Sincronia: Near-Optimal Network Design for Coflows

Paper #xyz, 18 Pages

ABSTRACT

We present Sincronia, a near-optimal network design for coflows that can be implemented on top on any transport layer (for flows) that supports priority scheduling. Sincronia achieves this using a key technical result — we show that given a "right" ordering of coflows, it is possible to achieve average coflow completion time within $4\times$ of the optimal as long as (co)flows are prioritized with respect to the ordering (irrespective of the rates assigned to individual flows).

Sincronia uses a simple greedy mechanism to periodically order all unfinished coflows; each host sets priorities for its flows using corresponding coflow order and offloads the flow scheduling and rate allocation to the underlying priority-enabled transport layer. We evaluate Sincronia over a real testbed comprising 16-servers and commodity switches, and using simulations across a variety of workloads. Evaluation results suggest that Sincronia not only admits a practical, near-optimal design but also improves upon state-of-the-art network designs for coflows (sometimes by as much as $8\times$).

1. INTRODUCTION

Traditionally, networks have used the abstraction of a "flow" to capture a sequence of packets between a single source and a single destination. This abstraction has been a mainstay for decades and for a good reason — network designs were optimized for precisely the performance metrics important to traditional applications (*e.g.*, file transfers, web access, etc.) — latency and/or throughput for a point-to-point connection. However, distributed applications running across datacenter networks use programming models (*e.g.*, Bulk Synchronous Programming model [1] and partition-aggregate model [2]) that require optimizing performance for a collection of flows rather than individual flows. The network still optimizes the performance of individual flows, leading to a fundamental mismatch between performance objectives of applications and the optimization objectives of network designs.

The coflow abstraction [3] mitigates this mismatch, allowing distributed applications to more precisely express their performance objectives to the network fabric. For instance, many distributed services with stringent performance constraints must essentially block until receiving all or almost all responses from hundreds or even thousands of remote servers (§2). Such services can specify a collection of flows as a coflow. The network fabric now optimizes for average Coflow Completion Time (CCT) [3, 4, 5], where the CCT of a coflow is defined as the time when some percentage, perhaps 100%, of flows in the coflow finish. Several recent evaluations show that optimizing for average CCT can significantly improve application-level performance [3,4,5].

There has been tremendous recent effort on network designs for coflows, both in networking [3,4,5,6,7,8] and in the theory community [9, 10, 11, 12, 13]. However, as we show in §3.1, heuristics developed in the former set of results can perform arbitrarily worse than optimal in terms of average CCT. In addition, all prior network designs require complex per-flow rate allocation with rate allocated to a flow being dependent on the rate allocated to other flows in the same coflow; such dependencies in flow rate allocation make it hard to realize these designs in practice for several reasons. First, per-flow rate allocation naturally requires knowledge about location of congestion in the network and paths taken by each (co)flow, making it hard to use these designs when congestion is in the fabric core and/or changes dynamically. Second, since rates allocated to flows are correlated, arrival of even one coflow may result in reallocation of rate for each and every flow in the network. Such reallocation is impractical in large datacenters where hundreds or thousands of coflows may arrive each second. As a result, a practical near-optimal network design for coflows still remains elusive.

This paper presents Sincronia, a new datacenter network design for coflows that achieves near-optimal average CCT without any explicit per-flow rate allocation mechanism. The high-level design of Sincronia can be summarized as:

- Time is divided into *epochs*;
- In each epoch, a subset of unfinished coflows are selected and "ordered" using a simple greedy algorithm;
- Each host independently sets a priority for its flows (based on the corresponding coflow's ordering), and offloads the flow to underlying priority-enabled transport mechanism;
- Coflows that arrive between epoch boundaries are greedily scheduled for work conservation.

Sincronia's minimalistic design is based on a key technical result — with a "right" ordering $\mathscr{O}$ of coflows, it is possible to achieve average CCT within 4× of the optimal[1] as long as (co)flow scheduling is "order-preserving" — if coflow C is ordered higher than coflow C', flows/packets in C must be prioritized over those in C'. From a practical perspective, this result is interesting because it shows that as long as flow scheduling is order-preserving, *any* per-flow rate allocation mechanism results in average CCT within 4× of the optimal. Using this result, Sincronia admits a practical, near-optimal network design for coflows (§3) — a simple greedy algorithm periodically orders the set of unfinished coflows; each host, without any explicit coordination with other hosts, sets priorities for its flows using corresponding coflow order and offloads the flow scheduling and rate allocation to the underlying priority-enabled transport layer.

Sincronia thus overcomes the aforementioned practical challenges in existing network designs for coflows by avoiding per-flow rate allocation and by being agnostic to the underlying transport layer. First, unlike prior works [3, 4, 6, 9, 10, 11, 12, 13], Sincronia design is independent of the underlying network topology, location of congestion in the fabric, and paths taken by each (co)flow. Second, coflow arrivals do not require explicit rate reallocation for existing flows, leading to a much more scalable design. Third, as we discuss in §4, Sincronia enables coexistence of flows and coflows, supporting backward compatibility. Finally, our transport-agnostic approach allows Sincronia to transparently respond to network failures and in-network congestion without any additional hardware requirements.

We have implemented Sincronia with TCP (using DiffServ [17] for priority scheduling). Our implementation is work conserving, efficiently handles online arrivals of coflows and allows Sincronia to co-exist with applications that use the traditional flow abstraction. We evaluate our Sincronia implementation with TCP/DiffServ on a 16-server testbed interconnected with a FatTree topology comprising 20 commodity switches. We have also incorporated Sincronia into simulators for several existing datacenter transport mechanisms including NDP [15], pFabric [14] and Fastpass [18]. We use these simulators to perform sensitivity analysis of Sincronia performance against variety of workloads, underlying transport mechanisms, number of coflows, etc. Our implementation and simulation results show that Sincronia not only provides near-optimal average CCT but also outperforms state-of-the-art network designs for coflows across all evaluated workloads (sometimes by as much as 8×).

[1]Under the standard assumption that the fabric core can sustain 100% throughput. That is, only the ingress and the egress access links are potential bottlenecks. This is the same assumption made in the big switch model for traditional abstraction of flows [14, 15, 16]. For the case when the congestion can be in the core of the network, no approximation algorithm is known even for the traditional abstraction of flows. While we use this assumption for our theoretical bounds, our implementation makes no such assumption and adapts well to in-network congestion.

Table 1: Examples of distributed programming frameworks that motivate the need for the coflow abstraction.

Distributed Frameworks	Coflow structure
Partition [2, 4, 6, 19, 20]	One-to-Many
Aggregate [1, 2, 5, 19, 20, 21]	Many-to-One
Dataflow Pipelines [2, 5, 19, 20, 21]	Many-to-Many
Global communication barriers [9, 22, 23]	All-to-All

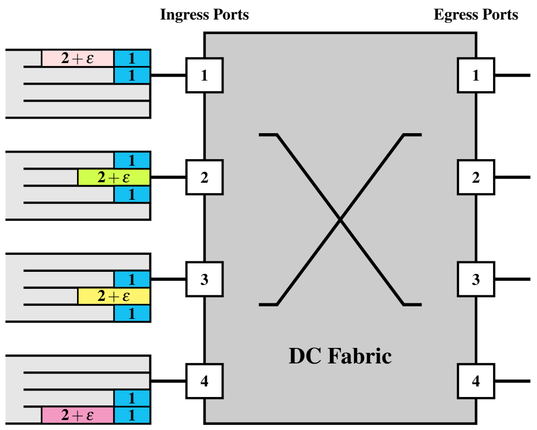


Figure 1: An instance of a coflow scheduling problem, used as a running example in the paper. The datacenter has 4 ingress and egress ports, with each ingress port having a "virtual output queue" for each egress port (see §2.2 for detailed model description). The example has 5 coflows. Coflow C1 has eight flows, with each ingress port sending unit amount of data to two egress ports; Coflow C2, C3 and C4 have one flow each sending $2+\varepsilon$ amount of data to one of the egress ports (the ε amount is only for breaking ties consistently). Coflow C1, C2, C3 and C4 being single flow coflow is only for simplicity; the ε amount of flow could be sent to any egress port without changing the results in §3.

2. SINCRONIA OVERVIEW

In this section, we briefly recall the coflow abstraction (§2.1) and formally define our optimization objective for network designs for coflows (§2.2). We also review some of the known results in coflow scheduling in §2.2.

2.1 The Coflow Abstraction

Most distributed programming frameworks differ in implementation details but have a communication stage that is often structured and takes place between successive computation stages (see Table 1). Often, execution of a task (or even an entire computation stage) cannot begin until all flows in the preceding communication stage have finished. A coflow [3, 4, 6] is a collection of such flows, with a shared performance goal (*e.g.*, minimizing the completion time of the last flow in a coflow). Figure 1 shows an example.

We assume that coflows are defined such that flows within a coflow are independent; that is, the input of a flow does not depend on the output of another flow within that coflow. Similar to most existing designs [4, 5, 8, 11, 12, 13], we focus on a clairvoyant design that assumes information about a coflow (set of flows, and corresponding sources, destinations and sizes) is known at coflow's arrival time but no earlier.

2.2 Problem Statement and Prior Results

We now describe the network model used for our theoretical bounds, and the network performance objective.

Conceptual Model (for theoretical bounds). Similar to near-optimal network designs for both traditional abstraction of network flows [14, 15] and coflows [4, 5, 9, 10, 11, 12, 13], we will abstract out the datacenter network fabric as one big switch that interconnects the servers. The ingress queues in the big switch model correspond to the NICs and the egress queues out of the big switch are at the last-hop TOR switches. The assumption in this model is that the fabric core can sustain 100% throughput and only the ingress and egress access links are potential congestion points. Under this model, each ingress port has flows from one or more coflows to various egress ports. For ease of exposition, we organize the flows in Virtual Output Queues at the ingress ports (see Figure 1). We use this abstraction to simplify our theoretical analysis and algorithmic description, but we do not enforce it in our design and experiments.

Performance Objective. More formally, we assume that the network is a big switch comprising m ingress ports $\{1,2,\ldots,m\}$ and m egress ports $\{m+1,m+2,\ldots,2m\}$. Unless mentioned otherwise, all ports have the same bandwidth. We are given a collection of n coflows $\mathbb{C} = \{1,2,\ldots,n\}$, indexed using c. Each coflow c may be assigned a weight w_c (default weight is 1), has an arrival time a_c and comprises a set of flows $\mathbb{F}_c$. The source, the destination and the size for each flow in the coflow is known at time a_c. The total amount of data sent by coflow c between ingress port i and egress port j is denoted by d_c^{ij} (see Table 2 for notation).

The completion time of a coflow (CCT_c) is the time when the last of its flows finishes. The average CCT for $\mathbb{C}$ is the average of individual completion times of all coflows $\sum_c \text{CCT}_c/n$. The weighted average CCT is similarly defined as $(\sum_c w_c \times \text{CCT}_c)/n$. Given this formulation, prior work has established that (detailed discussion of related work in §7):

NP**-Hardness [4].** Even when all coflows arrive at time 0 and their sizes are known in advance, the problem of minimizing average CCT is NP-hard (via reduction from concurrent open-shop scheduling problem [24]). Thus, the best we can hope is an approximation algorithm.

Lower Bounds [25, 26]. Even when the congestion is at the ingress or the egress ports (that is, the fabric can sustain 100% throughput), the only known lower bound is a natural generalization of the lower bound for flows — under a complexity-theoretic assumption somewhat stronger than $\mathsf{P}\neq\mathsf{NP}$, it is impossible to minimize (weighted) average coflow completion time within a factor of $2-\varepsilon$.

Necessity for Coordination [7]. There exists an instance of coflow scheduling problem, where a scheduling algorithm that does not use any coordination will achieve average CCT $\Omega(\sqrt{n})$ of the optimal. Thus, at least some coordination is necessary to achieve any meaningful approximation.

Table 2: Notation used in algorithms.

w_c	weight of coflow c (default = 1)
d_c^{ij}	data sent by coflow c between ingress port i & egress port j
$d_c^p = \begin{cases} \sum_i d_c^{pi} \\ \sum_j d_c^{jp} \end{cases}$	Total data sent by coflow c at port p if p is an ingress port if p is an egress port

3. SINCRONIA DESIGN

In this section, we present the core of Sincronia design — an offline algorithm for coflow scheduling; next section describes how Sincronia implementation incorporates this offline algorithm into an end-to-end network design that achieves near-optimal performance while scheduling coflows in an online and work conserving manner.

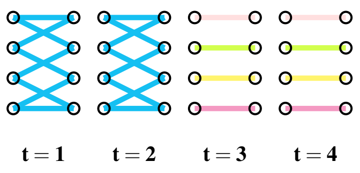
Our offline algorithm has two components. The first component is a combinatorial primal-dual greedy algorithm, Bottleneck-Select-Scale-Iterate, for ordering coflows (§3.1); the second component shows that any per-flow rate allocation mechanism that is work conserving, preemptive and schedules flows in the order of corresponding coflow ordering achieves average CCT within $4\times$ of the optimal (§3.2).

3.1 Coflow Ordering

Sincronia uses a primal-dual based greedy algorithm — Bottleneck-Select-Scale-Iterate (BSSI) — for ordering coflows (Algorithm 1). BSSI generalizes a near-optimal flow scheduling mechanism, “Shortest Remaining Processing Time” first (SRPT-first) [14], to the case of coflows. The main challenge in achieving such a generalization is to capture how scheduling a coflow impacts the completion time of other coflows in the network (across all ingress and egress ports). BSSI achieves this using a novel weight scaling step that is derived based on the primal-dual framework.

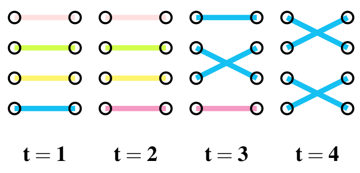
Specifically, as the name suggests, BSSI operates in four steps — bottleneck, select, scale, iterate. In its first two steps, BSSI generalizes SRPT-first policy for flows to the case of coflows. Intuitively, it does so using an alternative view of SRPT — Largest Remaining Processing Time last (LRPT-last). In particular, the first step finds the most bottlenecked ingress or egress port, say b, defined as the port that sends or receives the most amount of data across all unordered coflows; the second step then implements LRPT-last: it chooses the coflow with largest remaining processing time at port b (the coflow that sends most amount of data at the bottleneck port) and places this coflow the last among all unordered coflows. The third step in BSSI scales the weights of all unordered coflows to capture how ordering the coflow chosen in the second step impacts the completion time of all remaining coflows. The final step is to simply iterate on the set of unordered flows until all coflows are ordered.

An Example. We show the execution of Algorithm 1 on ex-



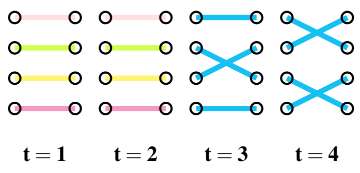
average CCT = $(2+4\times 4)/4 = 4.5$

(a) Varys



average CCT = $(3\times 2+3+4)/4 = 3.25$

(b) Sincronia



average CCT = $(4\times 2+4))/4 = 3$

(c) Optimal

Figure 2: Comparison of Sincronia against Varys and Optimal for the example of Figure 1; corresponding average CCTs are shown for $\varepsilon = 0$. Each figure shows a "matching" between ports at different time slots indicating the data being sent between ports at each time slot; for instance, the left part of (a) shows that Varys sends data from first ingress port to the first two egress ports in first two time steps, with two outgoing links representing equal rate allocation and the right part of (a) shows that Varys sends data from first ingress port to first egress port in third and fourth time steps at full rate. The orderings produced by Varys and Sincronia are {1, 2, 3, 4, 5} and {2, 3, 4, 1, 5} respectively (modulo various permutations within coflows 1,2,3 and 4). The optimal schedule requires ordering {2, 3, 4, 5, 1}. Sincronia is within 8% of the optimal and it gets closer as more coflows are added. Varys is 1.5× worse than the optimal and gets worse as more coflows are added.

Algorithm 1 Bottleneck-Select-*Scale*-Iterate Algorithm

```
$\mathbb{C} = [n]$                          ▷ Initial set of unscheduled coflows
procedure ORDER-COFLOWS($J$)
    for $k = n$ to 1 do      ▷ Note ordering is from last to first
        ▷ Find the most bottlenecked port
        $b \leftarrow \arg\max_p \sum_{c\in\mathbb{C}} d_c^p$
        ▷ Select weighted largest job to schedule last
        $\sigma(k) \leftarrow \arg\min_{c\in\mathbb{C}}(w_c/d_c^b)$
        ▷ Scale the weights
        $w_c \leftarrow w_c - w_{\sigma(k)} \times \frac{d_c^b}{d_{\sigma(k)}^b} \quad \forall c \in \mathbb{C}\setminus\{\sigma(k)\}$
        ▷ Iterate on updated set of unscheduled jobs
        $\mathbb{C} \leftarrow \mathbb{C}\setminus\{\sigma(k)\}$
    return $\sigma$                 ▷ Output the coflow permutation
```

Table 3: Execution of Algorithm 1 on example of Figure 1. The final ordering produced by the algorithm is $\{2,3,4,1,5\}$.

k	b	σ(k)	$\{w_1, w_2, w_3, w_4, w_5\}$	$\mathbb{C}$
–	–	–	$\{1,1,1,1,1\}$	$\{1,2,3,4,5\}$
5	4	5	$\{\varepsilon/(2+\varepsilon),1,1,1,0\}$	$\{1,2,3,4\}$
4	3	1	$\{0,1,1,1-\varepsilon/2,0\}$	$\{2,3,4\}$
3	3	4	$\{0,1,1,0,0\}$	$\{2,3\}$
2	2	3	$\{0,1,0,0,0\}$	$\{2\}$
1	1	2	$\{0,0,0,0,0\}$	$\emptyset$

ample of Figure 1 in Table 3. Figure 7 compares the performance of Varys [4] against Sincronia for this example demonstrating that, even for this simple example, Sincronia improves the average CCT of Varys by 1.38× and is within 1.08× of the optimal. It is not very hard to show that that the average CCT of Varys can be made arbitrarily worse compared to Sincronia (and optimal) by adding more ports and corresponding coflows, or by adding more coflows in the above example [27]. More interestingly, as we increase the number of ports and/or number of coflows, the average CCT of Sincronia converges to that of optimal.

3.2 Per-Flow Rate Allocation is Irrelevant

All prior network designs for coflows require complex per-flow rate allocation, where rates allocated to flows within a coflow are dependent on each other; for instance, Varys [4] allocates rates to flow in proportion to the respective flow sizes. Such dependencies in flow rate allocation make it hard to realize these designs in practice since changes in location of congestion in the network, transient failures and arrival of even one coflow may result in reallocation of rate to each and every flow in the network. Such reallocations are impractical in large datacenters where hundreds or thousands of coflows may arrive each second. We now discuss how Sincronia completely offloads the rate allocation and scheduling of individual flows to the underlying priority-enabled transport layer. We start with an intuitive discussion, followed by a more formal statement of the result.

High-level idea. Given a coflow ordering produced by our BSSI algorithm, we show that it is sufficient for Sincronia to schedule flows in an order-preserving manner; that is, at any time, a flow from coflow C is blocked if and only if either its ingress port or its egress port is busy serving a flow from a different coflow C' that is ordered before C. The reason this is sufficient is that once a strict ordering between coflows has been established, the proof simply requires finishing each coflow as soon as possible while following the ordering. The main insight is that if there are multiple flows within a coflow starting at an ingress port or ending at an egress port, sharing the link bandwidth does not improve the completion time of this coflow. For instance, in example of Figure 2(a), if we would have given full rate to one flow at each ingress port in the first time step and to the other flow in the other time step, the completion time of coflow 1 would not been unchanged (and so would have the overall average CCT). Figure 3 shows this for both Varys and optimal.

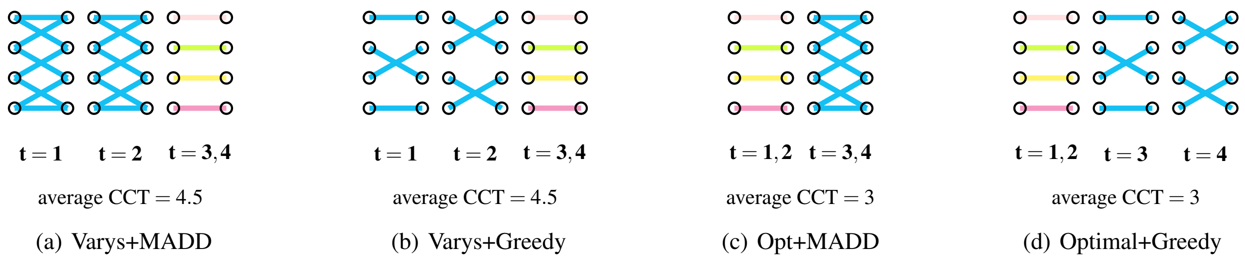


Figure 3: Intuition behind our results in §3.2 using the example of Figure 1. We use two per-flow rate allocation mechanisms in this example. The first one is weighted fair sharing proposed in Varys [4] (in this example, it simply allocates equal rates to all flows at any ingress or egress port). The second one is a greedy rate allocation mechanism that simply chooses one flow from the currently highest ordered coflow at each port, and assigns it the full rate (see Algorithm 2 in §4). The example shows that irrespective of the per-flow rate allocation mechanism used, both Varys and optimal achieve the same average CCT. We do not show Sincronia in this example because MADD ends up allocating non-equal rates to flows in the fist coflow and it is hard to depict it pictorially.

Formal statement of results. We now formally state the result regarding the irrelevance of per-flow rate allocation. The proofs for these results are multi-page long, and are provided in an anonymized technical report [27]. We start with some definitions.

Definition 1 *Let $\sigma : [n] \mapsto [n]$ be an ordering of coflows. A flow scheduling mechanism M is said to be **σ-order-preserving** if M blocks a flow f from coflow $\sigma(k)$ if and only if either its ingress port or its egress port is busy serving a flow from a coflow $\sigma(i)$, $i < k$ (preemption is allowed).*

We now state our main result:

Theorem 1 *When all coflows arrive at time 0, let $\mathscr{O}$ be the ordering of coflows produced by the Bottleneck-Select-Scale-Iterate algorithm and consider **any** work-conserving, pre-emptive and $\mathscr{O}$-order-preserving flow rate allocation scheme used in Sincronia. Then, under the big switch model, Sincronia achieves average coflow completion time within $4\times$ of optimal average coflow completion time.*

The full proof for the theorem is available in the anonymized technical report [27]. We give a high-level idea of the proof in §8. Next, we note that if work conservation is desired, preemption is necessary to achieve bounded average coflow completion time. In particular, we prove that:

Claim 1 *The average coflow completion time using any work-conserving, **non**-preemptive flow rate allocation scheme can be arbitrarily worse than the optimal average coflow completion time.*

Definition 2 *Let σ be an ordering of coflows. For any coflow $\sigma(k)$, let the **ordered load** for coflow $\sigma(k)$ on port p with respect to σ be: $\sum_i d^p_{\sigma(i)}$. Furthermore, let $\hat{p}(k)$ be the port with highest ordered load for $\sigma(k)$. That is*

$$\hat{p}(k) \leftarrow \arg\max_p \sum_{i=1}^{k} d^p_{\sigma(i)}$$

Definition 3 *Let σ be an ordering of coflows. Let $\mathscr{A}(\sigma)$ be the class of flow scheduling algorithms for which the completion time of each coflow $\sigma(k)$ is no earlier than its ordered load at port $\hat{p}(k)$.*

Note that $\mathscr{A}(\sigma)$ incorporates a large class of flow scheduling mechanisms — the only condition is that the last bit of coflow $\sigma(k)$ is sent no earlier than the ordered load at port $\hat{p}(k)$. Let $\texttt{OPT}(\mathscr{A}(\sigma))$ be the optimal average coflow completion time for the set of coflows across all flow scheduling mechanisms in $\mathscr{A}(\sigma)$.

Theorem 2 *Suppose all coflows arrive at time 0 and let σ be an ordering of coflows. Then **any** flow rate allocation scheme that is work-conserving, is pre-emptive and is σ-order-preserving achieves average coflow completion time within $2\times$ of $\texttt{OPT}(\mathscr{A}(\sigma))$.*

Again, the proof is available in [27]. Within the report, we also prove that this bound is tight for the class of work conserving and preemptive flow scheduling mechanisms.

4. SINCRONIA IMPLEMENTATION

In this section, we provide details on Sincronia implementation. We start with a description of the end-to-end system (§4.1). The remainder of the section focuses on three important aspects of the Sincronia implementation. First, the BSSI algorithm from previous section assumes that all coflows arrive at time 0. In §4.2, we discuss how Sincronia system incorporates the BSSI algorithm into an online algorithm, where coflows may arrive at arbitrary times, without giving up on theoretical guarantees on average CCT (although, as with all online algorithms, Sincronia guarantees are now in terms of competitive ratio). Second, we showed in previous section that Sincronia decouples coflow ordering from rate allocation to and scheduling of individual coflows. In §4.3, we discuss how this result allows Sincronia to be integrated with several existing datacenter transport mechanisms without any hardware modifications (in addition to what are required by these transport mechanisms), including TCP, NDP and pFabric. Finally, we discuss how Sincronia implements work conservation (§4.4), enables co-existence of flows and coflows and resolves various other practical issues (§4.5).

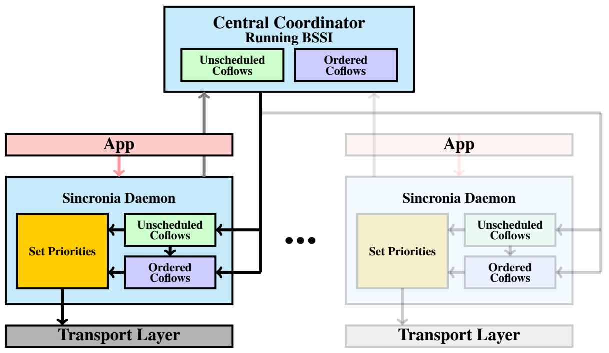


Figure 4: Sincronia end-to-end architecture. See §4.1 for description of various components.

4.1 End-to-end system design

We have implemented Sincronia in C++ using just 2000 lines of code. This includes a central coordinator and at each server, a shim layer that sits between the application and the transport layers (see Figure 4).

Applications, upon arrival of a coflow, inform the Sincronia daemon about the coflow (coflow ID, flows, and corresponding sources, destinations, sizes, etc.); the daemon adds this coflow to the list of "unordered coflows" (to be used for work conservation) and then uses the above information to `register` the (co)flow to the coordinator. The daemons also maintain a list of "ordered coflows", ones that have been assigned an ordering by the coordinator (as discussed below). When the ongoing flow finishes, the daemon picks one flow from the currently highest ordered coflow (if one exists) or from list of unordered coflows (if no ordered coflow exists), assigns the flow an appropriate priority, and sends it to the underlying transport layer. The daemon also `unregisters` the finished (co)flow from the coordinator. The priorities assigned to both ordered and unordered coflows depend on the underlying transport mechanism and are discussed in more depth in §4.3.

The coordinator performs the following tasks. It divides the time into epochs. At the starting of each epoch, the coordinator selects a subset of registered coflows (that have not finished yet) and uses the offline algorithm to order these coflows. We discuss several strategies for deciding the epoch size and for selecting the subset of coflows at the starting of epoch in §4.2. Once computed, the coordinator updates this "ordered coflow" list to all the servers that have unfinished coflows; we use several optimizations here such that the coordinator only informs the servers of the "delta", changes in ordered list, rather than resending the entire ordered list. In addition to this list of ordered coflows, the coordinator also maintains an ordered list of "unscheduled coflows", those that have been registered but not yet ordered (since ordering is done only at at the start of each epoch). The coordinator also periodically sends the unscheduled coflow list to the servers that have unfinished coflows.

As discussed in §2, there is a strong lower bound of $\Omega(\sqrt{n})$ on achievable approximation of average CCT using algorithms that do not use any centralization [7]; thus, some centralization is necessary. We note, however, that Sincronia requires minimal coordination, much lower than any other known mechanism — all of the existing network designs for coflows require the central coordinator to perform per-flow rate allocation, which may change upon each coflow arrival.

We emphasize that each server is oblivious to the epochs maintained at the coordinator (so, servers and the coordinator do not need to be synchronized).

4.2 From Offline to Online

We now discuss how Sincronia implementation incorporates the BSSI algorithm into a system that can efficiently handle the online case, where coflows may arrive at arbitrary times. An obvious way to incorporate BSSI algorithm into an online design is to use the approach taken by most heuristics proposed in prior works [4, 5] — run the offline algorithm upon each coflow arrival. However, in a large datacenter network, this may lead to high complexity since hundreds or thousands of coflows may arrive within each second (although we do use this algorithm as a baseline in our simulations and show that, if time taken to compute the ordering is zero, this algorithm performs extremely well).

Sincronia avoids this high complexity approach using a recently proposed framework [12] along with its own BSSI algorithm from the previous section. We provide a high-level description of the framework to keep the paper relatively self-contained. The framework works in three steps. First, time horizon is divided into exponentially increasing sized epochs. At the starting of each epoch, the framework runs an approximation algorithm to select a subset of unfinished coflows; this approximation algorithm is based on the standard "rounding techniques" over a very simple linear program. Once the subset of coflows is selected, any α-approximate offline algorithm can be used to order coflows arriving over time while providing $(8+\alpha)$-competitive ratio.

Sincronia implementation of this framework uses the BSSI algorithm (in the last step) to order coflows arriving in an online manner; we set smallest epoch size to be 100ms, with every subsequent epoch being $2\times$ larger. However, Sincronia makes two modifications in its implementation of the framework. The first modification is to avoid performance degradation due to large epoch sizes. Specifically, if epoch sizes grow arbitrarily large, increasingly larger number of coflows arrive within an epoch and have to wait until the starting of the next epoch to be scheduled (despite work conservation, as discussed in §4.4); if some of these coflows are small, they observe poor performance. Sincronia thus bounds the maximum epoch size and once this maximum size is reached, it "resets the time horizon". The second modification Sincronia implementation makes in using this framework is to incorporate a work conservation step; we describe this in more detail in §4.4. Since BSSI algorithm provides 4-approximation guarantees, using the above formula, Sincronia system achieves a competitive ratio of 12 for the online case.

4.3 Sincronia + Existing Transport Layers

We now discuss Sincronia implementation on top of several existing transport layer mechanisms for flows. We already described the Sincronia coordinator and daemon functionalities in §4.1; these remain unchanged across integrations with all transport layer mechanisms. The only thing that differs across implementations is how Sincronia daemons set the priorities for individual flows before offloading the flows to the underlying transport mechanism. We describe these priority-setting mechanisms for individual transport layer mechanisms below.

Sincronia + TCP. If the underlying network fabric supports infinite priorities, implementing Sincronia on top of TCP is straightforward — each server daemon, for any given flow, simply assigns it a priority equal to the order of the corresponding coflow, sets the priority using the priority bits in DiffServ [17], and sends the data over TCP. However, in practice, the underlying fabric may only support a small fixed number of priorities due to hardware limitations, or due to use of some priority levels for other purposes (*e.g.*, fault tolerance).

With finite number p of priorities, Sincronia implementation on top of TCP (with DiffServ) approximates the ideal Sincronia performance using a simple modification: the server daemon, when the time comes to assign a priority for any given flow, sets the priority of the flow to be the order of the corresponding coflow if the current order of the coflow is less than $p-1$, else it assigns priority p to the flow. As the list of active coflows is updated, the priority used for a flow is also updated accordingly. Note that priorities are updated upon each coflow departure and not upon each flow departure since there may be other flows in the coflow that may otherwise interfere at some switch in the network. When using unordered coflows for work conservation, the daemon also sets priority p. While not ideal, our experimental results in §5 show that Sincronia still achieves significant improvements in average CCT even with small number of priority levels; we also provide intuition for this result when discussing the implementation results.

Sincronia + pFabric. Sincronia implementation on top of pFabric admits an even simpler design. Since pFabric already supports infinite priority levels, we only need a minor change in pFabric priority assignment mechanism: each flow is now assigned a priority equal to the ordering of its coflow, rather than the size of the flow as in original pFabric paper [14]. A special "minimum priority level" is used for work conservation purposes using unordered coflows. Since Sincronia generates a total ordering across coflows, this implementation results in ideal Sincronia performance.

Sincronia + NDP. Finally, we discuss the implementation of Sincronia on top of recently proposed NDP mechanism.

Algorithm 2 Greedy Rate Allocation Algorithm

```
$\mathbb{C} = \sigma$ is the input coflow permutation
procedure GREEDYFLOWSCHEDULING($\sigma$)
    while $\mathbb{C}$ is not empty do
        for $i = 1$ to $|\mathbb{C}|$ do
            for $j = 1$ to $|C_i.flows|$ do
                if Ingress port of $C_i.flows(j)$ free then
                    if Egress port of $C_i.flows(j)$ free then
                        allocate entire BW to $C_i.flows(j)$
        Update flow sizes and link bandwidths
        GreedyFlowScheduling($\mathbb{C}$)
    return
```

This is particularly interesting because NDP handles incast and outcast traffic patterns in an elegant manner, precisely a use case for coflows. We extend the NDP implementation to support special prioritized scheduling required for Sincronia implementation (at the receiver as well as at the source and at intermediate network elements). We assume familiarity with NDP design; note that while the NDP design works without priorities, it does support assigning and using PULL packets in a prioritized manner. In our implementation, the NDP receiver sends a PULL packet to flows in order of corresponding coflow ordering. The server daemons, among all the received PULL packets, use the one for a flow in the currently highest ordered coflow. In-network priorities are not necessary (congestion is at the edge due to per-packet scheduling and packet spraying) but can be supported using TCP style priority assignment. Interestingly, NDP implementation of Sincronia converges to the greedy algorithm shown in Algorithm 2 that, in a converged state, assigns a single flow the full outgoing access link rate at any given point of time.

4.4 Prioritized Work Conservation

Sincronia coordinator runs the BSSI algorithm for coflow ordering at the starting of each epoch. Thus, coflows that arrive in the middle of an epoch (referred to as "orphan coflows") may not be assigned an ordering until the starting of the next epoch. While our flow scheduling mechanisms from the previous section are naturally work-conserving (because underlying transport layer mechanisms are work conserving), our preliminary implementation highlighted a potential performance issue. Orphan coflows, since unordered, end up fair sharing the bandwidth. For "short" orphan coflows, this could lead to a long tail — at an extreme case, consider an orphan coflow that has only two parallel flows, each transmitting unit amount of data and suppose the sources and the destinations for these flows are not occupied. Then, while this orphan coflow can be finished in unit amount of time, it ends up taking a lot longer since it is now sharing bandwidth with all other orphan coflows at the same port.

Sincronia uses a simple optimization for work conservation that alleviates this problem. Each orphan coflow is assigned an ordering among all the orphan coflows based on its "oracle completion time (OCT)", which is defined as the time the coflow would take to finish if it were the only coflow in the network. Note that the OCT of a coflow c is simply the maximum over all ports p, d_c^p. Specifically, the orphan coflows are chosen for work conservation in increasing order (smaller first) of the following metric:

$$\frac{\text{OCT}}{\text{current\_time + max\_epoch\_length - arrival\_time}}$$

The reason this metric is interesting for work conservation is that it finds the right balance between coflows that are "small" (that is, have small OCTs) and coflows that are large but have been waiting for a long while (that is, coflows that are being starved) while selecting coflows to use for work conservation. For instance, consider a time instant when there are three coflows in the system; two of these coflows arrived at time 20 and have OCTs equal to 1 and 10, respectively, and the third coflow arrived at time 0 and has OCT equal to 25. Suppose the max_epoch_length is 4. Then, when choosing a coflow for work conservation at time 21, the Sincronia daemon will choose the coflow with OCT equal to 1 first (the metric value being $1/5$), then choose the coflow with OCT equal to 25 that has been waiting for too long (the metric value being 1) and then finally send the coflow with OCT equal to 10. Thus, Sincronia is able to find the right balance between small coflows and starving coflows to choose from for work conservation purposes.

4.5 Other Practical Considerations

Finally, we discuss a few other techniques incorporated within Sincronia implementation to handle practical issues that may arise in real-world scenarios.

Co-existence of flows and coflows. In real-world datacenters, multiple applications co-exist; some of these applications may require the network fabric to optimize for coflow-based metrics while others may care about the performance of each individual flow. Prior results on network design for coflows enable coexistence of such applications by treating each individual flow as a coflow. However, this is quite restrictive since flow based applications may have different performance goals compared to coflow based applications. Sincronia partially handles such scenarios using coflow weights.

Specifically, while our discussion so far has focused on Sincronia design and implementation for achieving near-optimal performance in terms of average CCT, Sincronia achieves something much more general — it optimizes for "weighted" average CCT, as defined in §2. That is, it allows network operators to set a weight for each individual coflow. Network operators can use different weights for different applications (*e.g.*, those that require optimizing for flows and those that require optimizing for coflows), and the BSSI algorithm computes ordering of coflows that optimize for the weighted average CCT. Finding the right mechanism to set weights depends on the applications and is beyond the scope of this paper.

Achieving other performance objectives. Sincronia also allows achieving several other performance objectives using the coflow weights. For instance, there has been quite a bit of work in the community on deadline-aware scheduling for the traditional abstraction of network flows [14, 28, 29]. It is easy to show that assigning coflow weights inversely proportional to the deadlines results in Sincronia approximating Earliest Deadline First for coflows. Sincronia also supports admission control by assigning zero weights to coflows which would definitely miss their deadlines, hence scheduling them after other coflows. Finally, a natural use of coflow weights is to enable prioritization among various applications that require using the abstraction of network coflows.

Starvation Freedom. Minimizing average completion time necessitates starvation [4,14]. However, as discussed in §4.4, an interesting aspect of the prioritized work conservation mechanism used in Sincronia is that it alleviates starvation to some extent. Intuitively, since the choice of an unscheduled coflow for work conservation depends inversely on the "waiting time" of the coflow (difference between current time and the arrival time), as the waiting time of the coflow increases, its chances of being selected for work conservation purposes also improve.

5. EVALUATION

We now present evaluation results for Sincronia. We evaluate Sincronia along three dimensions using the set up described in §5.1. First, we evaluate Sincronia using simulations (§5.2); the main goal here is to understand the performance of Sincronia against the state-of-the-art [4], to understand the envelope of workloads for which Sincronia does and does not perform well and to perform sensitivity of Sincronia performance against a variety of workloads, varying number of coflows, network load, epoch sizes, and various underlying transport mechanisms[2]. We then evaluate the performance of our Sincronia implementation with TCP/DiffServ on a 16-server testbed interconnected with a FatTree topology comprising 20 commodity switches (§5.3) and compare it against coflow agnostic TCP based implementation[3]. The goal is to understand the performance of Sincronia on top of TCP (with DiffServ) over real-world topologies with varying loads and oversubscription ratios.

[2]For instance, it is hard to evaluate the performance of Sincronia with NDP and pFabric on our experimental testbed because these transport mechanisms require specialized switches.

[3]Despite trying for several weeks, we were not able to run the only available network design for coflows (Varys [4]) over our topology; this code is not maintained and the software stack for which this code was written has evolved. Nevertheless, we would like to thank the authors of Varys [4] to share their implementation with us and helping us to try and run their code.

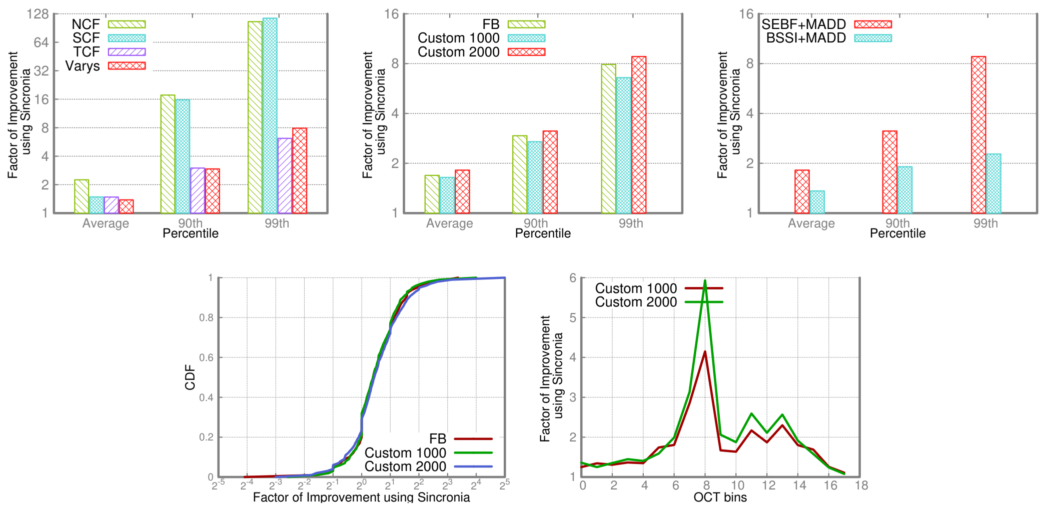


Figure 5: Comparison of Sincronia against Varys (and other heuristics) for Facebook traces (§5.1). (top left) For the original FB trace, Sincronia significantly improves CCT, both at average and high percentiles, when compared to existing heuristics for coflow scheduling; (top center) For different FB traces, Sincronia improves upon Varys both at average and high percentiles; (top right) Sincronia, when used with NDP, achieves better performance than either SEBF [4] or BSSI (§3) with per-flow rate allocation mechanism from [4]; (bottom left) the entire distribution of CCTs corresponding to results in top center; (bottom right) Sincronia improves upon Varys for coflows whose oracle completion time is neither too large nor too small; for coflows with extreme OCTs, both Sincronia and Varys make similar scheduling decisions.

Our evaluation results suggest that:

- Sincronia significantly improves upon state-of-the-art (*e.g.*, Varys [4]) across all evaluated workloads. In our simulations for the offline algorithms, when compared to Varys (which uses SEBF algorithm for ordering and MADD for per-flow rate allocation), Sincronia when used with NDP improves the CCT by $1.38\times$ on an average, by $2.93\times$ at 90 percentile and by $7.91\times$ at 99 percentile.
- Sincronia's end-to-end system handles online arrival of coflows very well, performing within $1.39\times$ of the oracle completion time on an average, within $2.57\times$ at 90 percentile and within $4.26\times$ at 99 percentiles. Note that these numbers are against oracle completion times and are thus, against absolutely best possible scenario (when there is only one coflow in the network).
- Sincronia's implementation with TCP running on our testbed improves the CCT when compared to a coflow agnostic TCP implementation by $18.61\times$ on an average, by $46.95\times$ at 90 percentiles, and $149.05\times$ at 99 percentiles. On the one hand, these results are a bit unfair — TCP is neither designed for coflows nor for minimizing average completion times. However, since we are unable to run existing network designs for coflows, these results give us some indication; for instance, Varys [4] reports to improve upon TCP by a factor of $1.85\times$ on an average, and roughly by the same number at 95 percentiles.

5.1 Setup

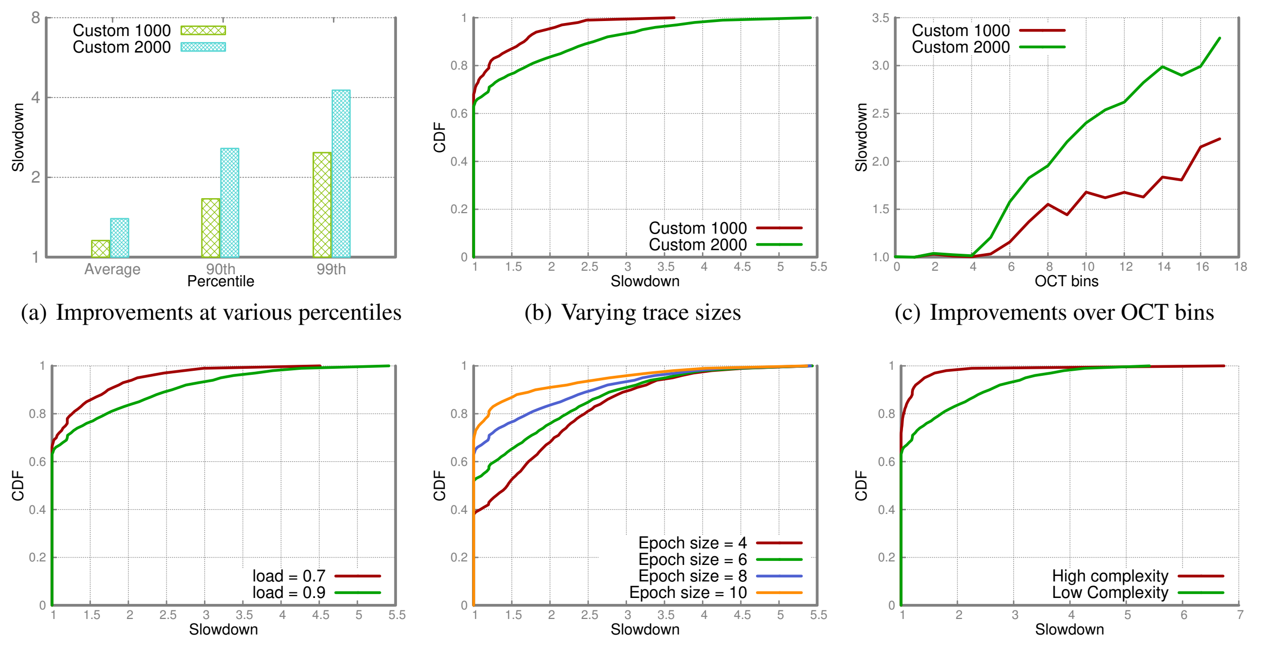
We now describe our simulation setup including the workloads and performance metrics.

Workload Generation. We primarily use two workloads in our evaluation. The first workload is a 526-coflow trace obtained after running MapReduce jobs for one hour on a 3000-machine cluster at Facebook, the one used in evaluations for all prior network designs [4]. Unfortunately, the Facebook trace makes some simplifying assumptions and is limited in number of coflows and imposes very low network load. The second workload is from a coflow workload generator [30] that allows us to upsample the Facebook trace to generate much larger workloads with similar characteristics and vary several parameters (*e.g.*, network size, network load, etc.) useful for our evaluations. We are thus able to work on traces that have characteristics similar to the Facebook trace but are much larger (*e.g.*, as many as 2000 coflows with inter-arrival times such that network load is as high as 0.9).

Metrics. Unless mentioned otherwise, we show the performance for Sincronia with NDP [15] (which converges to the greedy rate allocation mechanism in Algorithm 2). We compare the performance in terms of CCT on an average and at high percentiles. For the online algorithm, we compare the performance of Sincronia against the oracle completion times (the best possible result, since as discussed above, OCT is defined as the time taken by a coflow when its the only coflow in the network). Here, the slowdown of a coflow is defined as the ratio if the difference between its CCT and arrival time, and its OCT. That is, slowdown = (CCT - AT)/OCT.

5.2 Simulation Results

We now present simulation results for Sincronia.



(a) Improvements at various percentiles
(b) Varying trace sizes
(c) Improvements over OCT bins
(d) Varying Load
(e) Varying Epoch Size Bound
(f) Varying Work Conservation Heuristics

Figure 6: Comparison of Sincronia online algorithm.

All coflows arrive at time 0

We first evaluate the performance of Sincronia against known network designs for the case when all coflows arrive at time 0. While not a practical scenario, this evaluation allows us to tease out the performance benefits of Sincronia's BSSI algorithm against those used in prior network designs. Figure 5(a) shows the factor of improvements in CCT that Sincronia achieves on an average, and at 90th and 99th percentiles for the original FB trace. Sincronia improves upon Varys by 1.38$\times$ on average and 7.91$\times$ at the 99 percentile. Since Varys performs better than other heuristics, we focus our attention on Varys from here on.

Next, Figure 5(b) shows that Sincronia maintains its performance benefits against Varys across larger number of coflows. The benefits of Sincronia increase with increase in number of coflows; for 2000 coflows, we observe benefits of around 1.8$\times$ on an average and 8$\times$ at high percentiles. Figure 5(c) shows that using BSSI ordering even with MADD [4], improves upon Varys (SEBF and MADD). However, in most cases Sincronia along with NDP beats the performance achieved using the MADD rate allocation mechanism from Varys. Figure 5(d) plots the CDF of factor of improvement for Varys against Sincronia for various traces. We see that roughly, 25% of the coflows observe CCT improvement of more than 2$\times$ and less than 5% coflows are slowed down by Sincronia more than 2$\times$ when compared to Varys. We also observe that as the number of coflows increase, Sincronia performs improves when compared to Varys.

Figure 5(e) provides more insights into what kinds of coflows provide maximum improvement in performance of Sincronia against Varys. The average factor of improvement is measured across coflows binned by their oracle completion times (OCTs), each bin increasing by a factor of 2. For example, an OCT bin i in Figure 5(e) corresponds to the set of coflows having bottleneck size of $\in [2^i, 2^{i+1})$ MBs. We observe that the medium sized coflows have the maximum performance benefits using Sincronia for the offline case. This is as one would expect since for very coflows with extreme OCTs, Sincronia and Varys would make very similar decisions (the bottleneck, select and iterate steps of two algorithms are very similar). It is precisely the medium size coflows where the weight scaling technique in BSSI starts playing an important role.

Online arrival of coflows

We next evaluate the performance of Sincronia's online scheduler (Figure 6). The first figure shows the average (99 percentile) slowdown using Sincronia for Custom 1000 and 2000 workloads is 1.15 (2.48) and 1.40 (4.26) respectively. The second figure shows the CDF of slowdown using Sincronia for custom 1000 and 2000 workloads. Around 65 % coflows finish in their OCT and there is slowdown of around 2.5 (4.5) for 1000 (2000) coflow trace. The third figure shows the slowdown for coflows binned by their OCT (or bottleneck sizes). Coflows with larger bottleneck sizes observe larger slowdown by Sincronia. The fourth the slowdown with varying load values - 0.7 and 0.9 for 2000 coflow trace. Increasing the load increases the tail values - 4.5 for load 0.7 and 5.5 for load 0.9. The next figure shows Sincronia performance for varying epoch sizes for 2000 coflow

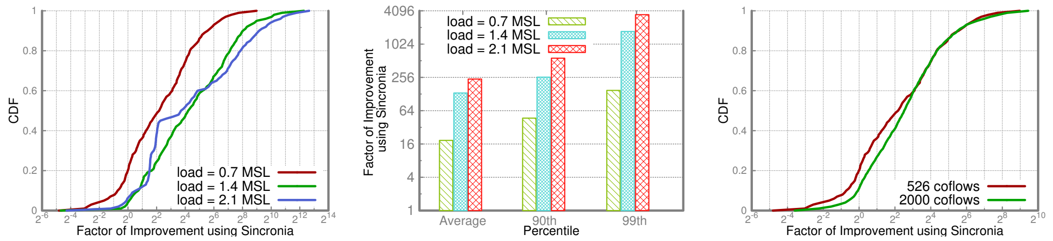


Figure 7: Performance of Sincronia implementation with TCP on our testbed topology.

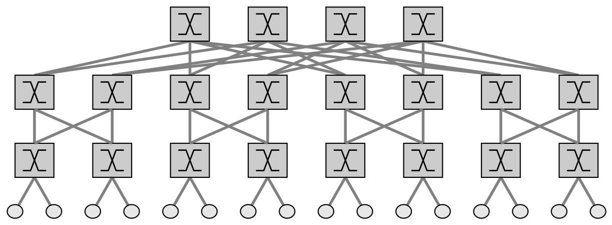


Figure 8: Testbed topology used in our experiments.

trace at load 0.9. Roughly 40% coflows finish at their OCTs for epoch size = 4 and 75% for epoch size = 10. There is no change in tail value - 5.5 slowdown. The final result shows the performance difference in using higher vs lower complexity online algorithm for Sincronia for 2000 coflow trace at load = 0.9. The high complexity version finishes roughly 80% coflows within their OCT and 60% for the low complexity version. The low complexity algorithm has a lower tail value for slowdown - 5.3 vs 6.7 for the high complexity algorithm.

5.3 Implementation Results

Cluster. The cluster used for experiments consists of 16 machines connected via a Fat-tree topology with 1Gbps of access link bandwidth. Figure 7(a) shows the CDF of factor of improvement using Sincronia against TCP for a 526 coflow workload generated to the coflow workload generator for 16 ingress/egress ports, for varying network loads - 0.7, 1.4 and 2.1 $\times$ the Maximum Sustainable Load for the network, defined as network load that can be sustained by the most bottlenecked server (coflows are by definition incast- and outcast-heavy and TCP performed extremely bad when such scenarios happened; thus, we present results for load as a function of MSL). Around 60 % coflows have ≥ 2 factor for improvement for load = 0.7 MSL and $\leq$ 10% have slowdown more than 2 $\times$ using Sincronia. For larger loads, the respective percentages are around 90% and less than 1% respectively. Figure 7(b) shows the percentiles of factor of improvements for various load values for the 526 coflow workload. Sincronia improves by around 18 $\times$ (149 $\times$) on average (99 percentile) for 0.7 MSL. The improvement increases to 239 $\times$ (3488 $\times$) for load = 2.1 MSL. Figure 7(c) shows the CDF plot for factor of improvement for increasing coflow workload at load = 0.7 MSL. The performance is fairly independent upon the number of coflows.

6. OPEN PROBLEMS

This section discusses a number of problems that remain open in the context of near-optimal network design for coflows.

Necessity of centralized solutions. Second, similar to [4, 5, 8, 31, 32], Sincronia requires a centralized controller to implement BSSI. As discussed in §2, there is a strong lower bound of $\Omega(\sqrt{n})$ on achievable approximation of average CCT using algorithms that do not use any centralization [7]. However, the "amount of centralization" required to mitigate this lower bound is unclear. While the amount of computation required by Sincronia's centralized scheduler is much lower than that of prior solutions (since Sincronia scheduler does not need to do per-flow rate allocation), it remains an open problem to decide the minimum amount of computation needed to be done at the centralized controller.

Extensions to Non-Clairvoyant scheduler. Sincronia, similar to [4,5,8,31], assumes that the information about a coflow is available at the arrival time (and no earlier). For many applications, this is indeed the case(see [4] and the discussion therein). Moreover, recent work has shown that it is possible to identify Coflows and their properties within reasonable estimate for many applications [32]. However, designing a non-clairvoyant scheduler based on the many ideas developed in Sincronia is an interesting future direction.

Resolving all the above problems is beyond the scope of a single paper. However, we believe that many of the insights developed in Sincronia may be useful in making progress on the above set of open problems.

7. RELATED WORK

Coflow scheduling. Orchestra [33] was the first scheduler which optimized for minimizing transfer completion times instead of job completion times. Varys [4] provided another centralized coflow scheduler implementing various coflow-oriented heuristics like SEBF-based scheduling and MADD for flow rate allocation. Baraat [5] provided a decentralized scheduler for coflows using FIFO with limited multiplexing to avoid head-of-line blocking. And to overcome the limitation of Varys being a clairvoyant scheduler, Aalo [7] provided a coflow scheduler assuming no prior coflow information at its arrival. The performance however degraded for decentralized and non-clairvoyant schedulers compared to their counterparts, which did not provide any performance guarantees beforehand. Moreover, unlike Sincronia, these network designs require a complete overhaul from the existing designs for flow-based scheduling for implementation.

Recent theory results. Recently, we have also had few theoretical results providing constant factor algorithms for coflow scheduling. The first bound of $\frac{67}{3}$ was given by Qiu et al. [9] using a deterministic algorithm and $9+\frac{16\sqrt{2}}{3}$ for the randomized case. The bounds were improved to $\frac{64}{3}$ and $8+16\frac{\sqrt{2}}{3}$ for the case with zero release dates. These were further improved by Ahmadi et al. [10] to 5 and 4 respectively with zero release dates case. Khuller et al [12] presented an online setting with deterministic 12-approximation and randomized 9.78 approximation. Most of these algorithms require solving linear programs for exponentially increasing number of constraints, rendering them unfeasible to implement in practice. Moreover, none of these provide performance obtained by the algorithms for real world traces in the average case scenarios. Sincronia provides the best known bounds achieved by [10, 12], that too using an iterative algorithm which is solvable in polynomial time.

8. PROOF OUTLINE FOR THEOREM 1

We consider the problem of Coflow Scheduling and provide a 4-approximation algorithm.

We use a linear programming relaxation to obtain a lower bound on the optimal value of the average Coflow completion time, and obtain our approximation guarantee by comparing the average CCT of our algorithm to this lower bound. However, our algorithm is purely combinatorial, in that it does not require the solution of an LP.

In our algorithm, we decouple the problem of obtaining a feasible schedule into two parts: we first obtain an ordering of coflows, and then obtain a feasible schedule by using a greedy rate allocation scheme that maintains this order.

To obtain an ordering of coflows, we also relax the problem by ignoring the dependencies between the input and output ports; more specifically, we consider an instance of a concurrent open shop problem where there are $2m$ machines corresponding to the m ingress and the m egress ports, and one job corresponding to each of the n coflows. The processing requirement for job c on machine k is the total load at port k due to coflow c, and the weight of job c is the weight associated with coflow c. Observe that the optimal value of this concurrent open shop instance is a lower bound on the optimal value of the original Coflow Scheduling instance since any feasible solution to the latter can be viewed as a feasible solution for the former with the same objective function value. Next, observe that there is an optimal solution for the concurrent open shop input in which the order of the jobs processed on each machine is the same: given any optimal solution, if we consider the last job on the most heavily loaded machine we can push that job to the end on each other machine while maintaining that none of the job completion times increases. We can repeat this to obtain a solution where the jobs are processed in the same order on each machine without increasing the objective function value. So we have reduced our problem to one of just finding an ordering of coflows, since given a ordering, determining a feasible schedule is straightforward.

We use a primal-dual algorithm to compute an ordering such that the weighted completion time of the jobs is at most twice the optimal value for the concurrent open shop instance; and by our lower bound argument, hence this weighted completion time is at most twice the optimal value for the Coflow Scheduling Problem as well.

For the second part of the problem, we present a greedy rate-allocation scheme that maintains the order returned by the primal-dual algorithm. More specifically, we ensure that a flow from input port i to output port j on a coflow c is scheduled only after all flows between this pair of ports from coflows of a higher priority have completed. So, if we consider the last flow processed for some coflow c, say from input port i to output port j, this property, coupled with the fact that the algorithm is work-conserving, implies that at least one of the ports i or j was busy for at least half the completion time of coflow c with processing flows from coflows with an equal or higher priority than coflow c. Since the total amount of work from higher priority coflows that can be done on these ports is at most the completion time of coflow c in the concurrent open shop instance, we arrive at the conclusion that the completion time of coflow c in our greedy rate-allocation scheme is at most twice the completion time of coflow c in the concurrent open shop instance. This result in fact extends to any rate allocation scheme that is preemptive, work-conserving, and maintains the ordering of the coflows. Combining the above two results directly yields the 4-approximation result.

9. CONCLUSION

We have presented Sincronia, a network design for coflows that provides near-theoretically-optimal performance and can be implemented on top on any Layer-4 mechanism (for flows) that supports priority scheduling. Sincronia achieves this using a key technical result — it is possible to decouple coflow scheduling from rate allocation to and scheduling of individual flows. This allows Sincronia to use a simple greedy mechanism to “order” all unfinished coflows; all flows within and across coflows can then be greedily scheduled using any Layer-4 mechanism that supports priority scheduling (without any per-flow rate allocation mechanism). We show via theoretical proofs, simulations and implementations over two existing transport layer mechanisms (TCP and NDP) that Sincronia is able to achieve near-optimal performance.

10. REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS operating systems review*, vol. 41, pp. 59–72, ACM, 2007.

[3] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pp. 31–36, ACM, 2012.

[4] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 443–454, ACM, 2014.

[5] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 431–442, ACM, 2014.

[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 98–109, ACM, 2011.

[7] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 393–406, ACM, 2015.

[8] A. Jajoo, R. Gandhi, and Y. C. Hu, "Graviton: Twisting space and time to speed-up coflows," *Network*, vol. 1, no. C2, p. C2, 2016.

[9] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pp. 294–303, ACM, 2015.

[10] S. Ahmadi, S. Khuller, M. Purohit, and S. Yang, "On scheduling co-flows," in *IPCO*, 2017.

[11] N. Garg, A. Kumar, and V. Pandit, "Order scheduling models: hardness and algorithms," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 96–107, Springer, 2007.

[12] S. Khuller, J. Li, P. Sturmfels, K. Sun, and P. Venkat, "Select and permute: An improved online framework for scheduling to minimize weighted completion time," *CoRR*, vol. abs/1704.06677, 2017.

[13] S. Im and M. Purohit, "A tight approximation for co-flow scheduling for minimizing total weighted completion time," *arXiv preprint arXiv:1707.04331*, 2017.

[14] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-optimal Datacenter Transport," in *SIGCOMM*, 2013.

[15] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. Moore, G. Antichi, and M. Wojcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *SIGCOMM*, 2017.

[16] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "phost: Distributed near-optimal datacenter transport over commodity network fabric," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, p. 1, ACM, 2015.

[17] "Configuration Guidelines for DiffServ Service Classes." https://tools.ietf.org/html/rfc4594.

[18] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 307–318, 2015.

[19] R. D. Datasets, "A fault-tolerant abstraction for in-memory cluster computing," *Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. NSDI*, 2012.

[20] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[21] "Apache Hive." http://hive.apache.org.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[23] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[24] T. A. Roemer, "A note on the complexity of the concurrent open shop problem," *Journal of Scheduling*, vol. 9, pp. 389–396, Aug 2006.

[25] N. Bansal and S. Khot, "Inapproximability of hypergraph vertex cover and applications to scheduling problems," in *International Colloquium on Automata, Languages, and Programming*, pp. 250–261, Springer, 2010.

[26] S. Sachdeva and R. Saket, "Optimal inapproximability for scheduling problems via structural hardness for hypergraph vertex cover," in *Computational Complexity (CCC), 2013 IEEE Conference on*, pp. 219–229, IEEE, 2013.

[27] "Sincronia TechReport." https://github.com/sincronia-coflows/sincronia.

[28] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing

flows quickly with preemptive scheduling," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 127–138, ACM, 2012.

[29] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 50–61, 2011.

[30] "Coflow Workload Generator." https://github.com/sakshamagarwals/coflow_workload_generator.

[31] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pp. 424–432, IEEE, 2015.

[32] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 160–173, ACM, 2016.

[33] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 98–109, ACM, 2011.

[34] L. A. Wolsey, "Formulating single machine scheduling problems with precedence constraints," in *Economic Decision–Making: Games, Econometrics and Optimisation* (J. J. Gabszewicz, J. Richard, and L. A. Wolsey, eds.), pp. 473 – 484, Amsterdam: North–Holland, 1990.

[35] M. Queyranne, "Structure of a simple scheduling polyhedron," *Mathematical Programming*, vol. 58, no. 1, pp. 263–285, 1993.

[36] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein, "Scheduling to minimize average completion time: Off-line and on-line approximation algorithms," *Mathematics of operations research*, vol. 22, no. 3, pp. 513–544, 1997.

[37] S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein, "Improved scheduling algorithms for minsum criteria," in *International Colloquium on Automata, Languages, and Programming*, pp. 646–657, Springer, 1996.

11. OFFLINE APPROXIMATION ALGORITHM

In this section, we prove that the algorithm described in section 3 is a 5-approximation algorithm for the general case and a 4-approximation algorithm when there are no release dates. We do this by dividing the problem into two parts - obtaining an ordering of the coflows to be scheduled, and a rate allocation algorithm that respects this ordering. We provide algorithms for each part, with a combined analysis. We first obtain an ordering of coflows by formulating a linear programming relaxation, from which we derive a primal-dual algorithm with provable guarantees. We then use a greedy algorithm for rate allocation, while maintaining the order derived from the primal-dual algorithm.

The algorithms are based on a linear program whose constraints are similar to those introduced by Wolsey [34] and Queyranne [35] for the problem of scheduling n jobs on a single machine: $\sum_{j\in S} p_jC_j \geq \frac{1}{2}\left[\left(\sum_{j\in S} p_j\right)^2 + \sum_{j\in S} p_j^2\right]$ for each subset $S \subseteq [n]$, where C_j, $j \in [n]$ are variables corresponding to the job completion time and p_j is the processing time of job j, for $j = 1,\ldots,n$. Although there are an exponential number of constraints, Queyranne [35] describes a polynomial-time separation oracle for the set of constraints.

The linear program we consider is not a relaxation of coflow scheduling(in the sense that a feasible schedule directly corresponds to a feasible LP solution), but we will prove that the optimal value of this LP is a lower bound on the weighted completion time of any feasible schedule. We focus on the total flow demands at each input and output port and introduce notation for the total flow at each ingress/egress port p: given a coflow k and the associated traffic matrix D_k, we define the total flow at port p

$$v_k^p = \begin{cases} \sum_j v_k(p,j), & \text{if } p \in \{1,\ldots,m\}; \\ \sum_i v_k(i,p), & \text{if } p \in \{m+1,\ldots,2m\}. \end{cases}$$

We also define $\rho(D_k) = \max_p\{v_k^p\}$ to be the maximum row or column sum in D_k.

The LP has a variable C_k for the completion time of each coflow k, and a variable C_k^p that corresponds to the time that coflow k is done sending or receiving flows on the ingress/egress port p:

$$\begin{aligned} \min \quad & \sum_{k=1}^{n} w_kC_k && \text{[LP-primal]} \\ \text{s.t.} \quad & C_k \geq C_k^p, \quad \forall p \in [2m], k \in [n]; && (1) \\ & C_k^p \geq r_k, \quad \forall p \in [2m], k \in [n]; && (2) \\ & \sum_{k\in S} v_k^p C_k^p \geq \frac{1}{2}\left[\left(\sum_{k\in S} v_k^p\right)^2 + \sum_{k\in S}\left(v_k^p\right)^2\right], \quad \forall p \in [2m], S \subseteq [n]. && (3) \end{aligned}$$

We prove that the optimal value of LP-primal provides a lower bound for the coflow scheduling problem.

Lemma 1 *Let* OPT *be an optimal schedule for the set of coflows K and let* $(C_k^{\mathsf{OPT}})_{k\in[n]}$ *be the completion times of the coflows in this optimal schedule. Furthermore, let* $C_1^*,\ldots,C_n^*$ *be the optimal values of* $C_1,\ldots,C_n$ *in LP-primal. Then* $\sum_{k=1}^n w_kC_k^* \leq \sum_{k=1}^n w_kC_k^{\mathsf{OPT}}$.

Proof To prove this, we establish a mapping from feasible schedules of Coflow$|r_k|\sum w_kC_k$ to feasible solutions of LP-primal.

Consider a feasible schedule and let $\hat{C}_k$ be the completion time of coflow k, $k = 1,\ldots,n$. Furthermore, assume that the coflows are re-indexed so that $\hat{C}_1 \leq \hat{C}_2 \leq \cdots \leq \hat{C}_n$.

We now construct a feasible solution to LP-primal. For each coflow k, we set $C_k^p = \hat{C}_k$ for each $p \in [2m]$ and set $C_k = \hat{C}_k$ as well. This satisfies constraints (1). Constraints

(2) follow since no coflow can complete before its release time. To see that constraints (3) hold, consider any set $S \subseteq [n]$ and $p \in [2m]$; then, since coflow k finishes after all of the coflows $\{1,\dots,k-1\}$ have finished, it must be true that $\hat{C}_k \geq \sum_{\kappa=1}^{k} v_\kappa^p$. Then,

$$\sum_{k\in S} v_k^p C_k^p = \sum_{k\in S} v_k^p \hat{C}_k \geq \sum_{k\in S} v_k^p \sum_{\kappa\in S, \kappa\leq k} v_\kappa^p$$
$$= \frac{1}{2}\left[\left(\sum_{k\in S} v_k^p\right)^2 + \sum_{k\in S} (v_k^p)^2\right].$$

To complete the proof, we note that the objective value of LP-primal is $\sum_{k=1}^n w_k C_k = \sum_{k=1}^n w_k \hat{C}_k$ and so every feasible schedule can be mapped to a feasible solution to LP-primal with the same value of the objective function. This proves that there exists a feasible solution to LP-primal whose value is equal to $\sum_{k=1}^n w_k C_k^{\mathsf{OPT}}$, and clearly $\sum_{k=1}^n w_k C_k^*$ is at most this value. ■

We now consider the dual of the linear program described above to analyze the combinatorial algorithm used to schedule coflows. We introduce a dual variable α_k^p for each coflow k and each port p, and a variable β_S^p for each port p and each subset of coflows $S \subseteq [n]$, to obtain the following linear program:

$$\max \quad \sum_p \sum_k \gamma_k^p r_k + \frac{1}{2}\sum_S \sum_p \beta_S^p [(\sum_{j\in S} v_k^p)^2 + \sum_{j\in S} (v_k^p)^2]$$
$$\text{s.t.} \quad \sum_p \alpha_k^p \leq w_k, \quad \forall k \in [n]; \tag{4}$$
$$\sum_{S\ni k} \beta_S^p v_k^p \leq \alpha_k^p - \gamma_k^p, \quad \forall p \in [2m], k \in [n]; \tag{5}$$
$$\alpha_k^p \geq 0, \quad \forall p \in [2m], k \in [n];$$
$$\beta_S^p \geq 0, \quad \forall p \in [2m], S \subseteq [n].$$

The primal-dual algorithm is described in Algorithm 3.

For ease of analysis, assume that the coflows are numbered so that $\sigma(k) = k, \forall k \in [n]$. If $(C_k^{\mathsf{OPT}})_{k\in[n]}$ are the optimal completion times given the set of coflows $\{1,\dots,n\}$, then we first prove bounds for the permutation produced by the above algorithm.

Lemma 2 *Algorithm 3 produces a permutation of coflows σ such that*

$$\sum_{k=1}^n w_k (\max_{j\leq k} r_j + 2\sum_{j\leq k} v_j^{b_k}) \leq 5 \sum_k w_k C_k^{\mathsf{OPT}}.$$

Following this, we prove an upper bound on the completion times of the coflows on using a rate allocation scheme that satisfies the properties of being work conserving(no idle time) and preemptive.

Lemma 3 *Let $\hat{C}_1,\dots,\hat{C}_n$ be the completion times of the coflows upon using any work conserving and flow-level preemptive rate allocation algorithm that ensures that it maintains the order returned by the primal-dual algorithm on each set of input-output ports. Then,*

$$\hat{C}_k \leq r_k + 2\sum_{j\leq k} v_j^{b_k}.$$

Since a greedy rate allocation scheme on the permutation

Algorithm 3 Greedy algorithm for scheduling coflows with release dates

```
$J = [n]$                    ▷ Initial set of unscheduled coflows
▷ $v_k(i,j)$ is the size of flow between ports $i$ and $j$ and belonging to coflow $k$ with weight $w_k$
▷ Flow sum at ingress ports for coflow $k$
$v_k^p = \sum_j v_k(i,j) \ \forall p \in \{1,2,\dots,m\}$
▷ Flow sum at egress ports for coflow $k$
$v_k^p = \sum_i v_k(i,j) \ \forall p \in \{m+1,m+2,\dots,2m\}$
procedure CoflowPermutation($J$)
    for $k = n$ to 1 do
        ▷ Find the bottleneck port
        $b_k \leftarrow \arg\max_p \sum_{j\in J} v_j^p$
        ▷ Find the latest release date
        r_max $\leftarrow \max_{j\in J} r_j$
        if r_max$\leq \frac{1}{2}\sum_{j\in J} v_j^{b_k}$ then
            ▷ Select weighted largest coflow to schedule
            $\sigma(k) \leftarrow \arg\min_{j\in J} \frac{w_j - \sum_{\ell>k}\beta_\ell v_j^{b_\ell}}{v_j^{b_k}}$
            $\beta_k \leftarrow \frac{w_{\sigma(k)} - \sum_{\ell>k}\beta_\ell v_{\sigma(k)}^{b_\ell}}{v_{\sigma(k)}^{b_k}}$
        else            ▷ Coflow with large release date exists
            ▷ Schedule latest released coflow
            $\sigma(k) \leftarrow \arg\max_{j\in J} r_j$
            $\gamma_{\sigma(k)} = w_{\sigma(k)} - \sum_{\ell>k}\beta_\ell v_{\sigma(k)}^{b_\ell}$
        ▷ Update set of unscheduled jobs
        $J \leftarrow J\backslash\sigma(k)$
    return $\sigma$                ▷ Output the coflow permutation
```

order given by the first algorithm satisfies the conditions of the Lemma 3, it is easy to see that by combining Lemma 2 and Lemma 3, we obtain Theorem 1.

Proof Lemma 2 We analyze the primal-dual algorithm by constructing a feasible dual solution and comparing the objective function value to the quantity of interest on the left hand side, $\sum_{k=1}^n w_k(\max_{j\leq k} r_j + 2\sum_{j\leq k} v_j^{b_k})$.

First set the new dual variables

$$\gamma_k^p = \begin{cases} \gamma_k, & \text{if } p = b_k; \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Note that the γ_k^p variables are non-zero if the **else** condition in the algorithm is triggered, and 0 otherwise. Furthermore, in the case that there is a coflow with a large release date, the variable corresponding to it is non-zero only for the port that has the most load at time of scheduling.

We set the dual variables

$$\alpha_k^p = \begin{cases} \sum_{\ell>k|b_\ell=p} \beta_\ell v_k^{b_\ell}, & \text{if } p \neq b_k; \\ w_k - \sum_{\ell>k|b_\ell\neq b_k} \beta_\ell v_k^{b_\ell}, & \text{if } p = b_k. \end{cases} \tag{7}$$

We set the variables

$$\beta_S^p = \begin{cases} \beta_k, & \text{if } S = [k] \text{ and } p = b_k; \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

These variables are now non-zero only if S represents the set of unscheduled coflows J at some iteration, and in that iteration, the **if** condition was satisfied in the algorithm, i.e., all the release dates were sufficiently small.

We observe some crucial properties of these variables

Lemma 4 *The dual variables α_k^p, β_S^p, and γ_k^p as defined in (7),(8), and (6), respectively, have the following properties:*

(a). $\alpha_k^p \geq 0, \forall k \in [n], p \in [2m]$;

(b). $\beta_S^p \geq 0, \forall S \subseteq [n], p \in [2m]$;

(c). $\gamma_k^p \geq 0, \forall k \in [n], p \in [2m]$;

(d). the dual constraints (4) are satisfied with equality;

(e). the dual constraints (5) are satisfied with equality;

(f). when γ_k^p is non-zero, $r_k \geq \frac{1}{2}\sum_{j\leq k} v_j^p$ and when $\beta_{[k]}^p$ is non-zero, $\max_{j\leq k} r_j \leq \frac{1}{2}\sum_{j\leq k} v_j^p$.

Proof(a,b). We prove this by induction on the iteration number k. When $k = n$, we have $\alpha_n^{b_n} = w_n \geq 0$ and $\alpha_n^p = 0, \forall p \neq b_n$. Furthermore, $\beta_{[n]}^{b_n} = \beta_n = \frac{w_n}{v_n^{b_n}} \geq 0$ and $\beta_{[n]}^p = 0, \forall p \neq b_n$.

When $k < n$, it is clear that $\alpha_k^p \geq 0, \forall p \neq b_k$, since by induction we know that $\beta_\ell \geq 0, \forall \ell > k$. We also have $\beta_{[k]}^p = 0, \forall p \neq b_k$. Now for the remaining variables $\alpha_k^{b_k}$ and $\beta_{[k]}^{b_k}(=\beta_k)$, we know that coflow $k+1$ was chosen as the minimizer of the modified weight-to-processing time ratio at iteration $k+1$, which gives us

$$\frac{w_k - \sum_{\ell=k+2}^n \beta_\ell v_k^{b_l}}{v_k^{b_{k+1}}} \geq \frac{w_{k+1} - \sum_{\ell=k+2}^n \beta_\ell v_{k+1}^{b_\ell}}{v_{k+1}^{b_{k+1}}}.$$

Noting that the right hand side is just β_{k+1} and rearranging yields $w_k - \sum_{\ell=k+1}^n \beta_\ell v_k^{b_\ell} \geq 0$. Dividing this by $v_k^{b_k}$ leads to $\beta_{[k]}^{b_k} \geq 0$, and from the induction hypotheses, since $\beta_\ell \geq 0, \forall \ell > k$, dropping terms from the summation only increases its value, and therefore $\alpha_k^{b_k} = w_k - \sum_{\ell>k|b_\ell\neq b_k} \beta_\ell v_k^{b_\ell} \geq w_k - \sum_{\ell=k+1}^n \beta_\ell v_k^{b_\ell} \geq 0$.

(c). This follows from the fact that $w_k \geq \sum_{l>k} \beta_l v_k^{p_l}$, as noted in the proof of part (a,b).

(d). The definition of α_k^p immediately yields the result $\sum_p \alpha_k^p = w_k, \forall k$.

(e). Recall that β_S^p is non-zero only when $S = [k]$ and $p = b_k$, for each $k \in [n]$; furthermore, $\gamma_k^p = 0$ when $p \neq b_k$. Therefore, whne $p \neq b_k$, we have $\gamma_k^p + \sum_{S\ni k} \beta_S^p v_k^p = \sum_{\ell>k|b_\ell=p} \beta_\ell v_k^{b_\ell} = \alpha_k^p$ by definition. If $p = b_k$ and $\gamma_k = 0$, then we have,

$$\begin{aligned}\sum_{S\ni k} \beta_S^{b_k} v_k^{b_k} &= \beta_k v_k^{b_k} + \sum_{\ell>k|b_\ell=b_k} \beta_\ell v_k^{b_\ell} \\ &= w_k - \sum_{\ell>k} \beta_\ell v_k^{b_\ell} + \sum_{\ell>k|b_\ell=b_k} \beta_\ell v_k^{b_\ell} \\ &= w_k - \sum_{\ell>k|b_\ell\neq b_k} \beta_\ell v_k^{b_\ell} \\ &= \alpha_k^{b_k},\end{aligned}$$

where the second equality follows from the definition of β_k.

For the remaining case, we look at port $p = p_k$ and assume that in iteration k, we have $\gamma_k \neq 0$. Then we know that $\beta_k = 0$, and so

$$\begin{aligned}\gamma_k^{b_k} + \sum_{S\ni k} \beta_S^{b_k} v_k^{b_k} &= \gamma_k^{b_k} + \sum_{\ell>k|b_\ell=b_k} \beta_\ell v_k^{b_\ell} \\ &= w_k - \sum_{\ell>k} \beta_\ell v_k^{b_\ell} + \sum_{\ell>k|b_\ell=b_k} \beta_\ell v_k^{b_\ell} \\ &= w_k - \sum_{\ell>k|b_\ell\neq b_k} \beta_\ell v_k^{b_\ell} \\ &= \alpha_k^{b_k},\end{aligned}$$

where the second equality follows from the definition of $\gamma_k(=\gamma_k^{b_k})$.

(f). This is evident from the **if-else** condition in the algorithm. (Note that γ_k^p and $\beta_{[k]}^p$ are zero for all $p \neq b_k$). ■

Now note that

$$\begin{aligned}&\sum_{j=1}^n w_k(\max_{i\leq j} r_i + 2\sum_{i\leq j} v_i^{b_j}) \\ &\overset{Lemma\ 4(d)}{=} \sum_{j=1}^n \Big(\sum_p \alpha_j^p\Big)(\max_{i\leq j} r_i + 2\sum_{i\leq j} v_i^{b_j}) \\ &\overset{Lemma\ 4(e)}{=} \sum_j \Big(\sum_p \gamma_j^p + \sum_p \sum_{S\ni j} \beta_S^p v_j^p\Big)(\max_{i\leq j} r_i + 2\sum_{i\leq j} v_i^{b_j}).\end{aligned} \tag{9}$$

We deal with the two terms separately. First,

$$\begin{aligned}\sum_j \sum_p \gamma_j^p (\max_{i\leq j} r_i + 2\sum_{i\leq j} v_i^{b_j}) &\overset{(a)}{=} \sum_j \sum_p \gamma_j^p \Big(r_j + 2\sum_{i\leq j} v_i^{b_j}\Big) \\ &\overset{(b)}{\leq} 5\sum_j \sum_p \gamma_j^p r_j,\end{aligned} \tag{10}$$

where (a) follows from the fact that when γ_k is non-zero (i.e., the **else** condition in the algorithm is triggered), the coflow scheduled in that iteration is the one with the maximum release date amongst the set of coflows yet to be scheduled, and (b) follows from Lemma 4(f).

Now looking at the second term,

$$\sum_j \Big(\sum_p \sum_{S \ni j} \beta_S^p v_j^p\Big)\Big(\max_{i \le j} r_i + 2\sum_{i \le j} v_i^{b_j}\Big)$$
$$= \sum_j \Big(\sum_{k \ge j} \beta_{[k]}^{b_k} v_j^{b_k}\Big)\Big(\max_{i \le j} r_i + 2\sum_{i \le j} v_i^{b_j}\Big)$$
$$= \sum_k \beta_{[k]}^{b_k} \sum_{j \le k} v_j^{b_k}\Big(\max_{i \le j} r_i + 2\sum_{i \le j} v_i^{b_j}\Big)$$
$$\le \sum_k \beta_{[k]}^{b_k} \sum_{j \le k} v_j^{b_k}\Big(\max_{i \le k} r_i + 2\sum_{i \le k} v_i^{b_k}\Big)$$
$$\overset{(a)}{\le} \frac{5}{2}\sum_k \beta_{[k]}^{b_k} \sum_{j \le k} v_j^{b_k} \sum_{i \le k} v_i^{b_k} = \frac{5}{2}\sum_k \beta_{[k]}^{b_k}\Big(\sum_{j \le k} v_j^{b_k}\Big)^2$$
$$\le \frac{5}{2}\sum_k \beta_{[k]}^{b_k}\Big(\sum_{j \le k}\big[v_k^{b_k}\big]^2 + \Big[\sum_{j \le k} v_k^{b_k}\Big]^2\Big)$$
$$= \frac{5}{2}\sum_p \sum_S \beta_S^p\Big(\sum_{j \le k}\big[v_k^{b_k}\big]^2 + \Big[\sum_{j \le k} v_k^{b_k}\Big]^2\Big), \qquad (11)$$

where (a) follows from Lemma 4(f).

Combining (10) and (11) with (9) and noting from weak duality that

$$\sum_p \sum_k \gamma_k^p r_k + \frac{1}{2}\sum_S \sum_p \beta_s^p[(\sum_{j \in S} v_k^p)^2 + \sum_{j \in S}(v_k^p)^2] \le \sum_k w_k C_k^{\mathsf{OPT}}$$

for any feasible dual solution, we get the desired result. ■

Proof Lemma 3

Consider any rate allocation algorithm $\mathscr{A}$ that is work conserving and follows the priorities established by the primal-dual algorithm. More specifically, given any flow from some input port p_i to output port p_o in a coflow k, no flow (p_i, p_o) from a later coflow $\ell : \ell > k$ is scheduled before all of the packets in this flow for coflow k have completed.

Now consider any coflow k, and let $\hat{C}_k$ be its completion time. Since the release time of this coflow is r_k, the flows of this coflow can only be scheduled after r_k. Since the coflow completes at time $\hat{C}_k$, there exists a packet from port p_i to port p_o that is sent at time $\hat{C}_k$. It follows from work conservation that this occurs because the ports p_i and p_o were never simultaneously free during the time interval $[r_k, \hat{C}_k)$. In fact, because of the strict adherence to the priorities as mentioned above, we can see that the ports p_i and p_o are neither free nor serving a flow from a coflow $\ell > k$ during this time interval(since otherwise, the corresponding flow from coflow k would have been picked by the algorithm).

Given the above statements, it immediately follows that at least one of the ports among p_i and p_o is busy serving flows from the set of coflows $\{1, \ldots, k\}$ for at least $\frac{1}{2}(\hat{C}_k - r_k)$ amount of time in the interval $[r_k, \hat{C}_k)$. Now noting that the total time a port p can be busy serving flows from a set S is $\sum_{j \in S} v_j^p$ and thus we get $\frac{1}{2}(C_k - r_k) \le \max\{\sum_{j \le k} v_j^{p_i}, \sum_{j \le k} v_j^{p_o}\}$. Combining with the fact that b_k was the most heavily loaded port and thus $\max\{\sum_{j \le k} v_j^{p_i}, \sum_{j \le k} v_j^{p_o}\} \le \sum_{j \le k} v_j^{b_k}$ and rearranging completes the proof. ■

We can improve this result for the case when there are no

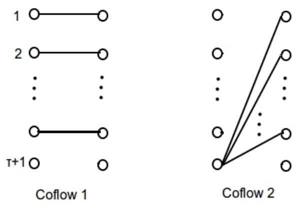


Figure 9: Coflow Instance

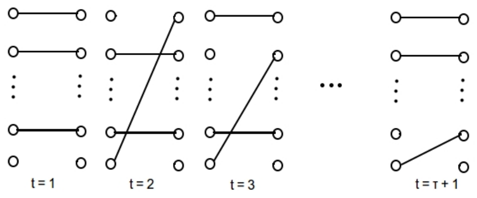


Figure 10: Optimal rate allocation

release dates, leading to a 4-approximation algorithm.

If $(C_k^{\mathsf{OPT}})_{k \in [n]}$ are the optimal completion times given the set of coflows $\{1, \ldots, n\}$, then

Lemma 5 *When there are no release dates, Algorithm 3 produces a permutation of coflows σ such that*

$$\sum_{k=1}^{n} w_k(\sum_{j \le k} v_j^{b_k}) \le 2\sum_k w_k C_k^{\mathsf{OPT}}$$

PROOF. Noting that all the γ_k^p dual variables are 0, the proof of Lemma 2 yields this result.

Combining this with the fact that the completion time of coflow k under the greedy rate allocation scheme is at most $2\sum_{j \le k} v_j^{b_k}$(follows from Lemma 3 by setting $r_k = 0, \forall k$), we obtain the 4-approximation guarantee.

An important point to note is that the only properties we used about the rate allocation scheme was that it was work conserving and that it preserved the order of the coflows returned by the primal-dual algorithm while scheduling. Thus any rate allocation which maintains these two properties provides, together with the primal-dual algorithm to provide an ordering, a 4-approximation algorithm to the coflow scheduling problem.

We now show that the bound given in Theorem 2 is tight. Let $\tau = \lceil \frac{3}{\varepsilon} \rceil$ and consider the coflow instance shown in Figure 9 on $\tau + 1$ ports and the permutation being that coflow 1 has higher priority. Let the weight of the first coflow be $w_1 = 1$ and that of the second coflow be $w_2 = \tau$. Now con-

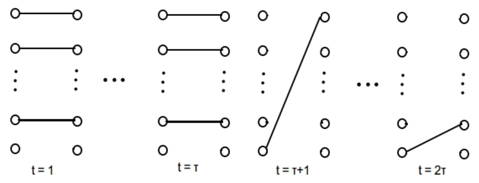
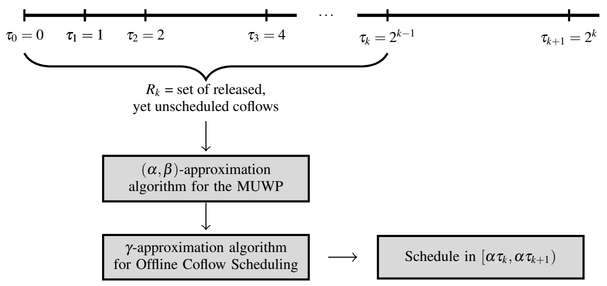


Figure 11: Poor rate allocation



$\tau_0 = 0$
$\tau_1 = 1$
$\tau_2 = 2$
$\tau_3 = 4$
$\tau_k = 2^{k-1}$
$\tau_{k+1} = 2^k$
R_k = set of released, yet unscheduled coflows
(α,β)-approximation algorithm for the MUWP
γ-approximation algorithm for Offline Coflow Scheduling
Schedule in $[\alpha\tau_k, \alpha\tau_{k+1})$

Figure 12: Algorithm to provide a $(2\alpha\beta+\gamma)$-competitive online algorithm, as described in Khuller et al.

sider the optimum rate allocation scheme that is work conserving, pre-emptive and order preserving, shown in Figure 10. The weighted completion time for this scheme is $(1+\tau)^2$. Now consider another rate allocation scheme within the same class, shown in Figure 11. The weighted completion time of this scheme is $\tau + 2\tau^2$. The ratio of this value to the optimal completion time is more than $2-\varepsilon$.

12. ONLINE APPROXIMATION ALGORITHM

For the online setting, we use the generalization of the framework of Hall et al. [36] by Khuller et al. [12] for a 12-competitive online algorithm and a 9.78-competitive randomized algorithm for this problem. Their framework makes use of a γ-approximation algorithm to the offline variant of the same problem and a (α,β)-approximation to the Minimum Unscheduled Weight Problem(MUWP) to obtain a $2\alpha\beta+\gamma$-competitive algorithm.

The MUWP with respect to the scheduling environment of coflows is as follows.

Definition 4 [MINIMUM UNSCHEDULED WEIGHT PROBLEM] *Given a set of coflows available at time* 0*, a weight for each coflow, and a deadline D, find a subset of coflows that can be scheduled to complete by time D while minimizing the weight of unscheduled coflows.*

Definition 5 *An algorithm is an (α,β)-approximation algorithm to the MUWP if it returns a set of coflows such that they can be scheduled to complete by time αD and ensures that the weight of the unscheduled coflows is at most β times the optimum solution to the MUWP with deadline D.*

We next briefly describe the algorithm used. As in Figure 12, divide the time horizon into a sequence of intervals whose lengths increase geometrically; let $\tau_0 = 0$, and for $k \geq 1$, $\tau_k = 2^{k-1}$. For each $k \geq 0$, let R_k be the set of coflows that have been released by τ_k but not yet scheduled. Then at each time τ_k, run the (α,β)-approximation algorithm for MUWP on the set R_k. We then run the γ-approximation algorithm on the resulting set of coflows and schedule them to run in the interval $[\alpha\tau_k, \alpha\tau_{k+1})$. Khuller et al. prove that this algorithm is $(2\alpha\beta+\gamma)$-competitive, with an added term of αW where W is the total weight of the coflows.

Using the $(2,2)$-approximation for the MUWP in the setting of order scheduling of Garg et al. [11] and the 4- approximation for the offline scheduling problem in this paper leads to a 12-competitive algorithm for online coflow scheduling.

The randomized algorithm achieves a better bound by modifying the framework so that the intervals lengths are not chosen deterministically. To be more specific, they take $\tau_k = \eta 2^k$ where $\eta = 2^{-X}$ and X is chosen uniformly from $(0,1]$. If B_j denotes the start of the interval in which coflow j is scheduled, then their framework produces a schedule with the weighted completion time being at most $2\alpha\beta\sum_j w_j B_j + \gamma\sum_j w_j C_j^{\mathrm{OPT}}$ where C_j^{OPT} is the completion time of coflow j in an optimal schedule.

Now using the result by Chakrabarti et al. [37] that $E[B_j] = \frac{1}{2\ln 2} C_j^{\mathrm{OPT}}$ when the intervals are chosen as specified above, and using linearity of expectation, Khuller et al. arrive at an algorithm that is $\left(\frac{1}{\ln 2}\alpha\beta + \gamma\right)$-competitive in expectation. Using the values of α, β and γ as described above leads to a 9.78-competitive randomized algorithm for online coflow scheduling.