A Case for Sampling-Based Learning Techniques in Coflow Scheduling

Akshay Jajoo, Y. Charlie Hu, *Fellow, IEEE*, and Xiaojun Lin, *Fellow, IEEE*

***Abstract*— Coflow scheduling improves data-intensive application performance by improving their networking performance. State-of-the-art online coflow schedulers in essence approximate the classic Shortest-Job-First (SJF) scheduling by learning the coflow *size* online. In particular, they use multiple priority queues to simultaneously accomplish two goals: to sieve long coflows from short coflows, and to schedule short coflows with high priorities. Such a mechanism pays high overhead in learning the coflow size: moving a large coflow across the queues delays small and other large coflows, and moving similar-sized coflows across the queues results in inadvertent round-robin scheduling. We propose PHILAE, a new online coflow scheduler that exploits the spatial dimension of coflows, *i.e.,* a coflow has many flows, to drastically reduce the overhead of coflow size *learning*. PHILAE pre-schedules sampled flows of each coflow and uses their sizes to estimate the average flow size of the coflow. It then resorts to Shortest Coflow First, where the notion of shortest is determined using the learned coflow sizes and coflow contention. We show that the sampling-based learning is robust to flow size skew and has the added benefit of much improved scalability from reduced coordinator-local agent interactions. Our evaluation using an Azure testbed, a publicly available production cluster trace from Facebook shows that compared to the prior art Aalo, PHILAE reduces the coflow completion time (CCT) in average (P90) cases by 1.50× (8.00×) on a 150-node testbed and 2.72× (9.78×) on a 900-node testbed. Evaluation using additional traces further demonstrates PHILAE's robustness to flow size skew.**

***Index Terms*— Coflows, flow scheduling, coflow completion time (CCT), sampling-based learning, online learning, coflow size prediction.**

I. Introduction

A. Motivation

IN BIG data analytics jobs, speeding up the communication stage (where the data is transferred between compute nodes) is important to speed up the jobs. However, improving network level metrics such as flow completion time may not translate into improvements at the application level metrics such as job completion time. The coflow abstraction [19] was proposed to bridge such a gap. The abstraction captures the collective network requirements of applications, as reduced coflow completion time (CCT) can directly lead to faster job completion time [20], [24].

There have been a number of efforts on network designs for coflows [7], [21], [29] that assume complete prior knowledge of coflow sizes (The coflow size is defined as the total size of its constituent flows.) However, in many practical settings, coflow characteristics are not known a priori. For example, multi-stage jobs pipeline data from one stage to the next as soon as the data is generated, which makes it difficult to know the size of each flow [22], [25]. A recent study [25] shows various other reasons why it is not very plausible to learn flow sizes from applications, for example, learning flow sizes from applications requires changing either the network stack or the applications.

Scheduling coflows in such *non-clairvoyant* settings, however, is challenging. The major challenge in developing an effective non-clairvoyant coflow scheduling scheme has centered around how to learn the coflow sizes online quickly and accurately, as once the coflow sizes (bytes to be transferred) can be estimated, one can apply variations of the classic Shortest-Job-First (SJF) algorithm such as Shortest Coflow First [21] or apply an LP solver (*e.g.,* [7]).

State-of-the-art online non-clairvoyant schedulers such as Saath [33], [34], [53], Graviton [32] and Aalo [18] try to approximate SCF (inspired by SJF scheduling in single-server queues, which is known to minimize the average completion time of tasks). They implement this idea using discrete priority queues, where all newly arriving coflows start from the highest priority queue, and move to lower priority queue as they send more data (without finishing), *i.e.,* cross the per-queue thresholds. In this way, the smaller coflows finish in high priority queues, while the larger coflows gradually move to the lower priority queues where they finish after smaller coflows.

To realize the above idea in scheduling coflows which have flows at many network ports, *i.e.,* in a distributed setting, Aalo uses a global coordinator to assign coflows to logical priority queues, and uses the total bytes sent by all flows of a coflow as its logical "length" in moving coflows across the queues. The logical priority queues are mapped to local priority queues at each port, and the individual local ports then schedule the flows in its local priority queues, *e.g.,* by enumerating flows from the highest to lowest priority queues and using FIFO to order the flows within each queue.

In essence, Aalo learns coflow sizes by actually scheduling the coflow, a "try and miss" approach to approximate SJF.

Manuscript received 2 December 2019; revised 17 May 2021; accepted 16 December 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Shin. Date of publication 31 January 2022; date of current version 18 August 2022. This work was supported in part by NSF under Grant CNS-2113893. An earlier conference version of this work was presented at USENIX ATC 2019 [35]. *(Corresponding author: Akshay Jajoo.)*

Akshay Jajoo was with the Department of Computer Science, Purdue University, West Lafayette, IN 47907 USA (e-mail: ajajoo@purdue.edu; akshayjajoo.research@gmail.com).

Y. Charlie Hu and Xiaojun Lin are with the Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA.

This article has supplementary downloadable material available at https://doi.org/10.1109/TNET.2021.3138923, provided by the authors.

Digital Object Identifier 10.1109/TNET.2021.3138923

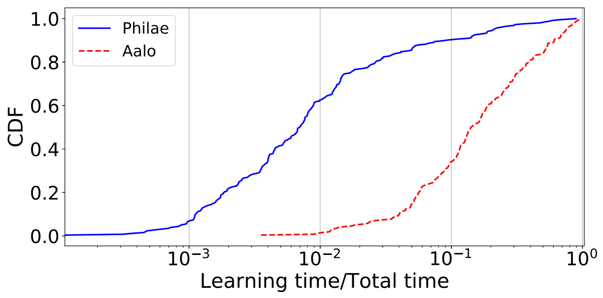


Fig. 1. CDF of learning overhead per coflow, *i.e.,* the time to reach the correct priority queue as a fraction of CCT, excluding coflows directly scheduled by PHILAE or finish in Aalo's first queue.

As coflow sizes are not known, in each queue, Aalo schedules each coflow for a fixed amount of data (try). If the coflow does not finish (miss), it is demoted to a lower priority queue. Afterwards, such a coflow will no longer block coflows in higher priority queues.

Using multiple priority queues to learn the relative coflow sizes of coflows this way, however, negatively affects the average CCT and the scalability of the coordinator:

(1) Intrinsic queue-transit overhead: Every coflow that Aalo transits through the queues before reaching its final queue worsens the average CCT because during transitions, such a coflow effectively *blocks other shorter* coflows in the earlier queues it went through, which would have been scheduled before this coflow starts in a perfect SJF.

(2) Overhead due to inadvertent round-robin: Although Aalo attempts to approximate SJF, it inadvertently ends up doing *round-robin* for coflows of similar sizes as it moves them across queues. Aalo assigns a *fixed threshold of data transfer* for each coflow in each queue. Assume there are "N" coflows in a queue that do not finish in that queue. Aalo schedules one coflow (chosen using FIFO) and demotes it to a lower priority queue when the coflow reaches the data threshold. At that point, the next coflow from the same queue is scheduled, which joins the previous coflow at a lower priority queue after exhausting its quantum, and this cycle continues as coflows of similar sizes move through the queues. Effectively, these coflows experience the round-robin scheduling which is known to have the worst average CCT [48], when jobs are of similar sizes.

(3) Limited scalability from frequent updates from local ports: To support the try-and-error style learning, the coordinator requires frequent updates from all local ports of the bytes sent for each coflow in order to move coflows across multiple queues timely. This results in high load on the central coordinator from receiving frequent updates and calculating and sending new rate allocations, which limits the scalability of the overall approach.

Empirical measurement We quantify the coflow size *learning overhead* of Aalo, defined as the portion of the bytes of a coflow that has been transferred (or the fraction of its CCT spent in doing so) before reaching its correct queue, using a trace from Facebook clusters [4] (see detailed methodology in §VII). Figure 1 shows that 40% of the coflows that moved beyond the initial queue reached the correct priority queue after spending more than 20% of their CCT moving across early queues.

B. Our Contribution

We propose PHILAE, a new non-clairvoyant coflow scheduler with a dramatically different approach to learning coflow sizes to enable online SJF. To leverage optimal scheduling SJF in coflow scheduling, it is vital to learn the coflow sizes quickly and accurately. PHILAE achieves this objective by exploiting the *spatial dimension* of coflows, *i.e.,* a coflow typically consists of many flows, via *sampling*, a highly effective technique used in large-scale surveys [43]. In particular, PHILAE pre-schedules sampled flows, called *pilot flows*, of each coflow and uses their measured size to estimate the coflow size. It then resorts to SCF (Smallest-Coflow-First) using the estimated size.

Intuitively, such a sampling scheme avoids all three sources of overhead in Aalo – Once the coflow sizes are learned, the coflows are assigned to the correct queues, which avoids the intrinsic queue-transit and round-robin effects. Further, a sampling-based design has an important benefit – it offers much higher scalability than priority-queue-based learning in Aalo. This is because unlike Aalo, after estimating the coflow size, PHILAE clients do not need to send periodic updates of bytes sent-so-far to the centralized coordinator.

Developing a complete non-clairvoyant coflow scheduler based on the simple sampling idea raises three questions: (1) Why is sampling more efficient than the priority-queue-based coflow size learning? (2) Will sampling be effective in the presence of skew of flow sizes? (3) How to design the complete scheduler architecture? We systematically address these questions with design rational, theoretical analysis, system design, prototyping, and extensive evaluation.

In summary, this paper makes the following contributions:

(1) Using a production datacenter trace from Facebook, we show that the prior art scheduler Aalo spends substantial amount of time and network bandwidth in learning coflow sizes, which negatively affects the CCT of coflows.

(2) We propose the novel idea of applying sampling in the spatial dimension of coflow to significantly reduce the overhead of online learning coflow sizes.

(3) We present theoretical underpinning explaining why sampling remains effective in the presence of flow size skew.

(4) We present the design and implementation of PHILAE.

(5) We extensively evaluate PHILAE via simulations and testbed experiments, and show that compared to the prior art, the new design reduces the average CCT by 1.51× for the Facebook coflow trace and by 1.36× for a trace with properties similar to a Microsoft production cluster.

(6) The CCT improvement mainly stems from reduced coflow size learning overhead. PHILAE reduces the median latency and data sent in finding the right queue for coflows in Aalo by 19.0× and 20.0×, respectively (§VII-B).

The rest of the paper is organized as follows. §II gives background on the existing studies of coflow scheduling and states the problem we are solving in this paper. §III presents the key idea behind our solution and presents qualitative, theoretical, and experimental arguments in its favor. §IV and §V discuss design, scalability, and implementation details of PHILAE. §VI summarizes the evaluation results with details presented in §VII (simulation) and §VIII (testbed). We then discuss related work and conclude.

II. Background and Problem Statement

We start with a brief review of the coflow abstraction and the need for non-clairvoyant coflow scheduling and state the network model. *We then give an overview of existing online coflow schedulers and formally state the problem.*

Coflow abstraction In data-parallel applications such as Hadoop [1] and Spark [2], the job completion time heavily depends on the completion time of the communication stage [12], [20]. The coflow abstraction [19] was proposed to speed up the communication stage to improve application performance. A coflow is defined as a set of flows between several nodes that accomplish a common task. For example, in map-reduce jobs, the set of all flows from all map to all reduce tasks in a single job forms a typical coflow. The coflow completion time (CCT) is defined as the time duration between when the first flow arrives and the last flow completes. In such applications, improving CCT is more important than improving individual flows' completion time (FCT) for improving the application performance [18], [21], [24], [31]–[34], [36].

Non-clairvoyant coflows Data-parallel directed acyclic graphs (DAGs) typically have multiple stages which are represented as multiple coflows with dependencies between them. Recent systems (*e.g.,* [3], [22], [30], [45]) employ optimizations that pipeline the consecutive computation stages which removes the barrier at the end of each coflow, making knowing flow sizes of each coflow beforehand difficult. A recent study [25] further shows various other reasons why it is not very plausible to learn flow sizes from applications, for example, learning flow sizes from applications requires changing either the network stack or the applications. Thus in this paper, we focus on *non-clairvoyant* coflow scheduling which do not assume knowledge about coflow characteristics such as flow sizes upon coflow arrival.

Non-blocking network fabric We assume the same non-blocking network fabric model in recent network designs for coflows [7], [18], [21], [32]–[34], where the datacenter network fabric is abstracted as a single non-blocking switch that interconnects all the servers, and each server (computing node) is abstracted as a network port that sends and receives flows. In such a model, the ports, *i.e.,* server uplinks and downlinks, are the only source of contention as the network core is assumed to be able to sustain all traffic injected into the network. We note that the abstraction is to simplify our description and analysis, and is not required or enforced in our evaluation.

A. Prior-Art on Non-Clairvoyant Coflow Scheduling

A classic approach to reduce average CCT is Shortest Coflow First (SCF) [18] (derived from classic SJF), where the coflow size is loosely defined as the total bytes of the coflow, *i.e.,* sum of length of all its flows. However, using SCF *online* is not practical as it requires prior knowledge about the coflow sizes. This is further complicated as coflows arrive and exit dynamically and by other cluster dynamics such as failures and stragglers.

Aalo [18] was proposed to schedule coflows online without any prior knowledge. The key idea in Aalo is to approximate SCF by learning Coflow length using discrete priority queues. In particular, it starts a newly arrived coflow in the highest priority queue and gradually moves it to the lower priority queues when the total data sent by the coflow exceeds the per-queue thresholds.

The above idea of learning the order of jobs in a priority queues was originally applied to scheduling jobs on a single server. To apply it to scheduling coflows with many constituent flows over many network ports, *i.e.,* in a distributed setting, Aalo uses a global coordinator to assign coflows to logical priority queues, and uses the total bytes sent by all flows of a coflow as its logical "length" in moving coflows across the queues. The logical priority queues are mapped to local priority queues at each port, and the individual local ports act *independently* in scheduling flows in its local priority queues, *e.g.,* by enumerating flows from the highest to lowest priority queues and using FIFO to order the flows within each queue.

Generally speaking, using multiple priority queues in Aalo in this way has three effects: (1) **Coflow segregation:** It segregates long coflows (who will move to low priority queues) from short coflows who will finish while in high priority queues; (2) **Finishing short coflows sooner:** Since high priority queues receive more bandwidth allocation, short coflows will finish sooner (than longer ones); (3) **Starvation avoidance:** Using the FIFO policy for intra-queue scheduling provides starvation avoidance, since at every scheduling slot, each queue at each port receives a fixed bandwidth allocation and FIFO ensures that every Coflow (its flow) in each queue is never moved back.

Similar to Aalo [18], Graviton [32] also uses a logical priority queue structure. Unlike Aalo, Graviton uses sophisticated policies to sort coflows within a queue based on their width (total number of ports that a coflow is present on). Saath [33] is another priority queue based online coflow scheduler that improves over Aalo with three high-level design priniciples: (1) it schedules flows of a coflow in an all-or-none fashion to prevent flows of a coflow from going out-of-sync; (2) it incorporates contention, *i.e.,* with how many other coflows a coflow is sharing ports with, into the metric for sorting coflows within a queue; (3) instead of using the total coflow size, it uses the length of the longest flow of a coflow to determine transition across priority queues, which helps in deciding the correct priority queue of a coflow faster.

B. Problem Statement

Our goal is to *develop an efficient non-clairvoyant coflow scheduler that optimizes the communication performance, in particular the average CCT, of data-intensive applications without prior knowledge, while guaranteeing starvation freedom and work conservation and being resilient to the* *network dynamics*. The problem of non-clairvoyant coflow scheduling is NP-hard because coflow scheduling even assuming all coflows arrive at time 0 and their size are known in advance is already NP-hard [21]. Thus practical non-clairvoyant coflow schedulers are approximation algorithms. Our approach is to dynamically prioritize coflows by efficiently learning their flow sizes online.

III. Key Idea

Our new non-clairvoyant coflow scheduler design, PHILAE, is based on a key observation about coflows that a coflow has a *spatial dimension*, *i.e.,* it typically consists of many flows. We thus propose to explicitly learn coflow sizes online by using *sampling*, a highly effective technique used in large-scale surveys [43]. In particular, PHILAE preschedules sampled flows, called *pilot flows*, of each coflow and uses their measured sizes to estimate the coflow size. It then resorts to SJF or variations using the estimated coflow sizes.

Developing a complete non-clairvoyant coflow scheduler based on the simple sampling idea raises three questions:

(1) Why is sampling more efficient than the priority-queue-based coflow size learning? Would scheduling the remaining flows after sampled pilot flows are completed adversely affect the coflow completion time?

(2) Will sampling be effective in the presence of skew of flow sizes?

(3) How to design the complete scheduler architecture? We answer the first two questions below, and present the complete architecture design in §IV.

A. Why is Sampling-Based Learning More Efficient than Priority-Queue-Based Learning?

Scheduling pilot flows first before the rest of the flows can potentially incur two sources of overhead. First, scheduling pilot flows of a newly arriving coflow consumes port bandwidth which can delay other coflows (with already estimated sizes). However, compared to the multi-queue based approach, the overhead is much smaller for two reasons: (1) PHILAE schedules only a small subset of the flows (*e.g.,* fewer than 1% for coflows with many flows). (2) Since the CCT of a coflow depends on the completion of its last flow, some of its earlier finishing flows could be delayed without affecting the CCT. PHILAE exploits this observation and schedules pilot flows on the least-busy ports to increase the odds that it only affects earlier finishing flows of other coflows.

Second, scheduling pilot flows first may elongate the CCT of the newly arriving coflow itself whose other flows cannot start until the pilot flows finish. This is again typically insignificant for two reasons: (1) A coflow (*e.g.,* from a MapReduce job) typically consists of flows from all sending ports to all receiving ports. Conceptually, pre-scheduling one out of multiple flows from each sender may not delay the coflow progress at that port, because all flows at that port have to be sent anyway. (2) Coflow scheduling is of high relevance in a busy cluster (when there is a backlog of coflows in the network), in which case the CCT of coflow is expected to be much higher than if it were the only coflow in the network, and hence the piloting overhead is further dwarfed by a coflow's actual CCT.

B. Why Is Sampling Effective in the Presence of Skew?

The flow sizes within a coflow may vary (*skew*). In this paper we measure skew as $\frac{max\ flow\ length}{min\ flow\ length}$. Other papers like Varys [21] have used metrics like coefficient of variation to measure the skew. We used the ratio $\frac{max\ flow\ length}{min\ flow\ length}$ because it allows us to analyze the learning error without assuming the specific distribution of flow-sizes.

Intuitively, if the skew across flow sizes is small, sampling even a small number of pilot flows will be sufficient to yield an accurate estimate. Interestingly, even if the skew across flow sizes is large, our experiment indicates that sampling is still highly effective. In the following, we give both the intuition and theoretical underpinning for why sampling is effective.

Consider, for example, two coflows and the simple setting where both coflows share the same set of ports. In order to improve the average CCT, we wish to schedule the shorter coflow ahead of the longer coflow. If the total sizes of the two coflows are very different, then even a moderate amount of estimation error of the coflow sizes will not alter their ordering. On the other hand, if the total sizes of the two coflows are close to each other, then indeed the estimation errors will likely alter their ordering. However, in this case since their sizes are not very different anyway, switching the order of these two coflows will not significantly affect the average CCT.

Analytic results To illustrate the above effect, we show that the gap between the CCT based on sampling and assuming perfect knowledge is small, even under general flow size distributions. Specifically, coflows C_1 and C_2 have cn_1 and cn_2 flows, respectively. Here, we assume that n_1 and n_2 are fixed constants. Thus, by taking c to be larger, we will be able to consider wider coflows.

Assume that each flow of C_1 (correspondingly, C_2) has a size that is distributed within a bounded interval $[a_1, b_1]$ ($[a_2, b_2]$) with mean μ_1 (μ_2), *i.i.d.* across flows. Let T^c be the total completion time when the exact flow sizes are known in advance. Let $\tilde{T}^c$ be the average CCT by sampling m_1 and m_2 flows from C_1 and C_2, respectively. Without loss of generality, we assume that $n_2\mu_2 \geq n_1\mu_1$. Then, using Hoeffding's Inequality, we can show that (see supplemental material, for detailed proof)

$$\lim_{c\to\infty}\frac{\tilde{T}^c-T^c}{T^c}\leq 4\exp\left[-\frac{2(n_2\mu_2-n_1\mu_1)^2}{\left(\frac{n_2(b_2-a_2)}{\sqrt{m_2}}+\frac{n_1(b_1-a_1)}{\sqrt{m_1}}\right)^2}\right]\frac{n_2\mu_2-n_1\mu_1}{n_2\mu_2+2n_1\mu_1} \tag{1}$$

(Note that here also we have used the fact that, since both coflows share the same set of ports and c is large, the CCT is asymptotically proportional to the coflow size.)

Equation (1) can be interpreted as follows. First, due to the first exponential term, the relative gap between $\tilde{T}^c$ and T^c decreases as $b_1 - a_1$ and $b_2 - a_2$ decrease. In other words, as the skew of each coflow decreases, sampling becomes more effective. Second, when $b_1 - a_1$ and $b_2 - a_2$ are fixed, if $n_2\mu_2 - n_1\mu_1$ is large (i.e., the two coflow sizes are very different), the value of the exponential function will be small. On the other hand, if $n_2\mu_2 - n_1\mu_1$ is close to zero (i.e., the two coflow sizes are close to each other), the numerator on the second term on the right hand side will be small. In both cases, the relative gap between $\tilde{T}^c$ and T^c will also be small, which is consistent with the intuition explained earlier. The largest gap occurs when $n_2\mu_2 - n_1\mu_1$ is on the same order as $\frac{n_2(b_2-a_2)}{\sqrt{m_2}} + \frac{n_1(b_1-a_1)}{\sqrt{m_1}}$. Finally, although these analytical results assume that both coflows share the same set of ports, similar conclusions on the impact of estimation errors due to sampling also apply under more general settings.

The above analytical results suggest that, when c is large, the relative performance gap for CCT is a function of the number of pilot flows sampled for each coflow, but is independent of the total number of flows in each coflow. In practice, large coflows will dominate the total CCT in the system. Thus, these results partly explain that, while in our experiments the number of pilot flows is never larger than 1% of the total number of flows, the performance of our proposed approach is already very good.

Finally, the above analytical results do not directly tell us how to choose the number of pilot flows, which likely depends on the probability distribution of the flow size. In practice, we do not know such distribution ahead of time. Further, while choosing a larger number of pilot flows reduces the estimation errors, it also incurs higher overhead and delay. Therefore, our design (§IV) needs to have practical solutions that carefully address these issues.

Error-correction. Readers familiar with the online learning and multi-armed bandit (MAB) literature [13], [15], [27], [40] will notice that our key idea above does not attempt to correct the errors in the initial sampling step. In particular, we did not use an iterative procedure to refine the initial estimates based on additional samples, e.g., as in the classical UCB (upper-confidence-bound) algorithm [13]. The reason is because, from our preliminary investigation (details are available in our online technical report [36].), we have found that straight-forward ways of applying UCB-type of algorithms do not work well for minimizing the total completion time. For instance, consider two coflows whose sizes are nearly identical. In order to identify which coflow is smaller, UCB-type of algorithms tend to alternately sample both coflows, which leads to nearly round-robin scheduling. While this is desirable for maximizing payoff (as in typical MAB problems), for minimizing completion time it becomes *sub-optimal*: Indeed, we should have instead let either one of coflows run to completion first, before the other coflow starts. Consistent with this intuition, our preliminary simulation results show that adding UCB-type of iterative error-correction actually degrades the performance of PHILAE: the average CCT improvement over Aalo reduces from 1.51× to 0.95×. (Further details are available in our online technical report [36].) While these preliminary results do not preclude the possibilities that other iterative sampling algorithms may outperform PHILAE, it does illustrate that straight-forward extensions of UCB-type of ideas may not work well. How to find iterative sampling algorithms outperforming PHILAE remains an interesting direction for future work.

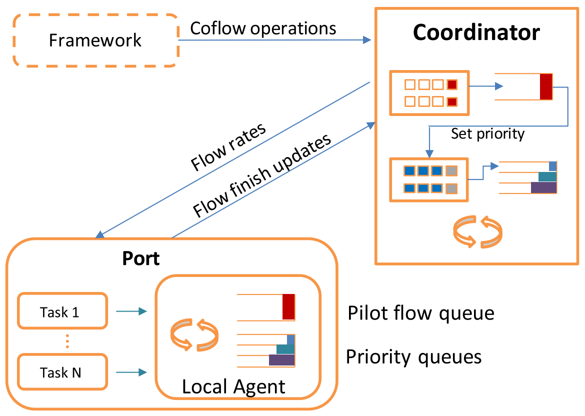


Fig. 2. PHILAE architecture.

IV. PHILAE DESIGN

In this section, we present the detailed design of PHILAE, which addresses three design challenges: (1) *Coflow size estimation:* How to choose and schedule the pilot flows for each newly arriving coflow? (2) *Starvation avoidance:* How to schedule coflows after size estimation using variations of SJF that avoid starvation? (3) *Inter-coflow scheduling:* How to schedule among all the coflows with estimated sizes?

A. PHILAE *Architecture*

Fig. 2 shows the PHILAE architecture PHILAE models the entire datacenter as a single big-switch with each computing node as an individual port. The scheduling function in PHILAE is divided among the following two entities: (1) a central coordinator - which is responsible for making decisions of flow priorities and sending rates, and (2) local agents that run on individual ports - they are responsible for executing the coordinator's decision. A computing framework such as Spark [50] first registers (removes) a coflow with the central coordinator when a job arrives (finishes). The central coordinator functions in an event-driven manner: upon a new coflow arrival, old coflow completion, or pilot flow completion, it calculates a new coflow schedule. The new schedule includes the following two pieces of information: (1) coflows that are to be scheduled in the next time slot, and (2) flow rates for the individual flows of a coflow. It then pushes this information to the local agents which use this information to allocate bandwidth to each flow. The local agents will follow the current schedule until they receive a new schedule.

B. *Sampling Pilot Flows for Coflow-Size Estimation*

As discussed in §III, PHILAE estimates the size of a coflow online by actually scheduling a subset of its flows (*pilot flows*) at their ports. We do not schedule the flows of a coflow other than the pilot flows until the completion of the pilot flows in order to avoid unnecessary extra blocking of other potentially shorter coflows.

How many pilot flows? When a new coflow arrives, PHILAE first needs to determine the number of pilot flows. As discussed at the end of §III, the number of pilot flows affects the trade-off between the coflow size estimation accuracy and scheduling overhead. For coflows with skewed flow sizes, accurately estimating the total coflow size potentially requires sampling the sizes of many pilot flows. However, scheduling pilot flows has associated overhead, *i.e.,* if the coflow turns out to be a large coflow and should have been scheduled to run later under SJF.

We explore several design options for choosing the number of pilot flow. Two natural design choices are using a constant number of pilot flows or a fixed fraction of the total number of flows of a coflow. In addition, we observe that typical coflows consist of flows between a set of senders (*e.g.,* mappers) and a set of receivers (*e.g.,* reducers) [23]. We thus include a third design choice of a fixed fraction of sending ports. This design also spreads the pilot flows to avoid having multiple pilot flows contending for the same sending ports. We empirically found that (§VII-B) limiting the pilot flows to 5% to 10% of the number of its sending ports (*e.g.,* mappers in a MapReduce coflow) strikes a good balance between estimation accuracy and overhead. We note the total number of flows sampled in this case is still under 1%.

Finally, we *estimate the total coflow size as* $S = f_i \cdot N$, where N is the number of flows in a coflow, and f_i is the average size of the sampled pilot flows.

Which flows to probe? Second, PHILAE needs to decide which ports to schedule the chosen number of probe flows for a coflow. For this, we use a simple heuristic where, upon the arrival of a new coflow, we select the ports for its pilot flows that are least busy, *i.e.,* having pilot flows from the least number of other coflows. PHILAE starts with the least busy sending port and iterates over receiving ports starting with the least busy receiving port and assigns the flow if it exists.[1] It then updates the statistics for the number of pilot flows scheduled at each port and repeats the above process. Such a choice will likely delay fewer coflows when the pilot flows are scheduled and hence reduce the elongation on their CCT. We note that such an online heuristic may not be optimal; more sophisticated algorithms can be derived by picking ports for multiple coflows together. However, we make this design choice for its simplicity and low time complexity to ensure that the coordinator makes fast decisions.

How to schedule pilot flows? In PHILAE, we prioritize the pilot flows of a new coflow over existing flows to accelerate learning the size of the new coflow. In particular, at each port, pilot flows have high priority over non-pilot flows. If there are multiple outstanding pilot flows (of different coflows) at a port, PHILAE schedules them in the FIFO order.

C. Coflow Scheduling With Starvation Avoidance

Once the sizes of coflows are learned, we can apply variations of the SJF policy to schedule them. However, it is well known that such policies can lead to starvation.

There are many ways to mitigate the starvation issue. However, a subtlety arises where *even slight difference in how starvation is addressed can result in different performance*. For example, the multiple priority queues in Aalo has the benefit of ensuring progress of all coflows, but assigning different time-quanta to different priority queues can result in different average CCT for the same workload. To ensure the fairness of performance comparison with Aalo, we need to ensure that both PHILAE and Aalo provide the same level of starvation freedom (or progress measure).

For this reason, in this paper, we inherit the multiple priority queue structure from Aalo for coflow scheduling. As in Aalo, PHILAE sorts the coflows among multiple priority queues. In particular, PHILAE uses N queues, Q_0 to Q_{N-1}, with each queue having lower queue threshold Q_q^{lo} and higher threshold Q_q^{hi}, where $\mathrm{Q}_0^{lo} = 0$, $\mathrm{Q}_{N-1}^{hi} = \infty$, $\mathrm{Q}_{q+1}^{lo} = \mathrm{Q}_q^{hi}$, and the queue thresholds grow exponentially, *i.e.,* $\mathrm{Q}_{q+1}^{hi} = \mathrm{E} \cdot \mathrm{Q}_q^{hi}$.

The overall coflow scheduling in PHILAE works as follows. After the coflow size is estimated using pilot flows, PHILAE assigns the coflow to the priority queue using inter-coflow policies discussed in §IV-D. Within a queue, we use FIFO to schedule coflows. Lastly, we use weighted sharing of network bandwidth among the queues, where a priority queue receives a network bandwidth based on its priority. As in Aalo, the weights decrease exponentially with decrease in the priority of the queues.

Using FIFO within the priority queue and weighted fair sharing among the queues together ensure the same starvation freedom and thus meaningful performance comparison between PHILAE and Aalo [18].

D. Inter-Coflow Scheduling Policies

In PHILAE, we explore six different scheduling policies based on different combinations of coflow size and contention, two size-based policies (*A, B*) as in Aalo, a contention-based, similar to the intra-queue policy used in Saath [33] (*C*), and three new contention-and-length-based policy (*D, E and F*):

We use the following parameters of a coflow to define the metrics in the above scheduling policies: (1) average flow length (l) from piloting, (2) number of flows (n), (3) number of sender and receiver ports (s, r), (4) total amount of data sent so far (d), (5) contention (c), defined as the number of other coflows sharing any ports with the given coflow, and (6) port-wise contention (c^p), defined as the number of other coflows blocked at a given port p.

(A) Smallest job first: Coflows are sorted based on coflow size ($l \cdot n$).

(B) Smallest remaining data first: Coflows are sorted based on remaining data ($l \cdot n - d$).

(C) Least global contention first: Coflows are sorted based on their global contention (c).

(D) Least length-weighted total-port contention first: Coflows are sorted based on the sum of port-wise contention times estimated flow length $\sum_p c^p \cdot l$.

(E) Least length-weighted max-port contention first: Coflows are sorted based on the maximum waiting time increase across ports $\max_p c^p \cdot l$.

(F) Least length-weighted contention first: Coflows are sorted based on the product of global contention and length

[1]Note that Philae chooses the least busy ports for sampling. This conditioning has very little impact on the distribution of the size of the sampled flows because flow sizes have no relation to the sending or receiving port. Hence, Philae's sampling of flows can be viewed as independent.

($c{\cdot}l$). This metric is indicative of average waiting time increase across ports.

PHILAE uses Policy D by default, as it results in the least average CCT (§VII). For all policies, we continue to use the priority-queue based scheduling, and the algorithms only differ in what metric they use in assigning coflows to the priority queues. In contrast, Aalo does not handle inter-coflow contention, and uses the total bytes sent so far (d) to move coflows across multiple priority queues.

E. Rate Allocation

Once the scheduling order of the coflows is determined, we need to determine the rates for the individual flows at each port. First, since we want to quickly finish the pilot flow, at any port that has pilot flows, PHILAE assigns the *entire* port bandwidth to the pilot flows. For the remaining ports, as discussed in §IV-C, across multiple queues, PHILAE assigns weighted shares of the port bandwidth, by assigning them varying numbers of scheduling intervals according to the weights assigned to each priority queues.

Second, at each scheduling interval, PHILAE assigns rates for the flows of the coflow in the head of the FIFO queue as follows. It assigns equal rates at all the ports containing its flows as there is no benefit in speeding-up its flows at certain ports when its CCT depends on the slowest flow. At each port, we could use max-min fairness to schedule the individual flows of the coflow (to different receivers), and then assign the rate of the slowest flow to all the flows in the coflow. Afterwards, the port-allocated bandwidths are incremented accordingly at the coordinator, which then allocates rates for the next coflow in the same FIFO queue, and so on. We provide the pseudocode for the rate assignment procedure in the paper's supplemental material.

Heusristic fast rate allocation For online coflow scheduling it is very important for the coordinator to make real-time decision in the given synchronization cycle, and therefore, the rate allocation process should be of low time complexity. While previous heuristics such as Smallest-Effective-Bottleneck-First and Minimum-Allocation-for-Desired-Duration in Varys [21] and Discretized Coflow-Aware Least-Attained Service in Aalo [18] have the advantage of minimizing bandwidth wastage, they slow down the coordinator and as a result the rate allocation process may fail to finish in a small synchronization cycle. For fast rate calculation, in our design, instead of usign max-min fairness to assign rates among the flows of the head-of-queue coflow, we use a simple scheme where we assign the *entire* available bandwidth corresponding to a priority queue at the sender and receiver ports to one flow of the coflow at the head of the FIFO queue at a time. We found that this simple scheme has very marginal effect on CCTs but makes the rate assignment process considerably faster.

F. Additional Design Issues

Thin coflow bypass Recall that, in PHILAE, when a new coflow arrives, PHILAE only schedules its pilot flows. All other flows of that coflow are delayed until the pilot flows finish and coflow size is known. However, such a design choice can inadvertently lead to higher CCTs for coflows, particularly for thin coflows, *e.g.,* a two-flow coflow would end up serializing scheduling its two flows, one for the piloting purpose.

TABLE I
COMPARISON OF FREQUENCY OF INTERACTIONS BETWEEN THE COORDINATOR AND LOCAL AGENTS

	Update of data sent	Update of flow completion	Rate calculation
PHILAE	No	Yes	Event triggered
Aalo	Periodic (δ)	Yes	Periodic (δ)

To avoid CCT degradations for thin coflows, we schedule all flows of a coflow if its width is under a threshold (set to 7 in PHILAE; §VII-F provides sensitivity analysis for thresholds).

Failure tolerance and recovery Cluster dynamics such as stragglers or node failure can delay some of the flows of a coflow or start new flows, increasing their CCT. The PHILAE design automatically self-adjusts to speed up coflows that are affected by cluster dynamics using the following mechanisms: (1) It adjusts the coflow size as the amount of data left by the coflow, which is essentially the difference between the size calculated using pilot flows and amount of data already sent. (2) It calculates contention only on the ports that have unfinished flows.

Work Conservation By default, PHILAE schedules non-pilot flows of a coflow only after all its pilot flows are over. This can lead to some ports being idle where the non-pilot flows are waiting for the pilot flows to finish. In such cases, PHILAE schedules non-pilot flows of coflows which are still in the sampling phase at those ports. In work conservation, the coflows are scheduled in the FIFO order of arrival of coflows.

G. Scalability Analysis

Compared to learning coflow sizes using priority queues (PQ-based) [18], [33], learning coflow sizes by sampling PHILAE not only reduces the learning overhead as discussed in §III-A and shown in §VII-B, but also significantly reduces the amount of interactions between the coordinator and local agents and thus makes the coordinator highly scalable, as summarized in Table I.

First, *PQ-based learning requires much more frequent update from local agents*. PQ-based learning estimates coflow sizes by incrementally moving coflows across priority queues according to the data sent by them so far. As such, the scheduler needs frequent updates (every δ ms) of data sent per coflow from the local agents. In contrast, PHILAE directly estimates a coflow's size upon the completion of all its pilot flows. The only updates PHILAE needs from the local agents are about the flow completion which is needed for updating contentions and removing flows from active consideration..

Second, *PQ-based learning results in much more frequent rate allocation*. In sampling-based approach, since coflow sizes are estimated only once, coflows are re-ordered only upon coflow completion or arrival events or in the case of contention based policies only when contention changes, which is triggered by completion of all the flows of a coflow at a port. In contrast, in PQ-based learning, at every δ interval, coflow data sent are updated and coflow priority may get updated, which will trigger new rate assignment.

Our scalability experiments in §VIII-C confirms that PHILAE achieves much higher scalability than Aalo.

V. IMPLEMENTATION

We implemented both PHILAE and Aalo scheduling policies in the same framework consisting of the global coordinator and local agents (Fig. 2), in 5.2 KLoC in C++.

Coordinator: The coordinator schedules the coflows based on the operations received from the registering framework. The key implementation challenge for the coordinator is that it needs to be fast in computing and updating the schedules. The PHILAE coordinator is optimized for speed using a variety of techniques including pipelining, process affinity, and concurrency whenever possible.

Local agents: The local agents update the global coordinator only upon completion of a flow, along with its length if it is a pilot flow. Local agents schedule the coflows based on the last schedule received from the coordinator. They comply to the last schedule until a new schedule is received. To intercept the packets from the flows, local agents require the compute framework to replace `datasend()`, `datarecv()` APIs with the corresponding PHILAE APIs, which incurs very small overhead.

Coflow operations: The global coordinator runs independently from, and is not coupled to, any compute framework, which makes it general enough to be used with any framework. It provides RESTful APIs to the frameworks for coflow operations: (a) `register()` for registering a new coflow when it enters, (b) `deregister()` for removing a coflow when it exits, and (c) `update()` for updating coflow status whenever there is a change in the coflow structure, *e.g.,* during task migration and restarts after node failures.

VI. EVALUATION HIGHLIGHTS

We evaluated PHILAE using a 150-node and a 900-node testbed cluster in Azure and using large scale simulations by utilizing a publicly available Hive/MapReduce trace collected from a 3000-machine, 150-rack Facebook production cluster [4] and multiple derived traces with varying degrees of flow size skew to measure PHILAE's robustness to skew.

Facebook (FB) trace: The trace contains 150 ports and 526 ($>7 \times 10^5$ flows) coflows, that are extracted from Hive/MapReduce jobs from a Facebook production cluster. Each coflow consists of pair-wise flows between a set of senders and a set of receivers.

Due to the lack of other publicly available coflow trace,[2] we derived three additional classes of traces using the original Facebook trace in order to more thoroughly evaluate PHILAE under varying coflow size skew:

Low-skew-filtered: Starting with the FB trace, we filtered out coflows that have skew (*max flow length/min flow length*) less than a constant k. We generated five traces in this class with $k = 1, 2, 3, 4, 5$. The filtered traces have 142, 100, 65, 51 and 43 coflows, respectively.

[2]A challenge that has also been faced by previous work on coflow scheduling such as [18], [29], [52] and [32].

Mantri-like: Starting with the FB trace, we adjusted the sizes of the flows sent by the mappers, keeping the total reducer data the same as given in the original trace, to match the skew of a large Microsoft production cluster trace as described in Mantri [12]. In particular, the sizes are adjusted so that the coefficients of variation across mapper data are0 about 0.34 in the 50^{th} percentile case and 3.1 in the 90^{th} percentile case. This trace has the same numbers of coflows and ports as the FB trace.

Wide-coflows-only: We filtered out all the coflows in the FB trace with the total number of flows ≤ 7, the default thin coflow bypass threshold (thinLimit) in PHILAE. The filtered trace has 269 coflows spreading over 150 ports.

The primary performance metrics used in the evaluation are CCT or CCT speedup, defined as the ratio of a CCT under other baseline algorithms and under PHILAE, piloting overhead, and coflow size estimation accuracy.

The highlights of our evaluation results are: **(1)** PHILAE significantly improves the CCTs. In simulation using the FB trace, the average CCT is improved by 1.51× over the prior art, Aalo. Individual CCT speedups are 1.78× in the median case (P90 = 9.58×). For the Mantri-like trace, the average CCT is improved by 1.36× and individual CCT speedups are 1.75× in the median case (P90 = 12.0×).

(2) The CCT improvement mainly stems from the reduction in the learning overhead (in terms of latency and amount of data sent) in determining the right queue for the coflows. Compared to Aalo, median reduction in the absolute latency in finding the right queue for coflows in PHILAE is 19.0×, and in absolute amount of data sent is 20.0× (§VII-B).

(3) PHILAE improvements are consistent when varying the skew among the flow sizes in a coflow (§VII-E).

(4) PHILAE is robust to parameter variation (§VII-F).

(5) The PHILAE coordinator is much more scalable than that of Aalo (§VIII-C).

VII. SIMULATION

We present detailed simulation results in this section, and the testbed evaluation of our prototype in §VIII.

Experimental setup Our simulated cluster uses the same number of nodes (sending and receiving network ports) as in the trace. As in [18], we assume full bisection bandwidth is available, and congestion can happen only at network ports.

The *default parameters* for Aalo and PHILAE in the experiments are: starting queue threshold (Q_0^{hi}) is 10MB, exponential threshold growth factor (E) is 10, number of queues (K) is set to 10, the weights assigned to individual priority queues decrease exponentially by a factor of 10, and the new schedule calculation interval δ is set to 8ms for the 150-node cluster,[3] the default suggested in its publicly available simulator [18]. In PHILAE, a new schedule is calculated on demand, upon arrival of a new coflow, completion of a coflow, or completion of all pilot flows of a coflow.

Finally, in PHILAE the threshold for thinLimit (T) is set to 7, the number of pilot flows assigned to wide coflows are $max(1, 0.05 \cdot S)$, where S is the number of senders, and

[3]8ms is the time to send 1MB of data.

TABLE II
PERFORMANCE IMPROVEMENT OVER AALO FOR VARYING PILOT FLOW SELECTION SCHEMES

	Constant	Proportional to number of senders					Proportional to number of flows	
	2	5%	10%	20%	50%	100%	1%	10%
Avg. error	13.21%	6.14%	5.42%	4.94%	5.53%	4.25%	4.15%	2.90%
Avg. CCT	1.27×	1.51×	1.45×	1.50×	1.50×	1.50×	1.43×	0.49×
P50 speedup	1.75×	1.78×	1.76×	1.71×	1.52×	1.40×	1.33×	0.69×
P90 speedup	9.00×	9.58×	9.00×	9.15×	8.33×	8.45×	8.23×	8.23×

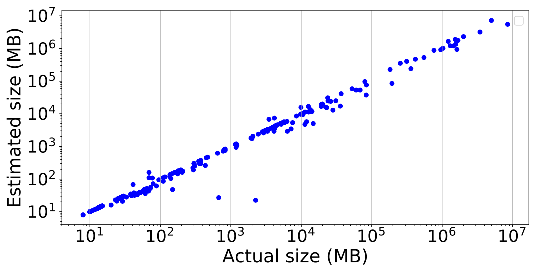


Fig. 3. PHILAE coflow size learning accuracy. Coflows that did not go through the piloting phase (48%) are not shown.

the default inter-coflow scheduling policy in PHILAE is Least length-weighted total-port contention.

A. Pilot Flow Selection Policies

We start by evaluating the impact of different policies in choosing the pilot flows for a coflow in PHILAE. Table II summarizes the improvement in average CCT of PHILAE over Aalo and average error in size estimation normalized to the actual coflow size, when varying the pilot flow selection policy while keeping other parameters as the default in PHILAE, using the FB trace.

Unsurprisingly, the estimation accuracy increases when increasing the number of pilot flows across the three selection schemes: constant, fraction of senders, and fraction of total flows. However, as the number of pilot flows increases (over the range of parameter choices), the CCT speedup (P50 and P90 of individual coflow CCT speedups) decreases. This is because the benefit from size estimation accuracy improvement from using additional pilot flows does not offset the added overhead from completing the additional pilot flows and the delay they incur to other coflows.

We find sampling 5% of the number of senders per coflow strikes a good trade-off between piloting overhead and size estimation accuracy leading to the best CCT reduction. We thus set it ($0.05 \cdot S$) as the default pilot flow selection policy.

B. Piloting Overhead and Accuracy

Next, using the default pilot selection policy, we evaluate PHILAE's effectiveness in estimating coflow sizes by sampling pilot flows. Fig. 3 shows a scatter plot of the actual coflow size vs. estimated size from running PHILAE under the default settings. We observe that PHILAE coflow's size estimation is highly accurate except for a few outliers. Overall, the average and standard deviation of relative estimation error are 0.06 and 0.15, respectively, and for the top 99% and 95% coflows (in terms of estimation accuracy), the average (standard deviation) of relative error are only 0.05 (0.12) and 0.03 (0.07) respectively. Interestingly, a few coflows experience large estimation errors, and we found they all have very high skew in their flow lengths; the mean standard deviation in flow lengths, normalized by the average length, of the bottom 1% (in terms of accuracy) ranges between 4.6 and 6.8.

Fig. 1 shows the cost of estimating the correct queue for each coflow in PHILAE and Aalo, measured as the time in learning the coflow size as a fraction of the coflow's CCT in PHILAE and Aalo. We see that under PHILAE, about 63% of the coflows spent less than 1% of their CCT in the learning phase, while under Aalo, 63% coflows reached the correct priority queue after spending up to 22% of their CCT moving across other queues. Compared to Aalo, PHILAE in the median case sends 20× less data in determining the right queue and reduces the latency in determining the right queue by 19×.

C. Inter-Coflow Scheduling Policies

PHILAE differs from Aalo in two ways: online size estimation and inter-flow scheduling policy. Here, we evaluate the effectiveness of the four inter-coflow scheduling policies of PHILAE discussed in §IV-D, keeping the remaining parameters as the default. Such evaluation allows us to decouple the contribution of sampling-based learning from the effect of scheduling policy difference.

Table III shows the CCT improvement of PHILAE under the four inter-flow scheduling policies over Aalo. We make the following observations.

First, PHILAE with the purely sized-based policy, **Smallest job first (A)**, which uses the same inter-queue and intra-queue scheduling policy as Aalo and only differs from Aalo in coflow size estimation, reduces the average CCT (P50) of Aalo by 1.40× (1.48×).

In contrast, the default PHILAE uses **Least length-weighted total-port contention (D)**, which uses the sum of size-weighted port contention to assign coflows to priority queues, and slightly outperforms the size-based policy A; it reduces the average CCT (P50) of Aalo by 1.51× (1.78×). This is because it captures the diversity of contention at different ports, which happens often in real distributed settings, and at the same time accounts for the coflow size by using length-weighted sum of the port-wise contention. The above results for policy A and policy D indicate that the primary improvement in PHILAE comes from its sampling-based coflow size estimation scheme.

TABLE III
CCT SPEEDUP IN PHILAE UNDER DIFFERENT INTER-COFLOW SCHEDULING POLICIES (§IV-D) OVER AALO

Priority estimation metric	P50	P90	Avg. CCT
Estimated size (A)	1.48×	8.27×	1.40×
Remaining size (B)	1.54×	8.34×	1.37×
Least global contention first (C)	0.75×	8.26×	0.13×
Length-weighted total-port contention (D) (PHILAE)	1.78×	9.58×	1.51×
Least length-weighted max-port contention first (E)	1.43×	9.00×	1.13×
Length-weighted global contention first (F)	1.68×	10.25×	1.22×

Shortest remaining time first (B) performs similarly as smallest job first. This is because the preemptive nature of SRTF will kick in only on arrival of new coflows. Also, although SRTF is advantageous for small coflows, since PHILAE already schedules thin coflows at high priority, many thin and thus small coflows are anyways being scheduled at high priority under both policies A and B, and as a result they perform similarly.

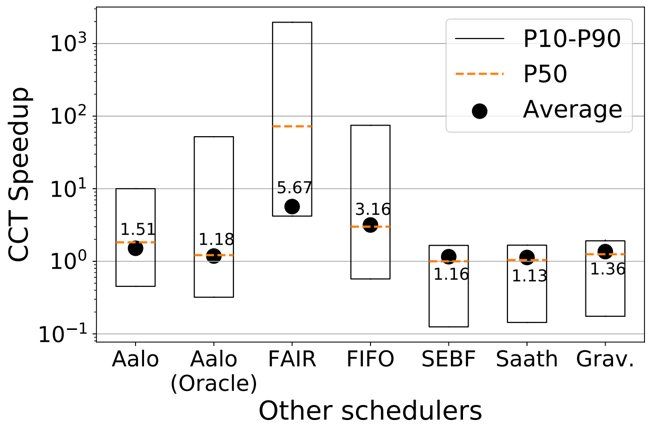
Next, **Least global contention first (C)** performs poorly. This is because global contention for a coflow is defined as the unique number of other coflows that share ports, and as a result such a policy completely ignores the size (length) of the coflows.

Finally, **Least length-weighted max-port contention first (E)** and **Least max length-weighted global contention first (F)** perform better than the above size-based or contention-only based policies for some coflows, by incorporating the flow length factor. However, these two policies sort coflows based on either an extreme notion (E) or a coarse notion (F) of contentions faced by coflows at their ports. As a result, wider coflows, which tend to have high contention at at-least one of their ports, are being scheduled with lower priority. This results in higher average CCT than **Least length-weighted total-port contention (D)**.

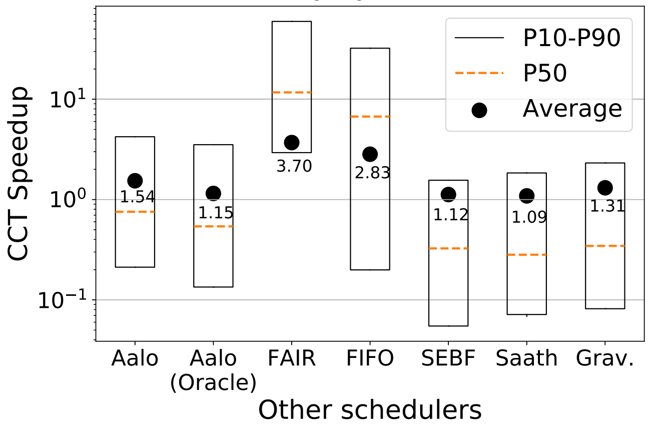
D. Average CCT Improvement

We now compare the CCT speedups of PHILAE against seven well-known coflow scheduling policies: (1) Aalo [18], (2) Aalo-Oracle, which is an oracle version of Aalo where the scheduler knows the final queue of a coflow upon its arrival time and directly starts the coflow from that queue, (3) SEBF in Varys [21] which assumes the knowledge of coflow sizes apriori and uses the Shortest Effective Bottleneck First policy, where the coflow whose slowest flow will finish first is scheduled first, (4) Saath [33], (5) Graviton [32], (6) FIFO, which is a single queue FIFO based coflow scheduler, and (7) FAIR, which uses per-flow fair sharing. All experiments use the default parameters discussed in the setup, including K, E, S, unless otherwise stated. The results are shown in Fig. 4(a). We make the following observations.

First, we compare CCT under PHILAE against under Aalo-Oracle, where Aalo-Oracle starts all coflows at the correct priority queues (*i.e.,* no learning overhead). PHILAE



(a) Using original FB trace.



(b) Using Wide-coflows-only trace.

Fig. 4. CCT speedup using PHILAE compared to using other coflow schedulers on different traces.

improves the average CCT by 1.18× and P50 CCT by 1.40×, respectively. Since Aalo-Oracle pays no overhead for coflow size estimation, its worse performance suggests that using length-weighted total-port contention in assigning coflows to the priority queues in PHILAE outperforms Aalo's size-based, contention-oblivious policy in assigning coflows to the queues.

Second, PHILAE improves the average CCT over Aalo by 1.51× (median) and P50 by 1.78. The significant *additional* improvement on top of the gain over Aalo-Oracle comes from fast and accurate estimation of the right queues for the coflows (Fig. 1).

Third, PHILAE, which requires no coflow size knowledge a priori, achieves comparable performance as SEBF [21]; it reduces the average CCT by 1.16×. Again this is because its total-port contention policy outperforms the contention-oblivious SEBF.

Fourth, we compare PHILAE against Saath [33], [34] and Graviton [32]. PHILAE improves over Saath [33], [34] and Graviton [32] because of its better and accurate learning. We note that Saath and Graviton like Aalo follow the priority queue based estimation technique. However, Graviton does a better job in sorting the coflows in their assigned queues. Hence, the speedup of Philae over Graviton is slightly lower when compared to Aalo. Average CCT speedup =1.36× and 1.31× for the full trace and wide-coflows-only trace, respectively. Similarly, the speedup of Philae over Saath

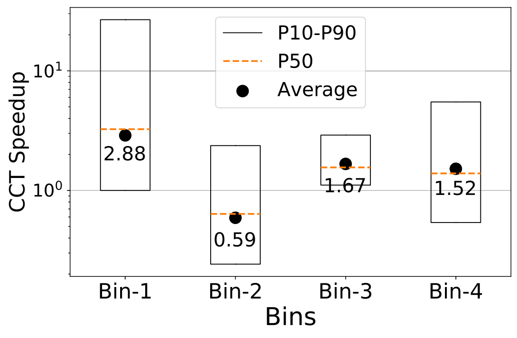


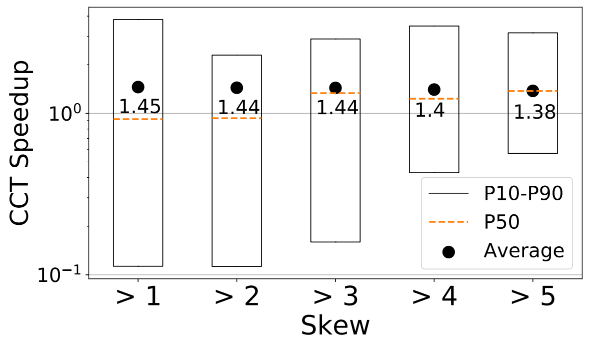
Fig. 5. Performance breakdown into bins shown in Table IV.

is lower because of the advanced scheduling policies in Saath like All-or-none. Average CCT speedup $=1.13\times$ and $1.09\times$ for the full trace and wide-coflows-only trace, respectively. However, the better performance of Saath and Graviton are not due to their learning technique, because they both use the priority queue based learning techniques similar to that in Aalo to learn coflow sizes. Rather, the gains are due to the more complex versions of the scheduling and coflow sorting techniques than those used in Aalo. Further, in practice Saath and Graviton face the same scalability issues as Aalo (because they have similar global coordinator architecture as they have similar learning techniques). For example, in a 150 port testbed experiment, we observed that Saath uses CPU $3.96\times$ ($3.24\times$) more than Philae in P50 (P90) case. Similarly, Aalo's CPU usage is also much higher than PHILAE, which is about 3.40x more than PHILAE. Since, in this paper our goal is to evaluate the new sampling-based coflow size estimation technique so we only use Aalo for detailed analysis.

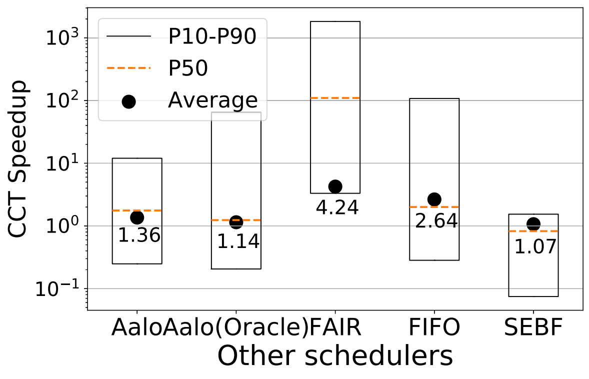
Finally, PHILAE significantly outperforms the single-queue FIFO-based coflow scheduler, with a median (P90) CCT speedup of 3.00 (77.96)$\times$ and average CCT speedup of $3.16\times$, and the un-coordinated flow-level fair-share scheduler, with a median (P90) CCT speedup of $70.82\times$ ($1947.12\times$) and average CCT speedup of $5.66\times$.

To gain insight into how different coflows are affected by PHILAE over Aalo, we group the coflows in the trace into four bins defined in Table IV, and show in Fig. 5 the CCT speedups for each bin. We see that PHILAE improves CCT for all coflows in bin 1 and 3 and for large fraction in bin-4. Most of the underperforming coflows fall in bin-2. Coflows in bin-2 have width >7 and size <100MB, *i.e.,* the flows are short but wide. Because the width exceeds the thinLimit, PHILAE schedules the pilot flows to estimate the coflow size first (§IV-B). Thus, although the remaining flows are short, they get delayed until the completion of the pilot flows, which results in CCT increase.

Finally, since thin coflows benefit from PHILAE's scheme of bypassing probing for thin coflows, we also compare PHILAE with other schemes using the Wide-coflows-only trace which consists of all coflows wider than the default thinLimit (7) in PHILAE. Fig. 4(b) shows that PHILAE continues to perform well, reducing the average CCT by $1.54\times$, $1.15\times$, and $1.12\times$ over Aalo, Aalo-Oracle, and SEBF, respectively.



(a) Using 5 Low-skew-filtered traces.



(b) Using the Mantri-like trace.

Fig. 6. CCT speedup using PHILAE compared to using other coflow schedulers on traces with different skew. In Fig. 6(a), the x-axis denotes the minimum skew in the 5 Low-skew-filtered traces.

TABLE IV

BINS BASED ON TOTAL COFLOW SIZE AND WIDTH (NUMBER OF FLOWS). THE NUMBERS IN BRACKETS DENOTE THE FRACTION OF COFLOWS IN THAT BIN

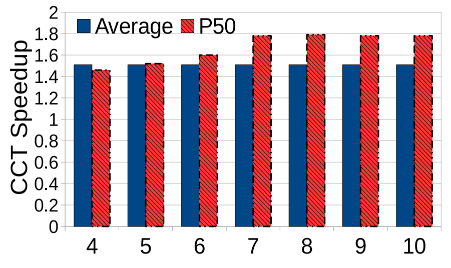
	width ≤ 7 (thin)	width > 7 (wide)
size $\leq$ 100MB (small)	bin-1 (44.3%)	bin-2 (24.1%)
size $>$ 100MB (large)	bin-3 (4.5%)	bin-4 (27.1%)

E. Robustness to Coflow Data Skew

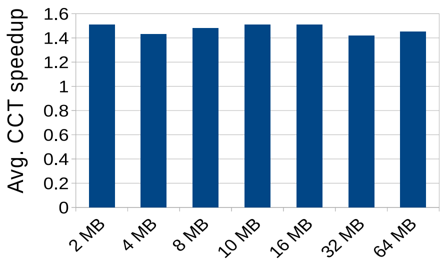
Next, we evaluate PHILAE's robustness to flow size skew by comparing it against Aalo using traces with varying degrees of skew. First, we evaluate PHILAE using the Mantri-like trace. Fig. 6(b) shows that PHILAE consistently outperforms prior-art coflow schedulers. In particular, PHILAE reduces the average CCT by 1.36x compared to Aalo. Second, we evaluate PHILAE using the Low-skew-filtered traces which have low skew coflows filtered out. Fig. 6(a) shows that PHILAE performs better than Aalo even with highly skewed traces and reduces the average CCT by $1.45\times$, $1.44\times$, $1.44\times$, $1.40\times$ and $1.38\times$ for the five Low-skew-filtered traces containing coflows with skew of at least 1, 2, 3, 4 and 5, respectively.

F. Sensitivity Analysis

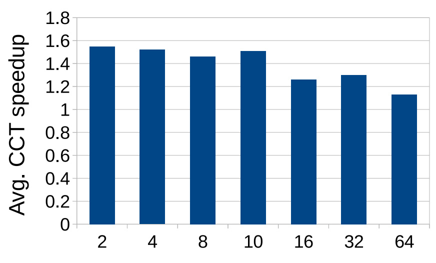
Compared to Aalo, PHILAE has only two additional paramaters: thinLimit and flow sampling rate. We already discussed the choice of sampling rate in §VII-A. Below, we evaluate the sensitivity of PHILAE to thinLimit and other design



(a) Thin coflow bypass threshold



(b) First queue capacity (Q_0^{hi})



(c) Exponent (E)

Fig. 7. [Simulation] PHILAE sensitivity analysis. We vary one parameter of PHILAE keeping rest same as default and compare it with Aalo.

parameters common to Aalo by varying one parameter at a time while keeping the rest as the default.

Thin coflow bypassing limit (T) In this experiment, we vary thinLimit (T) in PHILAE for bypassing coflows from the probing phase. The result in Fig. 7(a) shows that the average CCT remains almost the same as T increases. This is because the average CCT is dominated by wide and large coflows, which are not affected by thinLimit. However, the P50 speedup increases till $T = 7$ and tapers off after $T = 7$. The reason for the CCT improvement until $T = 7$ is that all flows of thin coflows (with width $\leq$ 7) are scheduled immediately upon arrival which improves their CCT, and the number of thin coflows is significant.

Start queue threshold (Q_0^{hi}) We next vary the threshold for the first priority queue from 2 MB to 64 MB. Fig. 7(b) shows the average CCT of PHILAE over Aalo. Overall, PHILAE is not very sensitive to the threshold of first priority queue and the CCT speedup over Aalo is within 8% of the default PHILAE (10 MB). The speedup appears to oscillate with a periodicity of 5x to 10x. For example, the speedups for 2 MB and 64 MB are close to that of the default (10 MB), while for 4 MB and 32 MB are lower. This can be explained by the impact of the first queue threshold on job segregation; with the default queue threshold growth factor of 10, every time the first queue threshold changes by close to 10x, the distribution of jobs across the queues become similar.

Multiplication factor (E) In this experiment, we vary the queue threshold growth factor from 2 to 64. Recall that the queue thresholds are computed as $Q_q^{hi} = Q_{q-1}^{hi} \cdot E$. Thus, as E grows, the number of queues decreases. As shown in Fig. 7(c), smaller queue threshold multiplication factor which leads to more queues performs better because of fine-grained priority segregation.

VIII. TESTBED EVALUATION

Next, we deployed PHILAE in a 150-machine Azure cluster and a 900-machine cluster to evaluate its performance and scalability.

Testbed setup: We rerun the FB trace on a Spark-like framework on a 150-node cluster in Microsoft Azure [5]. The coordinator runs on a Standard DS15 v2 server with 20-core 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor and 140GB memory. The local agents run on D2v2 with the same processor as the coordinator with 2-core and 7GB memory. The machines on which local agents run have 1 Gbps network bandwidth. Similarly as in simulations, our testbed evaluation keeps the same flow lengths and flow ports in trace replay. All the experiments use default parameters K, E, S and the default pilot flow selection policy.

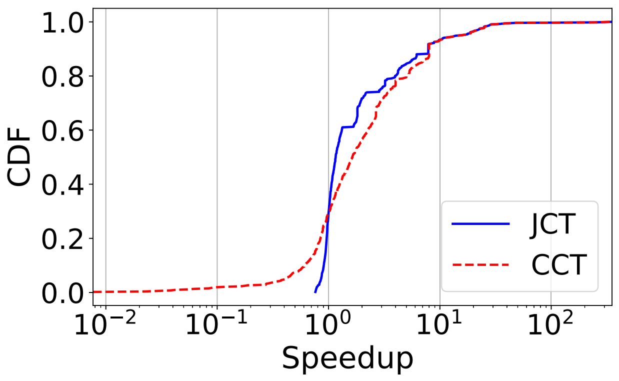


Fig. 8. [Testbed] Distribution of speedup in CCT and JCT in PHILAE using the FB trace.

TABLE V
[TESTBED] CCT IMPROVEMENT IN PHILAE AS COMPARED TO AALO

	P50	P90	Avg. CCT
FB Trace	1.63×	8.00×	1.50×
Wide-coflow-only	1.05×	2.14×	1.49×

A. CCT Improvement

In this experiment, we measure CCT improvements of PHILAE compared to Aalo. Fig. 8 shows the CDF of the CCT speedup of individual coflows under PHILAE compared to under Aalo. The average CCT improvement is 1.50× which is similar to the results in the simulation experiments. We also observe 1.63× P50 speedup and 8.00× P90 speedup.

We also evaluated PHILAE using the Wide-coflow-only trace. Table V shows that PHILAE achieves 1.52× improvement in average CCT over Aalo, similar to that using the full FB trace. This is because the improvement in average CCT is dominated by large coflows, PHILAE is speeding up large coflows, and the Wide-coflow-only trace consists of mostly large coflows.

B. Job Completion Time

Next, we evaluate how the improvement in CCT affects the job completion time (JCT). In data clusters, different jobs spend different fractions of their total job time in data shuffle.

TABLE VI

[Testbed] Average (Standard Deviation) Coordinator CPU Time (ms) per Scheduling Interval in 900-Port Runs. Philae Did Not Have to Calculate and Send New Rates in 66% of Intervals, Which Contributes to Its Low Average

	Rate Calc.	New Rate Send	Update Recv.	Total
Philae	2.99 (5.35)	4.90 (11.25)	6.89 (17.78)	14.80 (28.84)
Aalo	4.28 (4.14)	17.65 (20.90)	10.97 (19.98)	32.90 (34.09)

TABLE VII

[Testbed] Percentage of Scheduling Intervals Where Synchronization and Rate Calculation Took More Than δ for 150-Port and $\delta'(= 6 \times \delta)$ for 900-Port Runs

	150 ports	900 ports
Philae	1%	10%
Aalo	16%	37%

In this experiment, we used 526 jobs, each corresponding to one coflow in the FB trace. The fraction of time that the jobs spent in the shuffle phase follows the same distribution used in Aalo [18], *i.e.,* 61% jobs spent less than 25% of their total time in shuffle, 13% jobs spent 25-49%, another 14% jobs spent 50-74%, and the remaining spent over 75% of their total time in shuffle. Fig. 8 shows the CDF of individual speedups in JCT. Across all jobs, Philae reduces the job completion time by 1.16× in the median case and 7.87× in the 90^{th} percentile. This shows that improved CCT translates into better job completion time. As expected, the improvement in job completion time is smaller than the improvement in CCT because job completion time depends on the time spent in both compute and shuffle (communication) stages, and Philae improves only the communication stage.

C. Scalability

Finally, we evaluate the scalability of Philae by comparing its performance with Aalo on a 900-node cluster. To drive the evaluation, we derive a 900-port trace by replicating the FB trace 6 times across ports, *i.e.,* we replicated each job 6 times, keeping the arrival time for each copy the same but assigning sending and receiving ports in increments of 150 (the cluster size for the original trace). We also increased the scheduling interval δ by 6 times to $\delta' = 6 \times \delta$.

Philae achieved 2.72× (9.78×) speedup in average (P90) CCT over Aalo. The higher speedup compared to the 150-node runs (1.50×) comes from higher scalability of Philae. As summarized in Table VII, in 900-node runs, Aalo was not able to finish receiving updates, calculating new rates, and updating local agents of new rates within δ' in 37% of the intervals, whereas Philae only missed the deadline in 10% of the intervals. For 150-node runs these values are 16% for Aalo and 1% for Philae. The 21% increase in missed scheduling intervals in 900-node runs in Aalo resulted in local agents executing more frequently with outdated rates. As a result, Philae achieved even higher speedup in 900-node runs.

TABLE VIII

[Testbed] Mean Normalised Standard Deviation in CCT Among Philae and Aalo

	P10	P50	P90	Avg. CCT
Philae	6.1%	2.3%	0.1%	0.1%
Aalo	7.1%	4.4%	2.7%	1.6%

TABLE IX

[Testbed] Resource Usage in Philae and Aalo for 150 Ports Experiment

		Philae		Aalo	
		Overall	Busy	Overall	Busy
Coordinator	CPU (%)	5.0	10.4	17.0	27.2
	Memory (MB)	212	218	318	427
Local node	CPU (%)	4.3	4.6	4.5	4.6
	Memory (MB)	1.65	1.70	1.64	1.70

As discussed in §IV-G, Aalo's poorer coordinator scalability comes from more frequent updates from local agents and more frequent rate allocation, which result in longer coordinator CPU time in each scheduling interval. Table VI shows the average coordinator CPU usage per interval and its breakdown. We see that (1) on average Philae spends much less time than Aalo in receiving updates from local agents, because Philae does not need updates from local agents at every interval – on average in every scheduling interval Philae receives updates from 49 local agents whereas Aalo receives from 429 local agents, and (2) on average Philae spends much less time calculating new rates and send new rates. This is because rate calculation in Philae is triggered by events and Philae did not have to flush rates in 66% of the intervals.

D. Robustness to Network Error

As discussed in §V, unlike Aalo, Philae's coordinator does not need constant updates from local agents to sort coflows in priority queues. This simplifies Philae's design and makes it robust to network error. To evaluate the benefit of this property of Philae, we evaluated Aalo and Philae 5 times with the same configuration using the FB trace. Table VIII shows the mean-normalized standard deviation in the 10^{th}, 50^{th}, 90^{th} percentile and the average CCT across the 5 runs. The lower values for Philae indicates that it is more robust to network dynamics than Aalo.

E. Resource Utilization

Finally, we evaluate, for both Philae and Aalo, the resource utilization at the coordinator and the local agents (Table IX) in terms of CPU and memory usage. We measure the overheads in two cases: (1) Overall: average during the entire execution of the trace, (2) Busy: the 90-th percentile utilization indicating the performance during busy periods due to a large number of coflows arriving. As shown in Table IX, Philae agents have similar utilization as Aalo at the local nodes, where the CPU and memory utilization are minimal even during busy times. The global coordinator of Philae consumes much lower server resources than Aalo – the CPU utilization is 3.4× lower than Aalo on average, and 2.6× than Aalo during busy periods. This is due to PHILAE's event triggered communication and sampling-based learning, which significantly lowers its communication frequency with local agents when compared to Aalo. The lower resource utilization of the global coordinator enables PHILAE to scale to a lager cluster size than Aalo.

IX. RELATED WORK

Coflow scheduling: In this paper, we have shown PHILAE outperforms prior-art non-clairvoyant coflow scheduler Aalo from more efficient learning of coflow sizes online. In [18], Aalo was shown to outperform previous non-clairvoyant coflow schedulers Baraat [24] by using global coordination, and Orchestra [20] by avoiding head-of-line blocking.

Clairvoyant coflow schedulers such as Varys [21] and Sincronia [7] assume prior knowledge of coflows upon arrival. Varys runs a shortest-effective-bottleneck-first heuristic for inter-coflow scheduling and performs per-flow rate allocation at the coordinator. Sincronia improves the scalability of the centralized coordinator of Varys by only calculating the coflow ordering at the coordinator (by solving an LP) and offloading flow rate allocation to individual local agents. Sincronia is orthogonal to PHILAE; once coflow sizes are learned through sampling, ideas from Sincronia can be adopted in PHILAE to order coflows and offload rate allocation to local ports. *Prioritized work conservation in Sincronia helps in mitigating the effect of starvation, which arises as a result of ordering coflows in the optimal order. However, it still does not guarantee complete freedom from starvation.* CODA [52] tackles an orthogonal problem of identifying flows of individual coflows online.

However, recent studies [18], [25] have shown various reasons why it is not very plausible to learn flow sizes from applications beforehand. For example, many applications stream data as soon as data are generated and thus the application does not know the flow sizes until flow completion, and learning flow sizes from applications requires changing either the network stack or the applications.

Flow scheduling: There exist a rich body of prior work on flow scheduling. Efforts to minimize flow completion time (FCT), both with prior information (pFabric [9]) and without prior information (*e.g.,* Fastpass [44]), fall short in minimizing CCTs which depend on the completion of the last flow [21]. Similarly, Hedera [8] and MicroTE [14] schedule the flows with the goal of reducing the overall FCT, which again is different from reducing the overall CCT of coflows.

Speculative scheduling Recent works [16], [42] use the idea of online requirement estimation for scheduling in datacenter. In [39], recurring big data analytics jobs are scheduled using their history.

Job scheduling: There have been much work on scheduling in analytic systems and storage at scale by improving speculative tasks [11], [12], [37], [38], [51], improving locality [10], [49], and end-point flexibility [17], [47]. The coflow abstraction is complimentary to these work and their combination can further improve. This remains a future work.

Scheduling in parallel processors: Coflow scheduling by exploiting the spatial dimension bears similarity to scheduling processes on parallel processors and multi-cores, where many variations of FIFO [46], FIFO with backfilling [41] and gang scheduling [26] have been proposed.

X. CONCLUSION

State-of-the-art online coflow schedulers approximate the classic SJF by implicitly learning coflow sizes and pay a high penalty for large coflows. We propose the novel idea of sampling in the spatial dimension of coflows to explicitly and efficiently learn coflow sizes online to enable efficient online SJF scheduling. Our extensive simulation and testbed experiments show the new design offers significant performance improvement over prior art. Further, the sampling-in-spatial-dimension technique can be generalized to other distributed scheduling problems such as cluster job scheduling. Our simulator is available at [6].

REFERENCES

[1] *Apache Hadoop*. Accessed: Dec. 29, 2021. [Online]. Available: http://hadoop.apache.org

[2] *Apache Spark*. Accessed: Dec. 29, 2021. [Online]. Available: http://spark.apache.org

[3] *Apache Tez*. Accessed: Dec. 29, 2021. [Online]. Available: http://tez.apache.org

[4] *Coflow Trace From Facebook Datacenter*. Accessed: Dec. 29, 2021. [Online]. Available: https://github.com/coflow/coflow-benchmark

[5] *Microsoft Azure*. Accessed: Dec. 29, 2021. [Online]. Available: http://azure.microsoft.com

[6] *Philae Simulator*. Accessed: Dec. 29, 2021. [Online]. Available: https://github.com/coflowPhilae/simulator

[7] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: Near-optimal network design for coflows," in *Proc. SIGCOMM*, 2018, pp. 16–29.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. NSDI*, 2010, p. 19.

[9] M. Alizadeh *et al.*, "pFabric: Minimal near-optimal datacenter transport," in *Proc. SIGCOMM*, 2013, pp. 435–446.

[10] G. Ananthanarayanan *et al.*, "Scarlett: Coping with skewed content popularity in mapreduce clusters," in *Proc. 6th Conf. Comput. Syst. (EuroSys)*, New York, NY, USA, 2011, pp. 287–300.

[11] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. NSDI*, 2013, pp. 185–198.

[12] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. OSDI*, 2010, pp. 265–278.

[13] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, nos. 2–3, pp. 235–256, 2002.

[14] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE," in *Proc. 7th Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2011, p. 8.

[15] S. Bubeck and N. Cesa-Bianchi, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Found. Trends Mach. Learn.*, vol. 5, no. 1, pp. 1–122, 2012.

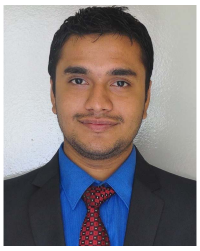
[16] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 239–252.

[17] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. ACM SIGCOMM Conf.*, Aug. 2013, pp. 231–242.

[18] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 393–406.

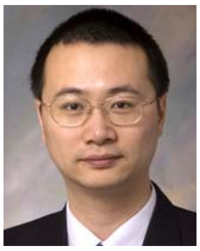
[19] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. HotNets*, New York, NY, USA, 2012, pp. 31–36.

[20] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 98–109.
[21] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 443–454.
[22] T. Condie, N. Conway, P. Alvaro, M. Joseph Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. NSDI*, Berkeley, CA, USA, 2010, p. 21.
[23] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[24] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. ACM Conf. SIGCOMM*, New York, NY, USA, Aug. 2014, pp. 431–442.
[25] V. Dukic, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla, "Is advance knowledge of flow sizes a plausible assumption?" in *NSDI*, Boston, MA, USA, 2019, pp. 565–580.
[26] G. Dror Feitelson and A. Morris Jette, "Improved utilization and responsiveness with gang scheduling," in *Proc. IPPS*. London, U.K.: Springer-Verlag, 1997, pp. 238–261.
[27] J. Gittins, K. Glazebrook, and R. Weber, *Multi-Armed Bandit Allocation Indices*. Hoboken, NJ, USA: Wiley, 2011.
[28] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *J. Amer. Stat. Assoc.*, vol. 58, no. 301, pp. 13–30, Mar. 1963.
[29] X. S. Huang, X. S. Sun, and T. S. Eugene Ng, "Sunflow: Efficient optical circuit scheduling for coflows," in *Proc. CoNEXT*, 2016, pp. 297–311.
[30] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. EuroSys*, 2007, pp. 59–72.
[31] A. Jajoo, "A study on the Morris Worm," 2021, *arXiv:2112.07647*.
[32] A. Jajoo, R. Gandhi, and Y. C. Hu, "Graviton: Twisting space and time to speed-up coflows," in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, Denver, CO, USA, 2016.
[33] A. Jajoo, R. Gandhi, Y. Charlie Hu, and C.-K. Koh, "Saath: Speeding up coflows by exploiting the spatial dimension," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol.*, 2017, pp. 439–450.
[34] A. Jajoo, R. Gandhi, Y. Charlie Hu, and C.-K. Koh, "Saath: Speeding up coflows by exploiting the spatial dimension," 2021, *arXiv:2111.08572*.
[35] A. Jajoo, Y. Charlie Hu, and X. Lin, "Your coflow has many flows: Sampling them for fun and speed," in *Proc. USENIX Annu. Tech. Conf.*, Renton, WA, USA, 2019, pp. 833–848.
[36] A. Jajoo, Y. Charlie Hu, and X. Lin, "A case for sampling based learning techniques in coflow scheduling," 2021, *arXiv:2108.11255*.
[37] A. Jajoo, Y. Charlie Hu, X. Lin, and N. Deng, "The case for task sampling based learning for cluster job scheduling," 2021, *arXiv:2108.10464*.
[38] A. Jajoo, Y. Charlie Hu Hu, X. Lin, and N. Deng, "A case for task sampling based learning for cluster job scheduling," in *Proc. 19th USENIX Conf. Netw. Syst. Design Implement.*, Berkeley, CA, USA, 2022.
[39] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proc. SIGCOMM*, 2015, pp. 407–420.
[40] T. L. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Adv. Appl. Math.*, vol. 6, no. 1, pp. 4–22, Mar. 1985.
[41] D. A. Lifka, "The ANL/IBM SP scheduling system," in *Proc. IPPS*. London, U.K.: Springer-Verlag, 1995, pp. 295–303.
[42] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proc. SIGCOMM*, New York, NY, USA, 2016, pp. 129–143.
[43] P. S. Levy and S. Lemeshow, *Sampling of Populations: Methods and Applications*, 4th ed. Hoboken, NJ, USA: Wiley, Jun. 2012.
[44] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized 'zero-queue' datacenter network," in *Proc. SIGCOMM*, 2014, pp. 307–318.
[45] J. Christopher Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A compiler and runtime for heterogeneous systems," in *Proc. SOSP*, New York, NY, USA, 2013, pp. 49–68.
[46] U. Schwiegelshohn and R. Yahyapour, "Analysis of first-come-first-serve parallel job scheduling," in *Proc. SODA*. Philadelphia, PA, USA: SIAM, 1998, pp. 629–638.
[47] D. Shue, J. Michael Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proc. OSDI*, 2012, pp. 349–362.
[48] A. Silberschatz, B. Peter Galvin, and G. Gagne, *Process Scheduling. Operating System Concepts*, 8th ed. Hoboken, NJ, USA: Wiley, 1998.
[49] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. EuroSys*, New York, NY, USA, 2010, pp. 265–278.
[50] M. Zaharia, M. Chowdhury, J. Michael Franklin, S. Shenker, and I. Stoica, "Spark," in *Proc. HotCloud*, 2010, p. 10.
[51] M. Zaharia, A. Konwinski, D. Anthony Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proc. OSDI*, Berkeley, CA, USA, 2008, pp. 29–42.
[52] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda," in *Proc. SIGCOMM*, 2016, pp. 160–173.
[53] A. Jajoo, "Exploiting the spatial dimension of big data jobs for efficient cluster job scheduling," Purdue Univ. Graduate School, 2020.



Akshay Jajoo received the B.Tech degree from IIT Guwahati, India, in 2015. He got his Ph.D. from Purdue in 2020 and then joined Bell Labs as a Researcher. Dr. Jajoo has also received DAAD WISE and Charpak Scholar. His research interests include cloud computing, computer networks, distributed systems, telecommunications, and AI/ML. He also has experience working in computer vision and RAM error correction. With his B.Tech, he was awarded President Shankar Dayal Sharma gold medal for academic and overall excellence. He was a Finalist in the Prestigious Honda YES award 2013.

Y. Charlie Hu (Fellow, IEEE) received the Ph.D. degree in computer science from Harvard University in 1997. From 1997 to 2001, he was a Research Scientist at Rice University. He is currently a Michael and Katherine Birck Professor of ECE with Purdue University. His research interests include mobile computing, operating systems, distributed systems, and wireless networking. He has published over 180 papers in these areas. He received the NSF CAREER Award in 2003. He is an ACM Distinguished Scientist.



Xiaojun Lin (Fellow, IEEE) received the B.S. degree from Zhongshan University, Guangzhou, China, in 1994, and the M.S. and Ph.D. degrees from Purdue University in 2000 and 2005, respectively. He is currently a Professor of ECE with Purdue University. His research interests include analysis, control and optimization of large and complex networked systems, including both communication networks and power grid. He received the NSF CAREER Award in 2007.