Accepted to appear in IEEE/ACM Transactions on Networking, 2022

Meeting Coflow Deadlines in Data Center Networks with Policy-based Selective Completion

Shouxi Luo, Pingzhi Fan, Huanlai Xing, Hongfang Yu

***Abstract*—Recently, the abstraction of *coflow* is introduced to capture the collective data transmission patterns among modern distributed data-parallel applications. During processing, coflows generally act as barriers; accordingly, time-sensitive applications prefer their coflows to complete within deadlines, and deadline-aware coflow scheduling becomes very crucial.**

Regarding these data-parallel applications, we notice that many of them, including *large-scale query systems*, *distributed iterative training*, and *erasure codes enabled storage*, are able to tolerate loss-bounded incomplete inputs by design. This tolerance indeed brings a flexible design space for the schedule of their coflows: when getting overloaded, the network can trade coflow completeness for the timeliness, and balance the completeness of different coflows on demand. Unfortunately, existing coflow schedulers neglect this tolerance, resulting in inflexible and inefficient bandwidth allocations.

In this paper, we explore this fundamental trade-off and design POCO, a POlicy-based COflow scheduler, along with a transport layer enhancement scheme, to achieve customizable selective coflow completion for emerging time-sensitive distributed applications. Internally, POCO employs a suite of novel designs along with admission controls to make *flexible*, *work-conserving*, and *performance-guaranteed* rate allocation to online coflow requests very efficiently. Extensive trace-based simulations indicate that POCO is highly flexible and achieves optimal coflow schedules respecting the requirements specified by applications.

***Index Terms*—Coflow, data center networks, flow scheduling**

I. INTRODUCTION

In modern cloud data centers, distributed data-parallel applications such as Hadoop, Spark, and EC-Cache, are widely employed to build large-scale data processing, analysis, and storage services [1–3]. In these systems, a job is split into multiple staged tasks carried out by a cluster in distributed manners. During processing, involved servers trigger groups of parallel, collective flows to move intermediate results from machines of the current stage to the next. These flows in the same group are abstracted as a *coflow* since they share the same performance goal and their completions act as the barrier of the distributed computation [2, 4]. For time-sensitive applications like web search, retail, recommendation systems, etc., the triggered coflows are generally bound with deadlines, implying the dates by which they should be finished [4, 5]. To deal with these transfers, existing deadline-aware (co)flow scheduling proposals directly reject a request if its deadline can not be met [4, 6], or admit all requests then dynamically preempt large-sized, less-emergency transfers in service to increase the amount of deadline-satisfied requests heuristically, without performance guarantee [5, 7, 8]. Unfortunately, such designs are proven to be sub-optimal for many emerging distributed applications.

Due to the approximate nature of the involved distributed computation [9, 10], or the redundant design employed for data transmission [3, 11], many of today's distributed applications are able to tolerate incomplete transmissions by design. For instance, in *large-scale query systems* like web search and advertisement selection, for each cache-missed request, a group of worker servers will report then aggregate their top-N results to generate the final response; a partial data transmission is acceptable to the application since it is a sample for the whole data thus still bringing benefits to the application [9]. Likewise, during the *distributed iterative training* of modern machine learning models, besides the tolerance of incomplete training data, models like *deep neural network* based image classification and natural language understanding, are robust to achieve comparable convergence rate over incomplete parameter updates [10]—Actually, the recent empirical study of [12] shows that many machine learning algorithms are bounded-loss tolerant; their end-to-end job performance would get little impacts in case that the randomized network data loss is below a certain fraction (typically 10%~35%). And for applications like *erasure codes enabled distributed storage system*, on object reads, because of the redundant self-coding designs, obtaining any k out of $(k + r)$ splits of the object residing in the cluster, are sufficient to serve the request [3, 11].

All the above observations demonstrate the ubiquity of tolerance on incomplete inputs among emerging distributed applications. Recently, by using this type of tolerance, Liu *et al.* propose a protocol with controlled packet loss called ATP, to perform approximate data transmission for approximate application [13]; Xia *et al.* design BTP, a Bounded-loss Tolerant transport Protocol, to remove the tail latency for the parameter synchronization process of distributed model training [12]. However, both ATP and BTP are oblivious of the coflow semantic among flows; their per-flow based designs are proved to be sub-optimal for the schedule of coflow [4]. To support incompleteness-tolerant coflow scheduling, Im *et al.* employ greedy designs to maximize the partial throughput of coflow [9]. However, the proposed Con-Myopic algorithm is unaware of the application requirements in terms of the

The work of Shouxi Luo was supported in part by NSFC under Project 62002300, in part by NSFSC under Project 2022NSFSC0944, and in part by the China Postdoctoral Science Foundation under Project 2019M663552. The work of Pingzhi Fan was supported in part by NSFC under Project 62020106001 and in part by the 111 Project (111-2-14). (*Corresponding author: Shouxi Luo*, e-mail: sxluo@swjtu.edu.cn)

Shouxi Luo, Pingzhi Fan, and Huanlai Xing are with Southwest Jiaotong University, Chengdu 611756, China.

Hongfang Yu is with the University of Electronic Science and Technology of China, Chengdu 611731, China.

The preliminary version of this work titled "Selective Coflow Completion for Time-sensitive Distributed Applications with Poco" is published in the 49th International Conference on Parallel Processing (ICPP) [1].

exact (coflow) completeness and timeliness. As a result, Con-Myopic provides no performance guarantee to time-sensitive applications. Moreover, Con-Myopic assumes that the data transmitted by one flow cannot be replaced by another. This is not always true as the aforementioned applications show counter-examples. For those applications, the data transmitted by all or portions of the flows in a coflow is exchangeable. Accordingly, the completeness of these flows is described by the total volume they deliver successfully. In these cases, the schedule of Con-Myopic is inflexible and inefficient.

In summary, emerging time-sensitive data-parallel applications are common to tolerate incomplete yet loss-bounded inputs. This brings an important yet overlooked design space for the schedule of their deadline-bounded coflows: in case the network is overloaded thus impossible to complete all tasks in time, we could trade coflow completeness for timeliness and trade one coflow's completeness for those of others.

This work. In this paper, we explore the fundamental trade-off between the time a coflow could take to complete and the completeness it would achieve. As different applications generally have various requirements on the completeness and timeliness of coflow, we extend the barrier definition of coflow to support partial completion and develop POCO, a POlicy-based COflow scheduler along with a transport layer enhancement scheme, to achieve customizable selective completion for them. To provide guaranteed performances, POCO involves admission controls for coflows arriving online. At the high-level, it provides a set of policy primitives, with which, distributed applications can precisely define their requirements of both the expired time and minimum completeness along with each coflow. Then, at the low-level, POCO translates these requirements into time-slotted linear constraints and formulate a Linear Program (LP) to solve. If the corresponding LP is infeasible, POCO rejects the request; otherwise, any feasible result of the problem yields a bandwidth allocation to admit the new request without sacrificing the requirements of others. With an affiliated flow based transport layer enhancement scheme, POCO would control the coflow traffic to occupy the network to complete respecting their schedules gracefully.

However, building LPs for the selective-completion schedule of coflow and solving them for rate scheduling are quite challenging. Firstly, coflow requests arrive online; although a coflow's detailed requirements would be available upon its arrival, it is impossible to get that information ahead of time [14, 15]; thus, greedily allocating all available bandwidth to admit an incoming request would be unfair to future requests, resulting in unfairness among their applications. Secondly, as we will show, the bandwidth allocation suggested by the LP might be non-work-conserving; POCO should not directly use the raw results for rate controls. Thirdly, as an online scheduler, the solving of involved LPs must be efficient.

To address these challenges, POCO *i*) employs a tunable model to control the level at which bandwidth in the future is allocated in admission control; *ii*) designs a post-processing to make work-conserving bandwidth allocations; *iii*) merges variables to compact the model, and more essentially, *iv*) develops a parallelizable core to speed up the LP solving by making use of the specific constraint structures of the problem.

Limits of POCO. As a centralized scheduler, POCO introduces scheduling delays. In practice, a coflow's actual duration depends on both the available network bandwidth and the amount of data it should deliver. For those small coflows that could complete within a very short time (e.g., one or two RTTs), the scheduling delay of POCO might be not negligible thus POCO could not help. In practice, there also exist many distributed applications like BSP-based distributed machine learning and user-facing approximate bigdata analytics whose triggered coflows are bulk and such delays are acceptable [4, 5, 16]. Poco is mainly designed for them.

Contributions. To sum up, we make these contributions:

- An analysis of the design space and desired proprieties of deadline-aware, loss-bounded coflow schedulers (§II).
- A high-level coflow abstraction along with an LP model that enables applications to express completeness requirements for deadline-sensitive coflows (§IV-A, §IV-B).
- A suite of schedule designs to compress the model size and make fair yet work-conserving bandwidth allocations for coflows incoming online (§IV-C, §IV-D).
- A transport enhancement scheme to carry out the rate schedule of POCO gracefully together with packet-level simulations confirm its benefits (§V).
- Extensive packet-level and flow-level evaluations assess the feasibility and effectiveness of POCO (§VI, §VII).

II. POCO GUIDELINES

A. Design Space

Consider that a group of flows $\mathcal{F}_e$ go through the same bottleneck link e with the capacity of c_e, and assume that the sending rate of flow f at time t is $r_f(t)$ ($r_f(t) \geq 0$). Obviously, as (1) shows, the volume that f can deliver before time t is determined by the integration of its allocated sending rate $r_f(t)$ over time, which is restrained by the rates allocated to all other flows (i.e., $\sum_{f' \in \mathcal{F}_e \setminus \{f\}} r_{f'}(t)$) in turn as (2) indicates—To avoid congestions, the aggregated rate of flows going through the same link should not exceed its capacity. Motivated by this, we obtain a foundational design space for the schedule of coflow: in a heavily-loaded network, by taking advantage of the application's tolerance of incomplete inputs, we can *i*) trade the achieved completeness for shorter completion times, and *ii*) trade one flow's completeness for those of others. Moreover, if the data delivered by a group of flows within a coflow ($\mathcal{F}_g$ for instance) is exchangeable, the network can balance the total task (ϕ_g for instance) among its sub-flows with respect to the network loads as (3) indicates.

$$v_f = \int_0^t r_f(t)dt \tag{1}$$

$$r_f(t) + \sum_{f' \in \mathcal{F}_e \setminus \{f\}} r_{f'}(t) \leq c_e \tag{2}$$

$$\sum_{f \in \mathcal{F}_g} v_f \geq \phi_g \tag{3}$$

B. Desirable Properties

By exploring the aforementioned trade-offs, POCO performs selective coflow completions for time-sensitive applications. To be practical, it must realize the following design goals.

Performance guarantees. First of all, to ensure the progress of distributed computation, applications usually have limited levels of tolerance. Hence, POCO should provide a service model with performance promises to applications. This property restricts POCO to be centralized since a global view of the entire system is needed for performance-guaranteed coflow scheduling.

High flexibility. Second, different applications are likely to have various performance requirements. Accordingly, POCO needs to be flexible enough to support various requirements.

Fairness. Third, coflows arrive online; the requirements of future coflow requests are agnostic ahead of time. Greedily allocating all available bandwidth to admit requests is unfair to future arrivals [17]. Thus, POCO should support configurable admission control.

Work-conservation. Fourth, to fully utilize the network and serve more requests, POCO is required to be work-conserving. That is to say, a link sits idle only if there is no traffic demand.

Scalability. Last but not least, as an online scheduler, POCO must decide whether to admit a request and schedule all flow sending rates to guarantee their performances effectively. For this purpose, the algorithms employed by POCO must run in real-time with low time complexity.

C. Reasonable Assumption

Before looking into the design details, in this part, we discuss the reasonable assumptions that POCO is built on.

Fine-grained centralized traffic control. As Section III will explain, POCO achieves performance-guaranteed coflow scheduling by precisely controlling the rates of all involved flows from its centralized controller. A recent study has demonstrated that, by taking advantage of the well-structured property of modern data center network topology, it is possible to control the rate of data center traffic precisely with a centralized controller at flowlet- or packet- levels [18, 19]. Thus, the centralized design of POCO is practical. Indeed, in POCO, the controller only needs to manage the rate of concerning traffic in a time-slotted manner; and the involved rate schedule is executed on the events of task arrival and completion. Moreover, to make sure that there is always enough bandwidth to implement the rate schedules computed by POCO, we can *i*) let POCO's controller perform rate allocations only on a portion (e.g., 95%) of the link/network capacity, and *ii*) assign a high priority to these scheduled flows [19, 20]. Then, rate allocations are with guarantees and background traffic would use the remaining bandwidth gracefully. As we will show in Section V, by employing rate limits and launching affiliated flows, POCO is able to make full use of the network respecting the scheduled rates.

Requirements are available upon task arrival. In line with numerous prior studies [4, 21, 22], we assume that a coflow's detailed requirements such as task volume, structure, remaining deadline, and level of required completeness, are known on its arrival and would not change over time. It is true that not all distributed applications hold the assumption [23]; but fortunately, many emerging production cluster applications are witnessed to have this property in practice [4, 12, 14, 15, 24]. For instance, the work of [24] empirically shows that many advanced data analytics jobs have predictable computation and communication structures. Likewise, it is reported that in some large-scale production data clusters, more than 60% of tasks are recurring and their task requirements can be estimated within low errors [25, 26]. Indeed, researchers have demonstrated that, for specific cluster computing applications, due to their specific designs, it is not hard to predict their traffic demands with high accuracy. As an example, for Mapreduce jobs on Hadoop and Spark, there already exist several tools to predict their inner traffic demands [14, 15]. Similarly, in emerging synchronized distributed machine learning applications, training servers generally iterate over the same dataset up to thousands of rounds; at each round, all the involved training servers would synchronize their local model updates with or without parameter servers, triggering periodic traffic patterns. Such application properties not only make their coflow demands knowable on their arrivals, but also enable operators and developers to measure and quantify *i*) the level at which the application could tolerate the incompleteness of inputs [12], and *ii*) the best deadline/latency that is needed for the optimization of revenue [4, 5]. Finally, as for the route of each flow, there already exist many tools to infer [18] or control [19, 27] on its packet arrivals.

III. POCO OVERVIEW

As Figure 1 sketches, POCO employs admission controls to provide promises of completeness and deadlines for coflows in the online scenario. On getting an incoming request, if POCO finds a way to meet its completeness- and deadline-requirements without violating those of any existing coflow, this new request could be admitted and a corresponding bandwidth allocation is already found. Otherwise, the request would get rejected; the application could either cancel the request, or relaxes its requirements then resubmits again. As we will show, with novel designs, POCO provides performance guarantees to admitted coflows in two basic dimensions—

- **Completeness**: flows involved in a coflow would deliver a certain percentage/amount of volume to meet pre-specified completeness requirements; and
- **Timeliness**: all the promised completeness is achieved within the given deadline.

Note that, once a coflow is admitted, POCO would insist on satisfying its completeness and deadline requirements. This never means that the bandwidth allocated to it is irrevocable. Indeed, to admit a new request, POCO is able to recycle some of the time-slotted bandwidth resources already allocated back on demand, provided the requirements of involved existing coflows would not be violated. To do so, POCO formulates the rate scheduling problem as a time-slotted LP, which guarantees that a request would be admitted if and only if a reasonable

rate schedule is obtained. Such a process is triggered upon the arrival of new coflow requests. Thus, in theory, by ensuring that the deadline-bounded time-slotted bandwidth resource each (co)flow obtain is not less than those suggested by the LP's solution, all admitted coflows would eventually achieve their required completeness within deadlines.

During the transmission, the actual rate that a flow would send at is generally larger than the values computed from the LP, due to two reasons as §IV-C will explain in detail. On one hand, to be fair to coflows arriving in the future, POCO would systematically limit the allocation of link capacities in future time slots when performing admission controls. And on the other, any feasible solution to the LP yields admissions; hence, the rate schedule suggested by the solution does not necessarily guarantee all the link capacities would be employed (a.k.a., work-conservation), even if the previous fairness-related design is disabled. Accordingly, to achieve work-conserving rate scheduling, POCO further adjusts the planned flow rates in pipelines, as Figure 1 summarizes.

In production, distributed applications controlled by POCO might coexist with many other applications, which generate both uncertain foreground and background traffic. Accordingly, only limiting the sending rates of POCO flows is not enough to deal with the network dynamics and make full use of the perishable available link capacity. To address the problem, POCO employs a readily-deployable transport layer enhancement scheme: i.e., for each transfer in a coflow, set up an affiliated subflow together with a rate-limited subflow, and assign them with increasing priorities to enhance the scheduling and make full use of the residual bandwidth.

Next, we describe how POCO builds then solves LPs efficiently to achieve performance-guaranteed, flexible, fair, and work-conserving coflow rate scheduling (§IV), then explain how an affiliated subflow based protocol enhancements help POCO carry out the scheduling gracefully (§V), in detail.

IV. SCHEDULER DESIGN

In this section, we first introduce the coflow abstraction (§IV-A) along with the network model (§IV-B) POCO provides to capture the flexible requirements raised by application, then describe the optimization designs that POCO adopts to achieve *fairness*, *work-conservation*, and *scalability* (§IV-C), and finally sketch out the specific block-angle structure residing in the generated LPs (§IV-D). By using these structures, advanced LP solvers could achieve accelerated solving. The design and optimization details of such solvers are beyond the scope of this paper; we refer the readers to [1, 28, 29].

A. Coflow Abstraction

As Figure 2 summarizes, POCO abstracts a coflow request, saying C_i for instance, by the set of its involved flows $\mathcal{F}_i = \{f_{i,1}, f_{i,2}, \cdots\}$, and the group of its associated completeness requirements $\mathcal{R}_i = \{\cdots, (G_{i,k}; \phi_{i,k}), \cdots\}$. Compared with the original coflow abstraction proposed by [2, 4], POCO mainly extends the coflow model to support partial completion. For the j-th subflow in $\mathcal{F}_i$, i.e., $f_{i,j}$, its task is to transmit data with remaining volume $v_{i,j}$ via established path $p_{i,j}$ within expired

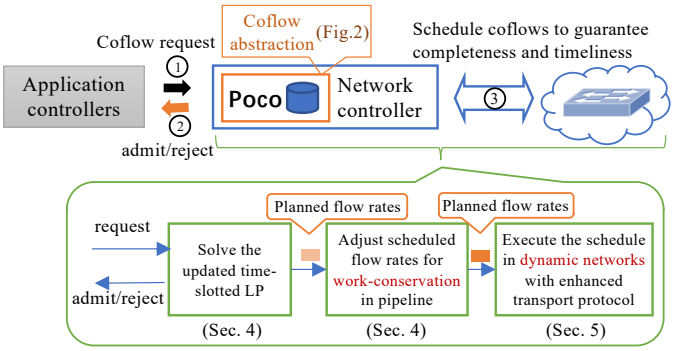


Fig. 1. The service model and internal workflow of POCO, in which the solving of time-slotted LP is triggered upon the arrival of coflow requests.

Grammar

C_i	$::= (\mathcal{F}_i; \mathcal{R}_i)$	Application-specified coflow request
$\mathcal{F}_i$	$::= \{\cdots, f_{i,j}, \cdots\}$	Transfer demands of cofow C_i
$\mathcal{R}_i$	$::= \{\cdots, (G_{i,k}; \phi_{i,k}), \cdots\}$	Completeness requirements
$f_{i,j}$	$::= (\tau_{i,j}; v_{i,j}; p_{i,j})$	Details of the j-th subflow in coflow i

More Notation

$\tau_{i,j}$	:	Expired time of flow $f_{i,j}$ (we have $\forall j : \tau_{i,j} = \tau_i$ in this paper)
$v_{i,j}$	:	Remaining volume of flow $f_{i,j}$
$p_{i,j}$	:	Path of flow $f_{i,j}$
$G_{i,k}$	:	Set of flow(s) in the k-th completeness group of coflow i
$\phi_{i,k}$	:	The k-th completeness requirement of coflow i

Fig. 2. Syntax of the coflow abstraction provided by POCO.

time $\tau_{i,j}$. Although our model allows $\tau_{i,j}$ vary among flows, in practice, a coflow represents a task, and thus flows belonging to the same coflow generally share the same deadline τ_i. In case the network is overloaded and a very strict hard deadline is desired, it is impossible to make full transmissions of all flows within their deadlines. Then, POCO makes selectively loss-bounded partial completions. The k-th restriction in $\mathcal{R}_i$ given by the application specifies that the total completed volume of flows in $G_{i,k}$ should not be less than $\phi_{i,k}$.

Obviously, the abstraction provided by POCO is very expressive. With it, applications can specify coflow requests along with both timeliness- and completeness- requirements easily. For transfers without deadlines, POCO simply treats them bound with a very loose expired time. And for coflows unable to tolerate incompleteness, POCO uses their exact volumes as the completeness requirements.

B. Network Model

Without loss of generality, consider that there are $n-1$ accepted yet uncompleted coflow requests, labeled $C_1, \cdots, C_{n-1}$, and the new incoming request to check is C_n. We assume that bandwidth is allocated in time slots with length Δ_T and we denote the rate of flow $f_{i,j}$ during time slot t by $r_{i,j,t}$. Then, the problem of finding a bandwidth allocation to admit request R_n and meet its requirements without violating those of others is straightforward to be formulated as the system of linear inequalities shown in (4). Here, $v_{i,j}$ and $\phi_{i,k}$ are the updated remaining flow size and uncompleted completeness volume requirement, respectively; $c_{e,t}$ denotes the available capacity of link e at time slot t that can be allocated to requests now.

$$(4)\begin{cases} \sum_{(i,j)\in G_{i,k}} \sum_{t=1}^{\tau_{i,j}} r_{i,j,t}\Delta_T \geq \phi_{i,k}, \quad \forall i,k & (4a) \\ \sum_{t=1}^{\tau_{i,j}} r_{i,j,t}\Delta_T \leq v_{i,j}, \quad \forall i,j & (4b) \\ \sum_{(i,j):e\in p_{i,j}} r_{i,j,t} \leq c_{e,t}, \quad \forall e,t & (4c) \\ r_{i,j,t} \geq 0, \quad \forall i,j,t & (4d) \end{cases}$$

It is obvious that, if constraints in (4) are infeasible, the request must be rejected; otherwise, any feasible $\{r_{i,j,t}\}$ satisfying (4) yields a bandwidth allocation that accepts C_n.[1]

C. Scheduling Algorithm

On getting a request, the straightforward design of POCO is to *i*) formulate the associated bandwidth allocation problem as an LP by introducing trivial objectives such as maximizing the total completed volume to the constraints of (4), as (5) shows;[2] then *ii*) employ off-the-shelf optimizer to solve, and *iii*) finally perform the admission control and rate scheduling based on the results. However, such a design is impractical, since *i*) the bandwidth of time slot in future might be over-allocated, resulting in unfairness to future arrivals; *ii*) more seriously, the rate schedule given by the LP does not guarantee work-conservation; and last but not least, *iii*) the LP involves too many variables, making the model solving time costly.

$$\text{Maximize} \sum_{i=1}^{n} \sum_{j=1}^{|F_i|} \sum_{t=1}^{\tau_{i,j}} r_{i,j,t}\Delta_T \quad s.t.\ (4) \tag{5}$$

Fairness. To be fair to future coflow arrivals, POCO systematically limits the allocation of link capacities in future time slots on performing admission control. Suppose that link e is with the capacity of c_e; motivated by the design of [17], POCO lets $c_{e,t} = c_e\beta(t)$, in which $\beta(t) = \min(1, exp(-(t-t_*)/t_o))$, t_* and t_o are two tunable parameters, receptively. By tuning them, POCO can control the level at which future link capacities are allocated. Note that, to be work-conserving in practice, for admitted requests, POCO should allocate all link capacities to serve until they complete or expire.

Work-conservation. As we will show, the rate schedules suggested by LPs do not guarantee work-conservation, even if fine-grained timeslots are employed and a very large t_* is used in $\beta(t)$. For instance, consider that two coflows C_i and C_j will appear at times 0 and 1, then expire at the same time 2, respectively. Accordingly, let Δ_T be one unit of time; then there are two time slots, t_1 with range $[0,1)$ and t_2 with range $[1,2)$. Suppose that each of these two incoming coflows involves only one subflow, saying $f_{i,1}$ and $f_{j,1}$, going through the same bottleneck link with capacity 2. The total volume and completeness requirement of $f_{i,1}$ are 2 and 1, respectively, while those of $f_{j,1}$ are 3 and 2, respectively. At time 0, the corresponding LP for the admission control of coflow C_i is as (7) shows. By solving the problem with either simplex or interior-point method, we might get the result of $r_{i,1,1} = 0, r_{i,1,2} = 2$ (indeed, this is exactly the solution given by Mosek 8.1.67 [31], a commercial off-the-shelf LP solver), yielding a bandwidth allocation to admit coflow C_i. However, such a schedule is not work-conserving since no traffic occurs in slot t_1. As a result, at time 1, coflow C_j would get rejected since there does not exist enough bandwidth to guarantee its requirements. For this specific instance, it is possible to achieve work-conserving bandwidth allocation by assigning decreasing weights to slotted rates in the objective (e.g., $\sum_{i=1}^{n}\sum_{j=1}^{|F_i|}\sum_{t=1}^{\tau_{i,j}}\frac{r_{i,j,t}}{\tau_{i,j}}$). However, such a design is impractical as POCO might under-allocate bandwidth in future slots on admission control for fairness.

$$(6)\begin{cases} 1 \leq r_{i,1,1} + r_{i,1,2} \leq 2 & (6a) \\ 0 \leq r_{i,1,1} \leq 2 & (6b) \\ 0 \leq r_{i,1,2} \leq 2 & (6c) \end{cases}$$

$$\text{Maximize} \quad r_{i,1,1} + r_{i,1,2} \quad s.t.\ (6) \tag{7}$$

To address this, POCO employs a post-processing to adjust the rate schedule given by LP. Basically, if there is remaining bandwidth in earlier slots, POCO greedily moves parts of a slot's task up, until no movement can be made. In the end, a work-conserving rate schedule is obtained. In case there are multiple flows going through the same under-loaded link, flows with unmet completeness requirements would occupy the available bandwidth before those whose completeness requirements are already satisfied; and for either requirement-unmet or met flows, residual slotted link capacities are allocated to them fairly or in non-decreasing order of their deadlines. Revisit the schedule of coflow i shown in (7) as an example, with the post-processing for work-conservation, its scheduled rates will be updated from $r_{i,1,1} = 0 \wedge r_{i,1,2} = 2$ to $r_{i,1,1} = 2 \wedge r_{i,1,2} = 0$.

Scalability. Because of the fine-grained slotted bandwidth allocation, the model involves a large number of variables, taking non-trivial time for LP solvers to deal with. In addition, the aforementioned process for work-conservation might also introduce significant delays since the number of slots to be checked could be huge. To overcome these, *i*) POCO lets flows that already meet their completeness requirements have the rate of 0 for model pruning and merges successive time slots between flow expiration events into a single one. Then, *ii*) POCO employs advanced solvers [1, 28, 29] to solve the compacted LP in parallel by leveraging the specific structure of its constraints. Finally, *iii*) POCO modifies the results given by the LP solver to make work-conserving adjustments. Next, we describe how POCO merges time slots and performs post progresses and leave the detail of why advanced solvers could be employed to §IV-D.

[1] To mitigate the impact of transmission delays, we can set either the deadline or completeness requirements involved in the model slight stricter than their original values.

[2] In this paper, POCO employs the objective of maximizing the total completed volume as a case study. With standard reformulation techniques [30], it is easy to extend POCO to support other types of schedules like maximizing the minimal gain of achieved completeness in a max-min fashion: i.e., Maximize $\min_{\forall i,k} \frac{1}{\phi_{i,k}} \sum_{(i,j)\in G_{i,k}} \sum_{t=1}^{\tau_{i,j}} r_{i,j,t}\Delta_T$.

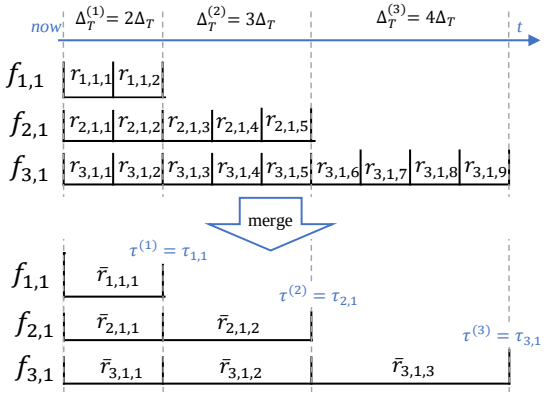


Fig. 3. An example showcases how time-slotted variables are merged. There are three flows, $f_{1,1}$, $f_{2,1}$, and $f_{3,1}$, with deadlines $2\Delta_T$, $5\Delta_T$ and $9\Delta_T$, respectively. After merging variables, the number of time-slotted rate variables are reduced from $2+5+9=16$ to $1+2+3=6$, which is independent of either flow lifespans or the width of configured time slot.

On performing admission controls, if no flow expires from time slot t_1 to t_2, it is reasonable to assume that all flow rates keep consistent during the interval, without impacting either the feasibility or optimality of the problem. Such a design is equivalent to adding the constraints of $r_{i,j,t}=r_{i,j,t-1}$ for unexpired flows $\{f_{i,j}:\forall\tau_{i,j}>t\}$ and slots $\{t:\nexists\tau_{i,j}=t\}$. Following this, merged time slots are with various lengths. Suppose that flows involve N diverse deadlines and let $\Omega(k)$ be the number of flows involving the k-th deadline; then, the number of merged rate variables would be reduced to $\sum_{k=1}^{N}k\Omega(k)$, which is independent of either flow lifespans or the setting of slot width. This would yield up to several orders of magnitude improvement in terms of the solving time. Denote the l-th expired time as $\tau^{(l)}$ and let $\tau^{(0)}$ be 0. Then, for the l-th merged time slot, starting from $\tau^{(l-1)}$ to $\tau^{(l)}$, the corresponding available capacity of link e for admission control, is corrected as $\beta_l c_e$, in which β_l is the corresponding correction factor of link capacity defined in (10). Accordingly, the problem of (5), along with its constraints (4), can be reformulated to (9), where $\pi_{i,j}$ is the interval index of the diverse expired time $\tau_{i,j}$ (11), $\Delta_T^{(l)}$ is the width of the l-th merged slot (12), and $\bar{r}_{i,j,l}$ is the rate of $f_{i,j}$ during that interval. For instance, consider that there are three flows namely $f_{1,1}$, $f_{2,1}$, and $f_{3,1}$, belonging to diverse coflows. Their deadlines are $2\Delta_T$, $5\Delta_T$, and $9\Delta_T$, respectively. As Figure 3 shows, with variable merging, POCO could reduce the number of time-slotted rate variables from 2+5+9 = 16 to 1+2+3 = 6.

$$
(8)\begin{cases}
\sum\limits_{(i,j)\in G_{i,k}}\sum\limits_{l=1}^{\pi_{i,j}}\Delta_T^{(l)}\bar{r}_{i,j,l}\geq\phi_{i,k}, & \forall i,k & (8a)\\
\sum\limits_{l=1}^{\pi_{i,j}}\Delta_T^{(l)}\bar{r}_{i,j,l}\leq v_{i,j}, & \forall i,j & (8b)\\
\sum\limits_{(i,j):e\in p_{i,j}}\bar{r}_{i,j,l}\leq\beta_l c_e, & \forall e,l & (8c)\\
\bar{r}_{i,j,l}\geq 0, & \forall i,j,l & (8d)
\end{cases}
$$

$$
\text{Maximize}\ \sum_{i=1}^{n}\sum_{j=1}^{|F_i|}\sum_{l=1}^{\pi_{i,j}}\bar{r}_{i,j,l}\Delta_T^{(l)}\quad s.t.\ (8) \tag{9}
$$

$$
\beta_l \coloneqq \frac{\sum_{t=\tau^{(l-1)}}^{\tau^{(l)}-1}\beta(t)}{\tau^{(l)}-\tau^{(l-1)}+1} \tag{10}
$$

$$
\pi_{i,j} \coloneqq \operatorname*{argmax}_{l}\{l:\tau^{(l)}\leq\tau_{i,j}\} \tag{11}
$$

$$
\Delta_T^{(l)} \coloneqq (\tau^{(l)}-\tau^{(l-1)})\Delta_T,\ l=1,\cdots,N \tag{12}
$$

Moreover, in case the number of coflows is extremely huge, we can fix the completion of some coflows in the LP solving to further reduce the number of involved variables, thus accelerating the solving process further.

As for the post bandwidth adjustment, POCO repeats to fully fill available slotted link capacities by moving parts of a slot's task up until no link capacity is left. Since the result of LP yields a bandwidth allocation to the future, POCO could work in the pipelines by keeping making work-conserving adjustments for slots only in the near future to keep low process delay.

D. Efficient Solver

To support large-scale selective coflow scheduling, the LP solver of POCO must be very efficient. According to the theory of linear optimization [30], Problem (9) can be reformulated into its matrix format as (13) shows. Here T stands for the operator of transpose; and $\boldsymbol{A}$, the constraint matrix, has the multiple-level primal block-angular structure as (14) specifies, which can be employed by advanced interior-point methods based solver to parallelize the costly algebraic operations during the solving, yielding orders of accelerations by making use of the abundant cores in modern servers. Now, we reformulate the problem of (9) to identify its block-angle structure.

$$
\text{Minimize}\ \ \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}\quad s.t.\ \boldsymbol{A}\boldsymbol{x}=\boldsymbol{b} \tag{13}
$$

$$
\boldsymbol{A}\boldsymbol{x}=\begin{bmatrix}\boldsymbol{A}_1 & & & \\ & \boldsymbol{A}_2 & & \\ & & \ddots & \\ & & & \boldsymbol{A}_n \\ \boldsymbol{B}_1 & \boldsymbol{B}_2 & \cdots & \boldsymbol{B}_n\ \boldsymbol{I}\end{bmatrix}\begin{bmatrix}\boldsymbol{x}_1\\ \boldsymbol{x}_2\\ \vdots\\ \boldsymbol{x}_n\\ \boldsymbol{x}^s\end{bmatrix}=\begin{bmatrix}\boldsymbol{b}_1\\ \boldsymbol{b}_2\\ \vdots\\ \boldsymbol{b}_n\\ \boldsymbol{b}^*\end{bmatrix} \tag{14}
$$

To identify the block-angular structure of $\boldsymbol{A}$, let us revisit the constraints specified in (8). Notably, the corresponding requirements to admit a new coflow request are made of two classes: *i*) meeting each coflow's minimum transfer demands within their deadlines (i.e., (8a)) without excusing the maximum size of each flow (i.e., (8b)); and *ii*) making sure that no link gets overloaded through the transmitting (i.e., (8c)). Let Γ be the set of all time-slotted link bandwidth resources that would be used by coflows, and further denote the corresponding link and time index of the o-th time-slotted link resource κ_o as κ_o^e and κ_o^t, respectively. Then, the associated time-slotted link capacity POCO could use for admission control now can be computed by $\beta_{\kappa_o^t}c_{\kappa_o^e}$. By using $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$ to indicate these two types of constraint matrices for coflow i, respectively, the linear constraints shown in (8) can be reformed as a primal block-angular structure as (14)

shows, in which $\boldsymbol{x}^s$ is the vector of slack variables for the constraints of link capacity at each time interval, and the details of $\boldsymbol{x}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{b}^*$ follow in (15), (16), and (17).

$$\begin{aligned}\boldsymbol{x}_i \coloneqq [&(\bar{r}_{i,1,1}, \cdots, \bar{r}_{i,1,\pi_{i,1}}, \hat{r}^+_{i,1}),\\ &\cdots,\\ &(\bar{r}_{i,|F_i|,1}, \cdots, \bar{r}_{i,|F_i|,\pi_{i,|F_i|}}, \hat{r}^+_{i,|F_i|}),\\ &\hat{r}^-_{i,1}, \cdots, \hat{r}^-_{i,|\mathcal{R}_i|}]^\mathsf{T}\end{aligned} \tag{15}$$

$$\boldsymbol{b}_i \coloneqq \left[v_{i,1}, \cdots, v_{i,|F_i|}, \phi_{i,1}, \cdots, \phi_{i,|R_i|}\right]^\mathsf{T} \tag{16}$$

$$\boldsymbol{b}^* \coloneqq \left[\beta_{\kappa^t_1} c_{\kappa^e_1}, \cdots, \beta_{\kappa^t_{|\Gamma|}} c_{\kappa^e_{|\Gamma|}}\right]^\mathsf{T} \tag{17}$$

As (15) shows, $\boldsymbol{x}_i$ is made up of the rate allocations of flows belonging to coflow i, along with a few slack variables $\{\hat{r}^+_{i,j}\}$ and $\{\hat{r}^-_{i,j}\}$. Likewise, as (18) indicates, the constraint matrix involved by each coflow also follows exactly the similar pattern of block-angular structure, in which, $\psi_{i,k,j}$ is either 1 or 0, indicating whether the k-th requirement $G_{i,k}$ involves $f_{i,j}$ or not. As for $\boldsymbol{B}_i$, constant $\hbar_{o,i,j,l}$ is either 1 or 0, indicating whether $f_{i,j}$ goes through link κ^e_o during time interval κ^t_o (23). Here, we also have $\kappa^t_o \equiv l$.

$$\boldsymbol{A}_i \coloneqq \left[\begin{array}{cccccc|cc} \boldsymbol{a}_{i,1} & 1 & & & & & & \\ & \boldsymbol{a}_{i,2} & 1 & & & & & \\ & & & \ddots & & & & \\ & & & & \boldsymbol{a}_{i,|F_i|} & 1 & & \\ \hline \psi_{i,1,1}\boldsymbol{a}_{i,1} & 0 & \cdots & \cdots & \psi_{i,1,|\mathcal{F}_i|}\boldsymbol{a}_{i,1,|F_i|} & 0 & -1 & \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \ddots \\ \psi_{i,|R_i|,1}\boldsymbol{a}_{i,1} & 0 & \cdots & \cdots & \psi_{i,|R_i|,|\mathcal{F}_i|}\boldsymbol{a}_{i,|F_i|} & 0 & & -1 \end{array}\right] \tag{18}$$

$$\boldsymbol{a}_{i,j} \coloneqq [\Delta_T^{(1)}, \cdots, \Delta_T^{(\pi_{i,j})}] \tag{19}$$

$$\psi_{i,k,j} \coloneqq \begin{cases} 1 & (i,j) \in G_{i,k} \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

$$\boldsymbol{B}_i \coloneqq \left[\begin{array}{ccc|c|c} \cdots & \hbar_{1,i,j,1} \cdots \hbar_{1,i,j,\pi_{i,j}} & 0 & \cdots & 0 \\ \ddots & \vdots \quad \ddots \quad \vdots & \vdots & \ddots & \vdots \\ \cdots & \hbar_{|\Gamma|,i,j,1} \cdots \hbar_{|\Gamma|,i,j,\pi_{i,j}} & 0 & \cdots & 0 \end{array}\right]_{j=1,\cdots,|F_i|} \tag{21}$$

$$\boldsymbol{B}_i \coloneqq \left[\begin{array}{ccc|c|c|c} \hbar_{1,i,1,1} \cdots \hbar_{1,i,1,\pi_{i,1}} & 0 & \cdots & \cdots & \hbar_{1,i,|F_i|,1} \cdots \hbar_{1,i,j,\pi_{i,|F_i|}} & 0 & 0 \\ \vdots \quad \ddots \quad \vdots & \vdots & \ddots & \ddots & \vdots \quad \ddots \quad \vdots & \vdots & \vdots \\ \hbar_{|\Gamma|,i,1,1} \cdots \hbar_{|\Gamma|,i,1,\pi_{i,1}} & 0 & \cdots & \cdots & \hbar_{|\Gamma|,i,|F_i|,1} \cdots \hbar_{|\Gamma|,i,|F_i|,\pi_{i,|F_i|}} & 0 & 0 \end{array}\right] \tag{22}$$

$$\hbar_{o,i,j,l} \coloneqq \begin{cases} 1 & \kappa^e_o \in p_{i,j} \wedge l \le \pi_{i,j} \\ 0 & otherwise \end{cases} \tag{23}$$

Regarding the value of $\boldsymbol{w}$ shown in (13), for rate variables involved in l-th merged time slots, their w_is are set to $-\Delta_T^{(l)}$, and for other slack variables, their w_is are set to 0.

V. Transport Layer Enhancement

The naive design to meet the QoS requirements of admitted coflows is limiting the sending rates of their flows respecting the scheduling strictly [32]. However, in practice, many applications coexist in clusters; some of them are not under the control of Poco and might have non-negligible impacts on Poco flows. Thus, only limiting Poco flows' rates is not enough. Moreover, to guarantee the completion of admitted coflow, the link capacities Poco employs for the LP-based coflow scheduling are conservative. Accordingly, the actual bandwidth coflows are able to use might be larger than the promised, hence a simple rate-limiting would prevent coflows from making full use of the network. In this section, we describe how Poco could address the problem elegantly with a readily-deployable transport layer enhancement scheme (§V-A), and verify its effectiveness with packet-level simulations (§V-B).

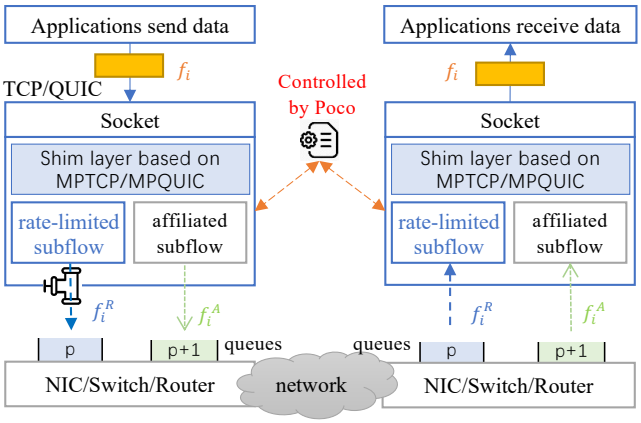


Fig. 4. At the low level, Poco would launch a rate-limited subflow along with an affiliated subflow for each Poco flow task.

A. Enhancement Designs

Similar to prior arts [32, 33], the transport layer enhancement of Poco involves two key designs. Firstly, by making use of the priority queues supported by modern data center networks, it lets different types of traffic make prioritized use of link capacities, such that the interference between them is controlled. Secondly, based on prioritized bandwidth allocation, for each Poco flow in a coflow, besides limiting its sending rates according to the schedule, Poco also launches an *affiliated* subflow with a lower priority to make full use of the residual bandwidth as Figure 4 illustrates. Indeed, as simulation results in §V-B and §VI-B will show, the design of *affiliated* flow not only makes Poco able to take advantage of dynamically available bandwidth but also tolerable to foreground bursts.

Basically, we can classify flows that are out of the control of Poco into two categories: i.e., *foreground* flows that are more important than Poco flows, and *background* flows that are less important than Poco flows. Figure 4 sketches how Poco takes advantage of the priority queues as well as how it launches two subflows to deliver f_i's data segments, cooperatively. To achieve the design goal, these subflows occupy two priority queues: the rate of subflow f_i^R is limited respecting the scheduling strictly and it delivers data segments using the priority queue of p; while the affiliated subflow f_i^A would deliver diverse data segments using the priority queue of $p+1$, which is used by other background traffic as well. Note that, the rate of f_i^A is not explicitly scheduled. Accordingly, all these affiliated flows, along with other background flows in the network, would share the possible residual bandwidth

fairly. Regarding foreground flows, they could use a priority equal to or higher than p. Following this way, the impacts of POCO flows on foreground flows and the impacts of background flows on POCO flows are reduced and controlled. Since switches in modern data center networks generally support 4-8 priority queues per port, such a design is readily-deployable.

As for the implementation, both the widely supported MPTCP (Multipath TCP) [33] and emerging MPQUIC (Multipath QUIC) [34] have already provided the ability to launch and manage multiple subflows for the delivery of a data stream cooperatively; we can extend them to achieve the transport layer enhancements required by POCO with techniques like eBPF. Regarding the rate-limiting, POCO can directly employ the widely available software-based solutions like Traffic Control (TC) [35], which can work with many of existing transport protocols like DCTCP, QUIC, etc. Thus, POCO is readily implementable and deployable. The full implementation and advanced optimization of the enhanced transport protocol are beyond the scope of this paper and we leave it as future work.

B. Behavioral Study

To verify the effectiveness of the aforementioned POCO protocol enhancement, we conduct packet-level simulations to study the detailed behavior of POCO-scheduled (co)flows in dynamic networks. The simulator is written in Python 3 and its behaviors are proven to be consistent with the results of both math model analysis and Mininet implementations. A more detailed description of its design can be founded at [11].

Firstly, let us consider a simple case in which two POCO flows f_1 and f_2 go through the same link with the capacity of 10×10^4 pps (packets per second), latency of 1ms, and their sending rates are scheduled to (24a) and (24b), respectively. Each port of the link is equipped with 4 priority queues and scheduled by a Deficit Weighted Round Robin (DWRR) controller with the associated weights of 1000, 100, 10, and 1. For each queue, it is able to hold 800 data packets at most and would mark en-queuing packets with ECTs once its queue occupancy reaches the threshold of 160 data packets. Besides POCO traffic, a background long-lived flow f_B appears at time 0.1s and another foreground flow f_F involving 400 packets appears at time 0.25s. Table I summarizes the simulation settings, in which the foreground flow f_F uses the highest/first priority, rate-limited subflows f_1^R and f_2^R employ the second priority, and both the affiliated subflow f_1^A and f_2^A along with the background flow f_B share the third priority. As for the transport layer protocol, all flows employ DCTCP.

$$r_1(t) = \begin{cases} 7 \times 10^4 pps & \text{if } \exists i \in N : 0 \le t - 0.2i < 0.1 \\ 2 \times 10^4 pps & \text{otherwise} \end{cases} \quad (24a)$$

$$r_2(t) = \begin{cases} 2 \times 10^4 pps & \text{if } \exists i \in N : 0 \le t - 0.2i < 0.1 \\ 7 \times 10^4 pps & \text{otherwise} \end{cases} \quad (24b)$$

Figure 5 shows the observed goodput of each (sub)flow. According to their rate schedules (24a) and (24b), there is about 10% link capacity left slack for possible foreground and background traffic. Obviously, with affiliated subflows f_1^A and f_2^A, POCO is able to make full use of the link bandwidth when there are no foreground and background flows. Let $n_i(t)$ and $\hat{n}_i(t)$ be the number of packets that are expected to be delivered and actual acked by POCO flow f_i; we denote the value of $\hat{n}_i(t) - n_i(t)$ as the *number of absent packets* for this task at time t, and count the change of those of f_1 and f_2 over time. For $\hat{n}_i(t) - n_i(t)$, the smaller value is better; and a negative value indicates that the actually sent data volume is larger than expected. As Figure 6a implies, despite the window of POCO flows collapses because of packet loss, their affiliated subflows enable f_1 and f_2 to tolerate the impacts of foreground bursty traffic f_F easily. Noticeably, because of transmission and acknowledgment delays, the number of absent packets is larger than 0 at the very beginning. In contrast, as Figure 6b shows, the naive rate-limiting scheme without affiliated subflows is impractical, since their number of absent packets would increase for each foreground burst. Besides DCTCP, we also use the protocol of TCP-Reno to rerun the tests and obtain consistent results, implying that the transport layer enhancement design of POCO is generic.

TABLE I
SETTINGS OF THE PACKET-LEVEL SIMULATION SHOWN IN FIGURE 5.

Flow	Description
f_1	A POCO flow task whose rate is scheduled following (24a).
f_1^R	The rate-limited subflow of POCO flow f_1.
f_1^A	The affiliated subflow of POCO flow f_1.
f_2	A POCO flow task whose rate is scheduled following (24b).
f_2^R	The rate-limited subflow of POCO flow f_2.
f_2^A	The affiliated subflow of POCO flow f_2.
f_F	A foreground flow with 400 packets starting at time 0.25s.
f_B	A background long-lived flow starting at time 0.1s.

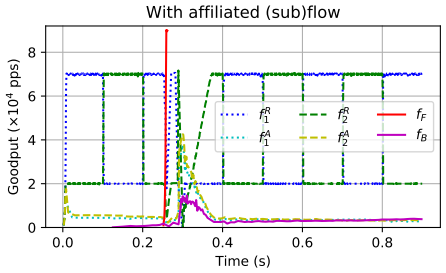
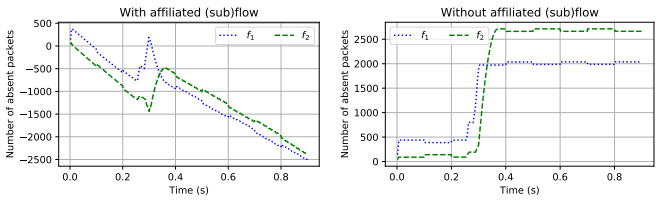
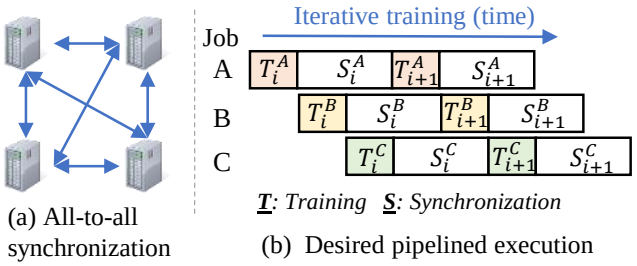


Fig. 5. The transport layer enhancement enables POCO to tolerate network dynamics and make full use of available bandwidth. Refer to Table I for simulation settings.



(a) With affiliated (sub)flow (b) Without affiliated (sub)flow

Fig. 6. Compared with its native implementation, affiliated flows enable POCO to tolerate burst traffic and make full use of available bandwidth.



(a) All-to-all synchronization

(b) Desired pipelined execution

Fig. 7. An example showcases the utility of POCO: there are three iterative training jobs, namely A, B, and C; during each round of training, each of them first makes exclusive use of the entire cluster then performs all-to-all model synchronizations (see (a)). To achieve fair and efficient use of the cluster, operators configure these three jobs to use exactly the same amount of time for each round of training computation and prefer their followed all-to-all data transmission to be completed in deadlines, such that the entire cluster could be fully used with pipelined scheduling as (b) shows.

VI. PACKET-LEVEL SIMULATION BASED CASE STUDIES

To showcase POCO's utility, in this section, we employ it to orchestrate coflows for concurrent Distributed Machine Learning (DML) jobs through packet-level simulations. Results confirm that, by exploring the tolerance of DML and trading transfers' completeness for deadlines, POCO enables concurrent training jobs to make perfect use of the cluster.

A. Demands and Settings

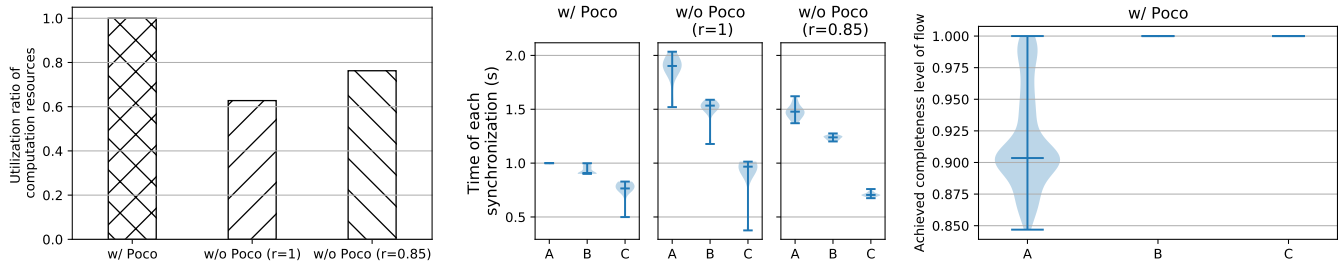
Let us consider the case that three distributed machine learning jobs A, B, and C, use a shared cluster involving 4 workers for data-parallel model training as Figure 7 shows. In every round of training, each job would make exclusive use of the entire cluster, followed by an all-to-all synchronization among the involved workers. To make fair and efficient use of the cluster, operators configure all jobs to use exactly the same amount of training times in each round and execute in pipelines so that one job could perform the synchronization when others are training using the cluster [36]. To do so, the coflow involved in synchronization should complete in deadlines; otherwise, the training would be blocked by the slow synchronization, resulting in under-utilizations of the cluster. We implement a simulator based on the one mentioned before to precisely simulate the behavior of such a training system under the schedule of POCO or not. For all jobs, the training is blocked until the dependent previous synchronization completes. When no POCO is employed, workers in the cluster directly launch DCTCP flows to conduct their all-to-all synchronizations.

As a concrete example, we assume that these 4 workers are networked to the same top-of-rack switch with bidirectional 1Gbps links. Each port at the switch support 4 priority queues scheduled by the DWRR controller using the weights of 1000, 100, 10, and 1, respectively. By default, foreground flows occupy the first queue, then the rate-limited flows for POCO-controlled (co)flows occupy the second one, and the affiliated flows together with the background flows share the third queue. To reduce queuing latency, the ECN marking threshold of each queue is 10KB. For jobs A, B, and C, we suppose that it takes 0.5s for each worker to conduct a round of training, and the data each worker needs to synchronize between every other partner worker is 30MB, 20MB, and 10MB, respectively. To achieve the perfect scheduling, operators would like every synchronization to complete within 1s. Regarding the tolerance of partial data transmission, based on the empirical study of [12], each worker is assumed to deliver at least 85% of the total data it should send, and obtain at least 85% of the data it was supported to receive. By default, all jobs are configured to conduct 100 rounds of training.

B. Simulation Results

Effectiveness of POCO scheduling. Figure 8 shows the observed utilization ratio of the cluster with and without the schedule of POCO during the training. Here, we assume that there is neither foreground nor background traffic in the cluster, thus, the protocol enhancement of POCO is disabled. Obviously, POCO successfully ensures that all synchronizations achieve their completeness requirements within deadlines, resulting in perfect and 100% use of the entire cluster. By contrast, without POCO the cluster is under-loaded (about 63%) because the training computations of both jobs A and B always be blocked by their slow synchronizations as Figure 8b demonstrates: i.e., the time costs of their synchronizations are always larger than the desired deadline of 1s. Furthermore, even when all flows are relaxed to only transmit 85% of its data volume, the training computations for both A and B are still blocked by slow synchronization, yielding the cluster utilization of about 77%. We also look into the detail of transfers' achieved completeness under the schedule of POCO. As Figure 8c shows, only flows triggered by job A do not achieve 100% completeness; indeed, most flows triggered by A deliver about 90% of their data, larger than the minimum completeness requirement of 85%. Results also show that the achieved completeness of several flows is slightly smaller than 85%; their completeness requirements are still satisfied because some other flows under the same volume constraint have contributed more, indicating the fact that POCO does trade some flows' completeness for that of others on demand.

Effectiveness of protocol enhancement. To study the effect of the proposed protocol enhancement, we consider a similar training scenario but reset all link capacities to 1.1Gbps, among which 90% are promised to the POCO scheduler for coflow scheduling. Besides (co)flows triggered by jobs A, B, and C, we assume that there is exactly another long-lived background flow, along with a burst foreground flow appearing at 7.01s, coexisting on each link. We increase the foreground flows' sizes until any job's synchronization could not achieve the completeness requirements within deadlines. We find that, when protocol enhancement is not enabled, due to the DWRR scheduling, the background long-lived flows make job A's synchronization missing its completeness requirements slightly, even when there is no foreground flow. To avoid this issue, we must limit the aggregated rate of background flows to their allocated rate of 1.1Gbps*0.1 = 0.11Gbps. Such a design removes the impacts of background traffic on POCO-controlled coflows, but also prevents them from making full use of other available bandwidth. Figure 9 shows the maximum amount of



(a) The cluster's utilization ratio (b) Distributions of the synchronization time costs (c) Distributions of flows' achieved completeness

Fig. 8. POCO enables concurrent training jobs A, B, and C, to make full use of the entire cluster as expected. Without POCO and even if every flow only completes 85% of its data, the cluster would be underloaded because the training computation would be blocked by the unscheduled, slow synchronization. In contrast, in POCO, most flows achieve 100% completeness, and only flows triggered by job A realize the completeness of about 90%.

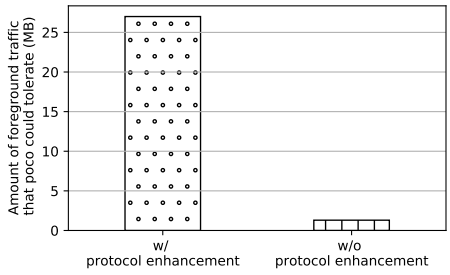


Fig. 9. The transport layer enhancement enables POCO to tolerate the impacts of foreground and background traffic greatly. For instance, with and without protocol enhancement designs, the size of a burst foreground flow that the POCO-enabled concurrent model training can tolerate are about 27MB and 1.3MB, respectively, yielding a performance gap larger than 20× in this case.

foreground flow that POCO-controlled DML synchronizations can tolerate, with and without the proposed protocol enhancement. Results show that observed thresholds are 27MB and 1.3MB, respectively, yielding a gap larger than 20×.

VII. FLOW-LEVEL EVALUATION

In this section, we evaluate POCO through trace-based simulations. We compare it with state-of-the-art deadline-aware coflow schedulers Varys [4], Con-Myopic [9], and the default baseline Fair-Sharing (FS). Extensive results indicate that POCO is flexible and robust to make very efficient tolerance-aware coflow scheduling:

1) POCO lets more coflows meet their requirements by trading the achieved completeness for the timeliness, and trading one coflow's completeness for those of others;
2) its scheduling algorithm makes very effective use of the network to provide guaranteed performance to time-sensitive coflow, outperforming that of the state-of-the-art Varys up to 1.25× and even more (the performance gain depends on the instance's settings).

A. Methodology

Workload. The coflow workloads employed in evaluations are generated using a coflow workload generator following the design provided by Varys [4]. In short, it unsamples the Facebook traces to the desired number of coflows, network load, cluster scale, etc., while keeping workload characteristics similar to the original Facebook trace. However, the Facebook trace does not involve the attribute requirements of deadline and completeness. In common with prior work [4, 5], for each coflow C_i, we set its deadline constraint to be $(1+z)\rho_i$, where ρ_i is the minimum completion time of coflow i in an empty network, and z is a random number following the uniform distribution $U[0;2x]$. As for the completeness requirements, we assume that each involves the requirements: $\mathcal{R}_i : \{(\mathcal{F}_i, \alpha_i \sum_{j=1}^{|\mathcal{F}_i|} v_{i,j})\}$, where α_i varies from 0 to 1. Unless mentioned otherwise, our tests use the baseline of 300 coflows, 1.0 network load, 0.9 completeness requirements; and x, the scale factor of the deadline (i.e., the mean of z), is set as 1.

Cluster. We find that simulations imply consistent results under diverse cluster scales. To reduce the simulation time, we consider a cluster comprising 60 servers here. In common with recent work, the entire cluster network is abstracted out as a non-blocking switch [4, 19], which interconnects all machines with 1 Gbps access links.

Flow-level simulator. As the packet-level study in §V-B has shown, by employing priority queues and affiliated flows, POCO would presciently control the sending rate of POCO flows respecting the rate schedules. To speed up the tests, instead of performing packet-level simulations, similar to that of Varys, we further develop flow-level simulators (in Python 3) to perform detailed replays of the aforementioned coflow traces, according to the scheduling policy of FS, Varys, Con-Myopic, and the proposed POCO, respectively. In short, FS is the max-min fair sharing policy adopted by TCP and its variations. Varys is the state-of-the-art deadline-guaranteed coflow scheduler, which performs admission controls by letting coflows finish exactly at their deadlines, then adjusting sending rates to achieve work-conservation [4]. Con-Myopic is the only existing scheduler designed to support partial completions; it greedy schedules coflows to maximize their marginal partial throughput without considering their exact deadlines [9]. POCO admits coflow requests based on the results of (9), then adjusts flow rates to achieve work-conservation. As for the back-end solver of POCO, our current implementation is a simple prototype based on Scipy and written in Python, slow than these highly-optimized commercial

LP solvers like Mosek and Gurobi. It is not unreasonable to speculate that by integrating our optimization designs, these commercial LP solvers could do even better. To accelerate the simulations, we mainly employ the Mosek solver as the core here. In all tests, rejected requests do not get resubmitted.

Theoretically, a fine-grained slotted model would ensure more efficient use of the network. However, smaller slots would increase the running time of the simulation greatly. Given that POCO is designed for bulk coflow tasks and following the design of [37], we suggest the use of $O(100)$ ms slots and let Δ_T be 500 ms in our simulation. As for the tunable parameter t_* and t_o for the control of available link capacity in future, we let t_* be $\frac{\tau_*}{(u\Delta_T)}$, and t_o be 1000, respectively, where τ_* is the 85-percentile of the involved coflows' ideal completion times and u is the average network load, both of which can be inferred from served coflows in practice.

Metrics. Regarding the performance metrics, we mainly consider the percentage of coflows that meet their requirements of deadline and completeness. For specific test cases, we also consider the (normalized) completed volume and achieved completeness under various scheduling schemes. For each parameter setting, we perform 8 trials.

B. Effectiveness

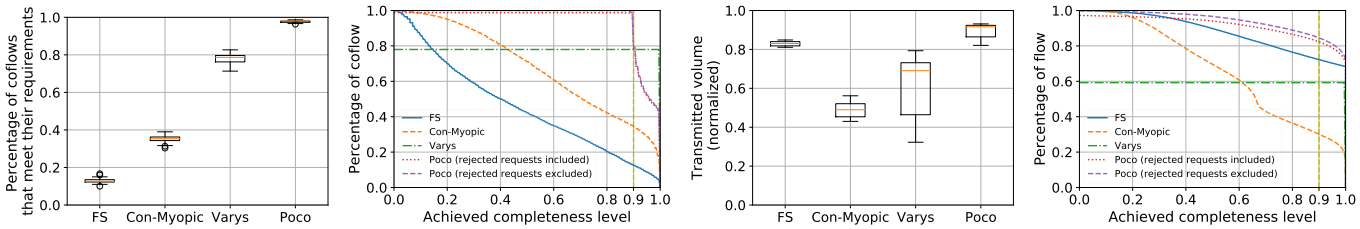
Case study. As Figure 10a shows, under the default parameter settings, FS, Con-Myopic, and Varys let about 13.1%, 35.2%, and 78.0% coflows meet their completeness and deadline requirements, respectively. In contrast, the average percentage achieved by POCO is 97.7%, yielding a performance gain of 7.46×, 2.78×, and 1.25×, respectively. For these test cases, Figure 10b gives their detailed Complementary Cumulative Distribution Function (CCDF) curves of all coflow requests. Recall that both POCO and Varys employ admission controls to provide performance guarantees; accordingly, their curves involve line segments. However, Varys neglects the tolerance nature of applications and always makes full completion for admitted coflow, resulting in performance loss compared with POCO. Regarding FS and Con-Myopic, they work poorly since the agnosticism of application requirements. Meanwhile, we also observe that the schedule of Con-Myopic does not guarantee work-conservation, since it is designed to maximize the marginal partial throughput at each slot [9].

To ascertain their performance details, we also count the total transmitted volume in each case (normalized by the total volume of all requests, Figure 10c) and the achieved completeness of each flow (Figure 10d). Obviously, POCO makes very efficient use of the network as it transmits nearly 89.4% of all the volume, accounting for about 92.6% of the total volume of the coflow requests it admits. As for Varys, it makes 100% deliveries for all the coflows it admits, accounting for about 62.8% of the total requested volume. A very interesting observation is that FS only lets about 13.1% of requests meet their requirements, however, its transmitted volume reaches 83.0% of the total. Such results imply that maximizing the network goodput/throughput does not necessarily optimize the completion of coflow. Thus, to perform efficient coflow scheduling, the awareness of both completeness and deadline is a must for the scheduler. As an example, POCO enables more coflow requirements to be met by trading completeness for timeliness and trading one coflow's completeness for those of others on demand. The illustration shown in Figure 10d confirms the awareness and flexibility of POCO: all admitted coflows do satisfy the completeness requirements of 0.9, yet at the flow-level, less than 86% of the admitted flows achieve the completeness level of 0.9 for their own tasks.

Impact of completeness. To investigate the impact of completeness, we change each α_i, the required completeness level, from 1 to 0.6, then rerun the tests and check the percentage of coflows that could meet their requirements under various schedule schemes. As Figure 11a illustrates, the results of Varys keep consistent, because it is unaware of the tolerance of completeness thus always performing 100% completions for all admitted requests. Conversely, with the relaxation of required completeness, all other three schemes schedule more coflows to meet their requirements. Especially, POCO is able to admit and satisfy all the requests, once their required completeness level is less than 0.8. Such results imply the ability of POCO on performing tolerance-aware scheduling, again. Moreover, we find that POCO still outperforms Varys by about 5%, even when all coflows require 100% completions. That is to say, the rate schedule algorithm adopted by POCO always makes more efficient use of the bandwidth than that of Varys. This is reasonable since the schedule of POCO is built upon LP and POCO obtains the optimal results in polynomial time. Besides, the results of FS and Con-Myopic also reveal that their percentages of met coflow increase linearly with the decrease of the required completeness level. This phenomenon is consistent with the observed distribution shown in Figure 10b.

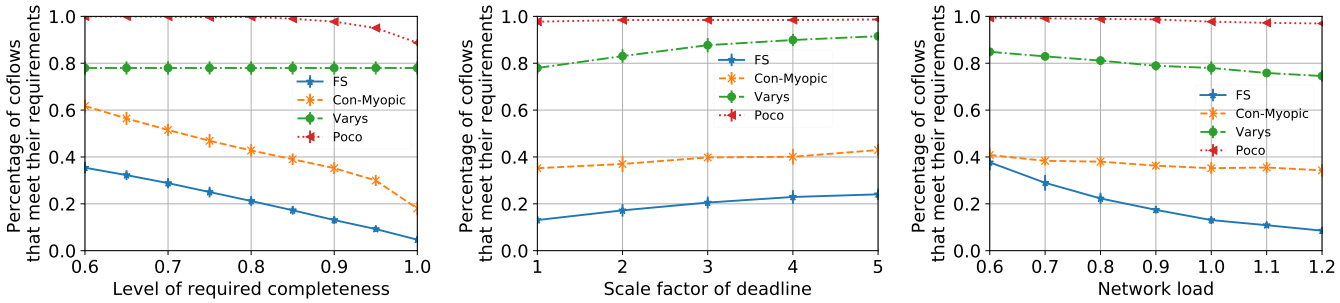
Impact of deadline. As the other requirement dimension of a coflow request, we then test how the number of requirement-satisfied coflow changes if coflows have looser deadlines. To this end, we increase x, i.e., the mean value of z, or the so-called scale factor of the deadline, from 1 to 5. As Figure 11b reveals, for all schemes but POCO, a significantly increased amount of coflows would meet their requirements when their deadlines get relaxed. However, the results of POCO have little change. This is reasonable since the relaxation of the deadline would not reduce the network load indeed. As POCO has already made very efficient use of the network to admit the request, there is little room to improve.

Impact of network load. Next, we test the change of requirement-satisfied coflow under various network loads. According to the trace generator, the tested coflow requests are assumed to arrive in a Poisson process whose rate is λ. We vary the network load from 0.6 to 1.2 by controlling the rate parameter λ. Because a coflow will get expired automatically after its deadline, there would be only a limited amount of coflows to each server even if the network load runs into a load value larger than 1. As Figure 11c indicates, for all schedule schemes, the percentages would reduce with the increase of network load, consistent with the fact that more requests will get completely served if the network load is light. We also



(a) Requirement-satisified coflow (b) Achieved completeness (c) Transmitted volume (d) Achieved completeness of flow

Fig. 10. The details of Fair-Sharing (FS), Con-Myopic, Varys, and POCO on scheduling coflows with 0.9-completeness and "$x = 1$"-deadline requirements.



(a) Impact of completeness requirement (b) Impact of deadline requirement (c) Impact of network load

Fig. 11. Different from the significant performance degradation of FS, Con-Myopic, and Varys, the percentage of coflows admitted by POCO only decreases slightly with the increase of required completeness and network load. As well, POCO always outperforms all other scheduling algorithms greatly.

notice that once the network load is under 0.6, POCO would let all requests meet their requirements simultaneously. This reflects that POCO does make very efficient rate allocations.

Impact of the fairness parameter. To study the effectiveness of the configurable fairness parameter, we increase the arrival rate of coflows several times to make the network be overloaded, and vary the value of parameter τ_* from 0.6 to 1. Obviously, a smaller τ_* means fewer link capacities in the future could be allocated to admit the current incoming coflow requests, i.e., fairer to future requests. Since coflows used in these tests are with skewed sizes [4], this might result in improved admission rates. As expected, under several-fold overloaded scenarios, with τ_*'s value growing, the achieved amount of admitted coflows decrease slowly. That is to say, the design of controlled temporal fairness takes effect. However, in case the network load is light, the results keep almost the same. This is reasonable since the issue of temporal fairness occurs only when the network is heavily loaded.

Impact of skewed completeness requirements. By default, for coflow i, we assume that it involves the completeness requirement of $\mathcal{R}_i : \{(\mathcal{F}_i, \alpha_i \sum_{j=1}^{|\mathcal{F}_i|} v_{i,j})\}$. To study whether the skewness in completeness requirements would impact the effectiveness of POCO scheduling, for the j-th member flow in coflow i, we further add the "per-flow" completeness requirement of $\{(\{f_{i,j}\}, (\alpha_i + y(1-\alpha_i))v_{i,j})\}$, where y is a number randomly chosen from the uniform distribution of $U[-s, s]$ and $0 \le s \le 1$ is a tunable parameter indicating the level of skewness. We observe that, compared with the case of no per-flow completeness requirement, the percentage of coflows that meet their requirements under the schedule of POCO has a slight degradation, about 0.039 (0.025, respectively) when α_i is 0.9 (0.95, respectively). However, with the value of s increases from 0 to 1, the results stay the same, indicating that the skewness in completeness requirements has little effect on the performance of POCO scheduling.

VIII. Related Work

Since the seminal work of Chowdhury and Stoica [2, 4], researchers from both academia and industry have proposed a large number of algorithms to optimize the completion of coflow, such as minimizing their completion times [4, 22, 38], reducing the missed deadlines [4, 5], providing isolation guarantees [39], *etc* [9, 40–42]. As POCO aims at providing flexible rate scheduling for deadline-constrained coflow, we focus on the most related work here and refer the reader to [21] and [43] for comprehensive surveys.

Basically, existing scheduling mechanisms designed for deadline-sensitive coflows can be classified into two classes, respecting whether performance guarantees are provided or not. In the former case, network schedulers mainly involve admission controls to ensure that serving the incoming request would not hurt the deadlines of others [4, 6]. However, a straightforward admission control design (e.g., Varys [4]) would result in unfairness, since admitting a big request might over-allocate the bandwidth in the future. In the latter case, best-effort coflow deliveries are acceptable, several preemptive algorithms such as D^2-CAS [5], Chronos [7], and OLPA [44] have been proposed. In this paper, POCO is designed to provide guaranteed yet flexible performances and it achieves fairness with tunable admission control. By putting POCO in

the context of customizable and intent-based network design, we are not the first who design flexible schedulers for coflow. For this type of purpose, Chen *et al.* developed a utility-based scheduling model to support diverging application requirements, in which, the formulation of utility function describes the performance goals desired by applications [41]. Nevertheless, all existing coflow scheduling study overlooks the tolerance of emerging distributed applications. They are designed based on the *all-or-nothing* service model, resulting in loss of flexibility and performance. Different from them, we explore this type of tolerance [10, 12, 13, 45] and design POCO to trade completeness for timeliness on demand.

The recently proposed Con-Myopic [9] supports partial coflow completions. However, it schedules coflows without considering their exact deadlines thus providing no performance guarantees. Such a characteristic limits its applicability since the performance would be unpredictable. The protocol of ATP proposed by Liu *et al.* [13] and the loss-bounded protocol proposed by Xia [12] perform approximate data transmission for application. However, they focus on designing new transport protocols and do not consider the possible coflow traffic pattern triggered by applications; thus, they are inefficient in performing controllable tolerance-aware coflow scheduling. A similar problem of scheduling deadline-bounded packets to maximize their timely throughputs have also been studied in the context of wireless network, which has completely decentralized solutions [46, 47]. However, this type of problem is essentially different from the one targeted by POCO in two aspects. Firstly, their deadline requirements are bound to the delivery of each packet rather than the entire (co)flow. Moreover, different flows in [46, 47] are independent without the relationship of coflow—Each of them would continuously generate deadline-bounded packets in streaming, As flows in a coflow could go across different or even link-disjoint paths and might not share endpoints, distributed algorithms making scheduling decisions based on each packet/flow's own information and the local network state without a global view (e.g., [46]), are unable to explore the design space raised by the tolerance of application at the level of coflow.

The design of POCO is motived by the observation that a lot of emerging distributed applications are able to tolerate incomplete inputs. Indeed, such a phenomenon has been widely employed for the design of computing (e.g., big data analytics), raising the area of approximate computation; a lot of new systems have been proposed for better performance or energy efficiency [48, 49]. Different from them, POCO employs the tolerance for efficient and flexible data transmission.

IX. CONCLUSION

Nowadays, an increasing number of emerging time-sensitive distributed applications are able to tolerate loss-bounded inputs by design [3, 12, 13, 41], yielding novel design space and trade-offs for the schedule of their coflow transmissions. Accordingly, this paper studied this type of trade-off and proposed POCO, a policy-based coflow scheduler, to achieve tolerance-aware coflow scheduling based on applications' requirements. As confirmed by extensive trace-driven simulations, by trading loss-bounded completeness for timeliness and trading one coflow's completeness for those of others on demand, POCO was able to achieve optimal bandwidth allocations respecting user-specific requirements.

REFERENCES

[1] S. Luo, P. Fan, H. Xing, and H. Yu, "Selective coflow completion for time-sensitive distributed applications with poco," in *49th ICPP*, 2020.

[2] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *11th HotNets*, 2012, pp. 31–36.

[3] K. V. Rashmi, M. Chowdhury *et al.*, "Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding," in *OSDI*, 2016, pp. 401–417.

[4] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *SIGCOMM*, Aug. 2014, pp. 443–454.

[5] S. Luo, H. Yu, and L. Li, "Decentralized deadline-aware coflow scheduling for datacenter networks," in *IEEE ICC*, May 2016, pp. 1–6.

[6] S. M. Srinivasan, T. Truong-Huu, and M. Gurusamy, "Deadline-aware scheduling and flexible bandwidth allocation for big-data transfers," *IEEE Access*, vol. 6, pp. 74 400–74 415, 2018.

[7] S. Ma, J. Jiang, B. Li, and B. Li, "Chronos: Meeting coflow deadlines in data center networks," in *IEEE ICC*, May 2016, pp. 1–6.

[8] S. Luo, H. Yu, K. Li, and H. Xing, "Efficient file dissemination in data center networks with priority-based adaptive multicast," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 6, pp. 1161–1175, 2020.

[9] S. Im, M. Shadloo, and Z. Zheng, "Online partial throughput maximization for multidimensional coflow," in *IEEE INFOCOM*, April 2018, pp. 2042–2050.

[10] C. Yu, H. Tang *et al.*, "Distributed learning over unreliable networks," in *36th ICML*, vol. 97, 09–15 Jun 2019, pp. 7202–7212.

[11] S. Luo, T. Ma *et al.*, "Efficient multi-source data delivery in edge cloud with rateless parallel push," *IEEE Internet Things J.*, pp. 1–1, 2020.

[12] J. Xia, G. Zeng *et al.*, "Rethinking transport layer design for distributed machine learning," in *3rd APNet*, 2019, pp. 22–28.

[13] K. Liu, S.-Y. Tsai, and Y. Zhang, "Atp: a datacenter approximate transmission protocol," *preprint arXiv:1901.01632*, 2019.

[14] Y. Peng, K. Chen *et al.*, "Towards comprehensive traffic forecasting in cloud computing: Design and application," *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 2210–2222, Aug 2016.

[15] H. Wang, L. Chen *et al.*, "Flowprophet: Generic and accurate traffic prediction for data-parallel cluster computing," in *35th IEEE ICDCS*, June 2015, pp. 349–358.

[16] J. Verbraeken, M. Wolting *et al.*, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, Mar. 2020.

[17] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula, "Calendaring for wide area networks," in *SIGCOMM*, 2014, pp. 515–526.

[18] J. Perry, H. Balakrishnan, and D. Shah, "Flowtune: Flowlet control for datacenter networks," in *NSDI*, 2017, pp. 421–435.

[19] J. Perry, A. Ousterhout *et al.*, "Fastpass: A centralized "zero-queue" datacenter network," in *SIGCOMM*, 2014, pp. 307–318.

[20] Y. Pan, C. Tian *et al.*, "Support ecn in multi-queue datacenter networks via per-port marking with selective blindness," in *38th IEEE ICDCS*, July 2018, pp. 33–42.

[21] S. Wang, J. Zhang *et al.*, "A survey of coflow scheduling schemes for data center networks," *IEEE Commun. Mag.*, vol. 56, no. 6, pp. 179–185, June 2018.

[22] B. Tian, C. Tian, H. Dai, and B. Wang, "Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time," in *IEEE INFOCOM*, April 2018, pp. 864–872.

[23] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *SIGCOMM*, 2015, pp. 393–406.

[24] S. Venkataraman, Z. Yang *et al.*, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *NSDI*, Mar. 2016, pp. 363–378.

[25] V. Jalaparti, P. Bodik *et al.*, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *SIGCOMM*, 2015, pp. 407–420.

[26] J. Rasley, K. Karanasos *et al.*, "Efficient queue management for cluster scheduling," in *EuroSys*, 2016.

[27] S. Hu, K. Chen *et al.*, "Explicit path control in commodity data centers: Design and applications," in *NSDI*, May 2015, pp. 15–28.

[28] S. Bocanegra, J. Castro, and A. R. Oliveira, "Improving an interior-point approach for large block-angular problems by hybrid preconditioners," *Eur J Oper Res*, vol. 231, no. 2, pp. 263–273, 2013.

[29] J. Gondzio and R. Sarkissian, "Parallel interior-point solver for structured linear programs," *Appl Manag Sci*, vol. 96, no. 3, pp. 561–584, Jun 2003.

[30] D. Bertsimas and J. Tsitsiklis, *Introduction to Linear Optimization*, 1st ed. Athena Scientific, 1997.

[31] E. D. Andersen and K. D. Andersen, "The mosek interior point optimizer for linear programming: An implementation of the homogeneous algorithm," in *High Performance Optimization*. Springer US, 2000, pp. 197–232.

[32] C. Huang, J. Zhang, T. Huang, and Y. Liu, "Providing guaranteed network performance across tenants: Advances, challenges and opportunities," *China Commun*, vol. 18, no. 2, pp. 152–174, 2021.

[33] B. S. Ali, K. Chen, and I. Khan, "Towards efficient, work-conserving, and fair bandwidth guarantee in cloud datacenters," *IEEE Access*, vol. 7, pp. 109 134–109 150, 2019.

[34] Q. D. Coninck and O. Bonaventure, "Multipath Extensions for QUIC (MP-QUIC)," Internet Engineering Task Force, Internet-Draft draft-deconinck-quic-multipath-06, Nov. 2020, work in Progress.

[35] K. He, W. Qin *et al.*, "Low latency software rate limiters for cloud networks," in *1st APNet*, 2017, p. 78–84.

[36] W. Xiao, R. Bhardwaj *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *OSDI*, Oct. 2018, pp. 595–610.

[37] X. Jin, Y. Li *et al.*, "Optimizing bulk transfers with software-defined optical wan," in *SIGCOMM*, 2016, p. 87–100.

[38] Q. Zhou, K. Wang *et al.*, "Fast coflow scheduling via traffic compression and stage pipelining in datacenter networks," *IEEE Trans. Comput.*, vol. 68, no. 12, pp. 1755–1771, Dec 2019.

[39] L. Wang, W. Wang, and B. Li, "Utopia: Near-optimal coflow scheduling with isolation guarantee," in *IEEE INFOCOM*, April 2018, pp. 891–899.

[40] S. Liu, L. Chen, and B. Li, "Siphon: Expediting inter-datacenter coflows in wide-area data analytics," in *USENIX ATC*, 2018, pp. 507–518.

[41] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *IEEE INFOCOM*, April 2016, pp. 1–9.

[42] S. Luo, H. Yu *et al.*, "Towards practical and near-optimal coflow scheduling for data center networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3366–3380, Nov 2016.

[43] K. Wang, Q. Zhou, S. Guo, and J. Luo, "Cluster frameworks for efficient scheduling and resource allocation in data center networks: A survey," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 3560–3580, Fourthquarter 2018.

[44] S. Tseng and A. Tang, "Coflow deadline scheduling via network-aware optimization," in *56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Oct 2018, pp. 829–833.

[45] F. Betzel, K. Khatamifard *et al.*, "Approximate communication: Techniques for reducing communication bottlenecks in large-scale parallel systems," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1:1–1:32, Jan. 2018.

[46] R. Singh and P. R. Kumar, "Throughput optimal decentralized scheduling of multihop networks with end-to-end deadline constraints: Unreliable links," *IEEE Trans. Autom. Control*, vol. 64, no. 1, pp. 127–142, 2019.

[47] K. S. Kim, C.-p. Li, and E. Modiano, "Scheduling multicast traffic with deadlines in wireless networks," in *IEEE INFOCOM*, 2014, pp. 2193–2201.

[48] Z. Wen, D. L. Quoc *et al.*, "Approxiot: Approximate analytics for edge computing," in *38th IEEE ICDCS*, July 2018, pp. 411–421.

[49] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016.