Efficient Coflow Scheduling with Varys

Mosharaf Chowdhury[1], Yuan Zhong[2], Ion Stoica[1]
[1]UC Berkeley, [2]Columbia University
{mosharaf, istoica}@cs.berkeley.edu, yz2561@columbia.edu

ABSTRACT

Communication in data-parallel applications often involves a collection of parallel flows. Traditional techniques to optimize flow-level metrics do not perform well in optimizing such collections, because the network is largely agnostic to application-level requirements. The recently proposed coflow abstraction bridges this gap and creates new opportunities for network scheduling. In this paper, we address inter-coflow scheduling for two different objectives: decreasing communication time of data-intensive jobs and guaranteeing predictable communication time. We introduce the *concurrent open shop scheduling with coupled resources* problem, analyze its complexity, and propose effective heuristics to optimize either objective. We present Varys, a system that enables data-intensive frameworks to use coflows and the proposed algorithms while maintaining high network utilization and guaranteeing starvation freedom. EC2 deployments and trace-driven simulations show that communication stages complete up to $3.16\times$ faster on average and up to $2\times$ more coflows meet their deadlines using Varys in comparison to per-flow mechanisms. Moreover, Varys outperforms non-preemptive coflow schedulers by more than $5\times$.

Categories and Subject Descriptors

C.2 [**Computer-communication networks**]: Distributed systems—*Cloud computing*

Keywords

Coflow; data-intensive applications; datacenter networks

1 Introduction

Although many data-intensive jobs are network-bound [7,9,15,23], network-level optimizations [7, 8, 12, 25] remain agnostic to job-specific communication requirements. This mismatch often hurts application-level performance, even when network-oriented metrics like flow completion time (FCT) or fairness improve [15].

Despite the differences among data-intensive frameworks [3, 4, 18,26,29,37,41], their communication is structured and takes place between groups of machines in successive computation stages [16]. Often a communication stage cannot finish until all its flows have completed [15, 19]. The recently proposed coflow abstraction [16] represents such collections of parallel flows to convey job-specific communication requirements – for example, minimizing completion time or meeting a deadline – to the network and enables application-aware network scheduling (§2). Indeed, optimizing a coflow's completion time (CCT) decreases the completion time of corresponding job [15].

However, jobs from one or more frameworks create multiple coflows in a shared cluster. Analysis of production traces shows wide variations in coflow characteristics in terms of total size, the number of parallel flows, and the size of individual flows (§4). Simple scheduling mechanisms like FIFO and its variants [15, 19], which are attractive for the ease of decentralization, do not perform well in such an environment – one large coflow can slow down many smaller ones or result in many missed deadlines.

Simply applying a shortest- or smallest-first heuristic, the predominant way to solve most scheduling problems, is not sufficient either (§5). Inter-coflow scheduling is different from scheduling individual flows [8, 25], because each coflow involves multiple parallel flows. It also differs from related problems like scheduling parallel tasks [9, 40] or caching parallel blocks [10]; unlike CPU or memory, the network involves *coupled resources* – each flow's progress depends on its rates at both source and destination. We show that these coupled constraints make *permutation schedules* – scheduling coflows one after another without interleaving their flows – suboptimal. Consequently, centralized scheduling becomes impractical, because the scheduler might need to preempt flows or recalculate their rates at arbitrary points in time even when no new flows start or complete.

In this paper, we study the inter-coflow scheduling problem for arbitrary coflows and focus on two objectives: improving application-level performance by minimizing CCTs and guaranteeing predictable completions within coflow deadlines. We prove this problem to be strongly NP-hard for either objective and focus on developing effective heuristics. We propose a coflow scheduling heuristic that – together with a complementary flow-level rate allocation algorithm – makes centralized coflow scheduling feasible by rescheduling only on coflow arrivals and completions.

In the presence of coupled constraints, the bottleneck endpoints of a coflow determine its completion time. We propose the *Smallest-Effective-Bottleneck-First* (SEBF) heuristic that greedily schedules a coflow based on its bottleneck's completion time. We then use the *Minimum-Allocation-for-Desired-Duration* (MADD) algorithm to allocate rates to its individual flows. The key idea behind MADD is to *slow down all the flows in a coflow to match the completion time of the flow that will take the longest to finish*. As a result, other coexisting coflows can make progress and the average

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626315.

CCT decreases. While the combination of SEBF and MADD is not necessarily optimal, we have found it to work well in practice.

For guaranteed coflow completions, we use admission control; i.e., we do not admit any coflow that cannot meet its deadline without violating someone else's. Once admitted, we use MADD to *complete all the flows of a coflow exactly at the coflow deadline* for guaranteed completion using the minimum amount of bandwidth.

We have implemented the proposed algorithms in a system called Varys[1] (§6), which provides a simple API that allows data-parallel frameworks to express their communication requirements as coflows with minimal changes to the framework. User-written jobs can take advantage of coflows *without* any modifications.

We deployed Varys on a 100-machine EC2 cluster and evaluated it (§7) by replaying production traces from Facebook. Varys improved CCTs both on average (up to 3.16×) and at high percentiles (3.84× at the 95th percentile) in comparison to per-flow fair sharing. Hence, end-to-end completion times of jobs, specially the communication-heavy ones, decreased. The aggregate network utilization remained the same, and there was no starvation. In trace-driven simulations, we found Varys to be 3.66× better than fair sharing, 5.53× better than per-flow prioritization, and 5.65× better than FIFO schedulers Moreover, in EC2 experiments (simulations), Varys allowed up to 2× (1.44×) more coflows to meet their deadlines in comparison to per-flow schemes; it marginally outperformed resource reservation mechanisms [11] as well.

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

2 Background and Motivation

This section overviews the coflow abstraction (§2.1) and our conceptual model of the datacenter fabric (§2.2), and illustrates the advantages of using coflows (§2.3).

2.1 The Coflow Abstraction

A coflow [16] is a collection of flows that share a common performance goal, e.g., minimizing the completion time of the latest flow or ensuring that flows meet a common deadline. We assume that the amount of data each flow needs to transfer is known before it starts [8, 15, 19, 25]. The flows of a coflow are independent in that the input of a flow does not depend on the output of another in the same coflow, and the endpoints of these flows can be in one or more machines. Examples of coflows include the shuffle between the mappers and the reducers in MapReduce [18] and the communication stage in the bulk-synchronous parallel (BSP) model [37].

Coflows can express most communication patterns between successive computation stages of data-parallel applications (Table 1) [16]. Note that traditional point-to-point communication is still a coflow with a single flow.

2.2 Network Model

In our analysis, we consider a network model where the entire datacenter fabric is abstracted out as one non-blocking switch [8, 11, 27, 33] interconnecting all the machines (Figure 1), and we focus only on its ingress and egress ports (e.g., machine NICs). This abstraction is attractive because of its simplicity, yet it is practical because of recent advances in full bisection bandwidth topologies [22, 32] and techniques for enforcing edge constraints into the network [11, 21]. Note that we use this abstraction to simplify our analysis, but we do not enforce it in our experiments (§7).

In this model, each ingress port has some flows from one or more coflows to various egress ports. For ease of exposition, we organize them in Virtual Output Queues [31] at the ingress ports as shown in Figure 1. In this case, there are two coflows C_1 and C_2; C_1 has three flows transferring 1, 2, and 4 units of data, while C_2 has two flows transferring 2 data units each.

[1] Pronounced \'vä-ris\.

Communication in Data-Parallel Apps	Coflow Structure
Comm. in dataflow pipelines [3, 4, 26, 41]	Many-to-Many
Global communication barriers [29, 37]	All-to-All
Broadcast [3, 26, 41]	One-to-Many
Aggregation [3, 4, 18, 26, 41]	Many-to-One
Parallel read/write on dist. storage [13, 14, 17]	Many One-to-One

Table 1: Coflows in data-parallel cluster applications.

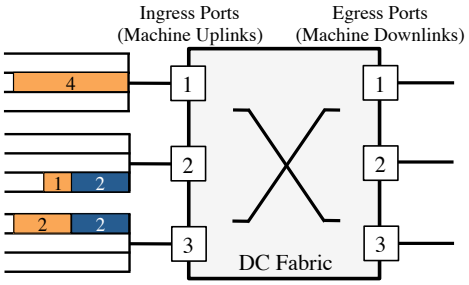


Figure 1: Coflow scheduling over a 3 × 3 datacenter fabric with three ingress/egress ports. Flows in ingress ports are organized by destinations and color-coded by coflows – C_1 in orange/light and C_2 in blue/dark.

2.3 Potential Benefits of Inter-Coflow Scheduling

While the network cares about flow-level metrics such as FCT and per-flow fairness, they can be suboptimal for minimizing the time applications spend in communication. Instead of improving network-level metrics that can be at odds with application-level goals, coflows improve performance through application-aware management of network resources.

Consider Figure 1. Assuming both coflows to arrive at the same time, Figure 2 compares four different schedules. Per-flow fairness (Figure 2a) ensures max-min fairness among flows in each link. However, fairness among flows of even the same coflow can increase CCT [15]. WSS (Figure 2c) – the optimal algorithm in homogeneous networks – is up to 1.5× faster than per-flow fairness for individual coflows [15]; but for multiple coflows, it minimizes the completion time across all coflows and increases the average CCT. Recently proposed shortest-flow-first prioritization mechanisms [8, 25] (Figure 2b) decrease average FCT, but they increase the average CCT by interleaving flows from different coflows. Finally, the optimal schedule (Figure 2d) minimizes the average CCT by finishing flows in the coflow order (C_2 followed by C_1). The FIFO schedule [15, 19] would have been as good as the optimal if C_2 arrived before C_1, but it could be as bad as per-flow fair sharing or WSS if C_2 arrived later.

Deadline-Sensitive Communication Assume that C_1 and C_2 have the same deadline of 2 time units – C_1 would never meet its deadline as its minimum CCT is 4. Using per-flow fairness or WSS, both C_1 and C_2 miss their deadlines. Using earliest-deadline-first (EDF) across flows [25], C_2 meets its deadline only 25% of the time. However, the optimal coflow schedule does not admit C_1, and C_2 always succeeds.

Note that egress ports do not experience any contention in these examples; when they do, coflow-aware scheduling can be even more effective.

3 Varys Overview

Varys is a coordinated coflow scheduler to optimize either the performance or the predictability of communication in data-intensive applications. In this section, we present a brief overview of Varys

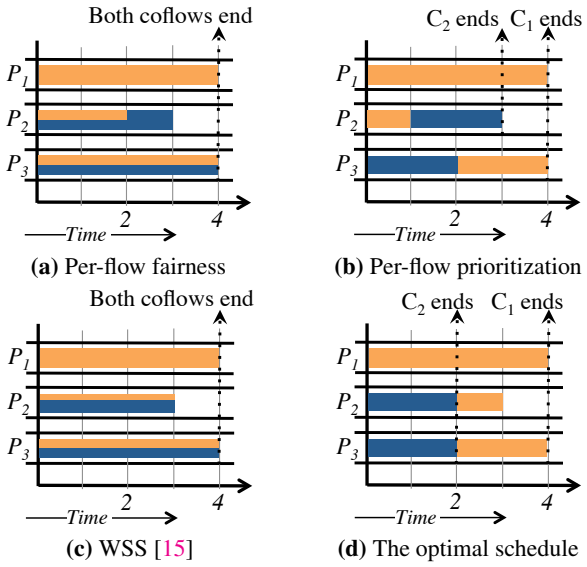


Figure 2: Allocation of *ingress* port capacities (vertical axis) using different mechanisms for the coflows in Figure 1. Each port can transfer one unit of data in one time unit. The average FCT and CCT for (a) per-flow fairness are 3.4 and 4 time units; (b) per-flow prioritization are 2.8 and 3.5 time units; (c) Weighted Shuffle Scheduling (WSS) are 3.6 and 4 time units; and (d) the optimal schedule are 3 and 3 time units.

to help the reader follow the measurements of coflows in production clusters (§4), analysis and design of inter-coflow scheduling algorithms (§5), and Varys's design details (§6).

3.1 Problem Statement

When improving performance, given a coflow with information about its individual flows, their size, and endpoints, Varys must decide *when to start* its flows and *at what rate* to serve them to minimize the average CCT of the cluster. It can preempt existing coflows to avoid head-of-line blocking, but it must also avoid starvation. Information about a coflow is unknown prior to its arrival.

When optimizing predictability, Varys must *admit a new coflow* if it can be completed within its deadline without violating deadline guarantees of the already-admitted ones.

Irrespective of the objective, the inter-coflow scheduling problem is NP-hard (§5). Varys implements a scheduler that exploits the variations in coflow characteristics (§4) to perform reasonably well in realistic settings.

3.2 Architectural Overview

Varys master schedules coflows from different frameworks using global coordination (Figure 3). It works in two modes: it either tries to minimize CCT or to meet deadlines. For the latter, it uses admission control, and rejected coflows must be resubmitted later. Frameworks use a client library to interact with Varys to register and define coflows (§6.1). The master aggregates all interactions to create a global view of the network and determines rates of flows in each coflow (§6.2) that are enforced by the client library.

Varys daemons, one on each machine, handle *time-decoupled* coflows, where senders and receivers are not simultaneously active. Instead of hogging the CPU, sender tasks (e.g., mappers) of data-intensive applications often complete after writing their output to the disk. Whenever corresponding receivers (e.g., reducers) are ready, Varys daemons serve them by coordinating with the master. Varys daemons use the same client library as other tasks. Additionally, these daemons send periodic measurements of the network usage at each machine to Varys master. The master aggregates them

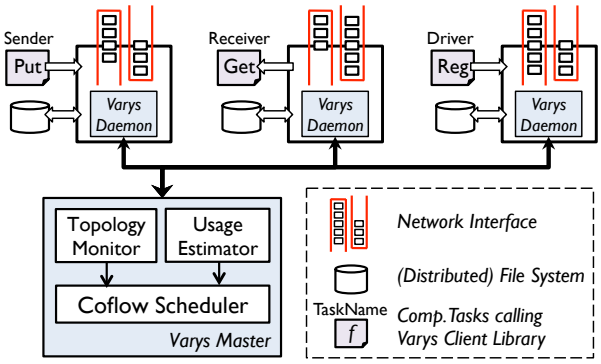


Figure 3: Varys architecture. Computation frameworks interact with Varys through a client library.

using existing techniques [17] to estimate current utilizations and use remaining bandwidth ($Rem(.)$) during scheduling (§5.3).

We have implemented Varys in the application layer out of practicality – it can readily be deployed in the cloud, while providing large improvements for both objectives we consider (§7).

Fault Tolerance Failures of Varys agents do not hamper job execution, since data can be transferred using regular TCP flows in their absence. Varys agents store soft states that can be rebuilt quickly upon restart. In case of task failures and consequent restarts, corresponding flows are restarted too; other flows of the same coflow, however, are not paused.

Scalability Varys reschedules only on coflow arrival and completion events. We did not observe the typical number of concurrent coflows (tens to hundreds [15, 33]) to limit its scalability. Varys batches control messages at $O(100)$ milliseconds intervals to reduce coordination overheads, which affect small coflows (§7.2). Fortunately, most traffic in data-intensive clusters are from large coflows (§4). Hence, we do not use Varys for coflows with bottlenecks smaller than 25 MB in size.

4 Coflows in Production

The impact of coflows on job durations and their network footprint have received extensive attention [15, 17, 33, 35, 39]. We focus on understanding their structural characteristics by analyzing traces from a 3000-machine, 150-rack Hive/MapReduce data warehouse at Facebook [17]. We highlight two attributes – wide variety in coflow structures and disproportionate footprint of few large coflows – that motivate and guide Varys's design.

4.1 Diversity of Coflow Structures

Because a coflow consists of multiple parallel flows, it cannot be characterized just by its size. We define the *length* of a coflow to be the size of its largest flow in bytes, its *width* to be the number of parallel flows, and its *size* to be the sum of all its flows in bytes. Furthermore, we consider *skew*, i.e., the coefficient of variation of its flows in terms of their size.

The key takeaway from the CDF plots in Figure 4 is that coflows vary widely in all four characteristics. We observe that while more than 40% coflows are short ($\leq$1 MB in length), flows in some coflows can be very large. Similarly, more than 60% narrow coflows (with at most 50 flows) reside with coflows that consist of millions of flows. Furthermore, we see variations of flow sizes within the same coflow (Figure 4c), which underpins the possible improvements from inter-coflow scheduling – e.g., the optimal schedule in Figure 2 outperforms the rest, primarily because it exploits the skew in C_1. Note that we rounded up flow sizes to 1 MB to calculate skew in Figure 4c to ignore small variations.

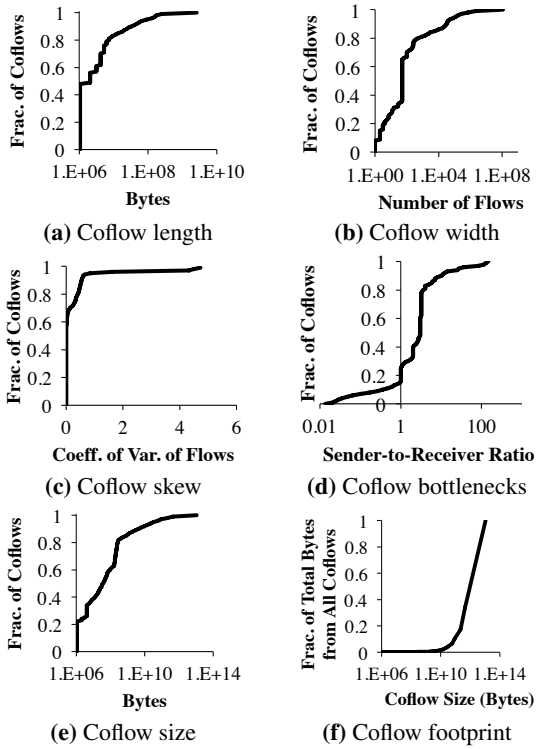


Figure 4: Coflows in production vary widely in (a) length, (b) width, (c) skew of flow sizes, (c) bottleneck locations, and (e) total size. Moreover, (f) numerous small coflows have tiny network footprint.

Identifying bottlenecks and exploiting them is the key to improvements. Figure 4d shows that the ratio of sender and receiver ports/machines[2] can be very different across coflows, and senders can be bottlenecks in almost a third of the coflows.

We found that length and width of a coflow have little correlation; a coflow can have many small flows as well as few large flows. However, as Figure 4b and Figure 4c suggest, width and skew are indeed correlated – as width increases, the probability of variations among flows increases as well. We also observed large variation in coflow size (Figure 4e).

4.2 Heavy-Tailed Distribution of Coflow Size

Data-intensive jobs in production clusters are known to follow heavy-tailed distributions in terms of their number of tasks, size of input, and output size [10, 17]. We observe the same for coflow size as well. Figure 4f presents the fraction of total coflows contributed by coflows of different size. Comparing it with Figure 4e, we see that almost all traffic are generated by a handful of large coflows – 98% (99.6%) of the relevant bytes belong to only 8% (15%) of the coflows that are more than 10 GB (1 GB) in size. This allows Varys to focus only on scheduling large coflows, where gains significantly outweigh coordination overheads.

5 Coflow Scheduling: Analytical Results

The inter-coflow scheduling problem is NP-hard. In this section, we provide insights into its complexity (§5.1) and discuss desirable properties of an ideal scheduler along with associated tradeoffs (§5.2). Based on our understanding, we develop two inter-coflow scheduling algorithms: one to minimize CCTs (§5.3) and another to guarantee coflow completions within their deadlines (§5.4).

Detailed analysis and proofs can be found in the appendix.

[2]One machine can have multiple tasks of the same coflow.

5.1 Problem Formulation and Complexity

We consider two objectives for optimizing data-intensive communication: either *minimizing* the average CCT or improving predictability by *maximizing* the number of coflows that meet deadlines (§A). Achieving either objective is NP-hard, even when

1. all coflows can start at the same time,
2. information about their flows are known beforehand, and
3. ingress and egress ports have the same capacity.

We prove it by reducing the concurrent open-shop scheduling problem [34] to inter-coflow scheduling (**Theorem A.1**).

The online inter-coflow scheduling problem is even harder because of the following reasons:

1. *Capacity constraints.* Ingress and egress ports of the datacenter fabric have finite, possibly heterogeneous, capacities. Hence, the optimal solution must find the best *ordering* of flows to dispatch at each ingress port and simultaneously calculate the best *matching* at the egress ports. Furthermore, when optimizing predictability, it must decide whether or not to *admit* a coflow.
2. *Lack of future knowledge.* Arrival times and characteristics of new coflows and their flows cannot be predicted.

Because the rate of any flow depends on its allocations at both ingress and egress ports, we refer to the inter-coflow scheduling problem as an instance of the concurrent open shop scheduling *with coupled resources* (**Remark A.2**). To the best of our knowledge, this variation of the problem – with ordering and matching requirements – has not appeared in the literature prior to this work.

5.2 Desirable Properties and Tradeoffs

Efficient scheduling (minimizing completion times) and predictable scheduling (guaranteeing coflow completions within their deadlines) are inherently conflicting. The former requires *preemptive* solutions to avoid *head-of-line* blocking. Shortest-remaining-time-first (SRTF) for optimally scheduling flows on a single link is an example [25]. Preemption, in the worst case, can lead to starvation; e.g., SRTF starves long flows. The latter, on the contrary, requires admission control to provide guarantees.

We expect an ideal scheduler to satisfy the following goals in addition to its primary objective.

1. *Starvation freedom.* Coflows, irrespective of their characteristics, should not starve for arbitrarily long periods.
2. *Work-conserving allocation.* Available resources should be used as much as possible.

The former ensures eventual completion of coflows irrespective of system load. The latter avoids underutilization of the network, which intuitively should result in lower CCTs and higher admissions. However, both are at odds with our primary objectives (§B).

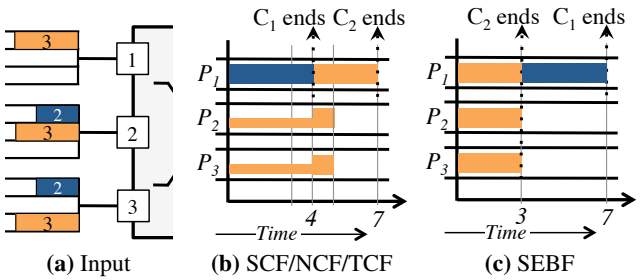
Predictable scheduling has an additional goal.

3. *Guaranteed completion.* If admitted, a coflow must complete within its deadline.

In the following, we present algorithms that achieve high network utilization, and ensure starvation freedom when minimizing CCT (§5.3) and guarantees completion of admitted coflows when maximizing predictability (§5.4).

5.3 Inter-Coflow Scheduling to Minimize CCT

Given the complexity, instead of finding an optimal algorithm, we focus on understanding what an offline optimal schedule might look like under simplifying assumptions. Next, we compute the minimum time to complete a single coflow. We use this result as a building block to devise a scheduling heuristic and an iterative



(a) Input **(b)** SCF/NCF/TCF **(c)** SEBF

Figure 5: Allocations of *egress* port capacities (vertical axis) for the coflows in (a) on a 3×3 fabric for different coflow scheduling heuristics (§5.3.2).

bandwidth allocation algorithm. We conclude by presenting the necessary steps to transform our offline solution to an online one with guarantees for starvation freedom and work conservation.

5.3.1 Solution Approach

Consider the offline problem of scheduling $|\mathbb{C}|$ coflows ($\mathbb{C} = \{C_1, C_2, \ldots, C_{|\mathbb{C}|}\}$) that arrived at time 0. The optimality of the shortest-processing-time-first (SPTF) heuristic on a single link suggests that shortest- or smallest-first schedules are the most effective in minimizing completion times [8, 25]. However, in the multi-link scenario, links can have different schedules. This makes the search space exponentially large – there are $((|\mathbb{C}|P)!)^P$ possible solutions when scheduling $|\mathbb{C}|$ coflows with P^2 flows each on a $P \times P$ fabric!

If we remove the capacity constraints from either ingress or egress ports, under the assumptions of Section 5.1, the coflow scheduling problem simplifies to the traditional concurrent open shop scheduling problem, which has optimal *permutation schedules* [30]; meaning, scheduling coflows one after another is sufficient, and searching within the $|\mathbb{C}|!$ possible schedules is enough. Unfortunately, permutation schedules can be suboptimal for coupled resources (**Theorem C.1**), which can lead to flow preemptions at arbitrary points in time – not just at coflow arrivals and completions (**Remark C.2**). To avoid incessant rescheduling, we restrict ourselves to permutation schedules.

5.3.2 The Smallest-Effective-Bottleneck-First Heuristic

Once scheduled, a coflow can impact the completion times of all other coflows scheduled after it. Our primary goal is to minimize the opportunity lost in scheduling each coflow.

Given the optimality of the shortest- or smallest-first policy in minimizing the average FCT [8, 25], a natural choice for scheduling coflows would be to approximate that with a *Shortest-Coflow-First (SCF)* heuristic. However, SCF does not take into account the width of a coflow. A width-based alternative to SCF is the *Narrowest-Coflow-First (NCF)* heuristic, but NCF cannot differentiate between a short coflow from a long one. A *smallesT-Coflow-First (TCF)* heuristic is a better alternative than the two – while SCF can be influenced just by a single long flow (i.e., coflow length) and NCF relies only on coflow width, TCF responds to both.

However, the completion time of coflow actually depends on its bottleneck. A coflow C must transfer $\sum_j d_{ij}$ amount of data through each ingress port i (P_i^{in}) and $\sum_i d_{ij}$ through each egress port j (P_j^{out}), where d_{ij} is the amount of data to transfer from P_i^{in} to P_j^{out} at rate r_{ij}. The minimum CCT (Γ) becomes

$$\Gamma = \max\left(\max_i \frac{\sum_j d_{ij}}{Rem(P_i^{\text{in}})}, \max_j \frac{\sum_i d_{ij}}{Rem(P_j^{\text{out}})}\right) \quad (1)$$

where $Rem(.)$ denotes the remaining bandwidth of an ingress or egress port estimated by Varys measurements. The former argument of Equation (1) represents the minimum time to transfer $\sum_{ij} d_{ij}$ ($= Size(C)$) amount of data through the input ports, and the latter is for the output ports.

We propose the *Smallest-Effective-Bottleneck-First (SEBF)* heuristic that considers a coflow's length, width, size, and skew to schedule it in the smallest-Γ-first order. Figure 5 shows an example: although C_2 (orange/light) is bigger than C_1 in length, width, and size, SEBF schedules it first to reduce the average CCT to 5 time units from 5.5. While no heuristic is perfect, we found SEBF to be $1.14\times$, $1.36\times$, and $1.66\times$ faster than TCF, SCF, and NCF, respectively, in trace-driven simulations and even more in synthetic ones. Additionally, SEBF performs $\geq 5\times$ better than non-preemptive coflow schedulers (§7.4).

Pseudocode 1 Coflow Scheduling to Minimize CCT

```
1:  procedure allocBandwidth(Coflows ℂ, Rem(.), Bool cct)
2:      for all C ∈ ℂ do
3:          τ = Γ^C (Calculated using Equation (1))
4:          if not cct then
5:              τ = D^C
6:          end if
7:          for all d_ij ∈ C do                          ▷ MADD
8:              r_ij = d_ij / τ
9:              Update Rem(P_i^in) and Rem(P_j^out)
10:         end for
11:     end for
12: end procedure

13: procedure minCCTOffline(Coflows ℂ, C, Rem(.))
14:     ℂ′ = SORT_ASC (ℂ ∪ C) using SEBF
15:     allocBandwidth(ℂ′, Rem(.), true)
16:     Distribute unused bandwidth to C ∈ ℂ′   ▷ Work conserv. (§5.3.4)
17:     return ℂ′
18: end procedure

19: procedure minCCTOnline(Coflows ℂ, C, Rem(.))
20:     if timeSinceLastDelta() < T then    ▷ T-interval: Decrease CCT
21:         ℂ′ = minCCTOffline(ℂ, C, Rem(.))
22:         Update ℂzero, the set of starved coflows
23:     else                                ▷ δ-interval: Starvation freedom
24:         C* = ⋃ C for all C ∈ ℂzero
25:         Apply MADD on C*
26:         Schedule a call to minCCTOnline(.) after δ interval
27:     end if
28: end procedure
```

5.3.3 Minimum-Allocation-for-Desired-Duration

Given a schedule of coflows $\mathbb{C}' = (C_1, C_2, \ldots, C_{|\mathbb{C}|})$, the next step is to determine the rates of individual flows. While single-link optimal heuristics would allocate the entire bandwidth of the link to the scheduled flow, we observe that completing a flow faster than the bottleneck does not impact the CCT in coflow scheduling.

Given Γ, *the minimum completion time of a coflow can be attained as long as all flows finish at time* Γ. We can ensure that by setting the rates (r_{ij}) of each flow to d_{ij}/Γ. We refer to this algorithm (lines 7–10 in Pseudocode 1) as MADD; this generalizes WSS [15], which works only for homogeneous links. *MADD allocates the least amount of bandwidth to complete a coflow in minimum possible time.*

We use MADD as a building block to allocate rates for the given schedule. We apply MADD to each coflow $C_i \in \mathbb{C}'$ to ensure its fastest completion using minimum bandwidth, and we iteratively distribute (line 15) its unused bandwidth to coflows C_j ($i < j \leq |\mathbb{C}|$). Once C_i completes, the iterative procedure is repeated to complete C_{i+1} and to distribute its unused bandwidth. We stop after $C_{|\mathbb{C}|}$ completes.

Pseudocode 2 Coflow Scheduling to Guarantee Completion

```
1:  procedure MEETDEADLINE(Coflows ℂ, C, Rem(.))
2:      allocBandwidth(ℂ, Rem(.), false)  ▷ Recalc. min rates for C ∈ ℂ
3:      if Γ^C ≤ D^C then                 ▷ Admission control
4:          ℂ′ = Enqueue C to ℂ           ▷ Add C in the arrival order
5:          allocBandwidth(ℂ′, Rem(.), false)
6:          Distribute unused bandwidth to C ∈ ℂ′   ▷ Work conservation
7:          return true
8:      end if
9:      Distribute unused bandwidth to C ∈ ℂ
10:     return false
11: end procedure
```

5.3.4 From Offline to Online

Work-Conserving Allocation Letting resources idle – as the offline iterative MADD might do – can hurt performance in the online case. We introduce the following backfilling pass in MINCCTOFFLINE (line 16 of Pseudocode 1) to utilize the unallocated bandwidth throughout the fabric as much as possible. For each ingress port P_i^{in}, we allocate its remaining bandwidth to the coflows in $\mathbb{C}'$; for each active coflow C in P_i^{in}, $Rem(P_i^{\text{in}})$ is allocated to C's flows in their current r_{ij} ratios, subject to capacity constraints in corresponding P_j^{out}.

Avoiding Perpetual Starvation Preemption to maintain an SEBF schedule while optimizing CCT may lead to starvation. To avoid perpetual starvation, we introduce the following simple adjustment to the offline algorithm.

We fix tunable parameters T and δ ($T \gg \delta$) and alternate the overall algorithm (MINCCTONLINE) between time intervals of length T and δ. For a time period of length T, we use MINCCTOFFLINE to minimize CCT. At the end of time T, we consider all coflows in the system which have not received any service during the last T-interval ($\mathbb{C}_{\text{zero}}$). We treat all of them as one collective coflow, and apply MADD for a time period of length δ (lines 24–26 in Pseudocode 1). All coflows that were served during the last T-interval do not receive any service during this δ-interval. At the end of the δ-interval, we revert back, and repeat.

The adjustments ensure that *all* coflows receive non-zero service in every $(T + \delta)$ interval and eventually complete. This is similar to ensuring at least one ticket for each process in lottery scheduling [38]. Avoiding starvation comes at the cost of an increased average CCT. At every $(T + \delta)$ interval, the total CCT increases by at most $|\mathbb{C}|\delta$, and it depends on the ratio of T and δ (§6.2).

5.4 Inter-Coflow Scheduling to Guarantee Deadline

To guarantee a coflow's completion within deadline (D^C), completing its bottlenecks as fast as possible has no benefits. *A coflow can meet its deadline using minimum bandwidth as long as all flows finish exactly at the deadline*. We can achieve that by setting $r_{ij} = d_{ij}/D^C$ using MADD.

To provide guarantees in the online scenario, we introduce admission control (line 3 in Pseudocode 2). We admit a coflow C, if and only if it can meet its deadline without violating that of any existing coflow. Specifically, we recalculate the minimum bandwidth required to complete all existing coflows within their deadlines (line 2 in Pseudocode 2) and check if the minimum CCT of C, $\Gamma^C \leq D^C$. Otherwise, C is rejected. An admitted coflow is never preempted, and a coflow is never rejected if it can safely be admitted. Hence, there is no risk of starvation. We use the backfilling procedure from before for work conservation.

`VarysClient` Methods	Caller
`register`(*numFlows*, [*options*]) $\Longrightarrow$ *coflowId*	Driver
`put`(*coflowId*, *dataId*, *content*, [*options*])	Sender
`get`(*coflowId*, *dataId*) $\Longrightarrow$ *content*	Receiver
`unregister`(*coflowId*)	Driver

Table 2: The Coflow API

6 Design Details

We have implemented Varys in about 5,000 lines of Scala with extensive use of `Akka` [1] for messaging and the `Kryo` serialization library [5]. This section illustrates how frameworks interact with Varys (§6.1) and discusses how Varys schedules coflows (§6.2).

6.1 Varys Client Library: The Coflow API

Varys client library provides an API similar to DOT [36] to abstract away the underlying scheduling and communication mechanisms. Cluster frameworks (e.g., Spark, Hadoop, or Dryad) must create `VarysClient` objects to invoke the API and interact with Varys. User jobs, however, do *not* require any modifications.

The coflow API has four primary methods (Table 2). Framework drivers initialize a coflow through `register()`, which returns a unique *coflowId* from Varys master. *numFlows* is a hint for the scheduler on when to consider the coflow READY to be scheduled. Additional information or hints (e.g., coflow deadline or dependencies) can be given through *options* – an optional list of key-value pairs. The matching `unregister()` signals coflow completion.

A sender initiates the transfer of a *content* with an identifier *dataId* using `put()`. A *content* can be a file on disk or an object in memory. For example, a mapper would `put()` r pieces of *content* for r reducers in a MapReduce job, and the Spark driver would `put()` a common piece of data to be broadcasted from memory to its workers (we omit corresponding `put()` signatures for brevity). The *dataId* of a *content* is unique within a coflow. Any flow created to transfer *dataId* belongs to the coflow with the specified *coflowId*.

A receiver indicates its interest in a *content* using its *dataId* through `get()`. Only after receiving a `get()` request, the scheduler can create and consider a flow for scheduling. Receiver tasks learn the *dataId*s of interest from respective framework drivers. Varys scheduler determines when, from where, and at what rate to retrieve each requested *dataId*. `VarysClient` enforces scheduler-determined rates at the application layer and notifies the master upon completion of `get()`.

Usage Example Consider shuffle – the predominant communication pattern in cluster frameworks. Shuffle transfers the output of each task (mapper) in one computation stage to the tasks (reducers) in the next. The following example shows how to enable a 3×2 shuffle (with 3 mappers and 2 reducers) to take advantage of inter-coflow scheduling. Assume that all entities interacting with Varys have their own instances of `VarysClient` objects named `client`.

First, the driver registers the shuffle indicating that Varys should consider it READY after receiving `get()` for all six flows.

```
val cId = client.register(6)
```

When scheduling each task, the driver passes along the `cId` for the shuffle. Each mapper m uses `cId` when calling `put()` for each reducer r.

```
// Read from DFS, run user-written map method,
// and write intermediate data to disk.
// Now, invoke the coflow API.
for (r <- reducers)
  client.put(cId, dId-m-r, content-m-r)
```

Pseudocode 3 Message Handlers in Varys Scheduler

```
1: procedure ONCOFLOWREGISTER(Coflow C)
2:     Mark C as UNREADY
3:     ℂcur = ℂcur ∪ {C}                    ▷ ℂcur is the set of all coflows
4: end procedure

5: procedure ONCOFLOWUNREGISTER(Coflow C)
6:     ℂcur = ℂcur \ {C}
7:     ℂready = ℂ.filter(READY)
8:     Call appropriate scheduler from Section 5
9: end procedure

10: procedure ONFLOWPUT(Flow f, Coflow C)
11:     Update Size(C) and relevant data structures
12: end procedure

13: procedure ONFLOWGET(Flow f, Coflow C)
14:     Update relevant data structures
15:     Mark C as READY after get() is called numFlows times
16:     if C is READY then
17:         ℂready = ℂ.filter(READY)
18:         Call appropriate scheduler from Section 5
19:     end if
20: end procedure
```

In the snippet above, `dId-m-r` is an application-defined unique identifier for individual pieces of data and `content-m-r` is the corresponding path to disk. This is an example of time-decoupled communication.

Reducers use `cId` to retrieve the shuffled pieces of content by the mappers (served by Varys daemons).

```
// Shuffle using the coflow API.
for (m <- mappers)
  content-m-r = client.get(cId, dId-m-r)
// Now, sort, combine, and write to DFS.
```

Once all reducers are done, the driver terminates the coflow.

```
client.unregister(cId)
```

Note that the example abstracts away some details – e.g., the pipelining between shuffle and sort phases in reducers, which can be handled by providing a `VarysInputStream` implementation.

Replacing the communication layer of a data-intensive framework with just the aforementioned changes can enable *all* its user jobs to take advantage of inter-coflow scheduling.

6.2 Inter-Coflow Scheduling in Varys

Varys implements the algorithms in Section 5 for inter-coflow scheduling, which are called upon coflow arrival and completion events. $Rem(.)$ is calculated from the aggregated measurements collected by Varys daemons. Pseudocode 3 lists the key event handlers in Varys. Initially, all coflows are marked UNREADY (ONCOFLOWREGISTER). The size of a coflow is updated as `VarysClient` instances periodically notify flow-related events using ONFLOWPUT and ONFLOWGET. A coflow is considered READY to be scheduled after *numFlows* `get()` calls (ONFLOWGET). Varys master groups the new allocations calculated by the scheduler by respective `VarysClient`s and sends the changes asynchronously.

Choice of T and δ Values A smaller δ in Pseudocode 1 ensures a lower impact on the average CCT. However, too small a δ can cause the underlying transport protocol (e.g., TCP) to behave erratically due to significant variation of available bandwidth over short time intervals. We suggest δ to be $O(100)$ milliseconds and T to be $O(1)$ seconds.

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

Table 3: Jobs binned by time spent in communication.

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

Table 4: Coflows binned by width and length.

7 Evaluation

We evaluated Varys through a set of experiments on 100-machine EC2 [2] clusters using a Hive/MapReduce trace collected from a large production cluster at Facebook. For a larger scale evaluation, we used a trace-driven simulator that performs a detailed replay of task logs from the same trace. The highlights are:

- For communication-dominated jobs, Varys improves the average (95th percentile) CCT and job completion time by up to $3.16\times$ ($3.84\times$) and $2.5\times$ ($2.94\times$), respectively, over per-flow fairness. Across all jobs, the improvements are $1.85\times$ ($1.74\times$) and $1.25\times$ ($1.15\times$) (§7.2).
- Simulations show that Varys improves the average (95th percentile) CCT by $5.53\times$ ($5.8\times$) over per-flow prioritization mechanisms (§7.2).
- Varys enables almost $2\times$ more coflows to meet their deadlines in EC2 experiments (§7.3).
- All coflows eventually complete using Varys without starvation, and simulations show Varys to be $5.65\times$ faster than a non-preemptive solution ($7.7\times$ at the 95th percentile) (§7.4).

7.1 Methodology

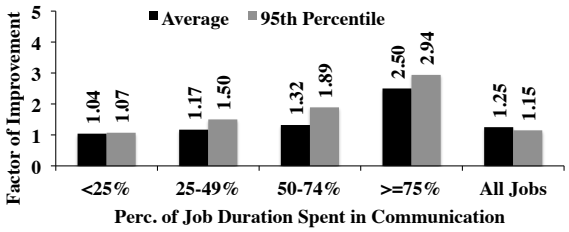
Workload Our workload is based on a Hive/MapReduce trace at Facebook that was collected on a 3000-machine cluster with 150 racks. The original cluster had a $10:1$ core-to-rack oversubscription ratio with a total bisection bandwidth of 300 Gbps. We scale down jobs accordingly to match the maximum possible 100 Gbps bisection bandwidth of our deployment. During the derivation, we preserve the original workload's communication characteristics.

We consider jobs with non-zero shuffle and divide them into bins (Table 3) based on the fraction of their durations spent in shuffle. Table 4 divides the coflows in these jobs into four categories based on their characteristics. Based on the distributions of coflow lengths and widths (Figure 4), we consider a coflow to be *short* if its longest flow is less than 5 MB and *narrow* if it involves at most 50 flows.

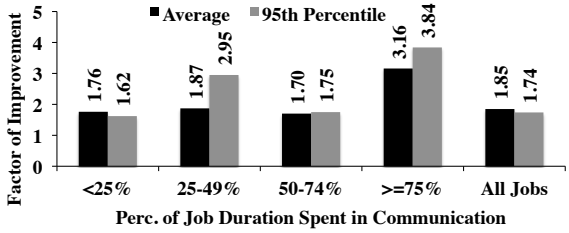
For deadline-constrained experiments, we set the deadline of a coflow to be its minimum completion time in an empty network (Γ_{empty}) multiplied by $\big(1 + \mathsf{U}(0, x)\big)$, where $\mathsf{U}(0, x)$ is a uniformly random number between 0 and x. Unless otherwise specified, $x = 1$. The minimum deadline is 200 milliseconds.

Cluster Our experiments use extra large high-memory EC2 instances, which appear to occupy entire physical machines and have enough memory to perform all experiments without introducing disk overheads. We observed bandwidths close to 800 Mbps per machine on clusters of 100 machines. We use a compute engine similar to Spark [41] that uses the coflow API. We use $\delta = 200$ milliseconds and $T = 2$ seconds as defaults.

Simulator We use a trace-driven simulator to gain more insights into Varys's performance at a larger scale. The simulator performs a detailed task-level replay of the Facebook trace. It preserves input-



(a) Improvements in job completion times



(b) Improvements in time spent in communication

Figure 6: [EC2] Average and 95th percentile improvements in job and communication completion times over per-flow fairness using Varys.

to-output ratios of tasks, locality constraints, and inter-arrival times between jobs. It 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, where

$$\text{Factor of Improvement} = \frac{\text{Current Duration}}{\text{Modified Duration}}$$

For deadline-sensitive coflows, the primary metric is the percentage of coflows that meet their deadlines.

The baseline for our deployment is TCP fair-sharing. We compare the trace-driven simulator against per-flow fairness as well. Due to the lack of implementations of per-flow prioritization mechanisms [8, 25], we compare against them only in simulation.

7.2 Varys's Performance in Minimizing CCT

Figure 6a shows that inter-coflow scheduling reduced the average and 95th percentile completion times of communication-dominated jobs by up to $2.5\times$ and $2.94\times$, respectively, in EC2 experiments. Corresponding average and 95th percentile improvements in the average CCT (CommTime) were up to $3.16\times$ and $3.84\times$ (Figure 6b). Note that varying improvements in the average CCT in different bins are not correlated, because it depends more on coflow characteristics than that of jobs. However, as expected, jobs become increasingly faster as the communication represent a higher fraction of their completion times. Across all bins, the average end-to-end completion times improved by $1.25\times$ and the average CCT improved by $1.85\times$; corresponding 95th percentile improvements were $1.15\times$ and $1.74\times$.

Figure 7 shows that Varys improves CCT for diverse coflow characteristics. Because bottlenecks are not directly correlated with a coflow's length or width, pairwise comparisons across bins – specially those involving bin-2 and bin-3 – are harder. We do observe more improvements for coflows in bin-1 than bin-4 in terms of average CCT, even though their 95th percentile improvements contradict. This is due to coordination overheads in Varys – recall that Varys does not handle small coflows to avoid fixed overheads.

Figure 8a presents comparative CDFs of CCTs for all coflows. Per-flow fairness performs better – $1.08\times$ on average and $1.25\times$ at the 95th percentile – only for some of the tiny, sub-second (<500 milliseconds) coflows, which still use TCP fair sharing. As coflows become larger, the advantages of coflow scheduling becomes more prominent. We elaborate on Varys's overheads next; later, we show simulation results that shed more light on the performance of small coflows in the absence of coordination overheads.

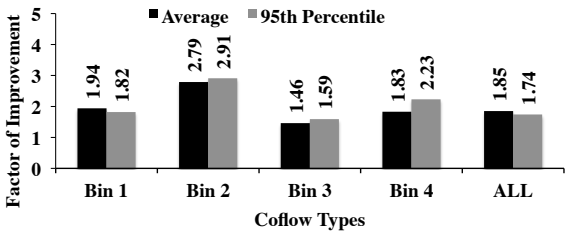
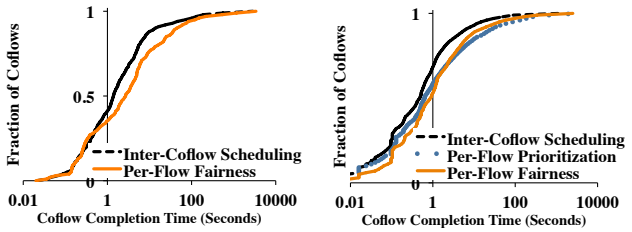


Figure 7: [EC2] Improvements in the average and 95th percentile CCTs using coflows w.r.t. the default per-flow fairness mechanism.



(a) EC2 **(b)** Simulation

Figure 8: CCT distributions for Varys, per-flow fairness, and per-flow prioritization schemes (a) in EC2 deployment and (b) in simulation. Note that the X-axes are in logarithmic scale.

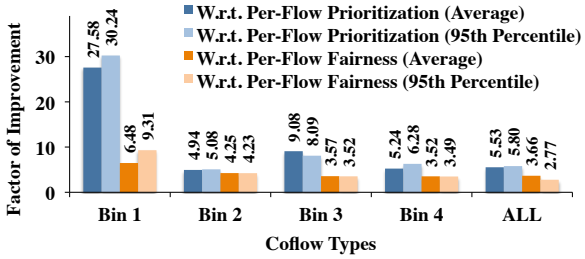


Figure 9: [Simulation] Improvements in the average and 95th percentile CCTs using inter-coflow scheduling.

Overheads Control plane messages to and from the Varys master are the primary sources of overheads. Multiple messages from the same endpoint are batched whenever possible. At peak load, we observed a throughput of 4000+ messages/second at the master. The scheduling algorithm took 17 milliseconds on average to calculate new schedules on coflow arrival or departure. The average time to distribute new schedules across the cluster was 30 milliseconds.

An additional source of overhead is the synchronization time before a coflow becomes READY for scheduling. Recall that a coflow waits for *numFlows* `get()` calls; hence, a single belated `get()` can block the entire coflow. In our experiments, the average duration to receive all `get()` calls was 44 milliseconds with 151 milliseconds being the 95th percentile.

A large fraction of these overheads could be avoided in the presence of in-network isolation of control plane messages [21].

Trace-Driven Simulation We compared the performance of inter-coflow scheduling against per-flow fairness and prioritization schemes in simulations. Without coordination overheads, the improvements are noticeably larger (Figure 9) – the average and 95th percentile CCTs improved by $3.66\times$ and $2.77\times$ over per-flow fairness and by $5.53\times$ and $5.8\times$ over per-flow prioritization.

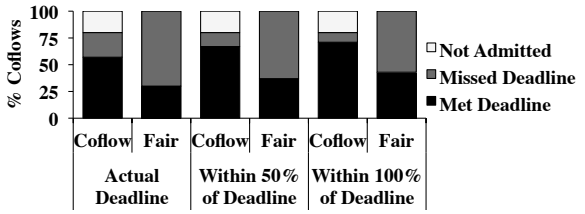


Figure 10: [EC2] Percentage of coflows that meet deadline using Varys in comparison to per-flow fairness. Increased deadlines improve performance.

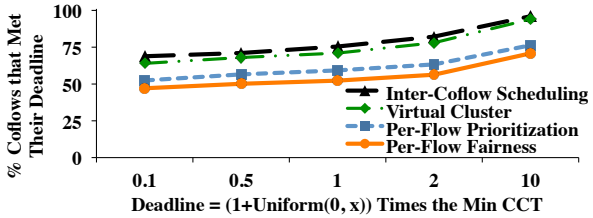


Figure 11: [Simulation] More coflows meet deadline using inter-coflow scheduling than using per-flow fairness and prioritization schemes.

Note that comparative improvements for bin-1 w.r.t. other bins are significantly larger than that in experiments because of the absence of scheduler coordination overheads. We observe larger absolute values of improvements in Figure 9 in comparison to the ones in Figure 7. Primary factors for this phenomenon include instant scheduling, zero-latency setup/cleanup/update of coflows, and perfectly timed flow arrivals (i.e., coflows are READY to be scheduled upon arrival) in the simulation. In the absence of these overheads, we see in Figure 8b that Varys can indeed outperform per-flow schemes even for sub-second coflows.

What About Per-Flow Prioritization? Figure 9 highlights that per-flow prioritization mechanisms are even worse (by 1.52×) than per-flow fairness provided by TCP when optimizing CCTs. The primary reason is indiscriminate interleaving across coflows – while all flows make some progress using flow-level fairness, per-flow prioritization favors only the small flows irrespective of the progress of their parent coflows. However, as expected, flow-level prioritization is still 1.08× faster than per-flow fairness in terms of the average FCT. Figure 8b presents the distribution of CCTs using per-flow prioritization in comparison to other approaches.

How Far are We From the Optimal? While finding the optimal schedule is infeasible, we tried to find an optimistic estimation of possible improvements by comparing against an *offline* 2-approximation combinatorial ordering heuristic for coflows *without* coupled resources [30]. We found that the average CCT did not change using the combinatorial approach. For bin-1 to bin-4, the changes were 1.14×, 0.96×, 1.46×, and 0.92×, respectively.

7.3 Varys's Performance for Deadline-Sensitive Coflows

Inter-coflow scheduling allowed almost 2× more coflows to complete within corresponding deadlines in EC2 experiments (Figure 10) – 57% coflows met their deadlines using Varys as opposed to 30% using the default mechanism. Coflows across different bins experienced similar results, which is expected because Varys does not differentiate between coflows when optimizing for deadlines.

Recall that because the original trace did not contain coflow-specific deadlines, we introduced them based on the minimum CCT of coflows (§7.1). Hence, we did not expect 100% admission rate. However, a quarter of the admitted coflows failed to meet their deadlines. This goes back to the lack of network support in estimating utilizations and enforcing Varys-determined allocations: Varys admitted more coflows than it should have had, which themselves missed their deadlines and caused some others to miss as well. Trace-driven simulations later shed more light on this.

To understand how far off the failed coflows were, we analyzed if they could complete with slightly longer deadlines. After doubling the deadlines, we found that almost 94% of the admitted coflows succeeded using Varys.

Trace-Driven Simulation In trace-driven simulations, for the default case (x=1), Varys admitted 75% of the coflows and *all* of them met their deadlines (Figure 11). Note that the admission rate is lower than that in our experiments. Prioritization schemes fared better than per-flow fairness unlike when the objective was minimizing CCT: 59% coflows completed within their deadlines in comparison to 52% using fair sharing.

As we changed the deadlines of all coflows by varying x from 0.1 to 10, comparative performance of all the approaches remained almost the same. Performance across bins were consistent as well.

	# Coflows	%SN	%LN	%SW	%LW
Mix-N	800	**48%**	**40%**	2%	10%
Mix-W	369	22%	15%	**24%**	**39%**
Mix-S	526	**50%**	17%	**10%**	23%
Mix-L	272	39%	**27%**	3%	**31%**

Table 5: Four extreme coflow mixes from the Facebook trace.

What About Reservation Schemes? Because the impact of admission control is similar to reserving resources, we compared our performance with that of the Virtual Cluster (VC) abstraction [11], where all machines can communicate at the same maximum rate through a virtual non-blocking switch. The VC abstraction admitted and completed slightly fewer coflows (73%) than Varys (75%), because reservation using VCs is more conservative.

7.4 Impact of Preemption

While minimizing CCT, preemption-based mechanisms can starve certain coflows when the system is overloaded. Varys takes precautions (§5.3.4) to avoid such scenarios. As expected, we did not observe any perpetual starvation during experiments or simulations.

What About a Non-Preemptive Scheduler? Processing coflows in their arrival order (i.e., FIFO) avoids starvation [15]. However, simulations confirmed that head-of-line blocking significantly hurts performance – specially, the short coflows in bin-1 and bin-3.

We found that processing coflows in the FIFO order can result in 24.64×, 5.44×, 34.2×, and 5.03× slower completion times for bin-1 to bin-4. The average (95th percentile) CCT became 5.65× (7.7×) slower than that using Varys.

7.5 Impact on Network Utilization

To understand Varys's impact on network utilization, we compared the ratios of *makespans* in the original workload as well as the ones in Table 5. Given a fixed workload, a change in makespan means a change in aggregate network utilization.

We did not observe significant changes in makespan in our EC2 experiments – the exact factors of improvements were 1.02×, 1.06×, 1.01×, 0.97×, and 1.03× for the five workloads. This is expected because while Varys is not work-conserving at every point in time, its overall utilization is the same as non-coflow approaches.

Makespans for both per-flow fairness and coflow-enabled schedules were the same in the trace-driven simulation.

7.6 Impact of Coflow Mix

To explore the impact of changes in the coflow mix, we selected four extreme hours (Table 5) from the trace and performed hour-long experiments on EC2. These hours were chosen based on the

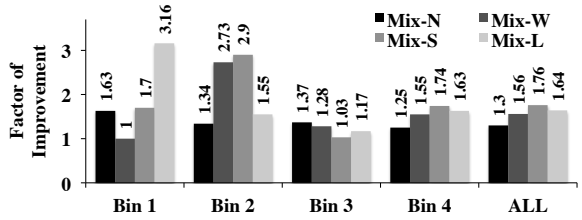


Figure 12: [EC2] Improvements in the average CCT using coflows for different coflow mixes from Table 5.

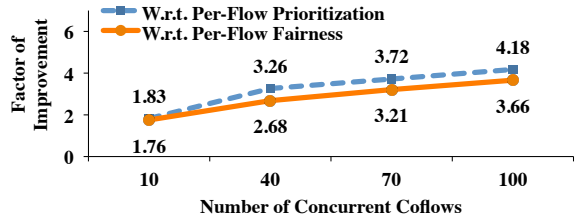


Figure 13: [Simulation] Improvements in the average CCT for varying numbers of concurrent coflows.

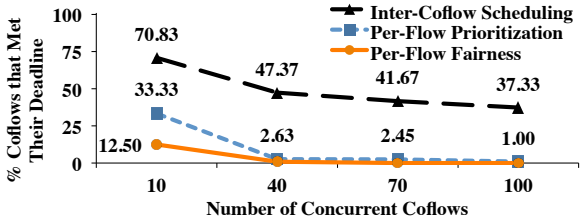


Figure 14: [Simulation] Changes in the percentage of coflows that meet deadlines for varying numbers of concurrent coflows.

high percentage of certain types of coflows (e.g., narrow ones in Mix-N) during those periods.

Figure 12 shows that the average CCT improves irrespective of the mix, albeit in varying degrees. Observations made earlier (§7.2) still hold for each mix. However, identifying the exact reason(s) for different levels of improvements is difficult. This is due to the online nature of the experiments – the overall degree of improvement depends on the instantaneous interplay of concurrent coflows. We also did not observe any clear correlation between the number of coflows or workload size and corresponding improvements.

7.7 Impact of Cluster/Network Load

So far we have evaluated Varys's improvements in online settings, where the number of concurrent coflows varied over time. To better understand the impact of network load, we used the same coflow mix as the original trace but varied the number of concurrent coflows in an offline setting. We see in Figure 13 that Varys's improvements increase with increased concurrency: per-flow mechanisms fall increasingly further behind as they ignore the structures of more coflows. Also, flow-level fairness consistently outperforms per-flow prioritization mechanisms in terms of the average CCT.

Deadline-Sensitive Coflows We performed a similar analysis for deadline-sensitive coflows. Because in this case Varys's performance depends on the arrival order, we randomized the coflow order across runs and present their average in Figure 14. We observe that as the number of coexisting coflows increases, a large number of coflows (37% for 100 concurrent coflows) meet their deadlines using Varys; per-flow mechanisms completely stop working even before. Also, per-flow prioritization outperforms (however marginally) flow-level fairness for coflows with deadlines.

8 Discussion

Scheduling With Unknown Flow Sizes Knowing or estimating exact flow sizes is difficult in frameworks that push data to the next stage as soon as possible [26], and without known flow sizes, preemption becomes impractical. FIFO scheduling can solve this problem, but it suffers from head-of-line blocking (§7.4). We believe that coflow-level fairness can be a nice compromise between these two extremes. However, the definition and associated properties of fairness at the coflow level is an open problem.

Decentralized SEBF+MADD Varys's centralized design makes it less useful for small coflows (§3); however, small coflows contribute less than 1% of the traffic in data-intensive clusters (§4). Furthermore, in-network isolation of control plane messages [21] or faster signaling channels like RDMA [20] can reduce Varys's application-layer signaling overheads (§7.2) to support even smaller coflows. We see a decentralized approximation of our algorithms as the most viable way to make Varys useful for low-latency coflows. This requires new algorithms and possible changes to network devices, unlike our application-layer design.

Handling Coflow Dependencies While most jobs require only a single coflow, dataflow pipelines (e.g., Dryad, Spark) can create multiple coflows with dependencies between them [16]. A simple approach to support coflow dependencies would be to order first by ancestry and then breaking ties using SEBF. Some variation of the Critical-Path Method [28] might perform even better. We leave it as a topic of future work. Note that dependencies can be passed along to the scheduler through *options* in the `register()` method.

Multi-Wave Coflows Large jobs often schedule mappers in multiple waves [10]. A job can create separate coflows for each wave. Alternatively, if the job uses its wave-width (i.e., the number of parallel mappers) as *numFlows* in `register()`, Varys can handle each wave separately. Applications can convey information about wave-induced coflows to the scheduler as dependencies.

In-Network Bottlenecks Varys performs well even when the network is not a non-blocking switch (§7). If likely bottleneck locations are known, e.g., rack-to-core links are typically oversubscribed [17], Varys can be extended to allocate rack-to-core bandwidth instead of NIC bandwidth. When bottlenecks are unknown, e.g., due to in-network failures, routing, or load imbalance, Varys can react based on bandwidth estimations collected by its daemons. Nonetheless, designing and deploying coflow-aware routing protocols and load balancing techniques remain an open challenge.

9 Related Work

Coflow Schedulers Varys improves over Orchestra [15] in four major ways. First, Orchestra primarily optimizes individual coflows and uses FIFO among them; whereas, Varys uses an efficient coflow scheduler to significantly outperform FIFO. Second, Varys supports deadlines and ensures guaranteed coflow completion. Third, Varys uses a rate-based approach instead of manipulating the number of TCP flows, which breaks if all coflows do not share the same endpoints. Finally, Varys supports coflows from multiple frameworks like Mesos [24] handles non-network resources.

Baraat [19] is a FIFO-based decentralized coflow scheduler focusing on small coflows. It uses fair sharing to avoid head-of-line blocking and does not support deadlines. Furthermore, we formulate the coflow scheduling problem and analyze its characteristics.

Datacenter Traffic Management Hedera [7] manages flows using a centralized scheduler to increase network throughput, and MicroTE [12] adapts to traffic variations by leveraging their short-term predictability. However, both work with flows and are unsuit-

able for optimizing CCTs. Sinbad [17] uses endpoint flexible transfers for load balancing. Once it makes network-aware placement decisions, Varys can optimize cross-rack write coflows.

High Capacity Networks Full bisection bandwidth topologies [22,32] do not imply contention freedom. In the presence of skewed data and hotspot distributions [17], managing edge bandwidth is still necessary. Inter-coflow scheduling improves performance and predictability even in these high capacity networks.

Traffic Reduction Techniques Data locality [18], both disk [9,40] and memory [10], reduces network usage only during reads. The amount of network usage due to intermediate data communication can be reduced by pushing filters toward the sources [6, 23]. Our approach is complementary; i.e., it can be applied to whatever data traverses the network after applying those techniques.

Network Sharing Among Tenants Fair sharing of network resources between multiple tenants has received considerable attention [11, 33, 35, 39]. Our work is complementary; we focus on optimizing performance of concurrent coflows within a single administrative domain, instead of achieving fairness among competing entities. Moreover, we focus on performance and predictability as opposed to the more debated notion of fairness.

Concurrent Open Shop Scheduling Inter-coflow scheduling has its roots in the concurrent open shop scheduling problem [34], which is strongly NP-hard for even two machines. Even in the offline scenario, the best known result is a 2-approximation algorithm [30], and it is inapproximable within a factor strictly less than $6/5$ if P$\neq$NP [30]. Our setting is different as follows. First, machines are not independent; i.e., links are coupled because each flow involves a source and a destination. Second, jobs are not known a priori; i.e., coflows arrive in an online fashion.

10 Concluding Remarks

The coflow abstraction [16] effectively enables application-aware network scheduling. We have implemented coflows in a system called Varys and introduced the *concurrent open shop scheduling with coupled resources* problem. To minimize coflow completion times (CCT), we proposed the SEBF heuristic to schedule coflows and the MADD algorithm to allocate bandwidth to their flows. Together, they decrease the average CCT without starving any coflow and maintain high network utilization. Through EC2 deployments and trace-driven simulations, we showed that Varys outperforms per-flow mechanisms by up to $3.16\times$ and non-preemptive coflow schedulers by more than $5\times$. Furthermore, by applying MADD in conjunction with admission control, Varys allowed up to $2\times$ more coflows to meet their deadlines in comparison to per-flow schemes.

In conclusion, this paper is only a first step in understanding the intricacies of inter-coflow scheduling and opens up a variety of exciting research problems, which include scheduling without knowing flow sizes, exploring the notion of coflow fairness, decentralizing the proposed algorithms, and handling coflow dependencies.

Acknowledgments

We thank Mohammad Alizadeh, Justine Sherry, Rachit Agarwal, Peter Bailis, Ganesh Ananthanarayanan, Tathagata Das, Ali Ghodsi, Gautam Kumar, David Zats, Matei Zaharia, the AMPLab members, our shepherd Nandita Dukkipati, and the anonymous reviewers of NSDI'14 and SIGCOMM'14 for useful feedback. 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, Apple, Inc., Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, WANdisco and Yahoo!.

11 References

[1] Akka. http://akka.io.
[2] Amazon EC2. http://aws.amazon.com/ec2.
[3] Apache Hadoop. http://hadoop.apache.org.
[4] Apache Hive. http://hive.apache.org.
[5] Kryo serialization library. https://code.google.com/p/kryo.
[6] S. Agarwal et al. Reoptimizing data parallel computing. In *NSDI'12*.
[7] M. Al-Fares et al. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*. 2010.
[8] M. Alizadeh et al. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*. 2013.
[9] G. Ananthanarayanan et al. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*. 2010.
[10] G. Ananthanarayanan et al. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*. 2012.
[11] H. Ballani et al. Towards predictable datacenter networks. In *SIGCOMM*. 2011.
[12] T. Benson et al. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*. 2011.
[13] D. Borthakur. The Hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007.
[14] R. Chaiken et al. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*. 2008.
[15] M. Chowdhury et al. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*. 2011.
[16] M. Chowdhury et al. Coflow: A networking abstraction for cluster applications. In *HotNets-XI*, pages 31–36. 2012.
[17] M. Chowdhury et al. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*. 2013.
[18] J. Dean et al. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. 2004.
[19] F. Dogar et al. Decentralized task-aware scheduling for data center networks. Technical Report MSR-TR-2013-96, 2013.
[20] A. Dragojević et al. FaRM: Fast remote memory. In *NSDI*. 2014.
[21] A. D. Ferguson et al. Participatory networking: An API for application control of SDNs. In *SIGCOMM*. 2013.
[22] A. Greenberg et al. VL2: A scalable and flexible data center network. In *SIGCOMM*. 2009.
[23] Z. Guo et al. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *OSDI*. 2012.
[24] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*. 2011.
[25] C.-Y. Hong et al. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*. 2012.
[26] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72. 2007.
[27] N. Kang et al. Optimizing the "One Big Switch" abstraction in Software-Defined Networks. In *CoNEXT*. 2013.
[28] J. E. Kelley. Critical-path planning and scheduling: Mathematical basis. *Operations Research*, 9(3):296–320, 1961.
[29] G. Malewicz et al. Pregel: A system for large-scale graph processing. In *SIGMOD*. 2010.
[30] M. Mastrolilli et al. Minimizing the sum of weighted completion times in a concurrent open shop. *Operations Research Letters*, 38(5):390–395, 2010.
[31] N. McKeown et al. Achieving 100% throughput in an input-queued switch. *IEEE Transactions on Communications*, 47(8), 1999.
[32] R. N. Mysore et al. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, pages 39–50. 2009.
[33] L. Popa et al. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*. 2012.
[34] T. A. Roemer. A note on the complexity of the concurrent open shop problem. *Journal of Scheduling*, 9(4):389–396, 2006.
[35] A. Shieh et al. Sharing the data center network. In *NSDI*. 2011.
[36] N. Tolia et al. An architecture for internet data transfer. In *NSDI'06*.
[37] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
[38] C. A. Waldspurger et al. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*. 1994.

[39] D. Xie et al. The only constant is change: Incorporating time-varying network reservations in data centers. In *SIGCOMM*. 2012.
[40] M. Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*. 2010.
[41] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. 2012.

APPENDIX

A Problem Formulation and Complexity

Each coflow $C(\mathbf{D})$ is a collection of flows over the datacenter backplane with P ingress and P egress ports, where the $P \times P$ matrix $\mathbf{D} = [d_{ij}]_{P\times P}$ represents the structure of C. For each non-zero element $d_{ij} \in \mathbf{D}$, a flow f_{ij} transfers d_{ij} amount of data from the ith ingress port (P_i^{in}) to the jth egress port (P_j^{out}) across the backplane at rate r_{ij}, which is determined by the scheduling algorithm.

If C_k represents the time for all flows of the kth coflow to finish and $r_{ij}^k(t)$ the bandwidth allocated to f_{ij} of the kth coflow at time t, the objective of minimizing CCT ($\mathsf{O}(.)$) in the offline case can be represented as follows.

$$\text{Minimize} \sum_{k=1}^{K} C_k \tag{2}$$

$$\text{Subject to} \sum_{j',k} r_{ij'}^k(t) \le 1 \qquad \forall t, \forall i; \tag{3}$$

$$\sum_{i',k} r_{i'j}^k(t) \le 1 \qquad \forall t, \forall j; \tag{4}$$

$$\sum_{t=1}^{C_k} r_{ij}^k(t) \ge d_{ij}^k \qquad \forall i, j, k. \tag{5}$$

The first two inequalities are the capacity constraints on ingress and egress ports. The third inequality ensures that all flows of the kth coflow finish by time C_k.

By introducing a binary variable U_k to denote whether a coflow finished within its deadline D_k, we can express the objective of maximizing the number of coflows that meet their deadlines ($\mathsf{Z}(.)$) in the offline case as follows.

$$\text{Maximize} \sum_{k=1}^{K} U_k \tag{6}$$

Subject to inequalities (3), (4), and (5);

$$\text{Where } U_k = \begin{cases} 1 & C_k \le D_k \\ 0 & C_k > D_k \end{cases}$$

Optimizing either objective (O or Z) is NP-hard.

Theorem A.1 *Even under the assumptions of Section 5.1, optimizing* O *or* Z *in the offline case is NP-hard for all* $P \ge 2$.

Proof Sketch We reduce the NP-hard concurrent open shop scheduling problem [34] to the coflow scheduling problem. Consider a network fabric with only 2 ingress and egress ports ($P = 2$) and all links have the same capacity (without loss of generality, we can let this capacity be 1). Since there are only 2 ports, all coflows are of the form $C(\mathbf{D})$, where $\mathbf{D} = (d_{ij})_{i,j=1}^2$ is a 2×2 data matrix. Suppose that n coflows arrive at time 0, and let $\mathbf{D}^k = (d_{ij}^k)_{i,j=1}^2$ be the matrix of the kth coflow. Moreover, assume for all k, $d_{ij}^k = 0$ if $i = j$. In other words, every coflow only consists of 2 flows, one sending data from ingress port P_1^{in} to egress port P_2^{out}, and the other sending from ingress port P_2^{in} to egress port P_1^{out}.

Consider now an equivalent concurrent open shop scheduling problem with 2 identical machines (hence the same capacity). Suppose n jobs arrive at time 0, and the kth job has d_{12}^k amount of work for machine 1 and d_{21}^k for machine 2. Since this is NP-hard [34], the coflow scheduling problem described above is NP-hard as well. ■

Remark A.2 Given the close relation between concurrent open shop scheduling and coflow scheduling, it is natural to expect that techniques to express concurrent open shop scheduling as a mixed-integer program and using standard LP relaxation techniques to derive approximation algorithms [30, 34] would readily extend to our case. However, they do not, because the coupled constraints (3) and (4) make permutation schedules sub-optimal (**Theorem C.1**). We leave the investigation of these topics as future work.

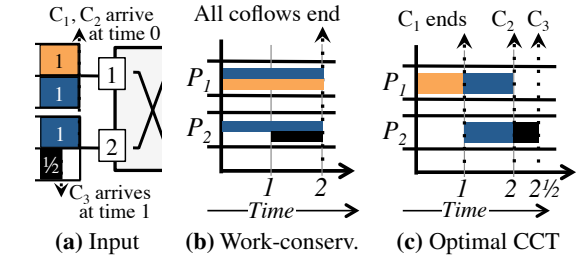


Figure 15: Allocation of *ingress* port capacities (vertical axis) for the coflows in (a) on a 2×2 datacenter fabric for (b) a work-conserving and (c) a CCT-optimized schedule. While the former is work-conserving and achieves higher utilization, the latter has a lower average CCT.

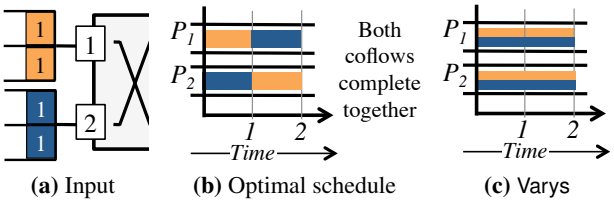


Figure 16: Flow-interleaved allocation of *ingress* port capacities (vertical axis) for the coflows in (a) for CCT-optimality (b).

B Tradeoffs in Optimizing CCT

With Work Conservation Consider Figure 15a. Coflows C_1 and C_2 arrive at time 0 with one and two flows, respectively. Each flow transfers unit data. C_3 arrives one time unit later and uses a single flow to send 0.5 data unit. Figure 15b shows the work-conserving solution, which finishes in 2 time units for an average CCT of 1.67 time units. The optimal solution (Figure 15c), however, takes 2.5 time units for the *same* amount of data (i.e., it lowers utilization); still, it has a $1.11\times$ lower average CCT (1.5 time units).

With Avoiding Starvation The tradeoff between minimum completion time and starvation is well-known for flows (tasks) on individual links (machines) – longer flows starve if a continuous stream of short flows keep arriving. The same tradeoff holds for coflows, because the datacenter fabric and coflows generalize links and flows, respectively.

C Ordering Properties of Coflow Schedules

Theorem C.1 *Permutation schedule is not optimal for minimizing the average CCT.*

Proof Sketch Both permutation schedules – C_1 before C_2 and C_2 before C_1 – would be suboptimal for the example in Figure 16a. ■

Remark C.2 In Varys, SEBF would schedule C_1 before C_2 (arbitrarily breaking the tie), and iterative MADD will allocate the minimum bandwidth to quickly finish C_1 and then give the remaining bandwidth to C_2 (Figure 16c). The average CCT will be the same as the optimal for this example.