Efficient Coflow Scheduling Without Prior Knowledge

Mosharaf Chowdhury, Ion Stoica
UC Berkeley
{mosharaf, istoica}@cs.berkeley.edu

ABSTRACT

Inter-coflow scheduling improves application-level communication performance in data-parallel clusters. However, existing efficient schedulers require a priori coflow information and ignore cluster dynamics like pipelining, task failures, and speculative executions, which limit their applicability. Schedulers without prior knowledge compromise on performance to avoid head-of-line blocking. In this paper, we present Aalo that strikes a balance and efficiently schedules coflows *without* prior knowledge.

Aalo employs *Discretized Coflow-Aware Least-Attained Service* (D-CLAS) to separate coflows into a small number of priority queues based on how much they have already sent across the cluster. By performing prioritization across queues and by scheduling coflows in the FIFO order within each queue, Aalo's non-clairvoyant scheduler reduces coflow completion times while guaranteeing starvation freedom. EC2 deployments and trace-driven simulations show that communication stages complete $1.93\times$ faster on average and $3.59\times$ faster at the 95th percentile using Aalo in comparison to per-flow mechanisms. Aalo's performance is comparable to that of solutions using prior knowledge, and Aalo outperforms them in presence of cluster dynamics.

CCS Concepts

•**Networks** → **Cloud computing;**

Keywords

Coflow; data-intensive applications; datacenter networks

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom

© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3542-3/15/08…$15.00

DOI: http://dx.doi.org/10.1145/2785956.2787480

1 Introduction

Communication is crucial for analytics at scale [19, 8, 12, 20, 25]. Yet, until recently, researchers and practitioners have largely overlooked application-level requirements when improving network-level metrics like flow-level fairness and flow completion time (FCT) [29, 10, 16, 8, 14]. The coflow abstraction [18] bridges this gap by exposing application-level semantics to the network. It builds upon the *all-or-nothing* property observed in many aspects of data-parallel computing like task scheduling [51, 12] and distributed cache allocation [11]; for the network, it means all flows must complete for the completion of a communication stage. Indeed, decreasing a coflow's completion time (CCT) can lead to faster completion of corresponding job [20, 25, 19].

However, inter-coflow scheduling to minimize the average CCT is NP-hard [20]. Existing FIFO-based solutions, e.g., Baraat [25] and Orchestra [19], compromise on performance by multiplexing coflows to avoid head-of-line blocking. Varys [20] improves performance using heuristics like smallest-bottleneck-first and smallest-total-size-first, but it assumes *complete* prior knowledge of coflow characteristics like the number of flows, their sizes, and endpoints.

Unfortunately, in many cases, coflow characteristics are unknown a priori. Multi-stage jobs use pipelining between successive computation stages [30, 22, 46, 3] – i.e., data is transferred as soon as it is generated – making it hard to know the size of each flow. Moreover, a single stage may consist of multiple waves [11],[1] preventing all flows within a coflow from starting together. Finally, task failures and speculation [50, 30, 24] result in redundant flows; meaning, the exact number of flows or their endpoints cannot be determined until a coflow has completed. Consequently, coflow schedulers that rely on prior knowledge remain inapplicable to a large number of use cases.

In this paper, we present a coordinated inter-coflow scheduler – called *Coflow-Aware Least-Attained Service* (CLAS) – to minimize the average CCT *without* any prior knowledge of coflow characteristics. CLAS generalizes the classic least-attained service (LAS) scheduling discipline [45] to coflows. However, instead of independently considering the number of bytes sent by each flow, CLAS takes into account the *total*

[1] A *wave* is defined as the set of parallel *tasks* from the same *stage* of a *job* that have been scheduled together.

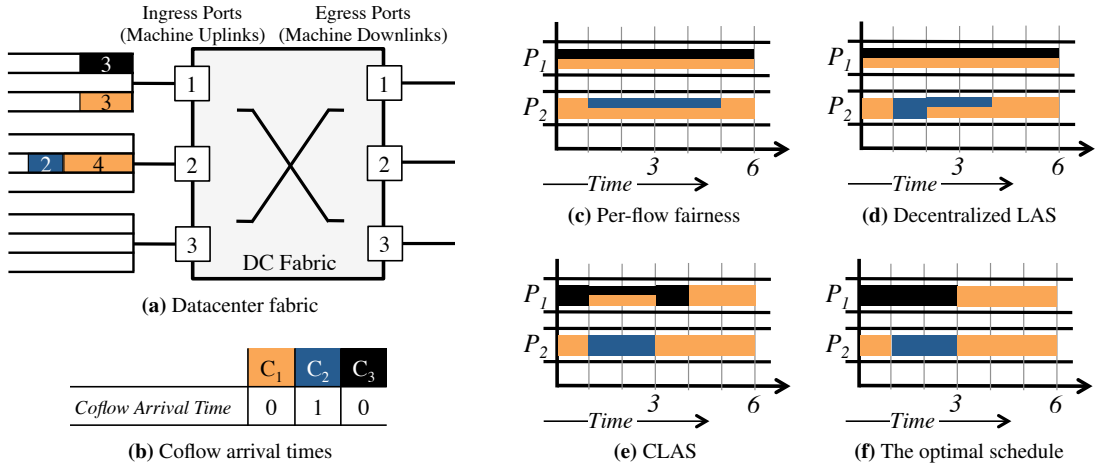


Figure 1: Online coflow scheduling over a 3×3 datacenter fabric with three ingress/egress ports (a). Flows in ingress ports are organized by destinations and color-coded by coflows – C_1 in orange/light, C_2 in blue/dark, and C_3 in black. Coflows arrive online over time (b). Assuming each port can transfer one unit of data in one time unit, (c)–(f) depict the allocations of *ingress* port capacities (vertical axis) for different mechanisms: The average CCT for (c) per-flow fairness is 5.33 time units; (d) decentralized LAS is 5 time units; (e) CLAS with instant coordination is 4 time units; and (f) the optimal schedule is 3.67 time units.

number of bytes sent by *all* the flows of a coflow. In particular, CLAS *assigns each coflow a priority that is decreasing in the total number of bytes the coflow has already sent.* As a result, smaller coflows have higher priorities than larger ones, which helps in reducing the average CCT. Note that for heavy-tailed distributions of coflow sizes, CLAS approximates the smallest-total-size-first heuristic,[2] which has been shown to work well for realistic workloads [20].

For light-tailed distributions of coflow sizes, however, a straightforward implementation of CLAS can lead to fine-grained sharing,[3] which is known to be suboptimal for minimizing the average CCT [19, 25, 20]. The optimal schedule in such cases is FIFO [25].

We address this dilemma by *discretizing* coflow priorities. Instead of decreasing a coflow's priority based on every byte it sends, we decrease its priority only when the number of bytes it has sent exceeds some predefined thresholds. We call this discipline *Discretized CLAS*, or D-CLAS for short (§4). In particular, we use exponentially-spaced thresholds, where the i-th threshold equals b^i, $(b > 1)$.

We implement D-CLAS using a multi-level scheduler, where each queue maintains all the coflows with the same priority. Within each queue, coflows follow the FIFO order. Across queues, we use weighted fair queuing at the coflow granularity, where weights are based on the queues' priorities. Using weighted sharing, instead of strict priorities, avoids starvation because each queue is guaranteed to receive some non-zero service. By approximating FIFO (as in Baraat [25]) for light-tailed coflows and smallest-coflow-first (as in Varys [20]) for heavy-tailed coflows, the proposed scheduler works well in practice.

We have implemented D-CLAS in Aalo,[4] a system that supports coflow dependencies and pipelines, and works well in presence of cluster dynamics like multi-wave scheduling (§5). Aalo requires *no* prior knowledge of coflow characteristics, e.g., coflow size, number of flows in the coflow, or its endpoints. While Aalo needs to track the total number of bytes sent by a coflow to update its priority,[5] this requires only loose coordination as priority thresholds are coarse. Moreover, coflows whose total sizes are smaller than the first priority threshold require no coordination. Aalo runs *without* any changes to the network or user jobs, and data-parallel applications require minimal changes to use it (§6).

We deployed Aalo on a 100-machine EC2 cluster and evaluated it by replaying production traces from Facebook and with TPC-DS [6] queries (§7). Aalo improved CCTs both on average (up to 2.25×) and at high percentiles (2.93× at the 95th percentile) w.r.t. per-flow fair sharing, which decreased corresponding job completion times. Aalo's average improvements were within 12% of Varys for single-stage, single-wave coflows, and it outperformed Varys for multi-stage, multi-wave coflows by up to 3.7× by removing artificial barriers and through dependency-aware scheduling. In trace-driven simulations, we found Aalo to perform 2.7× better than per-flow fair sharing and up to 16× better than fully decentralized solutions that suffer significantly due to the lack of coordination. Simulations show that Aalo performs well across a wide range of parameter space and coflow distributions.

We discuss current limitations and relevant future research in Section 8 and compare Aalo to related work in Section 9.

[2]Under the heavy-tailed distribution assumption, the number of bytes already sent is a good predictor of the actual coflow size [41].

[3]Consider two identical coflows, C_A and C_B, that start at the same time. As soon as we send data from coflow C_A, its priority will decrease, and we will have to schedule coflow C_B. Thus, both coflows will continuously be interleaved and will finish roughly at the same time – both taking twice as much time as a single coflow in isolation.

[4]In Bangla, Aalo (pronounced \'ä-lō\) means light.

[5]As stated by **Theorem A.1** in Appendix A, *any* coflow scheduler's performance can drop dramatically in the absence of coordination.

2 Motivation

Before presenting our design, it is important to understand the challenges and opportunities in non-clairvoyant coflow scheduling for data-parallel directed acyclic graphs (DAGs).

2.1 Background

Non-Clairvoyant Coflows A coflow is a collection of parallel flows with distributed endpoints, and it completes after all its flows have completed [18, 20, 19]. Jobs with one coflow finish faster when coflows complete faster [20]. Data-parallel DAGs [30, 50, 46, 2, 3] with multiple stages can be represented by multiple coflows with dependencies between them. However, push-forward pipelining between subsequent computation stages of a DAG [22, 30, 46, 3] removes barriers at the end of coflows, and knowing flow sizes becomes infeasible. Due to multi-wave scheduling [11], all flows of a coflow do not start at the same time either.

Hence, unlike existing work [19, 20, 25], we *do not assume* anything about a coflow's characteristics like the number of flows, endpoints, or waves, the size of each flow, their arrival times, or the presence of barriers.

Non-Blocking Fabric In our analysis, we abstract out the entire datacenter fabric as one non-blocking switch [10, 15, 20, 9, 31, 26] and consider machine uplinks and downlinks as the only sources of contention (Figure 1a). This model is attractive for its simplicity, and recent advances in datacenter fabrics [9, 28, 40] make it practical as well. However, we use this abstraction only to simplify our analysis; we do not require nor enforce this in our evaluation (§7).

2.2 Challenges

An efficient non-clairvoyant [39] coflow scheduler must address two primary challenges:

1. *Scheduling without complete knowledge:* Without a priori knowledge of coflows, heuristics like smallest-bottleneck-first [20] are inapplicable – one cannot schedule coflows based on unknown bottlenecks. Worse, redundant flows from restarted and speculative tasks unpredictably affect a coflow's structure and bottlenecks. While FIFO-based schedulers (e.g., FIFO-LM in Baraat [25]) do not need complete knowledge, they multiplex to avoid head-of-line blocking, losing performance.
2. *Need for coordination:* Coordination is the key to performance in coflow scheduling. We show analytically (**Theorem A.1**) and empirically (§7.2.1, §7.6) that fully decentralized schedulers like Baraat [25] can perform poorly in data-parallel clusters because *local-only* observations are poor indicators of CCTs of large coflows. Fully centralized solutions like Varys [20], on the contrary, introduce high overheads for small coflows.

2.3 Potential Gains

Given the advantages of coflow scheduling and the inability of clairvoyant schedulers to support dynamic coflow modifications and dependencies, a *loosely-coordinated* non-clairvoyant coflow scheduler can strike a balance between performance and flexibility.

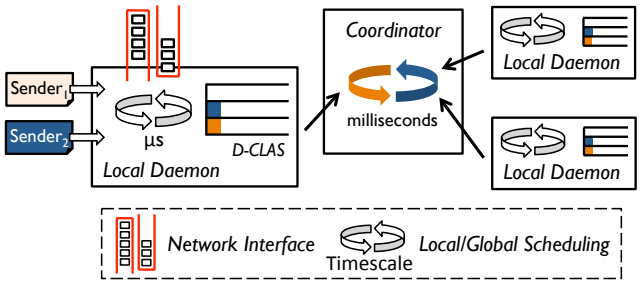


Figure 2: Aalo architecture. Computation frameworks interact with their local Aalo daemons using a client library, and the daemons periodically coordinate to determine the global ordering of coflows.

Consider the example in Figure 1 that compares three non-clairvoyant mechanisms against the optimal clairvoyant schedule. Per-flow fair sharing (Figure 1c) ensures max-min fairness in each link, but it suffers by ignoring coflows [19, 20]. Applying least-attained service (LAS) [42, 45, 14] in a decentralized manner (Figure 1d) does not help, because local observations cannot predict a coflow's actual size – e.g., it shares P_1 equally between C_1 and C_3, being oblivious to C_1's flow in P_2. The FIFO-LM schedule [25] would be at least as bad. Taking the total size of coflows into account through global coordination significantly decreases the average CCT (Figure 1e). The optimal solution (Figure 1f) exploits complete knowledge for the minimum average CCT. The FIFO schedule [19] would have resulted in a lower average CCT (4.67 time units) than decentralized LAS if C_3 was scheduled before C_1, and it would have been the same if C_1 was scheduled before C_3.

This example considers only single-stage coflows without egress contention. Coordinated coflow scheduling can be even more effective in both scenarios (§7).

3 Aalo Overview

Aalo uses a non-clairvoyant coflow scheduler that optimizes the communication performance of data-intensive applications without a priori knowledge, while being resilient to the dynamics of job schedulers and data-parallel clusters. This section briefly overviews Aalo to help the reader follow the analysis and design of its scheduling algorithms (§4), mechanisms to handle dynamic events (§5), and design details (§6) presented in subsequent sections.

3.1 Problem Statement

Our goal is *dynamically prioritizing coflows without prior knowledge of their characteristics while respecting coflow dependencies*. This problem – *non-clairvoyant coflow scheduling with precedence constraints* – is NP-hard, because coflow scheduling with complete knowledge is NP-hard too [20]. In addition to minimizing CCTs, we must guarantee starvation freedom and work conservation.

3.2 Architectural Overview

Aalo uses a loosely-coordinated architecture (Figure 2), because full decentralization can render coflow scheduling pointless (**Theorem A.1**). It implements global and local controls at two time granularities:

• *Long-term global coordination:* Aalo daemons send locally-observed coflow sizes to a central coordinator every $O(10)$ milliseconds. The coordinator determines the global coflow ordering using the D-CLAS framework (§4) and periodically sends out the updated schedule and globally-observed coflow sizes to all the daemons.

• *Short-term local prioritization:* Each daemon schedules coflows using the last-known global information. In between resynchronization, newly-arrived coflows are enqueued in the highest-priority queue. While flows from new and likely to be small[6] coflows receive high priority in the short term, Aalo daemons realign themselves with the global schedule as soon as updated information arrives. A flow that has just completed is replaced with a same-destination flow from the next coflow in the schedule for work conservation.

Frameworks use a client library to interact with the coordinator over the network to define coflows and their dependencies (§6). To send data, they must use the Aalo-provided `OutputStream`. The coordinator has an ID generator that creates unique CoflowIds while taking coflow dependencies into account (§5.1).

We have implemented Aalo in the application layer *without* any changes or support from the underlying network. We have deployed it in the cloud, and it performs well even for sub-second data analytics jobs (§7).

Fault Tolerance Aalo handles three failure scenarios that include its own failures and that of the clients using it. First, failure of a Aalo daemon does not hamper job execution, since the client library automatically falls back to regular TCP fair sharing until the daemon is restarted. Upon restart, the daemon remains in inconsistent state only until the next coordination step. Second, when the coordinator fails, client libraries keep track of locally-observed size until it has been restarted, while periodically trying to reconnect. Finally, in case of task failures and consequent restarts, relevant flows are restarted by corresponding job schedulers. Such flows are treated like a new wave in a coflow, and their additional traffic is added up to the current size of that coflow (§5.2).

Scalability The faster Aalo daemons can coordinate, the better it performs. The number of coordination messages is linear with the number of daemons and *independent* of coflows. It is not a bottleneck for clusters with $O(100)$ machines, and our evaluation suggests that Aalo can scale up to $O(10,000)$ machines with minimal performance loss (§7.6). Most coflows are small and scheduled through local decisions; hence, unlike Varys, Aalo handles tiny coflows well.

4 Scheduling Without Prior Knowledge

In this section, we present an efficient coflow scheduler for minimizing CCTs *without* a priori information. First, we discuss the complexity and requirements of such a scheduler (§4.1). Next, we describe a priority discretization framework (§4.2) that we use to discuss the tradeoffs in designing an efficient, non-clairvoyant scheduler (§4.3). Based on our understanding, we develop discretized Coflow-Aware Least-Attained Service (D-CLAS) – a mechanism to prioritize coflows and a set of policies to schedule them without starvation (§4.4). Finally, we compare our proposal with existing coflow schedulers (§4.5).

For brevity of exposition, we present the mechanisms in the context of single-stage, single-wave coflows. We extend them to handle multi-stage, multi-wave coflows as well as task failures and speculation in Section 5.

4.1 Complexity and Desirable Properties

The offline coflow scheduling problem – i.e., when all coflows arrive together and their characteristics are known a priori – is NP-hard [20]. Consequently, the non-clairvoyant coflow scheduling problem is NP-hard as well.

In the non-clairvoyant setting, smallest-bottleneck-first [20] – the best-performing clairvoyant heuristic – becomes inapplicable. This is because the bottleneck of a coflow is revealed only *after* it has completed. Instead, one must schedule coflows based on an attribute that

1. *can approximate its clairvoyant counterpart* using current observations, and
2. *involves all the flows* to avoid the drawbacks from the lack of coordination (**Theorem A.1**).

Note that a coflow's bottleneck can change over time and due to task failures and restarts, failing the first requirement.

In addition to minimizing the average CCT, the non-clairvoyant scheduler must

1. *guarantee starvation freedom* for bounded CCTs,
2. *ensure work conservation* to increase utilization, and
3. *decrease coordination requirements* for scalability.

Coflow-Aware Least-Attained Service (CLAS) Although the smallest-total-size-first heuristic had been shown to perform marginally worse ($1.14\times$) than smallest-bottleneck-first in the clairvoyant setting [20], it becomes a viable option in the non-clairvoyant case. The *current size* of a coflow – i.e., how much it has already sent throughout the entire cluster – meets both criteria. This is because unlike a coflow's bottleneck, it *monotonically* increases with each flow regardless of start time or endpoints. As a result, setting a coflow's priority that decreases with it's current size can ensure that smaller coflows finish faster, which, in turn, minimizes the average CCT. Furthermore, it is a good indicator of actual size [41], because coflow size typically follows heavy-tailed distribution [20, 11].

We refer to this scheme as Coordinated or Coflow-Aware Least-Attained Service (CLAS). Note that CLAS reduces to the well-known Least-Attained Service (LAS) [42, 45] scheduling discipline in case of a single link.

4.2 Priority Discretization

Unfortunately, using continuous priorities derived from coflow sizes can degenerate into fair sharing (§B), which increases the average CCT [19, 25, 20]. Coordination needed to find global coflow sizes poses an additional challenge. We must be able to preempt at opportune moments to decrease CCT without requiring excessive coordination.

[6]For data-parallel analytics, 60% (85%) coflows are less than 100 MB (1 GB) in total size [20, 25].

In the following, we describe a priority *discretization* framework to eventually design an efficient, non-clairvoyant coflow scheduler. Unlike classic non-clairvoyant schedulers – least-attained service (LAS) in single links [42, 45] and multi-level feedback queues (MLFQ) in operating systems [23, 21, 13] – that perform fair sharing in presence of similar flows/tasks to provide interactivity, our solution improves the average CCT even in presence of identical coflows.

Multi-Level Coflow Scheduling A multi-level coflow scheduler consists of K queues $(Q_1, Q_2, \ldots, Q_K)$, with queue priorities decreasing from Q_1 to Q_K. The i-th queue contains coflows of size within $[Q_i^{\text{lo}}, Q_i^{\text{hi}})$. Note that $Q_1^{\text{lo}} = 0$, $Q_K^{\text{hi}} = \infty$, and $Q_{i+1}^{\text{lo}} = Q_i^{\text{hi}}$.

Actions taken during three lifecycle events determine a coflow's priority.

- *Arrival:* New coflows enter the highest priority queue Q_1 when they start.
- *Activity:* A coflow is demoted to Q_{i+1} from Q_i, when its size crosses queue threshold Q_i^{hi}.
- *Completion:* Coflows are removed from their current queues upon completion.

The first two ensure coflow prioritization based on its current size, while the last is for completeness.

4.3 Tradeoffs in Designing Coflow Schedulers

Given the multi-level framework, a coflow scheduler can be characterized by its information source, queue structure, and scheduling disciplines at different granularities. Tradeoffs made while navigating this solution space result in diverse algorithms, ranging from centralized shortest-first to decentralized FIFO [19, 20, 25] and many in between. We elaborate on the key tradeoffs below.

Information Source There are two primary categories of coflow schedulers: *clairvoyant* schedulers use a priori information and *non-clairvoyant* ones do not. Non-clairvoyant schedulers have one more decision to make: whether to use globally-coordinated coflow sizes or to rely on local information. The former is accurate but more time consuming. The latter diverges (**Theorem A.1**) for coflows with large skews, which is common in production clusters [17, 20].

Queue Structure A scheduler must also determine the number of queues it wants to use and their thresholds. On the one hand, FIFO-derived schemes (e.g., Orchestra, Baraat) use exactly one queue.[7] FIFO works well when coflows follow light-tailed distributions [25]. Clairvoyant efficient schedulers (e.g., Varys), on the other hand, can be considered to have as many queues as there are coflows. They perform the best when coflow sizes are *known* and are heavy-tailed [20]. At both extremes, queue thresholds are irrelevant.

For solutions in between, determining an ideal number of queues and corresponding thresholds is difficult; even for tasks on a single machine, no optimal solution exists [13]. Increasing the number of levels/queues is appealing, but fine-grained prioritization can collapse to fair sharing when coflow sizes are unknown and hurt CCTs. More queues also generate more "queue-change" events and increase coordination requirements.

[7] Baraat takes advantage of multiple queues in switches to enable multiplexing, but logically all coflows are in the same queue.

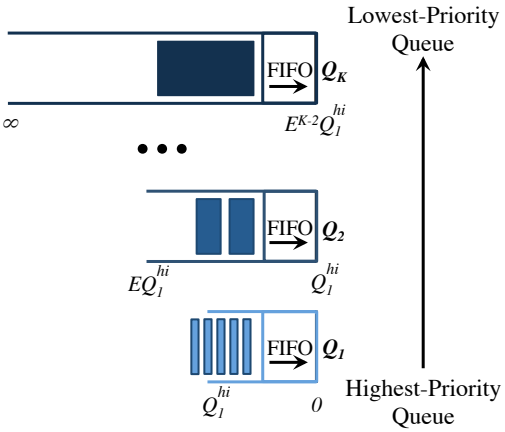


Figure 3: Discretized Coflow-Aware Least-Attained Service. Consecutive queues hold coflows with exponentially larger size.

Scheduling Disciplines Finally, a coflow scheduler must decide on scheduling disciplines at three different granularities: (i) across queues, (ii) among coflows in each queue, and (iii) among flows within each coflow. The first is relevant when $K > 1$, while the second is necessary when queues have more than one coflow. In the absence of flow size information, size-based rate allocation algorithms like WSS [19] and MADD [20] cannot be used; max-min fairness similar to TCP is the best alternative for scheduling individual flows.

4.4 Discretized Coflow-Aware Least-Attained Service

We propose *Discretized CLAS* or D-CLAS that use more than one priority queues, i.e., $K > 1$, to enable prioritization. The key challenge, however, is finding a suitable K that provides sufficient opportunities for preemption, yet small enough to not require excessive coordination.

To achieve our goals, each queue in D-CLAS contains *exponentially larger* coflows than its immediately higher-priority queue (Figure 3). Formally, $Q_{i+1}^{\text{hi}} = E \times Q_i^{\text{hi}}$, where the factor E determines how much bigger coflows in one queue are from that in another. Consequently, the number of queues remains small and can be expressed as an E-based logarithmic function of the maximum coflow size.

The final component in defining our queue structure is determining Q_1^{hi}. Because global coordination, irrespective of mechanism, has an associated time penalty depending on the scale of the cluster, we want coflows that are too small to be globally coordinated in Q_1. Larger coflows reside in increasingly more stable, lower-priority queues $(Q_2, \ldots, Q_K)$.

While we typically use $E = 10$ and $Q_1^{\text{hi}} = 10$ MB in our cluster, simulations show that for $K > 1$, a wide range of K, E, Q_1^{hi} combinations work well (§7.5).

Non-Clairvoyant Efficient Schedulers D-CLAS clusters similar coflows together and allows us to implement different scheduling disciplines among queues and among coflows within each queue (Pseudocode 1). It uses weighted sharing

	Orchestra [19]	Varys [20]	Baraat [25]	Aalo
On-Arrival Knowledge	Clairvoyant	Clairvoyant	Non-clairvoyant	Non-clairvoyant
Coflow Size Information	Global	Global	Local	Global Approx.
Number of Queues (K)	One	Num Coflows	One	$\log_E$(Max Size)
Queue Thresholds	N/A	Exact Size	N/A	$Q_{i+1}^{\text{hi}} = E \times Q_i^{\text{hi}}$
Queue Scheduling	N/A	Strict Priority	N/A	Weighted
Coflow Scheduling in Each Queue	FIFO	N/A	FIFO	FIFO
Flow Scheduling	WSS	MADD	Max-Min	Max-Min
Work Conservation	Next Coflow	Next Queue	Next Coflow	Weighted Among Queues
Starvation Avoidance	N/A	Promote to Q_1	N/A	N/A
HOL Blocking Avoidance	Multiplexing	N/A	Multiplexing	N/A

Table 1: Qualitative comparison of coflow scheduling algorithms. Typically, $E = 10$ for D-CLAS.

Pseudocode 1 D-CLAS Scheduler to Minimize CCT

```
1:  procedure RESCHEDULE(Queues Q, ExcessPolicy E(.))
2:    while Fabric is not saturated do                    ▷ Allocate
3:      for all i ∈ [1, K] do
4:        for all Coflow C ∈ Q_i do                       ▷ Sorted by CoflowId
5:          for all Flow f ∈ C do
6:            f.rate = Max-min fair share                 ▷ Fair schedule flows
7:            Update Q_i.share based on f.rate
8:        Distribute unused Q_i.share using E(.)          ▷ Work conserv.
9:  end procedure

10: procedure D-CLAS
11:   W = ∑ Q_i.weight
12:   for all i ∈ [1, K] do
13:     Q_i.share = Q_i.weight / W                        ▷ Weighted sharing b/n queues
14:   reschedule(Q, Max-Min among Q_{j≠i})
15: end procedure
```

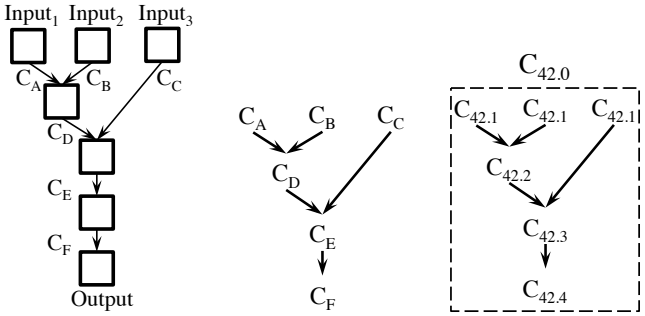
among queues, where queue weights decrease with lowered priority; i.e., Q_i.weight $\geq Q_{i+1}$.weight at line 13 in Pseudocode 1. Excess share of any queue is divided among unsaturated queues in proportion to their weights using max-min fairness (line 14).

Within each queue, it uses FIFO scheduling (line 4) so that coflows can proceed until they reach queue threshold or complete. Minimizing interleaving between coflows in the same queue minimizes CCTs, and large coflows are preempted after crossing queue thresholds. Hence, D-CLAS does not suffer from HOL blocking. As mentioned earlier, without prior knowledge, flows within each coflow use max-min fairness (line 6).

Starvation Avoidance Given non-zero weights to each queue, all queues are guaranteed to make progress. Hence, D-CLAS is starvation free. We did not observe any perpetual starvation in our experiments or simulations either.

4.5 Summary

Table 1 summarizes the key characteristics of the schedulers discussed in this section. D-CLAS minimizes the average CCT by prioritizing significantly different coflows across queues and FIFO ordering similar coflows in the same queue. It does so without starvation, and it approximates FIFO schedulers for light-tailed and priority schedulers for heavy-tailed coflow distributions.



(a) Query Plan **(b)** Dependencies **(c)** CoflowIDs

Figure 4: Coflow dependencies in TPC-DS query-42 [6]: (a) Query plan generated by Shark [48]; boxes and arrows respectively represent computation and communication stages. (b) Finishes-Before relationships between coflows are represented by arrows. (c) CoflowIds assigned by Aalo.

5 Handling Uncertainties

So far we have only considered "ideal" coflows from single-stage, single-wave jobs without task failures or stragglers. In this section, we remove each of these assumptions and extend the proposed schedulers to perform well in realistic settings. We start by considering multi-stage dataflow DAGs (§5.1). Next, we consider dynamic coflow modifications due to job scheduler events like multi-wave scheduling and cluster activities like restarted and speculative tasks (§5.2).

5.1 Multi-Stage Dataflow DAGs

The primary concern in coflow scheduling in the context of multi-stage jobs [30, 50, 3, 2] is the divergence of CCT and job completion time. Minimizing CCTs might not always result in faster jobs – one must carefully handle coflow *dependencies* within the same DAG (Figure 4).

We define a coflow C_F to be dependent on another coflow C_E if the consumer computation stage of C_E is the producer of C_F. Depending on pipelining between successive computation stages, there can be two types of dependencies.

1. *Starts-After ($C_E \longmapsto C_F$):* In presence of explicit barriers [2], C_F cannot start until C_E has finished.
2. *Finishes-Before ($C_E \longrightarrow C_F$):* With pipelining between successive stages [30, 22], C_F can coexist with C_E but it cannot finish until C_E has finished.

Note that coflows in different branches of a DAG can be un-

Pseudocode 2 Coflow ID Generation

```
1: NextCoflowID = 0                                          ▷ Initialization
2: procedure NEWCOFLOWID(CoflowId pId, Coflows 𝒫)
3:   if pId == Nil then
4:     newId = NextCoflowID++                                ▷ Unique external id
5:     return newId.0
6:   else
7:     sId = 1 + max_{C∈𝒫} C.sId                             ▷ Ordered internal id
8:     return pId.sId
9: end procedure
```

related to each other.

Job schedulers identify coflow dependencies while building query plans (Figure 4a). They can make Aalo aware of these dependencies all at once, or in a coflow-by-coflow basis. *Given coflow dependencies, we want to efficiently schedule them to minimize corresponding job completion times.*

We make two observations about coflow dependencies. First, coflows from the same job should be treated as a single entity. Second, within each entity, dependent coflows must be deprioritized during contention. The former ensures that minimizing CCTs directly affect job completion times, while the latter prevents circular dependencies. For example, all six coflows must complete in Figure 4a, and dependent coflows cannot complete without their parents in Figure 4b.

We simultaneously achieve both objectives by encoding the DAG identifier and internal coflow dependencies in the CoflowId. Specifically, we extend the CoflowId with an *internal* component in addition to its *external* component (Pseudocode 2). While the external part of a CoflowId uniquely identifies the DAG it belongs to, the internal part ensures ordering of coflows within the same DAG (Figure 4c). Our schedulers process coflows in each queue in the FIFO order based on their external components, and they break ties between coflows with the same external component using their internal CoflowIds (line 4 in Pseudocode 1).

Note that optimal DAG scheduling is NP-hard (§9). Our approach is similar to the Critical-Path Method [33] and resolves dependencies in each branch of a DAG, but it does not provide any guarantees for the entire DAG.

5.2 Dynamic Coflow Modifications

A flow can start only after its source and destination tasks have been scheduled. Tasks of large jobs are often scheduled in multiple waves depending on cluster capacity [11]. Hence, flows of such jobs are also created in batches, and waiting for all flows of a stage to start only halts a job. Because the number of tasks in each wave can dynamically change, Aalo must react without a priori knowledge. The same is true for unpredictable cluster events like failures and stragglers. Both result in restart or replication of some tasks and corresponding flows, and Aalo must efficiently handle them as well.

Aalo can handle all three events without any changes to its schedulers. As long as flows use the appropriate CoflowId, how much a coflow has sent *always* increases regardless of multiple waves and tasks being restarted or replicated.

6 Design Details

We have implemented Aalo in about $4,000$ lines of Scala code that provides a pipelined coflow API (§6.1) and implements (§6.2) the proposed schedulers.

6.1 Pipelined Coflow API

Aalo provides a simple coflow API that requires just replacing `OutputStream`s with `AaloOutputStream`. Any `InputStream` can be used in conjunction with `AaloOutputStream`. It also provides two additional methods for coflow creation and completion – `register()` and `unregister()`, respectively.

The `InputStream`-`AaloOutputStream` combination is non-blocking. Meaning, there is no artificial barrier after a coflow, and senders (receivers) start sending (receiving) without blocking. As they send (receive) more bytes, Aalo observes their total size, perform efficient coflow scheduling, and throttles when required. Consequently, small coflows proceed in the FIFO order without coordination overhead. The entire process is transparent to applications.

Usage Example *Any* sender can use coflows by wrapping its `OutputStream` with `AaloOutputStream`.

For example, for a shuffle to use Aalo, the driver first registers it to receive a unique CoflowId.

```
val sId = register()
```

Note that the driver does not need to define the number of flows before a coflow starts.

Later, each mapper must use `AaloOutputStream` for sending data. One mapper can create multiple `AaloOutputStream` instances, one for each reducer connection (i.e., socket `sock`), in concurrent threads.

```
val out = new AaloOutputStream(sock, sId)
```

Reducers can use any `InputStream` instances to receive their inputs. They can also overlap subsequent computation with data reception instead of waiting for the entire input. Once all reducers complete, the driver terminates the shuffle.

```
unregister(sId)
```

Defining Dependencies Coflows can specify their parent(s) during registration, and Aalo uses this information to generate CoflowIds (Pseudocode 2). In our running example, if the shuffle (`sId`) depended on an earlier broadcast (`bId`) – common in many Spark [50] jobs – the driver would have defined `bId` as a dependency during registration as follows.

```
val sId = register({bId})
```

`sId` and `bId` will share the same external CoflowId, but `sId` will have lower priority if it contends with `bId`.

6.2 Coflow Scheduling in Aalo

Aalo daemons resynchronize every Δ milliseconds. Each daemon sends the locally-observed coflow sizes to the coordinator every Δ interval. Similarly, the coordinator sends out the globally-coordinated coflow order and corresponding

sizes every Δ interval. Furthermore, the coordinator sends out explicit ON/OFF signals for individual flows in order to avoid receiver-side contentions and to expedite sender-receiver rate convergence.

In between updates, daemons make decisions based on current knowledge, which can be off by at most Δ milliseconds from the global information. Because traffic-generating coflows are large, daemons are almost always in sync about their order; only tiny coflows are handled by local decisions to avoid synchronization overheads.

Choice of Δ Aalo daemons are more closely in sync as Δ decreases. We suggest Δ to be $O(10)$ milliseconds, and our evaluation shows that a 100-machine EC2 cluster can resynchronize within 8 milliseconds on average (§7.6).

7 Evaluation

We evaluated Aalo through a series of experiments on 100-machine EC2 [1] clusters using traces from production clusters and an industrial benchmark. For larger-scale evaluations, we used a trace-driven simulator that performs a detailed replay of task logs. The highlights are:

- For communication-dominated jobs, Aalo improves the average (95th percentile) CCT and job completion time by up to 2.25× (2.93×) and 1.57× (1.77×), respectively, over per-flow fairness. Aalo improvements are, on average, within 12% of Varys (§7.2).
- As suggested by our analysis, coordination is the key to performance – independent local decisions (e.g., in [25]) can lead to more than 16× performance loss (§7.2.1).
- Aalo outperforms per-flow fairness and Varys for multi-wave (§7.3) and DAG (§7.4) workloads by up to 3.7×.
- Aalo's improvements are stable over a wide range of parameter combinations for any $K \geq 2$ (§7.5).
- Aalo coordinator can scale to $O(10,000)$ daemons with minimal performance loss (§7.6).

7.1 Methodology

Workload Our workload is based on a Hive/MapReduce trace collected by Chowdhury et al. [20, Figure 4] from a 3000-machine, 150-rack Facebook cluster. The original cluster had a 10 : 1 core-to-rack oversubscription ratio and a total bisection bandwidth of 300 Gbps. We scale down jobs accordingly to match the maximum possible 100 Gbps bisection bandwidth of our deployment while preserving their communication characteristics.

Additionally, we use TPC-DS [6] queries from the Cloudera benchmark [7, 4] to evaluate Aalo on DAG workloads. The query plans were generated using Shark [48].

Job/Coflow Bins We present our results by categorizing jobs based on their time spent in communication (Table 2) and by distinguishing coflows based on their lengths and widths (Table 3). Specifically, we consider a coflow to be *short* if its longest flow is less than 5 MB and *narrow* if it has at most 50 flows. Note that coflow sizes, like jobs, follow heavy-tailed distributions in data-intensive clusters [20].

Shuffle Dur.	$< 25\%$	25–49%	50–74%	$\geq 75\%$
% of Jobs	61%	13%	14%	12%

Table 2: Jobs binned by time spent in communication.

Coflow Bin	**1 (SN)**	**2 (LN)**	**3 (SW)**	**4 (LW)**
% of Coflows	52%	16%	15%	17%
% of Bytes	0.01%	0.67%	0.22%	99.10%

Table 3: Coflows binned by their length (**S**hort and **L**ong) and their width (**N**arrow and **W**ide).

Cluster Our experiments use extra-large high-memory (`m2.4xlarge`) EC2 instances. We observed bandwidths close to 900 Mbps per machine on clusters of 100 machines. We use a compute engine similar to Spark [50] that uses the coflow API (§6.1) and use $\Delta = 10$ milliseconds, $E = K = 10$, and $Q_1^{\text{hi}} = 10$ MB as defaults.

Simulator For larger-scale evaluation, we use a trace-driven flow-level simulator that performs a detailed task-level replay of the Facebook trace. It preserves input-to-output ratios of tasks, locality constraints, and inter-arrival times between jobs and runs at $10s$ decision intervals for faster completion.

Metrics Our primary metric for comparison is the improvement in average completion times of coflows and jobs (when its last task finished) in the workload. We measure it as the completion time of a scheme normalized by Aalo's completion time; i.e.,

$$\text{Normalized Comp. Time} = \frac{\text{Compared Duration}}{\text{Aalo's Duration}}$$

If the normalized completion time of a scheme is greater (smaller) than one, Aalo is faster (slower).

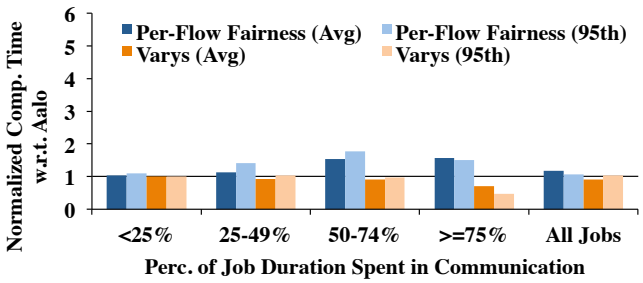
We contrast Aalo with TCP fair sharing and the open-source[8] implementation of Varys that uses a clairvoyant, smallest-bottleneck-first scheduler. Due to the lack of readily-deployable implementations of Baraat [25], we compare against it only in simulation. We present Aalo's results for D-CLAS with Q_i.weight = $K - i + 1$.

7.2 Aalo's Overall Improvements

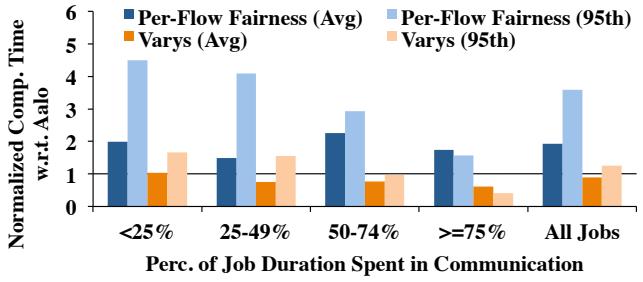
Figure 5a shows that Aalo reduced the average and 95th percentile completion times of communication-dominated jobs by up to 1.57× and 1.77×, respectively, in EC2 experiments in comparison to TCP-based per-flow fairness. Corresponding improvements in the average CCT (CommTime) were up to 2.25× and 2.93× (Figure 5b). As expected, jobs become increasingly faster as their time spent in communication increase. Across all bins, the average end-to-end completion times improved by 1.18× and the average CCT improved by 1.93×; corresponding 95th percentile improvements were 1.06× and 3.59×.

Varying improvements in the average CCT across bins in Figure 5b are not correlated, as it depends more on coflow characteristics than that of jobs. Figure 6 shows that Aalo improved the average CCT over per-flow fair sharing regardless of coflow width and length distributions. We observe

[8] https://github.com/coflow



(a) Improvements in job completion times



(b) Improvements in time spent in communication

Figure 5: [EC2] Average and 95th percentile improvements in job and communication completion times using Aalo over per-flow fairness and Varys.

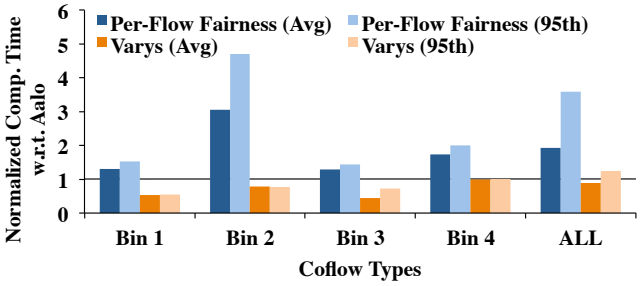


Figure 6: [EC2] Improvements in the average and 95th percentile CCTs using Aalo over per-flow fairness and Varys.

more improvements in bin-2 and bin-4 over bin-1 and bin-3, respectively, because longer coflows give Aalo more opportunities for better estimation.

Finally, Figure 7 presents comparative CDFs of CCTs for all coflows. Across a wide range of coflow durations – milliseconds to hours – Aalo matches or outperforms TCP fair sharing. As mentioned earlier, Aalo's advantages keep increasing with longer coflows.

What About Clairvoyant Coflow Schedulers? To understand how far we are from clairvoyant solutions, we have compared Aalo against Varys, which uses *complete* knowledge of a coflow's individual flows. Figure 5 shows that across all jobs, the average job and coflow completion times using Aalo stay within 12% of Varys. At worst, Aalo is 1.43× worse than Varys for 12% of the jobs.

Figure 6 presents a clearer picture of where Aalo is performing worse. For the largest coflows in bin-4 – sources of almost all the bytes – Aalo performs the same as Varys; it is only for the smaller coflows, specially the short ones in bin-1 and bin-3, Aalo suffers from its lack of foresight.

However, it still does not explain why Varys performs

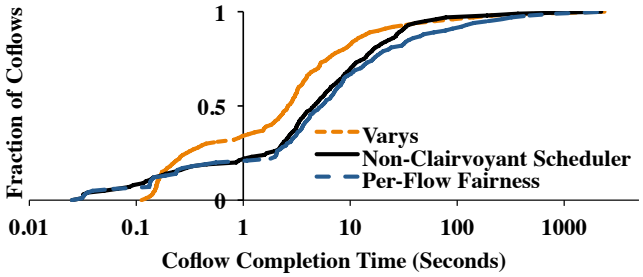


Figure 7: [EC2] CCT distributions for Aalo, Varys, and per-flow fairness mechanism. The X-axis is in log scale.

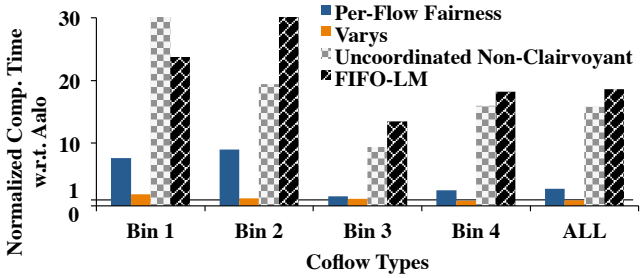


Figure 8: [Simulation] Average improvements in CCTs using Aalo. 95th percentile results are similar.

so much better than Aalo for coflows of durations between 200ms to 30s (Figure 7) given that Δ is only 10ms! Closer examination revealed this to be an isolation issue [43, 36]: Varys delays large coflows in presence of small ones and uses explicit rates for each flow. Because Aalo cannot explicitly control rates without a priori information, interference between coflows with few flows with very large coflows results in performance loss. Reliance on slow-to-react TCP for flow-level scheduling worsens the impact. We confirmed this by performing width-bounded experiments – we reduced the number of flows by 10× while keeping same coflow sizes; this reduced the gap between the two CDFs from ≤ 6× to ≤ 2× in the worst case.

Scheduling Overheads Because coflows smaller than the first priority threshold are scheduled without coordination, Aalo easily outperforms Varys for sub-200ms coflows (Figure 7). For larger coflows, Aalo's average and 99th percentile coordination overheads were 8ms and 19ms, respectively, in our 100-machine cluster – an order of magnitude smaller than Varys due to Aalo' loose coordination requirements. Almost all of it were spent in communicating coordinated decisions. Impact of scheduling overheads on Aalo's performance is minimal, even at much larger scales (§7.6).

7.2.1 Trace-Driven Simulation

We compared Aalo against per-flow fairness, Varys, and non-clairvoyant scheduling *without* coordination in simulations (Figure 8). Similar to EC2 experiments, Aalo outperformed flow-level fairness with average and 95th percentile improvements being 2.7× and 2.18×.

Figure 8 shows that Aalo outperforms Varys for smaller coflows in bin-1 to bin-3 in the absence of any coordination overheads. However, Varys performed 1.25× better than Aalo for coflows longer than 10s (not visible in Figure 9).

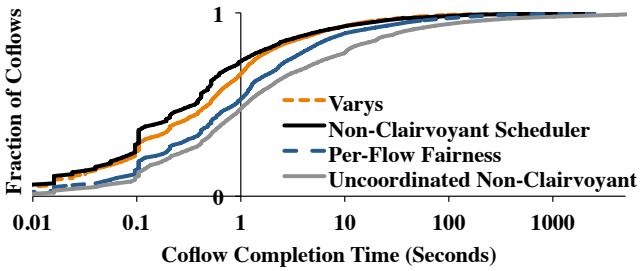


Figure 9: [Simulation] CCT distributions for Aalo, Varys, per-flow fairness, and uncoordinated non-clairvoyant coflow scheduling. X-axis is in log scale.

Number of Waves in Coflow	1	2	3	4
Max Waves = 1	100%			
Max Waves = 2	90%	10%		
Max Waves = 4	81%	9%	4%	6%

Table 4: Coflows binned by the number of waves.

What About Aalo *Without* Coordination? Given that Aalo takes few milliseconds to coordinate, we need to understand the importance of coordination. Simulations show that coflow scheduling without coordination can be significantly worse than even simple TCP fair sharing. On average, Aalo performed 15.8× better than its uncoordinated counterpart, bolstering our worst-case analysis (**Theorem A.1**). Experiments with increasing Δ suggest the same (§7.6).

What About FIFO with Limited Multiplexing in Baraat [25]? We found that FIFO-LM can be significantly worse than Aalo (18.6×) due to its lack of coordination: each switch takes locally-correct, but globally-inconsistent, scheduling decisions. Fair sharing among heavy coflows further worsens it. We had been careful – as the authors in [25] have pointed out – to select the threshold that each switch uses to consider a coflow heavy. Figure 8 shows the results for FIFO-LM's threshold set at the 80-th percentile of the coflow size distribution; results for the threshold set to the 20-th, 40-th, 60-th, 70-th, and 90-th percentiles were worse. Aalo and FIFO-LM performs similar for small coflows following light-tailed distributions (not shown).

How Far are We From the Optimal? Finding the optimal schedule, even in the clairvoyant case, is an open problem [20]. Instead, we tried to find an optimistic estimation of possible improvements by comparing against an *offline* 2-approximation heuristic for coflows *without* coupled resources [37]. For bin-1 to bin-4, corresponding normalized completion times were 0.75×, 0.78×, 1.32×, and 1.15×, respectively. Across all bins, it was 1.19×.

7.3 Impact of Runtime Dynamics

So far we have only considered static coflows, where all flows of a coflow start together. However, operational events like multi-wave scheduling, task failures, and speculative execution can dynamically change a coflow's structure in the runtime (§5.2). Because of their logical similarity – i.e., tasks start in batches and the number of active flows cannot be known a priori – we focus only on the multi-wave case.

The number of waves in a stage depends on the number of

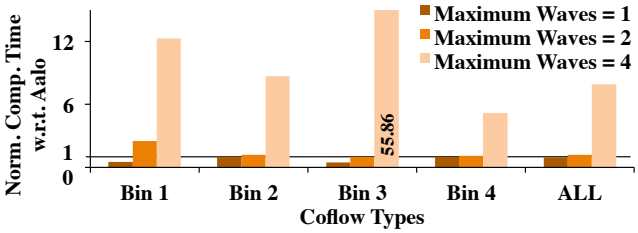


Figure 10: [EC2] Average improvements in CCTs w.r.t. Varys for multi-wave coflows.

senders (e.g., mappers in MapReduce) [11]. In these experiments, we used the same coflow mix as the original trace but varied the maximum number of concurrent senders in each wave while keeping all the receivers active, essentially fixing the maximum number of waves in each coflow. Table 4 shows the fraction of coflows with different number of waves; e.g., all coflows had exactly one wave in Section 7.2.

Figure 10 shows the importance of leveraging coflow relationships across waves. As the number of multi-wave coflows increased, Aalo moved from trailing Varys by 0.94× to outperforming it by 1.21× and 7.91×. Using Varys, one can take two approaches to handle multi-wave coflows – (i) creating separate coflows for each wave as they become available or (ii) introducing barriers to determine the bottleneck of the combined coflow – that both result in performance loss. In the former, Varys can efficiently schedule each wave but increases the stage-level CCT by ignoring the fact that all waves must finish for the stage to finish. The 56× improvement in bin-3 presents an extreme example: one *straggler* coflow was scheduled much later than the rest, increasing the entire stage's runtime. In the latter, artificial barriers decrease parallelism and network utilization. Aalo circumvents the dilemma by creating exactly one coflow per stage for any number of waves and by avoiding barriers.

Aalo's improvements over per-flow fairness (not shown) remained similar to that in Section 7.2.

7.4 Impact on DAG Scheduling

In this section, we evaluate Aalo using multi-stage jobs. Because the Facebook trace consists of only single-coflow jobs, we used the Cloudera industrial benchmark [7, 4] consisting of 20 TPC-DS queries. We ensured that each stage consists of a single wave, but multiple coflows from the same job can still run in parallel (Figure 4c).

Figure 11 shows that Aalo outperforms both per-flow fairness and Varys for DAGs that have more than one levels. Because Aalo does not introduce artificial barriers and can distinguish between coflows from different levels of the critical path, improvements over Varys (3.7× on average) are higher than that over per-flow fairness (1.7× on average).

7.5 Sensitivity Analysis

In this section, we first examine Aalo's sensitivity to the number of queues and their thresholds for heavy-tailed coflow size distributions. Later, we evaluate Aalo's performance for light-tailed distributions.

The Number of Queues (K) Aalo performs increasingly better than per-flow fairness as we increase the number of

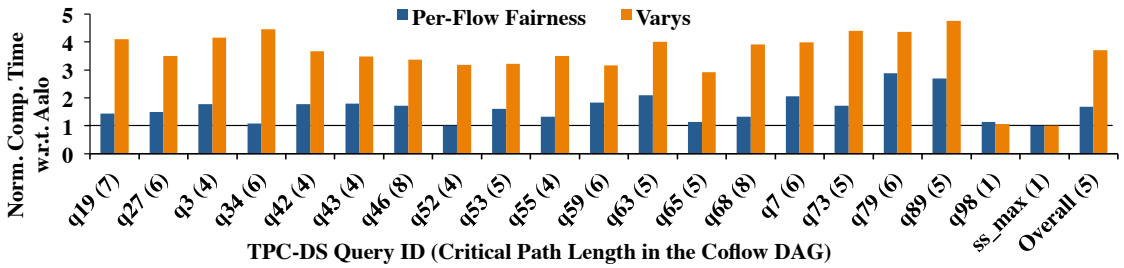
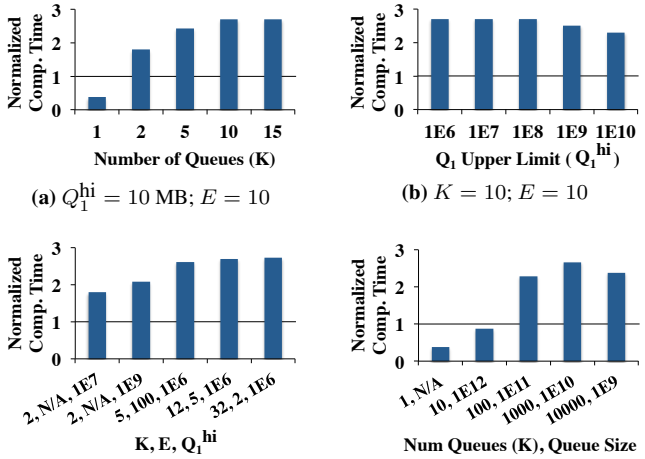


Figure 11: [EC2] Improvements in job-level communication times using Aalo for coflow DAGS in the Cloudera benchmark.



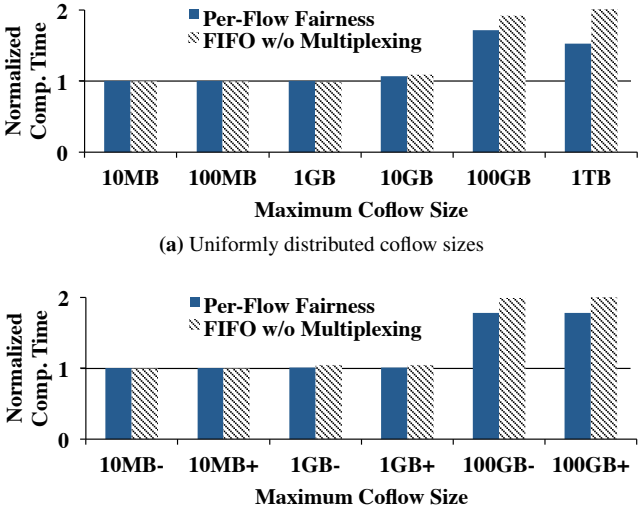
(a) $Q_1^{\text{hi}} = 10$ MB; $E = 10$

(b) $K = 10$; $E = 10$

(c) Exp.-Spaced Queues

(d) Equal-Sized Queues

Figure 12: [Simulation] Aalo's sensitivity (measured as improvements over per-flow fairness) to (a) the number of queues, (b) the size of the highest-priority queue, and (c) exponential and (d) linear queue thresholds.



(a) Uniformly distributed coflow sizes

(b) Fixed-size coflows

Figure 13: [Simulation] Improvements in average CCTs using Aalo (a) when coflow sizes are uniformly distributed up to different maximum values and (b) when all coflows have the same size.

queues (Figure 12a). However, we observe the largest jump as soon as Aalo starts avoiding head-of-line blocking for $K = 2$. Beyond that, we observe diminishing returns.

Queue Thresholds For more than one queues, Aalo must carefully determine their thresholds. Because we have defined queue thresholds as a function of the size of the initial queue Q_1^{hi} (§4.4), we focus on its impact on Aalo's performance. Recall that as we increase Q_1^{hi}, more coflows will be scheduled in the FIFO order in the highest-priority Q_1. Figure 12b shows that as we increase Q_1^{hi} up to 100 MB and schedule almost 60% of the coflows [20, Figure 4(e)] in the FIFO order, Aalo's performance remains steady. This is because all these coflows carry a tiny fraction of the total traffic ($\leq$ 0.1%). If we increase Q_1^{hi} further and start including increasingly larger coflows in the FIFO-scheduled Q_1, performance steadily deteriorates. Finally, Figure 12c demonstrates the interactions of E, the multiplicative factor used to determine queue thresholds, with K and Q_1^{hi}. We observe that for $K > 2$, Aalo's performance is steady for a wide range of (K, E, Q_1^{hi}) combinations.

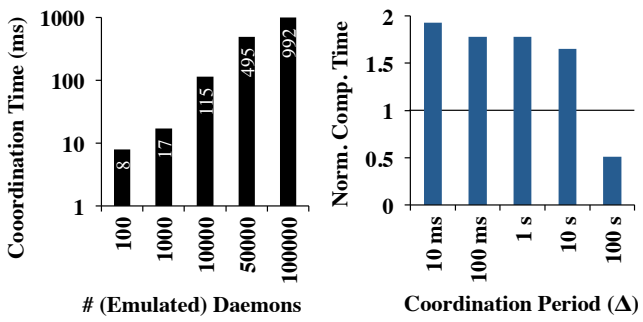
What About Non-Exponential Queue Thresholds? Instead of creating exponentially larger queues, one can create equal-sized queues. Given the maximum coflow size of 10 TB, Figure 12d shows Aalo's performance for varying number of equal-sized queues – it requires orders of magnitude more queues to attain performance similar to exponential spacing. Although creating logical queues is inexpensive at end hosts, more queues generate more "queue-change" events and increase coordination costs.

Impact of Coflow Size Distributions So far we have evaluated Aalo on coflows that follow heavy-tailed distribution. Here, we compare Aalo against per-flow fairness and a non-preemptive FIFO scheduler on coflows with uniformly-distributed and fixed sizes. We present the average results of ten simulated runs for each scenario with 100 coflows, where coflow structures follow the distribution in Table 3.

In Figure 13a, coflow sizes follow uniform distributions $\mathsf{U}(0, x)$, where we vary x. In Figure 13b, all coflows have the same size, and we select sizes slightly smaller and bigger than Aalo's queue thresholds. We observe that in both cases, Aalo matched or outperformed the competition. Aalo emulates the FIFO scheduler when coflow sizes are smaller than Q_1^{hi}(=10 MB). As coflows become larger, Aalo performs better by emulating the efficient Varys scheduler.

7.6 Aalo Scalability

To evaluate Aalo's scalability, we emulated running up to $100,000$ daemons on 100-machine EC2 clusters. Figure 14a



(a) Overheads at Scale **(b)** Impact of Δ

Figure 14: [EC2] Aalo scalability: (a) more daemons require longer coordination periods (Y-axis is in log scale), and (b) delayed coordination can hurt overall performance (measured as improvements over per-flow fairness).

presents the time to complete a coordination round averaged over 500 rounds for varying number of emulated daemons (e.g., 10,000 emulated daemons refer to each machine emulating 100 daemons). During each experiment, the coordinator transferred scheduling information for 100 concurrent coflows on average to each of the emulated daemons.

Even though we might be able to coordinate 100,000 daemons in 992ms, the coordination period (Δ) must be increased. To understand the impact of coordination on performance, we reran the earlier experiments (§7.2) for increasingly higher Δ (Figure 14b). For $\Delta = 1$s, Aalo's improvements over per-flow fairness dropped slightly from $1.93\times$ to $1.78\times$. For $\Delta > 1$s, performance started to drop faster and plummeted at $\Delta > 10$s. These trends hold across coflow bins and reinforce the need for coordination (**Theorem A.1**).

Because Δ must increase for Aalo to scale, sub-Δ coflows can further be improved if Aalo uses explicit switch/network support [27, 25]. However, we note that tiny coflows are still better off using Aalo than per-flow fairness schemes.

8 Discussion

Determining Optimal Queue Thresholds Finding the optimal number of queues and corresponding thresholds remains an open problem. Recent results in determining similar thresholds in the context of flows [14] do not immediately extend to coflows because of cross-flow dependencies. Dynamically changing these parameters based on online learning can be another direction of future work.

Decentralizing Aalo Decentralizing D-CLAS primarily depends on the following two factors.

1. *Decentralized calculation of coflow sizes, and*
2. *Avoiding receiver-side contentions without coordination.*

Approximate aggregation schemes like Push-Sum [34] can be good starting points to develop solutions for the former within reasonable time and accuracy. The latter is perhaps more difficult, because it relies on fast propagation of receiver feedbacks throughout the entire network for quick convergence of sender- and receiver-side rates. Both can improve from in-network support as used in CONGA [9].

Faster Interfaces and In-Network Bottlenecks As 10 GbE NICs become commonplace, a common concern is that scaling non-blocking fabrics might become cost prohibitive.[9] Aalo performs well even if the network is not non-blocking – for example, on the EC2 network used in the evaluation (§7). When bottleneck locations are known, e.g., rack-to-core links, Aalo can be modified to allocate rack-to-core bandwidth instead of NIC bandwidth [17]. For in-network bottlenecks, one can try enforcing coflows inside the network [52]. Nonetheless, designing, deploying, and enforcing distributed, coflow-aware routing and load balancing solutions remain largely unexplored.

9 Related Work

Coflow Schedulers Aalo's improvements over its clairvoyant predecessor Varys [20] are threefold. First, it schedules coflows *without* any prior knowledge, making coflows practical in presence of task failures and straggler mitigation techniques. Second, it supports pipelining and dependencies in multi-stage DAGs and multi-wave stages through a simpler, non-blocking API. Finally, unlike Varys, Aalo performs well even for tiny coflows by avoiding coordination. For larger coflows, however, Varys marginally outperforms Aalo by exploiting complete knowledge.

Aalo outperforms existing non-clairvoyant coflow schedulers, namely Orchestra [19] and Baraat [25], by avoiding head-of-line blocking unlike the former and by using global information unlike the latter. While Baraat's fully decentralized approach is effective for light-tailed coflow distributions, we prove in **Theorem A.1** that the lack coordination can be arbitrarily bad in the general case.

Qiu et al. have recently provided the first approximation algorithm for the clairvoyant coflow scheduling problem [44]. Similar results do not exist for the non-clairvoyant variation.

Flow Schedulers Coflows generalize traditional point-to-point flows by capturing the multipoint-to-multipoint aspect of data-parallel communication. While traffic managers like Hedera [8] and MicroTE [16] cannot directly be used to optimize coflows, they can be extended to perform coflow-aware throughput maximization and load balancing.

Transport-level mechanisms to minimize FCTs, both clairvoyant (e.g., PDQ [29], pFabric [10], and D^3 [47]) and non-clairvoyant (e.g., PIAS [14]), fall short in minimizing CCTs as well [20].

Non-Clairvoyant Schedulers Scheduling without prior knowledge is known as non-clairvoyant scheduling [39]. To address this problem in time-sharing systems, Corbató et al. proposed the multi-level feedback queue (MLFQ) algorithm [23], which was later analyzed by Coffman and Kleinrock [21]. Many variations of this approach exist in the literature [42, 45], e.g., foreground-background or least-attained service (LAS). In single machine (link), LAS performs almost as good as SRPT for heavy-tailed distributions of task (flow) sizes [45]. We prove that simply applying LAS through-

[9] A recent report from Google [5] suggests that it is indeed possible to build full-bisection bandwidth networks with up to 100,000 machines, each with 10 GbE NICs, for a total capacity of 1 Pbps.

out the fabric can be ineffective in the context of coflows (**Theorem A.1**). The closest instance of addressing a problem similar to ours is ATLAS [35], which controls concurrent accesses to multiple memory controllers in chip multiprocessor systems using coordinated LAS. However, ATLAS does not discretize LAS to ensure interactivity, and it does not consider coupled resources like the network.

DAG and Workflow Schedulers When the entire DAG and completion times of each stage are known, the Critical Path Method (CPM) [33, 32] is the best known algorithm to minimize end-to-end completion times. Without prior knowledge, several dynamic heuristics have been proposed with varying results [49]. Most data-parallel computation frameworks use breadth-first traversal of DAGs to determine priorities of each stage [50, 2, 3]. Aalo's heuristic enforces the *finishes-before* relationship between dependent coflows, but it cannot differentiate between independent coflows.

10 Conclusion

Aalo makes coflows more practical in data-parallel clusters in presence of multi-wave, multi-stage jobs and dynamic events like failures and speculations. It implements a non-clairvoyant, multi-level coflow scheduler (D-CLAS) that extends the classic LAS scheduling discipline to data-parallel clusters and addresses ensuing challenges through priority discretization. Aalo performs comparable to schedulers like Varys that use complete information. Using loose coordination, it can efficiently schedule tiny coflows and outperforms per-flow mechanisms *across the board* by up to $2.25\times$. Moreover, for DAGs and multi-wave coflows, Aalo outperforms both per-flow fairness mechanisms and Varys by up to $3.7\times$. Trace-driven simulations show Aalo to be $2.7\times$ faster than per-flow fairness and $16\times$ better than decentralized coflow schedulers.

Acknowledgments

We thank Yuan Zhong, Ali Ghodsi, Shivaram Venkataraman, CCN members, our shepherd Hitesh Ballani, and the anonymous reviewers of NSDI'15 and SIGCOMM'15 for useful feedback, and Kay Ousterhout for generating Shark query plans for the TPC-DS queries. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

11 References

[1] Amazon EC2. http://aws.amazon.com/ec2.
[2] Apache Hive. http://hive.apache.org.
[3] Apache Tez. http://tez.apache.org.
[4] Impala performance update: Now reaching DBMS-class speed. http://blog.cloudera.com/blog/2014/01/impala-performance-dbms-class-speed.
[5] A look inside Google's data center networks. http://googlecloudplatform.blogspot.com/2015/06/A-Look-Inside-Googles-Data-Center-Networks.html.
[6] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.
[7] TPC-DS kit for Impala. https://github.com/cloudera/impala-tpcds-kit.
[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.
[9] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.
[10] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. Mckeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*, 2013.
[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.
[12] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.
[13] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Scheduling: The multi-level feedback queue. In *Operating Systems: Three Easy Pieces*. 2014.
[14] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI*, 2015.
[15] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.
[16] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.
[17] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*, 2013.
[18] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *HotNets*, 2012.
[19] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.
[20] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.
[21] E. G. Coffman and L. Kleinrock. Feedback queueing models for time-shared systems. *Journal of the ACM*, 15(4):549–576, 1968.
[22] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. Mapreduce online. In *NSDI*, 2010.
[23] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Spring Joint Computer Conference*, pages 335–344, 1962.
[24] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[25] F. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *SIGCOMM*, 2014.
[26] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.
[27] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, 2013.
[28] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim,

P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[29] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.

[30] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[31] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" abstraction in Software-Defined Networks. In *CoNEXT*, 2013.

[32] J. E. Kelley. Critical-path planning and scheduling: Mathematical basis. *Operations Research*, 9(3):296–320, 1961.

[33] J. E. Kelley. The critical-path method: Resources planning and scheduling. *Industrial scheduling*, 13:347–365, 1963.

[34] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *FOCS*, 2003.

[35] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[36] G. Kumar, M. Chowdhury, S. Ratnasamy, and I. Stoica. A case for performance-centric network allocation. In *HotCloud*, 2012.

[37] M. Mastrolilli, M. Queyranne, A. S. Schulz, O. Svensson, and N. A. Uhan. Minimizing the sum of weighted completion times in a concurrent open shop. *Operations Research Letters*, 38(5):390–395, 2010.

[38] T. Moscibroda and O. Mutlu. Distributed order scheduling and its application to multi-core DRAM controllers. In *PODC*, 2008.

[39] R. Motwani, S. Phillips, and E. Torng. Nonclairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[40] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[41] J. Nair, A. Wierman, and B. Zwart. The fundamentals of heavy tails: Properties, emergence, and identification. In *SIGMETRICS*, 2013.

[42] M. Nuyens and A. Wierman. The Foreground–Background queue: A survey. *Performance Evaluation*, 65(3):286–307, 2008.

[43] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[44] Z. Qiu, C. Stein, and Y. Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *SPAA*, 2015.

[45] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack. Analysis of LAS scheduling for job size distributions with high variance. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):218–228, 2003.

[46] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *SOSP*, 2013.

[47] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[48] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, 2013.

[49] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. In *Metaheuristics for Scheduling in Distributed Computing Environments*, pages 173–214. 2008.

[50] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[51] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[52] Y. Zhao, K. Chen, W. Bai, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang. RAPIER: Integrating routing and scheduling for coflow-aware data center networks. In *INFOCOM*, 2015.

APPENDIX

A Coflow Scheduling w/ Local Knowledge

Theorem A.1 *Any coflow scheduling algorithm where schedulers do not coordinate, has a worst-case approximation ratio of $\Omega(\sqrt{n})$ for n concurrent coflows.*

Proof Sketch Consider n coflows $C_1, \ldots, C_n$ and a network fabric with $m \leq n$ input/output ports $P_1, P_2, \ldots, P_m$. Let us define $d^k_{i,j}$ as the amount of data the k-th coflow transfers from the i-th input port to the j-th output port.

For each input and output port, consider one coflow with just one flow that starts from that input port or is destined for that output port; i.e., for all coflows $C_k, k \in [1, m]$, let $d^k_{k,m-k+1} = 1$ and $d^k_{i,j} = 0$ for all $i \neq k$ and $j \neq m-k+1$. Next, consider the rest of the coflows to have exactly k flows that engage all input and output ports of the fabric; i.e., for all coflows $C_k, k \in [m+1, n]$, let $d^k_{i,m-i+1} = 1$ for all $i \in [1, m]$ and $d^k_{l,j} = 0$ for all $l \neq i$ and $j \neq m-i+1$. We have constructed an instance of *distributed order scheduling*, where n orders must be scheduled on m facilities [38]. The proof follows from [38, Theorem 5.1 on page 3]. ■

B Continuous vs. Discretized Prioritization

We consider the worst-case scenario when N identical coflows of size S arrive together, each taking $f(S)$ time to complete. Using continuous priorities, one would emulate a byte-by-byte round-robin scheduler, and the total CCT (T_{cont}) would approximate $N^2 f(S)$.

Using D-CLAS, all coflows will be in the k-th priority queue, i.e., $Q^{\text{lo}}_k \leq S < Q^{\text{hi}}_k$. Consequently, T_{disc} would be

$$N^2 f(Q^{\text{lo}}_k) + \frac{N(N+1) f(S - Q^{\text{lo}}_k)}{2}$$

where the former term refers to fair sharing until the k-th queue and the latter corresponds to FIFO in the k-th queue.

Even in the worst case, the normalized completion time ($T_{\text{cont}}/T_{\text{disc}}$) would approach $2\times$ from $1\times$ as S increases to Q^{hi}_k starting from Q^{lo}_k.

Note that the above holds only when a coflow's size accurately predicts it's completion time, which might not always be the case [20, §5.3.2]. Deriving a closed-form expression for the general case remains an open problem.