Your CoFlow has Many Flows: Sampling them for Fun and Speed

Techreport

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, moving all flows of a CoFlow across the queues further amplifies the above effect at all related ports.

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. Our evaluation using an Azure testbed and simulations of a publicly available production cluster trace from Facebook shows that compared to the prior art Aalo, Philae reduces the CoFlow completion time(CCT) in median (P90) cases by 1.78× (9.58×).

1 Introduction

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 [18] was proposed to bridge such a gap. The abstraction captures the collective network requirements of applications, which is then used to improve the network level performance that directly translates into application performance improvements.

Scheduling CoFlows in non-clairvoyant settings, *i.e.,* without any apriori knowledge, is a daunting task. Ideally, if the CoFlow sizes (bytes to be transfered) are known apriori, then one can apply variations of the classic Shortest-Job-First (SJF) algorithm (*e.g.,* [21]). Indeed, state-of-the-art online non-clairvoyant schedulers such as Aalo [19] in essence approximate SJF using priority queues, where all CoFlows start from the highest priority queue, and move to lower priority queue as they send more data (without finishing) and thus are deemed as larger CoFlows. 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.

In essence, Aalo assumes that the CoFlow sizes cannot be learned accurately without actually scheduling the CoFlow, and resorts to a "try and miss" approach to approximate SJF. As CoFlow sizes are not known, in each queue, Aalo schedules each CoFlow for a fixed amount of data (try) in the hope that the CoFlow finishes in that queue. 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.

However, using multiple priority queues to learn relative CoFlow sizes this way incurs three intertwined sources of overhead:

Intrinsic queue-transit overhead: Every CoFlow that Aalo transits through the queues before reaching its final queue worsens the average CCT as 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.

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 [40], when jobs are of similar sizes.

Overhead in scheduling all flows: To learn the CoFlow sizes, when a CoFlow arrives, Aalo schedules *all* its flows at all their ports. This replicates and hence amplifies the above intrinsic queue-transit overhead and round-robin overhead at all those ports.

In this paper, we propose Philae, a new online CoFlow scheduler with a dramatically different approach to learn CoFlow sizes to enable online SJF. Like prior-art, Philae is non-clairvoyant, *i.e.,* it does not require any prior knowledge, and also approximates Smallest CoFlow First (SCF) derived from SJF. For optimal scheduling using SJF (or SCF) 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, and by using *sampling*, a highly effective technique used in large-scale surveys [36]. 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 using the estimated job size.

Scheduling pilot flows first before the rest of the flows can potentially incur two overheads. 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.

We present the complete Philae design that addresses four major design issues: (1) How to select pilot flows for each newly arriving CoFlow? (2) How to schedule the pilot flows? (3) How to schedule among all the CoFlows with estimated sizes? (4) How to schedule flows within a scheduled CoFlow?

We have implemented and evaluated Philae using a prototype on a 150-node cluster in Microsoft Azure, and large-scale simulations utilizing a public MapReduce trace from Facebook by varying the degree of flow size skew. Our simulation results show that, compared to prior art Aalo, Philae reduces the CCT by 1.78× (median) and 9.58× (P90). We also evaluated Philae on a testbed by keeping the fraction of the time spent by each job in the communication phase the same as given in Aalo [19]. Our testbed evaluation shows that Philae reduces the average job completion time by 18% compared to Aalo.

In summary, this paper makes the following contributions:

- 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.
- We propose the novel idea of applying sampling in the spatial dimension of CoFlow to significantly reduce the overhead of online learning CoFlow sizes. The technique can be generalized to other scheduling problems in distributed settings such as cluster job scheduling [41].
- We present the complete design and implementation of Philae.
- 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.

2 Background

In this section, we provide a brief background on the CoFlow abstraction, prior-art Aalo scheduler, and discuss its weaknesses.

2.1 CoFlow Abstraction

In data-intensive applications such as Hadoop [1] and Spark [2], the job completion time heavily depends on the completion time of the communication stage [9, 20]. The CoFlow abstraction [18] 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 [19, 21, 23, 29, 30].

2.2 Aalo Scheduler

A classic way to reduce average CCT is SCF (Shortest CoFlow First) [19] derived from classic SJF, where the CoFlow job length is loosely defined as the total bytes of the CoFlow, *i.e.,* 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 cluster dynamics such as failures and stragglers.

Aalo [19] 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 as the total data sent by the CoFlow exceeds the per-queue thresholds.

The above idea is actually generic and applicable to scheduling sequential jobs (*e.g.,* single flows) on a single work engine (*e.g.,* a single network port). To apply it to scheduling CoFlows which have flows in 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 every CoFlow (its flow) in each queue is never moved back.

2.3 Drawback: High Learning Overhead

Using multiple discrete priority queues to learn the CoFlow length in this way, however, can incur high overhead, as we discussed in §1 and summarized here:

Intrinsic queue-transit overhead: As a CoFlow transits through the queues before reaching its final queue Q_f, it blocks *shorter* CoFlows that would finish in the queues before Q_f, worsening their CCT.

Overhead due to inadvertent round-robin: Moving CoFlows across multiple FIFO-based priority queues effectively results in round-robin for *CoFlows of similar sizes*, worsening their average CCT.

Overhead in scheduling all flows: Scheduling all flows of a CoFlow at their ports effectively replicates and hence amplifies the above two flavors of overhead.

Intuitively, the learning overhead of a CoFlow C1, defined as the size (or the time spent in doing so) is completed before reaching its correct queue, directly translates into the delay incurred on any other CoFlow C2 that was "shorter" or "of same size" but did not finish until after the learning of CoFlow C1 completed. In other words, if an oracle told us that C2 was shorter than or of the same size as C1 when C1 first arrived, we could have allocated all the bandwidth resources used for learning C1 before C2 finishes to C2, so C2 would finish sooner, without affecting the completion time of C1.[1]

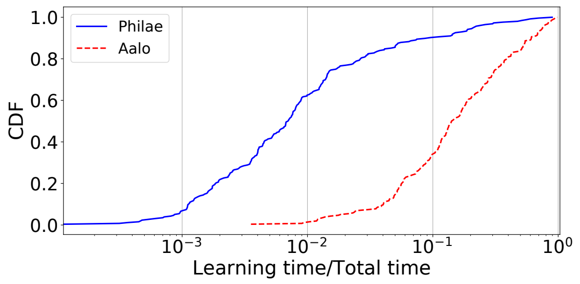


Figure 1. CDF of learning overhead per CoFlow, *i.e.,* the time to reach the correct priority queue as a fraction of CCT, excluding CoFlows directly scheduled by PHILAE or finish in Aalo's first queue.

Empirical measurement We quantify the overhead of CoFlow length learning using multiple queues under Aalo, using a trace from Facebook clusters [3]. We define the overhead as the time a CoFlow spent before reaching the correct (final) priority queue as a fraction of its CCT. 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.

3 Key Ideas

We propose PHILAE, a new online CoFlow scheduler with a very different approach to approximate online SJF. PHILAE makes a key observation about CoFlows that a CoFlow has a spatial dimension, *i.e.,* it typically consists of many flows, and preschedules sampled flows, called *pilot flows*, of each CoFlow and uses their measured size to estimate the CoFlow size. It then resorts to SJF or variations using the estimated CoFlow sizes.

Intuitively, such a sampling scheme avoids all three sources of overhead in Aalo (§2.3). Once the CoFlow sizes are learned, the CoFlows are assigned to the right correct queues, which avoids the intrinsic queue-transit and round-robin effects. Furthermore, sampling only a small subset of the flows avoids the amplification effect across multiple ports.

Conceptually, scheduling pilot flows first before the rest of the flows appears to negatively affect CCT in two ways. First, scheduling pilot flows of a newly arriving CoFlow consumes port bandwidth which can delay other CoFlows (with already estimated sizes). 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. We argue that there is going to be some overhead with any online CoFlow size learning scheme, and the key insight here is that compared to the multi-queue based approach, the overhead will be much smaller, a we discussed in §1.

[1]This is the same swapping argument in the optimality proof of SJF.

The more important question is whether and why sampling will work well in the presence of skew among the flow sizes of a CoFlow. We address this question below.

Why is sampling effective in the presence of skew? The flow sizes within a CoFlow may vary (*skew*). Such skew may lead to errors in estimating the total CoFlow size when we use samples from a few pilot flows. 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. We give both the intuition and theoretical underpinning for why sampling is effective below.

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 errors 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 either.

To illustrate the above effect, we use a simple model assuming that the flow sizes follow a normal distribution. 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 has a size that is normally distributed with mean μ_1 and variance σ_1^2, *i.i.d.* across flows. Similarly, assume that each flow of C_2 has a size that is normally distributed with mean μ_2 and variance σ_2^2, *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, we can show that,

$$\lim_{c\to\infty} \frac{\tilde{T}^c - T^c}{T^c} = Q\left(\frac{n_2\mu_2 - n_1\mu_1}{\sqrt{\frac{n_2^2\sigma_2^2}{m_2} + \frac{n_1^2\sigma_1^2}{m_1}}}\right)\frac{n_2\mu_2 - n_1\mu_1}{n_2\mu_2 + 2n_1\mu_1} \qquad (1)$$

where $Q(x) = \int_x^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}} du$ is the tail integration of the standard normal distribution. (Note that here 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, the relative gap between $\tilde{T}^c$ and T^c decreases as σ_1 and σ_2 decrease. In other words, as the skew of each CoFlow decreases, sampling becomes more effective. Second, when σ_1 and σ_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 Q 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 size 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 in the previous paragraph. The largest gap occurs when $n_2\mu_2 - n_1\mu_1$ is on the same order of the square root value.

Similar results can be obtained even when the flow size distribution is not normal. For example, if the flow size of C_1 (C_2) has a general distribution in a bounded interval $[a_1, b_1]$ ($[a_2, b_2]$) with mean μ_1 (μ_2). Then, we can show that,

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

Discussions similar to that for (1) can then be made here as well. 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. However, these 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 improves the estimation errors, it also incurs higher overhead and delay. Thus, our design (§4) need to have practical solutions that carefully address these issues.

Error-correction: Readers familiar with the online learning and multi-armed bandit (MAB) literature [11, 15, 25, 32] will notice that our key idea above does not attempt to correct the errors in the sampling step. In such literature, it is common to use an iterative procedure so that more samples are collected to correct any initial errors. Such an iterative procedure must carefully balance "exploration" (i.e., an arm that looks poor at the beginning may actually have high expected payoff in the long term) and "exploitation" (i.e., arms that already demonstrate high payoffs should be used more often). For instance, the Upper-confidence-bound (UCB) algorithm of [11] is a classical example that adjusts the sampled mean by a confidence interval that decreases with the number of samples, so that preference is given to arms that have only been sampled a small number of times, which may thus have large errors. In this way, UCB has been shown to balance exploitation and exploration. *However, in this paper we have not used this type of ideas due to a fundamental difference in our objective.* Specfically, our objective is to minimizing the average CCT, instead of maximizing the total payoff. To see the difficulty of UCB-type of algorithms, consider the case when we have two CoFlows whose sizes are nearly identical. A straight-forward way of using confidence intervals is to compute a lower-confidence-bound for the sampled mean, i.e., by subtracting the size of the confidence interval from the sampled mean[2]. In this way, preference is given to the CoFlow with a smaller size (according to the sampled mean) or with a smaller number of samples. Then, as new samples of the CoFlow are revealed, we recompute the confidence interval and reschedule all remaining flows accordingly. Under such a scheme, suppose that CoFlow C_1 initially has a smaller sampled mean than CoFlow C_2, but the sizes of their confidence intervals are initially the same. Thus, C_1 will be scheduled first. As new samples of C_1 becomes available, its confidence interval shrinks. It is then likely that C_2 will have a smaller lower-confidence-bound, and thus will be scheduled next. In this way, C_1 and C_2 are essentially scheduled in a round-robin manner because they turns becoming the one with the smaller lower-confidence-bound. As a result, they will both complete at about the same time. In contrast, a better strategy in this situation is to let one CoFlow runs until completion, before the other CoFlow starts. Due to this reason, in this paper we do not use confidence intervals nor an iterative procedure to correct sampling errors. Nonetheless, thanks to the reasons explained earlier by $eq.(1)$ and $eq.(2)$, we have observed that our proposed approach already produces superior results.

To verify the above intuition, we have conducted the following experiments with adding error correction in size estimation module(§4.2) the default Philae (§4). For this we need (1) a method to calculate the confidence interval on the estimated size; and (2) a method for error correction. In our experiments to evaluate Philae with error correction we use the following methods. (1) We calculate confidence interval of the estimated size by bootstraping[26]. Specifically, we resample 100 times from the same set of sampled flows (with repitition allowed) and then using the 100 averages calculate the lower bound of the confidence interval as $average - 3 * standard\_deviation$. (2) The error correction process triggers each time a specific set of flows complete in the following manner. Each set has the same number of flows as the number of pilot flows(§4.1), say p flows, for the CoFlow. These sets are ordered according to the time that the corresponding flows start, *i.e.,* first p flows that start after size estimation is over are grouped as the first set of error-correcting flows and the next p flows that start are grouped as second set, and so on. We perform the first round of error-correction (*i.e.,* recalculating the lower confidence bound using the method(1)(described above)) only after all flows from the first set are completed, and the second round after all flows from the second set are completed. In this way we avoid any sort of bias of the samples due to completion time. Using the above two features we evaluated three variants of the default Philae (§4) against Aalo based on the FB-CoFlow trace. (1) Philae-lower-confidence-bound: this variant uses the lower bound of the confidence interval of estimated size of a CoFlow as its estimated size (Note that in contrast the default Philae uses the unbiased mean size); (2) Philae-with-one-error-correction: on top of (1), we apply one round of error correction, after first set of p error-correction flows are completed, so that more flow sizes are used to update the estimate; (3) Philae-with-multiple-error-correction: on top of (1), error correction is applied multiple rounds till the CoFlow finishes. Our evaluation shows that variant (1) improves the average CCT over Aalo by 1.33× and median (P90) CCT speedup by 1.78× (10.75×), variant (2) improves average CCT by 1.27× and median (P90) CCT speedup by 1.59× (9.78×), and variant (3) degrades the average CCT by 0.95× and median (P90) CCT speedup is 1.06× (8.25×). However, the default Philae (without error correction) improves average CCT by 1.51× and median (P90) CCT speedup by 1.78× (9.58×) (§6.4). In summary, adding error-correction only seems to degrade the performance of Philae in all 3 variants

We note that this result does not preclude the possibilities that other iterative sampling algorithms may outperform Philae. Nonetheless, it illustrates that straight-forward extensions of UCB-type of ideas do not work well. How to find iterative sampling algorithms outperforming Philae remains an interesting direction for future work.

4 Philae Design

In this section, we elaborate key design principles in Philae. The key objective in Philae is to schedule CoFlows to *reduce the average CCT* without any prior knowledge of CoFlow sizes. Additionally, Philae aims to avoid starvation so that all CoFlows can continually make progress.

4.1 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 task in Philae is divided among (1) a central coordinator, and (2) local agents that run on individual ports. A computing framework such as Spark [42] first registers (removes) a CoFlow when a job arrives (finishes). Upon a new CoFlow arrival, old CoFlow completion, or pilot flow completion, the coordinator calculates a new CoFlow schedule, which includes (1) CoFlows that are to be scheduled in the next time slot, and (2) flow rates for the individual flows of a CoFlow, and pushes this information to the local agents which use this information

[2]It is more logical to use a lower-confidence-bound here because we aim to *minimize* the completion time, but not to *maximize* the payoffs.

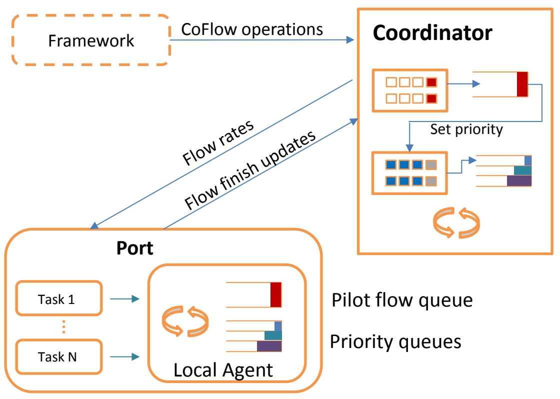


Figure 2. Philae architecture.

to allocate their bandwidth. The local agents will follow the current schedule until they receive a new schedule.

There are three major components in the Philae design: (1) *CoFlow size estimation:* How to choose and schedule the pilot flows for each newly arriving CoFlow? (2) *Inter-CoFlow scheduling:* How to schedule among all the CoFlows with estimated sizes? (3) *Intra-CoFlow scheduling:* How to allocate bandwidth among the flows once a CoFlow is scheduled to run? We discuss the above three design components and how to tackle other challenges, like starvation-freedom and work conservation.

4.2 Using pilot flows to estimate CoFlow size

As discussed in §3, Philae estimates the size of a CoFlow online by actually scheduling a subset of its flows (*pilot flows*) on their ports. We do not schedule the flows of a CoFlow other than the pilot flows until the completion of the pilot flows 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 §3, the number of pilot flows affects the trade-off between the CoFlow size estimation accuracy and scheduling overhead. In many workloads, the flow sizes tend to be skewed, which requires "sampling" the sizes of many pilot flows of a CoFlow to accurately estimate the total CoFlow size. 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.

Therefore, we will experiment with several ways of choosing the number of pilot flow. The 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 consists of flows between a set of senders (*e.g.,* mappers) and a set of receivers (*e.g.,* reducers) [22], 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 (§6.2) 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 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 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. 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.

4.3 Inter-CoFlow scheduling policies

Shortest job or remaining time first heuristics are shown to be optimal when minimizing average execution time when execution depends on single resource (*e.g.,* processes in single-CPU system or flows on a single link). Naturally, CoFlows can be scheduled using Shortest (or Smallest) CoFlow First. However, such scheduling can be suboptimal as it ignores *contention* of the CoFlow, *i.e.,* the number of other CoFlows blocked on its multiple ports, which can vary at different ports [30]. Effectively, when a CoFlow is scheduled, the increase in the waiting time of other CoFlows depends on the CoFlow size as well as the *contention* caused by scheduling individual flows at individual ports.

In Philae, we explore six different scheduling policies based on different combinations of CoFlow size and contention, 2 size-based algorithms (A, B) and 1 contention-based, similar to what is used in [30] (C), and 3 contention-and-length-based algorithms (D, E, F):

- **(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 contention first:** CoFlows are sorted based on their contention (c).
- **(D) Least length-weighted contention first:** CoFlows are sorted based on the increase in the waiting time for other CoFlows ($c \cdot l$)

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

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

Philae uses Algorithm F, least total-port contention first, as it results in the least average CCT (§6). For all six algorithms, we continue to use the priority-queue based scheduling, and the algorithms only differ on what metric they use in assigning CoFlows to the priority queues.

We use the following parameters of a CoFlow to define the metrics in scheduling algorithms: (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.

4.4 Intra-CoFlow scheduling: rate allocation

Once the scheduling order of the CoFlows is determined, we need to determine the rates for the individual flows of a CoFlow. One can simply assign the *entire* bandwidth at all sender and receiver ports to the CoFlow at the head of the queue. However, such bandwidth allocation can be wasteful when there are uneven numbers of senders and receivers in a CoFlow. For a CoFlow with s sender and r receiver ports, if the bandwidth of each port is B, the bandwidth assigned to each flow is $b = \frac{B}{max(s,r)}$, and the bandwidth utilization (fraction of bandwidth used) at each of the sender and receiver ports are $\frac{r}{max(s,r)}$ and $\frac{s}{max(s,r)}$, respectively, since there are r flows at each sender and s flows at each receiver. Thus, when $s \neq r$, either the sender or receiver ports will be underutilized. To improve bandwidth utilization, Philae assigns flow rates depending on the available bandwidth as follows.

In Philae, we assign the rate of the slowest flow to all the flows of a CoFlow as there is no benefit in speeding-up flows at certain ports when the CCT depends on the slowest flow. At a port, we use max-min fairness to schedule the individual flows of a CoFlow (to different receivers).

4.5 CoFlow scheduling with starvation avoidance

Once the CoFlow sizes are estimated, the natural way of scheduling CoFlows is to use shortest- or smallest- job first (SJF) (or their variations) because of their known optimality. In SJF, the CoFlows are scheduled in the increasing order of their sizes. Alternatively, Philae can use Shortest-Effective-Bottleneck first (SEBF) in Varys [21], which assumes known CoFlow size and width. However, neither approach accounts for starvation avoidance – an intrinsic drawback of any SJF-like online schemes.

There are many ways to mitigate the starvation issue. However, a subtlety arises where 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 comparison, we need to ensure that both Philae and Aalo provide the same level of starvation freedom (or progress measure). For this reason, we inherit the multiple priority queue structure from Aalo for CoFlow scheduling in this paper. 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}, and $\mathrm{Q}_0^{lo} = 0$, $\mathrm{Q}_{N-1}^{hi} = \infty$, $\mathrm{Q}_{q+1}^{lo} = \mathrm{Q}_q^{hi}$. Philae uses exponentially growing queue thresholds, *i.e.,* $\mathrm{Q}_{q+1}^{hi} = \mathrm{E} \cdot \mathrm{Q}_q^{hi}$. By default, Philae sets E = 10, K = 10, shown to be effective in our evaluation (§6.6).

Once we estimate the CoFlow size (using pilot flows), we assign it to the priority queue using inter-CoFlow policies discussed in §4.3. 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. In Philae, the weights decrease exponentially (with a factor of 10) 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 between both Philae and Aalo [19].

4.6 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.

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; §6.6 provides sensitivity analysis for thresholds).

4.7 Failure tolerance and recovery

Cluster dynamics such as node failure or stragglers can delay some of the flows of a CoFlow, increasing their CCT. The current Philae design automatically self-adjusts to speed up CoFlows that are slowed down due to cluster dynamics using the following mechanisms: (1) We adjust 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) We calculate contention only on the ports that have unfinished flows.

4.8 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.

5 Implementation

We implemented Philae 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.

As discussed in §4.1, the coordinator calculates a new schedule consisting of flow rates on-demand, upon three possible events – arrival of a new CoFlow, or completion of a CoFlow or pilot flows of a CoFlow, and pushes them to local agents right away. In practice, to avoid frequent rate calculation due to frequent events, Philae caps rate calculation to at most once every δ interval, *i.e.,* it performs at most one rate calculation at the end each interval of duration δ which incorporates all events happened in that interval.

Unlike prior online CoFlow schedulers[19, 30] Philae's coordinator does not need updates from local agents every interval(δ). It only needs updates on arrival and completion of flows. This design approach of Philae makes coordinator highly scalable and less prone to communication dynamics between coordinator and local-agents. We demonstrate this by large scale real testbed evaluation in §7. The coordinator also manages the explicit ON/OFF signals about individual flows to manage receiver side contention. The coordinator is stateless [3], as it makes scheduling decisions on the latest flow stats received from the local agents, which makes it easy to recover from failures.

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 CoFlow 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. Lastly, the local agents are optimized for low CPU and memory overhead (evaluated in §7.4), enabling them to fit well in the cloud settings [35].

[3]Coordinator only keeps track of receiver contention signals, which may only lose a fraction of bandwidth if the coordinator fails.

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.

6 Simulation

We evaluated Philae using a 150-node testbed cluster in Azure and using large scale simulations by utilizing a publicly available Hive/MapReduce trace collected from a 300-machine, 150-rack Facebook production cluster [3].

- **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. However, the trace only contains information about the total data received at each reducer and so the total data of a reducer is equally divided among all the incoming flows to a reducer.

Due to the lack of other publicly available CoFlow trace[4], we derived three additional 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.
- **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 [10]. In particular, the sizes are adjusted so that the coefficients of variation across mapper data are 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 highlights of our evaluation results are:

[4]A challenge that has been faced by previous work on CoFlows such as [19, 28, 29, 44].

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 speed ups 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 speed ups are 1.75× in the median case (P90 = 12.0×).
2. The CCT improvement mainly stems from the reduction in the penalty (in terms of latency and amount of data sent) in determining the right queue for the CoFlows. Compared to Aalo, median reduction in the latency in finding the right queue for CoFlows in Philae is 19.0×, and in data sent is 20.0×. (§6.2).
3. Philae improvements are consistent when varying the skew among the flow sizes in a CoFlow (§6.5).
4. Philae improvements are consistent when varying its parameters (§6.6).
5. The Philae prototype is fast and has small memory and CPU footprints (§7.4).

We present detailed simulation results in this section, and the testbed evaluation of our prototype in §7.

Experimental setup: Our simulated cluster uses the same number of nodes (sending and receiving network ports) as per the trace. As in Aalo [19], we assume full bisection bandwidth is available, and congestion can happen only at the 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 in Aalo the new schedule calculation interval δ is set to 8ms, the default suggested in it's publicly available simulator [19]. In Philae, a new schedule is calculated on-demand, upon arrival of a new CoFlow, or competition of CoFlow completion or pilot flows of a CoFlow. Finally, in Philae the threshold for thinLimit (T) is set to 7 and the number of pilot flows assigned to wide CoFlows are $max(1, 0.05 \cdot S)$, where S is the number of senders, and the default inter-CoFlow scheduling policy in Philae is Least length-weighted total-port contention.

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

6.1 Pilot flow selection policies

We start by evaluating the impact of different policies in choosing the pilot flows for a CoFlow in Philae. Table 1 summarizes the improvement in average CCT of Philae over Aalo and average error in size estimation normalized to the actual CoFlow size, when varying pilot flow selection while keeping other parameters as the default in Philae.

Unsurprisingly, the estimation accuracy increases with increasing number of pilot flows across the three schemes: constant, fraction of senders, and fraction of total flows. However, as the number of pilot flows increases, the CCT speedup (P50 and P90 of individual coFlow CCT speedups) decreases (among the range of parameter choices). 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 and thus set it ($0.05 \cdot S$) as the default pilot flow selection policy.

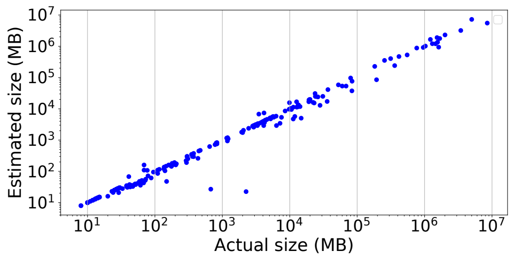


Figure 3. Philae CoFlow size learning accuracy. CoFlows that did not go through the piloting phase (48%) are not shown.

6.2 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 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% (95%) CoFlows (in terms of accuracy), the average and standard deviation of relative error are only 0.05 (0.03) and 0.12 (0.07). 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) falls in between 6.8 - 4.6.

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×.

Table 1. Performance improvement over Aalo for varying pilot flow selection schemes.

	Constant	Proportional to num(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%	2.9%
Avg. CCT	1.27x	1.51x	1.45x	1.50x	1.50x	1.50x	1.43x	0.49x
P50 speed up	1.75x	1.78x	1.76x	1.71x	1.52x	1.40x	1.33x	0.69x
P90 speed up	9.0x	9.58x	9.0x	9.15x	8.33x	8.45x	8.23x	8.23x

6.3 Inter-CoFlow scheduling policies

Next, we evaluate the effectiveness of the six inter-CoFlow scheduling policies of Philae discussed in §4.3, keeping the remaining parameters as the default.

Table 2 shows the CCT improvements of Philae over Aalo. We make the following observations.

First, **Smallest job first (A)** and **Shortest remaining time first (B)** perform reasonably well and similarly. This is because the pre-emptive 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 A and B and thus they are performing similarly.

Second, **Least contention first (C)** performs poorly. This is because 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.

Third, **Least length-weighted contention first (D)** and **Least max length-weighted port contention first (E)** 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 coarse notion (D) or extreme notion (E) 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 poor average CCT.

The **Least length-weighted total-port contention (F)** (Philae) which uses the sum of size-weighted port contention to assign CoFlows to priority queues out-performs all the other policies. This is because it captures the diversity of contention at different ports, which happen often in real distributed settings, and at the same time accounts for the CoFlow size by taking length-weighted sum of the port-wise contention.

6.4 Average CCT Improvement

We now compare the CCT speedups of Philae against 5 well-known CoFlow scheduling policies: (1) Aalo, (2) Aalo-Oracle, which is an oracle version of Aalo where the scheduler knows the final queue of the CoFlow upon its arrival time and directly start 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) FIFO, which is a single queue FIFO based CoFlow scheduler, and (5) FAIR, which uses per-flow fair sharing. We do not include Saath [30] in the comparison as it does not provide the same liveness guarantees as Philae. 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.

Table 2. CCT Speedup in Philae under different inter-CoFlow scheduling policies (§4.3) over Aalo.

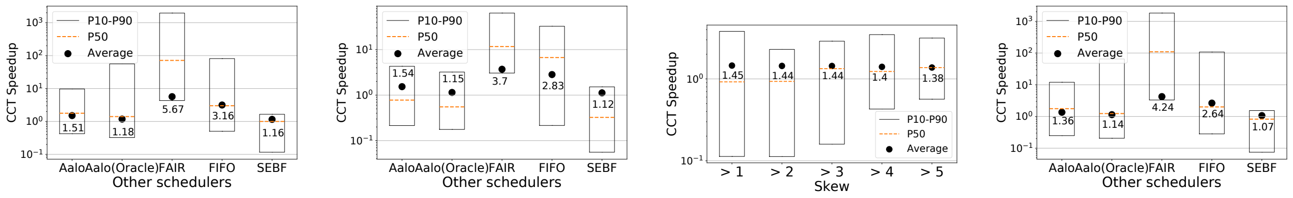
Priority estimation metric	P50	P90	Avg. CCT
Estimated size (A)	1.48x	8.27x	1.40x
Remaining size (B)	1.54x	8.34x	1.37x
Global Contention (C)	0.75x	8.26x	0.13x
Length-weighted global contention (D)	1.68x	10.25x	1.22x
Max of length-weighted port contention (E)	1.43x	9.0x	1.13x
Total length-weighted port contention (F) (Philae)	1.78x	9.58x	1.51x

First, to isolate any improvement due to improved scheduling policy from fast size estimation, we compare CCT under Philae against Aalo-Oracle, where we start all CoFlows at the right priority queue (*i.e.,* no learning overhead). Philae improves the average CCT by 1.18×, and P50 (P90) CCT by 1.4× (55.0×), respectively. This shows 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 1.78× (median) and 9.59× (P90). The significant additional improvement on top of the gain over Aalo-Oracle comes from fast and accurate estimation of the right queue for the CoFlows (Fig. 1).

Third, Philae, which requires no CoFlow size knowledge apriori, 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.

Next, we compare Philae against a FIFO-based CoFlow scheduler under which CoFlows are scheduled at individual ports, without any global coordination, in the order of their arrival time. In this scheme, there are no multiple priority queues. Lack of coordination, coupled with lack of priority queues, severely hampers the CCT. Philae achieves a



(a) Using original FB trace. (b) Using Wide-CoFlows-only trace. (c) Using 5 Low-skew-filtered traces. (d) Using the Mantri-like trace.

Figure 4. CCT speedup using PHILAE as compared to other CoFlow schedulers on different traces. In Fig. 4(c), the x-axis denotes the minimum skew in the 5 Low-skew-filtered traces.

Table 3. 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 ≤ 100MB (small)	bin-1 (44.3%)	bin-2 (24.1%)
size > 100MB (large)	bin-3 (4.5%)	bin-4 (27%)

median(P90) CCT speedup of 3.0 (77.96)× and improves the average CCT by 3.16× over the FIFO-based scheduler.

Finally, we compare PHILAE against an un-coordinated flow level scheduler, where there are no queues, and all the flows are scheduled upon arrival as per TCP. Again, lack of coordination, coupled with lack of priority queues and no notion of CoFlows, severely hampers the CCT, and PHILAE achieves median(P90) CCT speedup of 70.82× (1947×) and average CCT improvement of 5.66×.

In summary, our results above show that PHILAE is far more effective in reducing CoFlow completion time than existing non-clairvoyant CoFlow schedulers. Further, it achieves similar performance as SEBF which assumes prior knowledge of CoFlow sizes.

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 3, 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 (§4). Although the flows are short, they get delayed until the completion of the pilot flows, which results in CCT increase.

Finally, since thin CoFlows benefits 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 the results. We see that PHILAE continues to perform well, reducing the average CCT by 1.54×, 1.15×, and 1.12× over Aalo, Aalo-Oracle, and SEBF, respectively.

6.5 Robustness to CoFlow Data Skew

Previous work [21] has shown that most CoFlows in large production clusters have no skew, a majority among the remaining have very low skew, and only very few of them have high skew. Nevertheless, to rigoursoulsy evaluate PHILAE, we further compare it against Aalo using traces with varying skew patterns.

First, we evaluate PHILAE using the Mantri-like trace. Fig. 4(d) 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. 4(c) shows that PHILAE performs better than Aalo even with highly skewed traces and reduces the average CCT by 1.45×, 1.44×, 1.44×, 1.4× and 1.38× for the five Low-skew-filtered traces containing CoFlows with skew of at-least 1, 2, 3, 4 and 5, respectively.

6.6 Sensitivity Analysis

Next, we evaluate the sensitivity of PHILAE to different design parameters.

Thin CoFlow bypassing limit (T) In this experiment, we vary the thinLimit (T) in PHILAE for bypassing CoFlows from the probing phase. The results in Fig. 6(a) shows that the average CCT remains almost the same as T increases. However, the P50 speedup increases till $T = 7$, and tapers off after $T = 7$. The key reason for the CCT improvement until $T = 7$ is that all flows of the thin CoFlows (with width ≤ 7) are scheduled immediately upon arrival which improves their CCT (§4.6), and the majority of the CoFlows are thin.

Start queue threshold (Q_0^{hi}) We next vary the threshold for the first priority queue from 2 MB to 64 MB. Fig. 6(b) shows as the average CCT. PHILAE is not very sensitive to the threshold of first priority queue and fluctuation from default is at most 8%. For 2 MB the performance is similar to default PHILAE as the threshold for lower priority queues got doubled up and essentially did not make much difference in how CoFlows are being seggregated in queues. However, with the first queue threshold as 4 MB, the way CoFlows are being seggregated, especially the larger one's, is changing and so we have slight(8%) difference in performance as compared to default. However, with the threshold being 8 MB and 16 MB

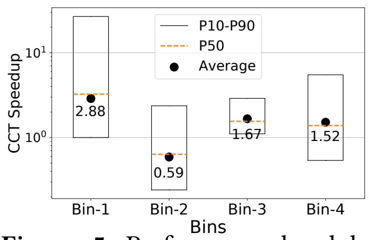
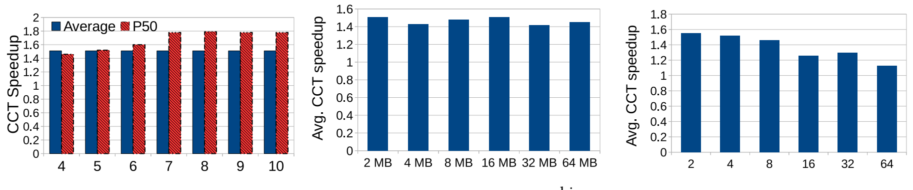


Figure 5. Performance breakdown into bins shown in Table 3.



(a) Thin CoFlow bypass threshold (b) First queue capacity(Q_0^{hi}) (c) Exponent (E)

Figure 6. PHILAE sensitivity analysis. We vary one parameter of PHILAE keeping rest same as default and compare it with default Aalo.

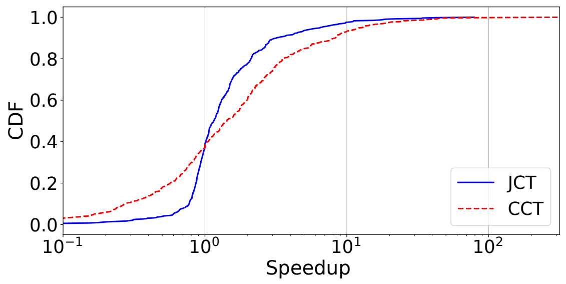


Figure 7. [Testbed] Distribution of speedup in CCT and JCT in PHILAE using the FB trace.

it is again similar to the default with again slight fluctuation on increasing the threshold further.

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. 6(c), smaller queue threshold multiplication factor which leads to more queues performs better because of fine-grained priority segregation.

7 Testbed Evaluation

Next, we deployed PHILAE in a 150-machine Azure cluster to evaluate PHILAE's performance.

Testbed setup: In testbed experiments, we rerun the FB trace on Spark-like framework on a 150-node cluster in Microsoft Azure [4]. The coordinator runs on a Standard F4s server with 4-core 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor and 8GB memory. The local agents run on D2v2 with the same processor with 2-core and 7GB memory. The machines on which local agents run have 1 Gbps network bandwidth. Similar to 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.

7.1 CCT Improvement

In this experiment, we measure CCT improvements of PHILAE compared to Aalo. Fig. 7 shows the CDF of the CCT speedup of individual CoFlows under PHILAE compared to

Table 4. [Testbed] CCT improvement in PHILAE as compared to Aalo.

	P50	P90	Avg. CCT
FB Trace	1.43×	8.29×	1.48×
Wide-CoFlow-only	0.99×	2.16×	1.52×

Aalo. The average CCT improvement is 1.48× which is similar to the results in the simulation experiments. We also observe 1.43× P50 speedup and 8.29× P90 speedup.

We also evaluated PHILAE using the Wide-CoFlow-only trace. Table 4 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 and PHILAE is speeding up large CoFlows, and the Wide-CoFlow-only trace consists of mostly large CoFlows.

7.2 Job Completion Time

Next, we evaluate how the improvements in CCT affects the job completion time. In data clusters, different jobs spend different fractions of their total job time in data shuffle. In this experiment, the fraction of time that the jobs spent in the shuffle phase follows the same distribution used in Aalo [19]. Fig. 7 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 3.26× in the 90th percentile. This shows that improved CCT translates into better job completion time. As expected, the improvement in job completion time is smaller than the improvements 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.

7.3 Robustness to network error

As discussed in §5, 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 5 shows the mean-normalized standard deviation in the 10th, 50th, 90th 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.

Table 5. [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 6. [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

7.4 Resource Utilization

Finally, we evaluate, for both Philae and Aalo, the resource utilization at the coordinator and the local agents (Table 6) 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 6, 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.

8 Related Work

CoFlow scheduling: Varys [21] is an offline CoFlows scheduler that assumes *prior* CoFlow information is available. In contrast, Philae is an online scheduler that provides roughly equal performance of Varys. Like Philae, Baraat [23], Aalo [19] and Saath [30] schedule CoFlows in online settings. Baraat is a de-centralized scheduler without coordination among the ports and suffers from the same limitations as Aalo. Saath [30] suffers with limitations similar to Aalo, and does not provide same liveness guarantees as Philae. CODA [44] tackles an orthogonal problem, of identifying flows of individual CoFlows.

Speculative scheduling Recent works ([16, 34]) use the idea of online requirement estimation for scheduling in datacenter. In [31], recurring big data analytics jobs are scheduled using their history.

Flow scheduling: There have been a rich body of prior work on flow scheduling. Efforts to minimize flow completion time (FCT), both with prior information (*e.g.*, PDQ [27], pFabric [6]) and without prior information (*e.g.*, Fastpass [37], PIAS [12], [13]), fall short in minimizing CCTs which depend on the completion of the last flow [21]. Similarly, Hedera [5] 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.

Job scheduling: There have been much work on scheduling in analytic systems and storage at scale by improving speculative tasks [8, 9, 43], improving locality [7, 41], and end-point flexibility [17, 39]. The CoFlow abstraction is complimentary to these work, and can benefit from them. Combining CoFlow with these approaches 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 [38], FIFO with backfilling [33] and gang scheduling [24] have been proposed.

9 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 CoFlow 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 – with the median CCT speedup of 1.78× in simulation and 1.43× on testbed for the publicly available Facebook CoFlow trace. Further, the sampling-in-spatial-dimension technique can be generalized to other distributed scheduling problems such as cluster job scheduling.

References

[1] Apache Hadoop. http://hadoop.apache.org.

[2] Apache Spark. http://spark.apache.org.

[3] CoFlow trace from Facebook datacenter. https://github.com/coflow/coflow-benchmark.

[4] Microsoft Azure. http://azure.microsoft.com.

[5] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI* 2010.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *ACM SIGCOMM* 2013.

[7] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *EuroSys* 2011.

[8] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI* 2013.

[9] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *OSDI* 2010.

[10] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 265–278 2010. http://dl.acm.org/citation.cfm?id=1924943.1924962

[11] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2-3 (2002), 235–256.

[12] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. PIAS: Practical Information-Agnostic Flow Scheduling for Data Center Networks. In *ACM HotNets* 2014.

[13] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 455–468 2015. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/bai

[14] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT* 2011.

[15] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning* 5, 1 (2012), 1–122.

[16] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 239–252 2017. DOI: https://doi.org/10.1145/3098822.3098840

[17] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging Endpoint Flexibility in Data-intensive Clusters. In *ACM SIGCOMM* 2013.

[18] Mosharaf Chowdhury and Ion Stoica. Coflow: A Networking Abstraction for Cluster Applications. In *HotNets-XI* 2012.

[19] Mosharaf Chowdhury and Ion Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *ACM SIGCOMM* 2015.

[20] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM* 2011.

[21] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *ACM SIGCOMM* 2014.

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI 2004*.

[23] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *ACM SIGCOMM* 2014.

[24] Dror G. Feitelson and Morris A. Jettee. Improved Utilization and Respon- siveness with Gang Scheduling. In *Job Scheduling Strategies for Parallel Processing* 2005.

[25] John Gittins, Kevin Glazebrook, and Richard Weber. 2011. *Multi-armed bandit allocation indices*. John Wiley & Sons.

[26] Tim Hesterberg, Shaun Monaghan, David S. Moore, Ashley Clipson, and Rachel Epstein. 2003. *Bootstrap Methods and Permutation Tests*. W. H. Freeman and Company, New York.

[27] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing Flows Quickly with Preemptive Scheduling. *ACM SIGCOMM* (2012).

[28] Xin Sunny Huang, Xiaoye Steven Sun, and T.S. Eugene Ng. Sunflow: Efficient Optical Circuit Scheduling for Coflows. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 297–311 2016. DOI: https://doi.org/10.1145/2999572.2999592

[29] Akshay Jajoo, Rohan Gandhi, and Y. Charlie Hu. Graviton: Twisting Space and Time to Speed-up CoFlows. In *USENIX HotCloud* 2016.

[30] Akshay Jajoo, Rohan Gandhi, Y. Charlie Hu, and Cheng-Kok Koh. Saath: Speeding Up CoFlows by Exploiting the Spatial Dimension. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '17)*. ACM, New York, NY, USA, 439–450 2017. DOI: https://doi.org/10.1145/3143361.3143364

[31] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *ACM SIGCOMM* 2015.

[32] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics* 6, 1 (1985), 4–22.

[33] David A. Lifka. The ANL/IBM SP Scheduling System. In *IPPS* 1998.

[34] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *ACM SIGCOMM* 2016.

[35] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *NSDI* 2013.

[36] Stanley Lemeshow Paul S. Levy. 2012. *Sampling of Populations: Methods and Applications* (4 ed.). Wiley.

[37] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *ACM SIGCOMM* 2014.

[38] Uwe Schwiegelshohn and Ramin Yahyapour. Analysis of First-come-first-serve Parallel Job Scheduling. In *SODA* 1998.

[39] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *USENIX OSDI* 2012.

[40] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2010. *Process Scheduling. Operating System Concepts* (8 ed.). John Wiley & Sons.

[41] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys* 2010.

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10 2010. http://dl.acm.org/citation.cfm?id=1863103.1863113

[43] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI'08* 2008.

[44] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *ACM SIGCOMM* 2015.