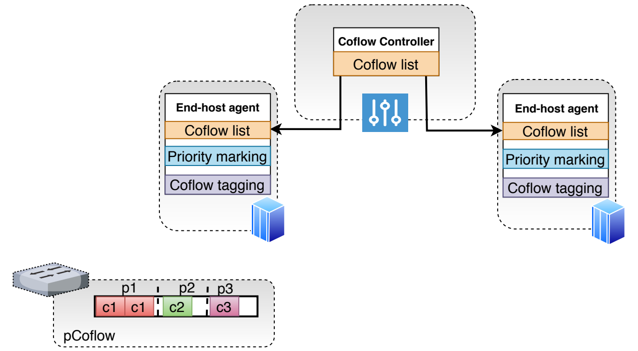

Fig. 4. pCoflow Architecture Overview

C. End-host

End-hosts are responsible for marking packets with the corresponding coflow priority and sending packets over the transport protocol. We exploit a shim layer between the application and the transport layer which continuously orders the coflows using e.g. information available from a centralized controller (e.g. Sincronia) [10]). The coflow order is translated by the end-host agent to DSCP values that map the highest order coflow to the highest priority level. Second highest priority is mapped to second highest priority, etc. and all remaining priorities are mapped to the lowest priority level. The shim layer also tags each packet with an unique `coflowID`, which is subsequently used by switches to avoid reordering when coflow order is updated by the end-host shim layer. The `coflowID` can be provided in an extra header (e.g. using GPE extension of VXLAN) or can be conveyed within the IP Identification field or TCP options.

D. In-Network Coflow-aware Scheduler

The main objective of our coflow aware programmable scheduler approach is to maintain coflow priorities for dequeuing operation while avoiding reordering when coflow priorities are switched at end-hosts. Consequently, we need to maintain the relative scheduling order of buffered packets with future packet arrivals, which will be implemented during the push-in operation. The main idea of our approach is to assigning coflows to a single queue as long as buffer space is available. In order to enable prioritization of coflows, we therefore partition a single queue into multiple virtual

priority bands, each one having a dedicated priority level and a certain buffer space. As long as there is space available at a given priority level, coflow packets matching that priority level can be inserted appropriately.

Packet reordering may only happen when the end-host increases the priority level of a given coflow. If there are still packets enqueued at the switch for the same flow at lower priority levels, the newly arriving packets are served first, leading to reordering. On the contrary, when new packets have a lower priority, there will be no reordering as packets waiting at higher priority levels will be served first. In order to avoid packet reordering due to end-host triggered coflow order update, we first identify the coflow priority by parsing the DSCP field in the packet header

The insert operation has to ensure that packets with highest priority are inserted at the first priority band of the queue, and packets with lower priority at lower priority bands. This makes sure that packets with higher priority are always served first implementing a strict priority queuing policy. In order to avoid that a change in coflow priorities may lead to packet reordering within a flow, we need to check to which coflow a packet belongs to. If a Sincronia triggered reorder increases the priority of coflow C_j, a packet may arrive at a switch with higher priority and several packets of the same coflow C_j may wait for transmission at a lower priority level. In this case, we temporarily do not use the higher priority as indicated by the end-host reordering but rather insert the newly arriving packet after packets of the same coflow C_j. This avoids reordering and makes transport layer transparent to priority changes. The drawback is a delayed response to priority changes in the switch.

Our packet scheduler needs to store (1) the bounds of priority bands, and (2) for each coflow the lowest priority band that has packets waiting to be served (Figure 5. When a packet with priority p_i that belongs to coflow C_j reaches the switch, the scheduler thus checks the position of the last packet enqueued at priority level p_i and the position of the last packet enqueued for coflow C_j and calculates the rank of the packet as in Equation 1.

$$rank = max(p_i, C_j) + 1 \tag{1}$$

pCoflow uses ECN to signal congestion to the end-host transport layer [23], [24]. When using a single queue with multiple priority bands and a single ECN marking threshold may lead to marking mostly lower priority packets which may lead to coflow starvation. To prevent this effect, *pCoflow* uses multiple ECN marking thresholds, one per priority band. If during enqueue we detect that the number of packets enqueued for a given priority p_i is larger than the ECN threshold for the given band, we set the ECN bit.

Although the minimum and maximum marking threshold of each priority level can be adjusted to react earlier to congestion and start marking packets before reaching the maximum threshold [25], congestion control algorithms can take several RTTs to adjust the sending rate after receiving ECN notification. Therefore, queue sizes may temporarily exceed the defined ECN thresholds. Dropping packets may be necessary when using transport protocols that do not react to ECN or if we want to guarantee a certain buffer space for each priority. However, enforcing packet drop reduces queue elasticity. *pCoflow* enables adaptive queue sizes by dynamically allowing priority bands to increase and shrink. It integrates such dynamic resizing with ECN marking to signal end-points to reduce their rate.

E. Implementation Feasibility

With today's switches, only a limited number of scheduling approaches is available whose parameters can be controlled by network operators. However, using programmable packet schedulers allows to implement custom scheduling disciplines tuned to application requirements. Indeed, using the push-in first-out (PIFO) scheduling abstraction [12], where packets can be pushed into an arbitrary position but always dequeued from the head, several scheduling approaches can be implemented on programmable data-planes such as P4 [11].

To implement our approach, we can leverage the PIFO abstraction [12], which allows enqueued packets to be pushed in arbitrary positions, given by the packet rank, while being dequeued from the head. PIFO assumes that packet ranks increase continuously within a flow and dynamic reordering of already enqueued packets is not supported. Consequently, we need to consider three main issues (i) extracting the coflow priority from the packet header; (ii) mapping packets to correct priority bands and computing the rank; and (iii) updating priority band bounds. For (i), we read the priority bits from the IP header ToS field. Figure 5 illustrates our approach.

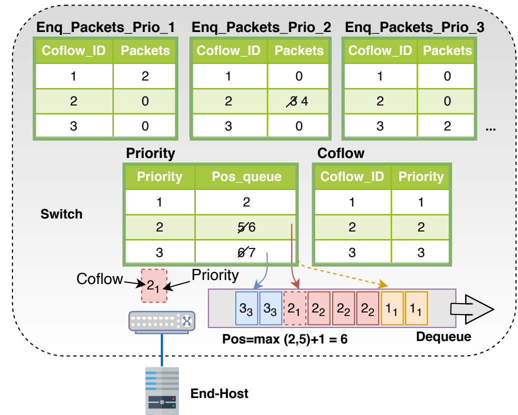

Fig. 5. pCoflow Queue with coflow and priority tracking

Mapping: We use registers `Priority` to store the bound information for each priority band. Assuming p priority bands, we use a register to encode the end of priority band p_i. For each coflow, we track the lowest priority band that still has packets enqueued in register array `Coflow`. If there is no packet enqueued for a coflow, we set the priority to 0. For calculating the PIFO rank at insert and avoid reordering, we first check `Coflow` to determine, which lowest priority band has packets enqueued (e.g. 2 in the example). Then, we look up the position of the last packet in this band using

`Priority`, which returns 5. We compare this with the rank of the last packet in the priority band that corresponds to the priority marked in the packet header (as the priority is one, the lookup returns 2). Equation 1 returns $rank = max(2,5) + 1 = 6$ which is used for PIFO insert operation.

Update: As in [10], we aim to map each coflow to the given priority band if the current order of the coflow is less than $p-1$, else we map it to band p. To avoid reordering when packets are enqueued at priority p_i and new packets for the same coflow arrive having higher priority, p_{i-k}, we track which bands have packets enqueued for each coflow using one register array `Enq_Packets` for each priority band. We update `Coflow` as follows. On enqueue of a packet at the end of priority band p_i, we update priority band bounds of all lower priority bands p_{i+k} (e.g. if $p_i == 2$ we will update bounds of bands 2, 3, 4...). We update `Enq_Packets` accordingly (e.g. indicating that priority band 2 has now 4 packets waiting for coflow 2). If there is no packet waiting to be transmitted in any queue (`Coflow` returned 0), we update `Enq_Packets` using the priority band corresponding to the packet priority. On dequeue, we update `Enq_Packets` for the priority band we dequeued from and coflow id. If after the dequeue there are no more packets in the current priority band, we sweep the remaining lower priority bands to find the lowest priority band that has still packets waiting and update `Coflow` for the given coflow id. If there are no more packets waiting, we set `Coflow` to zero. Finally, we also update `Priority` of the priority band corresponding to the packet that has been dequeued and all lower priority bands. To track the ECN marking threshold, we use counters per priority band. If we detect that an insert leads to more packets than allowed according to the ECN threshold for a given band, we mark the ECN bit. In our example (Figure 5), if the ECN threshold is set to 2 packets, when the packet belonging to coflow 2 and priority 1 arrives at the switch, the counter associated to priority 1 will return 2, and the ECN bit is set.

Remarks: Note, *pCoflow* downprioritizes temporarily all coflow packets if there are other packets waiting at lower priorities, which maybe not necessary. However, a more fine-granular per flow decision would require per flow tracking, which may lead to excessive switch resources. Note, that all sweeping operations through the multiple priority bands can be implemented as nested `if-else` statements as the number of bands is determined at compile time. As *pCoflow* does not require to maintain per-flow state (just per priority band and coflow), the required state variables is reasonably small. Increasing the priority bands *p* leads to more fine granular prioritization but requires more switch resources. Note, the scheme cannot be implemented on state-of-the-art hardware such as Tofino because registers in egress is not available in ingress, which however could be solved by packet recirculation at the expense of higher complexity. PIFO on the other hand supports not more than around 1000 flows [12]. A variant of our scheme supporting a fixed-size priority bands with limited reordering could be implemented using SP-PIFO [26].

IV. Evaluation and Results

We implemented our scheme in the NS2 simulator and used coflow traces for comparing our scheme against different configurations. **Topology:** We use a 3-tier Fat-tree topology with k=4. All links have a capacity of 40Gbps, except links connecting 64 servers to the ToRs, which have a capacity of 10Gbps. Each of the 8 ToRs has 8 servers connected that send coflows according to a given trace. Servers run a client application with the Sincronia shim layer that informs the Sincronia coordinator about the coflow information such as coflow id, number of flows and sources, and destination for each flow. It receives the coflow ordering and tags coflow priorities and coflow IDs. **Coflow Scheduler:** We use the online Sincronia algorithm from [10] to order coflows. We immediately recompute the order upon each coflow arrival and departure. As in [10], we map coflow order to the Diffserv option and use 8 priority levels. **Workload:** We use [27] to create a coflow trace having the same characteristics as the Facebook trace from [10]. The trace contains 150 coflows, which are composed of 2086 total flows. In total, the Intra-pod traffic was 32.8 GB and the Inter-pod traffic 25.4 GB. We increase the workload by reducing inter-coflow arrival rates. As in [14], [28], we group coflows into *short* if the longest flow of a coflow has less than 5MB. A coflow is classified as *narrow* if it has less than 50 flows leading to four categories: Short and Narrow (SN), Long and Narrow (LN), Short and Wide (SW) and Long and Wide (LW). **Network Layer Load Balancing:** We compare Equal-Cost Multi-path (ECMP) and HULA [9]. HULA is a flowlet-based load-balancing scheme that forwards flowlets over the least congested path. Fowlet gap is set to 500μs and probing interval is set to 200μs [9].
Transport Protocol and Queue: We use DCTP [29] with standard retransmission time-out of 3 RTTs and an RTO of 200us as in [30]. As baseline (deRED), we use 8 strict priority (SP) queues, each one holds [31] max. 500 packets. Each physical queue contains a single virtual RED queue ($min_th = 200$, $max_th = 400$) that starts marking ECN at $min_th = 200$ with a given probability and the scheduler maps flows to queues given by the Diffserv field. When using *pCoflow*, the single queue has 8 priority bands of aggregated size. Each priority band starts marking packets at $min_th = 200$ per band.

Results for BigSwitch: The first question we try to answer for *pCoflow* is, if it is better to drop the packets once the maximum number of packets per priority band is exceeded or allow to borrow space from lower priority bands? Figure 8 compares *pCoflow_Drop*, which drops packets once the limit for a band is reached (500) with *pCoflow_ECN* which adaptively adjusts queue bands, when using Sincronia for priority ordering. Dropping packets once the threshold is exceeded avoids coflow starvation by not allowing packets from other priorities to take queue space reserved for other priorities. On the other hand, allowing coflows to temporary exceed their reserved queue space enables flows to steadily reduce their sending rate. Although this decision may temporarily lead to coflow starvation, we note that coflows can only take more

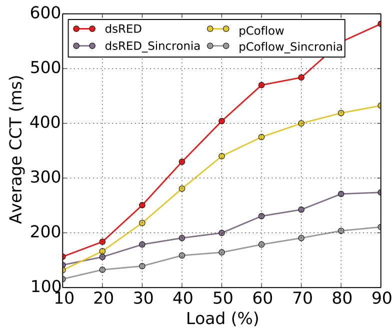

Fig. 6. Average CCT for BigSwitch

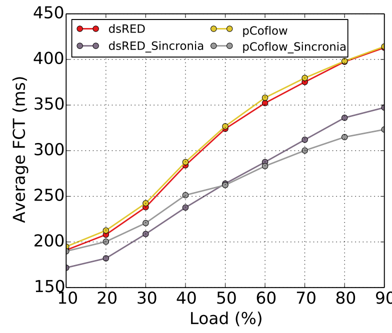

Fig. 7. Average FCT for BigSwitch

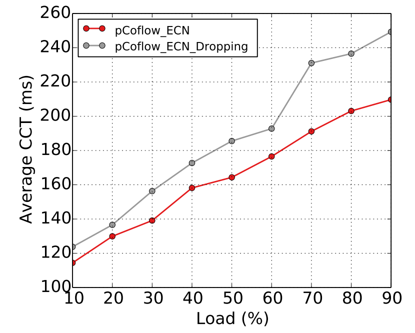

Fig. 8. pCoflow Queue ECN vs ECN-Dropping

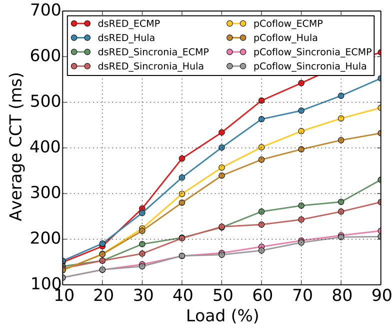

Fig. 9. Average CCT for fat-tree topology

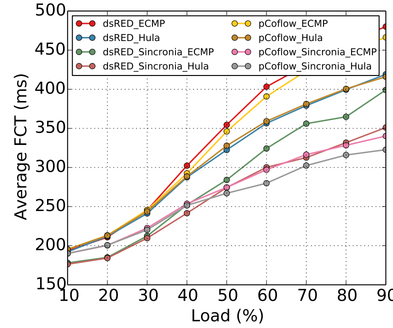

Fig. 10. Average FCT for fat-tree topology

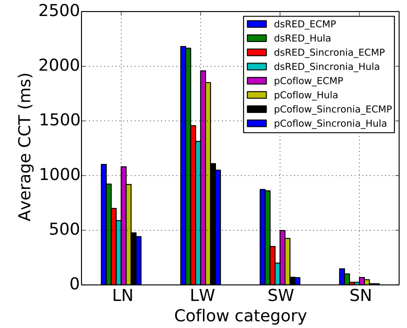

Fig. 11. Average CCT for coflows sorted by category for 90% load

space in the queue whenever there is space left from other coflows. In Figure 8), we observe that allowing coflows to temporally exceed the priority band limit of 500 packets, we can reduce the overall CCT. This is due to DCTCP which reacts upon the ECN marking. All remaining experiments for *pCoflow* are performed with adaptive queues and ECN marking. Figure 6 and Figure 7 show the average CCT and FCT for different combinations of load-balancing with and without Sincronia ordering. *pCoflow* improves both upon Sincronia and when not using Sincronia. When not using Sincronia, the benefits of *pCoflow* are attributed to the adaptive queue size. *pCoflow* improves upon multi-level dsRED queues when using Sincronia since we avoid reordering and leverage the full capacity of the queue. At higher load, the benefits of *pCoflow* are more pronounced, where the gap between our approach and the vanilla SP multi-level dsRED queue is in the range of 15-25%. This might stem from the fact that at high loads the traffic is more unstable, leading to more changes in priorities and therefore more reordering. However, as Figure 7 shows, by changing the insertion order of the packets, this can lead to an increase in the tailing of some flows and therefore leading to an increase of the FCT compared to the multi-level dsRED queue for low loads. For network loads higher than about 70%, *pCoflow* also reduces FCT compared to other approaches.

Fattree - Summary: Figure 9 and Figure 10 show the average CCT and FCT for the Fattree topology. The lowest CCT is achieved by *pCoflow* when used with HULA (see Figure 9). When using Sincronia, the difference between ECMP and HULA is not significant since HULA probe packets are used to identify least congested paths. Consequently, we map them to the highest priority queue or band. This can lead to a situation where low priority packets being forwarded to a more congested path. Moreover, Facebook traffic is characterized by having a one-to-many communication pattern, where a single node receives data from different nodes in the network. At low loads, the bottleneck might be located mainly on the links between the ToR and the server and therefore load-balancing plays a minor role. On the other hand, we can see that when we do not use Sincronia, the effect of load-balancing is more pronounced due to the congestion-aware load-balancing by HULA. When using Sincronia, *pCoflow* combined with HULA achieves a CCT reduction up to 27% compared to the fixed dsRED multi-level queuing when used with HULA. On the other hand, *pCoflow* can reduce CCT by 34% compared to dsRED multi-level queues when used with Sincronia and ECMP. Figure 11 analyzes the CCT for the 90% load case for each coflow category. As expected, long and wide (LW) coflows contribute to the highest CCT. This is because they are the coflows that transport the largest data volume. In addition, Sincronia benefits small coflows, as they have a higher probability to be assigned to a higher priority band. Surprisingly, the load-balancing scheme plays a less important role for large flows than expected, which may be caused by the unawareness of HULA of coflow properties. This leaves

room for an integrated design with *pCoflow*.

V. Related Work

There is extensive work related to scheduling coflows in data centers. Most of these schemes including [10], [14], [13], [17], [16] rely on prior knowledge about coflows (e.g. flow sizes, server pairs). Coflow scheduling methods can be divided into two groups: *distributed schedulers* and *centralized schedulers*. Distributed schemes including [17], [18], [32] are executed on each host where coflows are scheduled and sorted locally. On the contrary, centralized schemes such as [13], [14], [15], [16] rely on a central controller to order coflows. Indeed, having a global view enables better scheduling decisions [33] while facing a a large control overhead and, therefore, scalability is an issue. Aalo [28] uses priority queues to classify coflows according to the amount of data sent and does not need prior knowledge. Sincronia [10] overcomes the main centralized schedulers' problems by avoiding per-flow rate allocation. Sincronia achieves near-optimal average CCT and requires a transport layer proritizing flows according to coflow orderings. While most of the flows do not consider coflow routing, Rapier [13] and [34] demonstrate that combining scheduling and routing can lead to better performance.

VI. Conclusions and Future Work

We presented *pCoflow*, an in-network support to coflow scheduling. Our work integrates state-of-the-art end-host coflow reordering approaches with in-network packet prioritization performed in the switch. Our approach uses the PIFO scheduling abstraction to build a coflow aware packet scheduler which considers packet history during priority scheduling. *pCoflow* therefore avoids excessive packet reordering that potentially lead to wasteful restransmissions. Our approach improves upon coflow completion time and benefits from flowlet-based load-balancing schemes such as HULA. As future work, we aim for an integrated design by defining extensions and proper interactions between *pCoflow* and flowlet-based load-balancing schemes such as HULA.

Acknowledgment

Parts of this work is supported by the Knowledge Foundation of Sweden through the Profile HITS under Grant No.: 20140037.

References

[1] J. Dean *et al.*, "MapReduce: simplified data processing on large clusters," in *Communications of the ACM, Volume: 51, Issue: 1*. ACM, 2008.

[2] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," in *Hot Topics in Cloud Computing (HotCloud)*. USENIX Association, 2010.

[3] M. Isard *et al.*, "Dryad: distributed data-parallel programs from sequential building blocks," in *SIGOPS Operating Systems Review, Volume: 41, Issue: 3*. ACM, 2007.

[4] H. Kumar *et al.*, "Machine Intelligence at the NOC," in *Ericsson Blog*, 06 2018, [Online; accessed 21-April-2020].

[5] L. V. Le *et al.*, "SDN/NFV, Machine Learning, and Big Data Driven Network Slicing for 5G," in *2018 IEEE 5G World Forum (5GWF)*, 11 2018.

[6] V. Nejkovic *et al.*, "Big Data in 5G Distributed Applications," in *High-Performance Modelling and Simulation for Big Data Applications: Selected Results of the COST Action IC1406 cHiPSet*, J. Kołodziej *et al.*, Eds. Cham: Springer International Publishing, 2019, pp. 138–162.

[7] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012.

[8] M. Chowdhury *et al.*, "Coflow: a networking abstraction for cluster applications," in *Hot Topics in Networks (HotNets)*. ACM, 2012.

[9] N. Katta *et al.*, "Hula: Scalable load balancing using programmable data planes," in *Symposium on SDN Research (SOSR)*. ACM, 2016.

[10] S. Agarwal *et al.*, "Sincronia: Near-optimal Network Design for Coflows," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

[11] P. Bosshart *et al.*, "P4: Programming Protocol-independent Packet Processors," vol. 44, no. 3. New York, NY, USA: ACM, jul 2014, pp. 87–95.

[12] A. Sivaraman *et al.*, "Programmable Packet Scheduling at Line Rate," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[13] Y. Zhao *et al.*, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *International Conference on Computer Communications (INFOCOM)*. IEEE, 2015.

[14] M. Chowdhury *et al.*, "Efficient coflow scheduling with Varys," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.

[15] Y. Li *et al.*, "Efficient online coflow routing and scheduling," in *Mobile Ad Hoc Networking and Computing (MobiHoc)*. ACM, 2016.

[16] M. Chowdhury *et al.*, "Managing data transfers in computer clusters with orchestra," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2011.

[17] S. Luo *et al.*, "Minimizing average coflow completion time with decentralized scheduling," in *International Conference on Communications (ICC)*. IEEE, 2015.

[18] H. Susanto *et al.*, "Stream: Decentralized opportunistic inter-coflow scheduling for datacenter networks," in *International Conference on Network Protocols (ICNP)*. IEEE, 2016.

[19] M. Alizadeh *et al.*, "pfabric: Minimal near-optimal datacenter transport," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.

[20] ——, "Data Center TCP (DCTCP)," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.

[21] ——, "CONGA: Distributed congestion-aware load balancing for datacenters," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.

[22] S. Floyd, "TCP and Explicit Congestion Notification," in *Computer Communication Review, Volume: 24, Issue: 5*. ACM, 1994.

[23] H. Wu *et al.*, "Tuning ecn for data center networks," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012, pp. 25–36.

[24] W. Bai *et al.*, "Enabling {ECN} in multi-service multi-queue data centers," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 537–549.

[25] Y. Zhu *et al.*, "Congestion Control for Large-Scale RDMA Deployments," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.

[26] A. Alcoz *et al.*, "SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues ," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 59–76. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/alcoz

[27] S. Agarwal *et al.*, "Coflow workload generator," https://github.com/sincronia-coflow, 2018.

[28] M. Chowdhury *et al.*, "Efficient Coflow Scheduling Without Prior Knowledge," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.

[29] M. Alizadeh *et al.*, "Data Center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851192

[30] A. G. Alcoz *et al.*, "Sp-pifo: Approximating push-in first-out behaviors using strict-priority queues," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 59–76.

[31] M. A. Qadeer *et al.*, "Differentiated services with multiple random early detection algorithm using ns2 simulator," in *International Conference on Computer Science and Information Technology (ICCSIT)*. IEEE, 2009.

[32] F. R. Dogar *et al.*, "Decentralized task-aware scheduling for data center networks," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.
[33] S. Wang *et al.*, "A survey of coflow scheduling schemes for data center networks," in *Communications Magazine, Volume: 56, Issue: 6*. IEEE, 2018.
[34] H. Jahanjou *et al.*, "Asymptotically optimal approximation algorithms for coflow scheduling," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2017.