Sincronia: Near-Optimal Network Design for Coflows

Saksham Agarwal*
Cornell University

Shijin Rajakrishnan*
Cornell University

Akshay Narayan
MIT

Rachit Agarwal
Cornell University

David Shmoys
Cornell University

Amin Vahdat
Google

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, any per-flow rate allocation mechanism achieves average coflow completion time within 4× of the optimal as long as (co)flows are prioritized with respect to the ordering.

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×).

CCS CONCEPTS

• **Networks** → **Network protocol design**; • **Theory of computation** → *Scheduling algorithms*;

KEYWORDS

Coflow, Datacenter Networks, Approximation Algorithms

*The first two authors contributed equally to the paper.

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230569

ACM Reference Format:
Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-Optimal Network Design for Coflows. In *SIGCOMM '18: ACM SIGCOMM 2018 Conference, August 20–25, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3230543.3230569

1 INTRODUCTION

Traditionally, networks have used the abstraction of a "flow", that captures 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 latency and/or throughput for a point-to-point connection, precisely the performance metrics important to traditional applications (*e.g.*, file transfers, web access, etc.). However, distributed applications running across datacenter networks use programming models (*e.g.*, bulk synchronous programming and partition-aggregate model) 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 [7] 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) [7, 10, 12], 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 [7, 10, 12].

There has been tremendous recent effort on network designs for coflows, both in networking [7–10, 12] and in theory community [3, 19, 22]. However, prior designs require a centralized coordinator to perform complex per-flow rate allocations, with rate allocated to a flow being dependent on the rate allocated to other flows in the network. Such

centralized inter-dependent per-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 or departure of even one coflow may result in reallocation of rate for each and every flow in the network. Such reallocation is impractical in datacenters where 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 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 (co)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 based on corresponding coflow's ordering, and offloads scheduling of and rate allocation to individual flows 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, by avoiding per-flow rate allocation, Sincronia design is independent of underlying network topology, location of congestion in the fabric, and paths taken by each (co)flow. This also allows Sincronia to transparently respond to network failures. Second, coflow arrivals and departures do not require explicit rate reallocation for existing flows, leading to a much more scalable design. Third, Sincronia design using simple priority mechanisms enables coexistence of flows and coflows (§4), supporting backward compatibility. Finally, by being transport-agnostic, Sincronia admits efficient implementation on top of any existing transport mechanism that supports priority scheduling including TCP (using DiffServ [6] for priority scheduling), pHost [13] and pFabric [4]. We discuss implementation of Sincronia on top of several transport mechanisms in §4.3.

We have implemented Sincronia on top of TCP, with DiffServ [6] for priority scheduling. Our implementation is work conserving, efficiently handles online arrival of coflows, and allows coexistence of flows and coflows. We evaluate Sincronia implementation on a 16-server testbed interconnected with a FatTree topology comprising 20 commodity switches. We have also incorporated Sincronia into existing coflow simulators; we use these to perform sensitivity analysis of Sincronia performance against variety of workloads, number of coflows, network load, transport mechanisms, 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×).

2 SINCRONIA OVERVIEW

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

2.1 The Coflow Abstraction

Existing distributed programming frameworks [7, 9, 11, 17, 20, 21, 26, 27] often have a communication stage that is structured and takes place between successive computation stages. In these frameworks, execution of a task (or even an entire computation stage) cannot begin until all flows in the preceding communication stage have finished. A coflow [7, 10] 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 [10, 12, 14, 19], 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.

[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 [4, 13, 15]. For the case when the fabric core cannot sustain 100% throughput, 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.

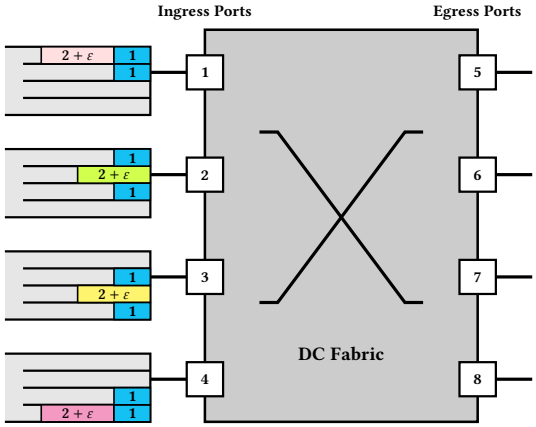


Figure 1: An instance of a coflow scheduling problem, used as a running example in the paper. The datacenter has 4 ingress and egress ports, with each ingress port having a "virtual output queue" for each egress port (see §2.2 for detailed model description). The example has 5 coflows. Coflow C1 has eight flows, with each ingress port sending unit amount of data to two egress ports; coflows C2, C3, C4 and C5 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 C2, C3, C4 and C5 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.

2.2 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 traditional abstraction of network flows [4, 13, 15] and coflows [3, 10, 12, 14, 19, 22], we will abstract out the datacenter network fabric as one big switch that interconnects the servers. In such a big switch model, the ingress queues correspond to the NICs and the egress queues to the last-hop TOR switches. The model assumes that the fabric core can sustain 100% throughput and only the ingress and egress queues are potential congestion points. Under this model, each ingress port has flows from one or more coflows to various egress ports (see Figure 1). For ease of exposition, we organize the flows in virtual output queues at the ingress ports. 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. 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 $\mathbb{F}_c$.

Table 1: Notation used in the paper.

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

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 1 for notation).

The completion time of a coflow (CCT_c) is the difference between the time when the last of its flows finishes and its arrival time a_c. 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 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 §6):

NP-**Hardness [10].** 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 [23]). Thus, the best we can hope for is an approximation algorithm.

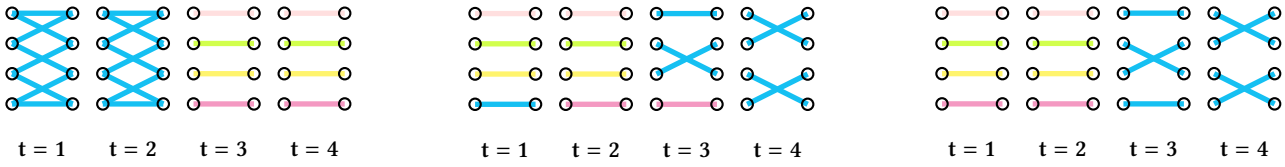
Lower Bounds [5, 24]. Even under the big switch model, the only known lower bound is a natural generalization of the lower bound for flows — under a complexity-theoretic assumption somewhat stronger than P≠NP, it is impossible to minimize (weighted) average CCT within a factor of $2-\varepsilon$.

Necessity for Coordination [8]. 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.

3 SINCRONIA DESIGN

In this section, we present the core of Sincronia design — an offline algorithm for scheduling coflows for the case when all coflows arrive at time 0; next section describes how Sincronia incorporates this 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 order of corresponding coflow ordering achieves average CCT within 4× of the optimal (§3.2).



average CCT = $(2 + 4 \times 4)/5 = 3.6$ (a) Varys

average CCT = $(3 \times 2 + 3 + 4)/5 = 2.6$ (b) Sincronia

average CCT = $(4 \times 2 + 4))/5 = 2.4$ (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 each time slot; a link indicates data being sent between ports in that time slot, with multiple links at a port indicating port bandwidth being equally allocated to each of the links in that time slot. For instance, (a) shows that Varys sends data from first ingress port to the first two egress ports in first two time steps (with each outgoing link getting equal rate allocation), and from first ingress port to first egress port in third and fourth time steps (with the outgoing link getting full rate). The orderings produced by Varys and Sincronia are {C1, C2, C3, C4, C5} and {C2, C3, C4, C1, C5}, respectively; the optimal schedule requires ordering {C2, C3, C4, C5, C1} (modulo permutations within coflows C2, C3, C4 and C5).

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 σ ▷ **Output the coflow ordering**

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) [4], 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. BSSI achieves this using a novel weight scaling step that is derived based on the primal-dual framework.

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 weighted processing time at port b 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.

Table 2: Execution of Algorithm 1 on Figure 1 example. In this example, we break ties in favor of ingress port with the largest index. The final ordering produced by the algorithm is {C2,C3,C4,C1,C5}.

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	C5	$\{\varepsilon/(2+\varepsilon), 1, 1, 1, 0\}$	$\{1, 2, 3, 4\}$
4	3	C1	$\{0, 1, 1, 1 - \varepsilon/2, 0\}$	$\{2, 3, 4\}$
3	3	C4	$\{0, 1, 1, 0, 0\}$	$\{2, 3\}$
2	2	C3	$\{0, 1, 0, 0, 0\}$	$\{2\}$
1	1	C2	$\{0, 0, 0, 0, 0\}$	$\emptyset$

An Example. Table 2 shows execution of Algorithm 1 on example of Figure 1. In first iteration ($k = 5$), the algorithm chooses bottleneck port b=4 and LRPT coflow $\sigma(5)$ = C5. The algorithm then scales the weights — in this example, it ends up reducing the weight of coflow C1 while keeping other weights unchanged. This reduction in weight allows C1 to be selected as the LRPT coflow in next iteration ($k = 4$). Figure 2 compares the performance of Sincronia against Varys [10] and optimal for this example. 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 in the above example [2].

average CCT = 3.6 (a) Varys + MADD

average CCT = 3.6 (b) Varys + Greedy

average CCT = 2.4 (c) Optimal + MADD

average CCT = 2.4 (d) Optimal + Greedy

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 [10] (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 first coflow and it is hard to depict it pictorially.

3.2 Per-Flow Rate Allocation is Irrelevant

As discussed earlier, prior network designs for coflows require a centralized coordinator to perform complex per-flow rate allocation, where rate allocated to a flow is dependent on rates allocated to other flows in the network; for instance, Varys [10] allocates rates to flows in proportion to their sizes. Such centralized inter-dependent per-flow rate allocations make it hard to realize these designs in practice since changes in location of congestion in the network, transient failures, and arrival or departure of even one coflow may result in reallocation of rate for each and every flow in the network. Such reallocations are impractical in large datacenters where thousands of coflows may arrive each second. We discuss how Sincronia completely offloads the rate allocation to 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 C1 would not have changed (and so, the same is true for the overall average CCT). Figure 3 demonstrates the irrelevance of per-flow rate allocation for both Varys and the optimal.

Formal statement of results. We now formally state the result regarding the irrelevance of per-flow rate allocation. The detailed proofs for these results are in the technical report [2]; we give a high-level idea of the proofs in §7.

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).*

Theorem 1. *Consider a set of coflows $\mathbb{C}$, all of which arrive at time 0. Let $\mathcal{O}$ be the ordering of coflows produced by the Bottleneck-Select-Scale-Iterate algorithm for $\mathbb{C}$ and consider **any** work-conserving, pre-emptive and $\mathcal{O}$-order-preserving flow rate allocation scheme used to schedule $\mathbb{C}$ in Sincronia. Then, under the big switch model, Sincronia achieves average coflow completion time within 4× of the optimal average coflow completion time for $\mathbb{C}$.*

It turns out that if work conservation is desired, preemption is necessary to achieve bounded average CCT:

Claim 1. *There exists a set of coflows $\mathbb{C}$ for which 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 for $\mathbb{C}$.*

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: $\sum_{i \le k} d^p_{\sigma(i)}$. 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 $\mathcal{A}(\sigma)$ 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 $\hat{p}(k)$ divided by the bandwidth of port $\hat{p}(k)$.*

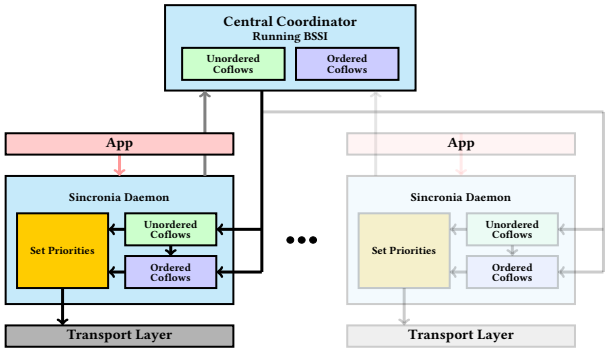


Figure 4: Sincronia end-to-end architecture. See §4.1 for description of various components.

$\mathcal{A}(\sigma)$ is indeed 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 $\texttt{OPT}(\mathcal{A}(\sigma))$ be the optimal average coflow completion time for the set of coflows across all flow scheduling mechanisms in $\mathcal{A}(\sigma)$.

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* $\texttt{OPT}(\mathcal{A}(\sigma))$*. This bound is tight.*

4 SINCRONIA IMPLEMENTATION

We now provide details on the 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 implementation. First, the BSSI algorithm from §3.1 assumes that all coflows arrive at time 0. In §4.2, we discuss how Sincronia implementation incorporates the BSSI algorithm to efficiently handle coflows arriving at arbitrary times. Second, we showed in §3.2 that Sincronia decouples coflow ordering from rate allocation to and scheduling of individual flows. In §4.3, we discuss how this result enables Sincronia to be efficiently integrated with existing datacenter transport mechanisms, including TCP, pHost and pFabric. Finally, we discuss how Sincronia implementation achieves work conservation (§4.4), enables co-existence of flows and coflows, and resolves various other practical issues (§4.5).

4.1 End-to-end system design

We have implemented Sincronia in C++ using just 3000 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 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 coflow 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. The coordinator maintains a list of "unordered coflows", those that have been registered but not yet ordered (ordering is done only at the start of each epoch). At the start of each epoch, the coordinator selects a subset of coflows from the unordered coflow list 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 removes the ordered coflows from the unordered coflow list, and sends the "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. Note that each server is oblivious to epochs maintained at the coordinator; hence, servers are not require to be time synchronized with the coordinator.

As discussed in §2, there is a lower bound of $\Omega(\sqrt{n})$ on achievable approximation for average CCT for mechanisms that do not use coordination [8]; thus, some coordination is necessary. However, Sincronia admits much simpler coordinator that existing network designs for coflows — in contrast to existing designs that require the central coordinator to perform per-flow rate allocation, Sincronia requires it to just order the coflows using its BSSI algorithm.

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 prior works (*e.g.*, Varys [10]) — run the offline algorithm upon each coflow arrival. However, in datacenter networks where thousands of coflows may arrive within each second, this may lead to high complexity (although we do use this algorithm as a baseline in our simulations).

Sincronia avoids this high complexity approach using a recently proposed framework [19] 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 start of each epoch, the framework runs an approximation algorithm to select a subset of unfinished coflows. This approximation algorithm, introduced by Garg et. al. [14], works as follows: formulate an integer program with a decision variable for each coflow to indicate whether (or not) it should be assigned to complete within that epoch; we add constraints to enforce that the total work required for the selected subset does not exceed the amount of data that can be transferred within the epoch, and we aim to maximize the total weight of the coflows assigned to this epoch. We solve the linear programming relaxation of this integer program, and select each coflow that is at least "half-assigned" by this optimal solution. It is easy to understand why this process loses a factor of 8 in the approximation guarantee. 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[2].

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× larger. However, Sincronia makes two modifications in its implementation of the framework. The first modification is to incorporate a work conservation step; we describe this in more detail in §4.4. The second modification Sincronia implementation makes in using this framework is to avoid performance degradation due to large epoch sizes. Indeed, 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); small coflows, in particular, observe poor performance. Sincronia thus bounds the maximum number of epochs and once this number is reached, it "resets the time horizon". Since BSSI algorithm provides 4-approximation guarantees, using the above formula, Sincronia system achieves a competitive ratio of 12 for the online case.

We emphasize that exponentially-increasing sized epochs are needed only for polynomial-time complexity of the online algorithm and for theoretical guarantees [19]. If these were not the goals, Sincronia could simply use suitably chosen fixed-sizes epochs. We evaluate this in §5 and show that Sincronia performance with exponentially-increasing sized epochs is very similar to that with fixed-sized epochs.

[2] As with all online algorithms, Sincronia guarantees for the online version are in terms of competitive ratio.

4.3 Sincronia + Existing Transport Layers

We now discuss Sincronia implementation on top of several existing transport layer mechanisms for flows. We already described Sincronia coordinator and daemon functionalities in §4.1; these remain unchanged across Sincronia implementation on top of various transport mechanisms. We focus on the only difference across implementations — assignment of priorities to individual flows before Sincronia daemon offloads the flows to the underlying transport mechanism.

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 [6], and sends the data over TCP. However, in practice, the underlying fabric may only support a 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 daemon sets the priority of a given 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. When using unordered coflows for work conservation, the daemon also sets priority p. While not ideal, our experimental results over real testbeds that support 8 priority levels (§5) show that Sincronia achieves significant improvements in average CCT even with small number of priority levels.

Sincronia + pHost. We now discuss Sincronia implementation on top of pHost, a receiver-driven transport layer mechanism. This is particularly interesting because pHost handles incast and outcast traffic patterns efficiently, precisely a use case for coflows. We extend the pHost 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 pHost design. In our implementation, the pHost receiver sends a `token` for packets in flows in order of corresponding coflow ordering. The server daemons, among all the received `tokens`, 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, pHost 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.

Algorithm 2 Greedy Rate Allocation Algorithm

```
All access links have uniform bandwidth
ℂ = σ is the input coflow ordering
procedure GreedyFlowScheduling(σ)
    while ℂ is not empty do
        for i = 1 to |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 available link bandwidth
        GreedyFlowScheduling(ℂ)
    return
```

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 [4]. A special "minimum priority level" is assigned to unordered coflows for work conservation purposes. Since Sincronia generates a total ordering across coflows, this implementation results in ideal Sincronia performance.

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, such fair sharing could lead to a long tail.

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 $\max_p(d_c^p/b_p)$, where b_p is the bandwidth at port p. Specifically, the orphan coflows are chosen to be scheduled for work conservation in increasing order (smaller first) of the following metric:

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

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 most datacenter networks, multiple applications co-exist, some of which 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. This may be 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 [4, 16, 25]. One way to handle deadlines in Sincronia is to assign coflow weights inversely proportional to their deadlines. Sincronia also supports admission control by assigning zero weights to coflows which would definitely miss their deadlines, hence scheduling them after other coflows. Designing coflow scheduling algorithms that provide provable guarantees for the case of deadlines is an interesting open problem (§6).

Starvation Freedom. Minimizing average completion time necessitates starvation [4, 10]. However, as discussed in §4.4, the prioritized work conservation mechanism used in Sincronia alleviates starvation to some extent. Intuitively, since the choice of an unordered 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[3]. We start by describing the workloads and performance metrics used in our evaluation (§5.1). We then evaluate Sincronia using simulations (§5.2); the main goal here is to understand the performance of Sincronia against the state-of-the-art [10], to understand the envelope of workloads for which Sincronia performs well, and to perform sensitivity analysis of Sincronia performance over a variety of parameters including workloads, number of coflows, network load and epoch sizes. We then evaluate the Sincronia implementation with TCP/DiffServ on a 16-server testbed interconnected with a FatTree topology comprising 20 commodity switches (§5.3).

5.1 Workloads and Performance Metrics

We use two workloads in our evaluation. The first workload is the one used in all prior network designs [10]: a 526-coflow trace obtained from one-hour long run of MapReduce jobs on a 3000-machine cluster at Facebook. Unfortunately, the Facebook trace makes several simplifying assumptions, is limited in number of coflows and imposes low network load. The second workload is a collection of traces generated using a coflow workload generator [1] that allows upsampling the Facebook trace to desired number of coflows, network load, etc., while keeping workload characteristics similar to the original Facebook trace. Unless mentioned otherwise, all results for these "custom" traces use the baseline of 2000 coflows, 0.9 network load and time horizon reset after 8 epochs. Varys does not support weighted coflow scheduling; thus, to make a fair comparison, we use unit weights for all coflows. We provide other setup details (*e.g.*, topology, link bandwidth, etc.) for simulations and implementation in respective subsections.

Metrics. For the offline algorithm, we evaluate the performance in terms of CCT, on an average and at high percentiles. For the online algorithm, we evaluate the performance of Sincronia using the slowdown metric; for a given coflow, the slowdown is defined as the ratio of its CCT and its OCT (recall from §4.4, the OCT of a coflow is its completion time when its the only coflow in the network). In presence of other coflows, CCT of a coflow may be much larger than its OCT; thus, our evaluation results for the online version of Sincronia are against the best possible baseline.

5.2 Simulation Results

We now evaluate Sincronia against the state-of-the-art [10] using simulations, and perform sensitivity analysis of Sincronia performance over a variety of parameters.

Setup. Varys uses a flow-level simulator in its evaluation. For a fair comparison, we incorporate Sincronia within Varys simulator, with BSSI for coflow ordering (Algorithm 1, §3.1) and greedy algorithm for flow scheduling (Algorithm 2, §4.3). We use the same setup used in Varys, modeling the network using a non-blocking 150-port switch, where each port corresponds to a top-of-the-rack switch in 3000-machine cluster.

Offline Algorithm. We first evaluate Sincronia against existing network designs for the offline case, that is, when all coflows arrive at time 0. While not a practical scenario, this allows us to tease out the performance benefits of Sincronia's BSSI algorithm against those used in prior network designs.

Figure 5(a)-5(c) show that Sincronia significantly improves upon state-of-the-art (Varys [10] and other heuristics) across all evaluated workloads. Specifically, Figure 5(a) shows that, when compared to TCF and Varys, Sincronia improves CCT by 1.7× on an average and by 7.9× at 99-th percentile. In addition, Figure 5(b) shows that Sincronia improvements over Varys increase with larger number of coflows in the trace, both on an average and at high percentiles. Since both Varys and Sincronia are work-conserving, CCT for some coflows has to degrade for improvement in CCT for other coflows. Figure 5(c) shows the CDF for distribution of improvement across all coflows; interestingly, Sincronia degrades the CCT for less than 20% of the coflows while improving the CCT for more than 65% of the coflows.

Figure 5(d)-5(e) provide more insights on how Sincronia achieves the observed improvements. Figure 5(d) shows that most of the improvements come from Sincronia's ordering algorithm, Bottleneck-Select-Scale-Iterate (BSSI) — even when BSSI is used with MADD, the rate allocation mechanism from Varys, we observe significant CCT improvements on an average and at tail. Figure 5(e) provides more insights

[3]Sincronia implementation and simulator are publicly available at: `https://github.com/sincronia-coflow`.

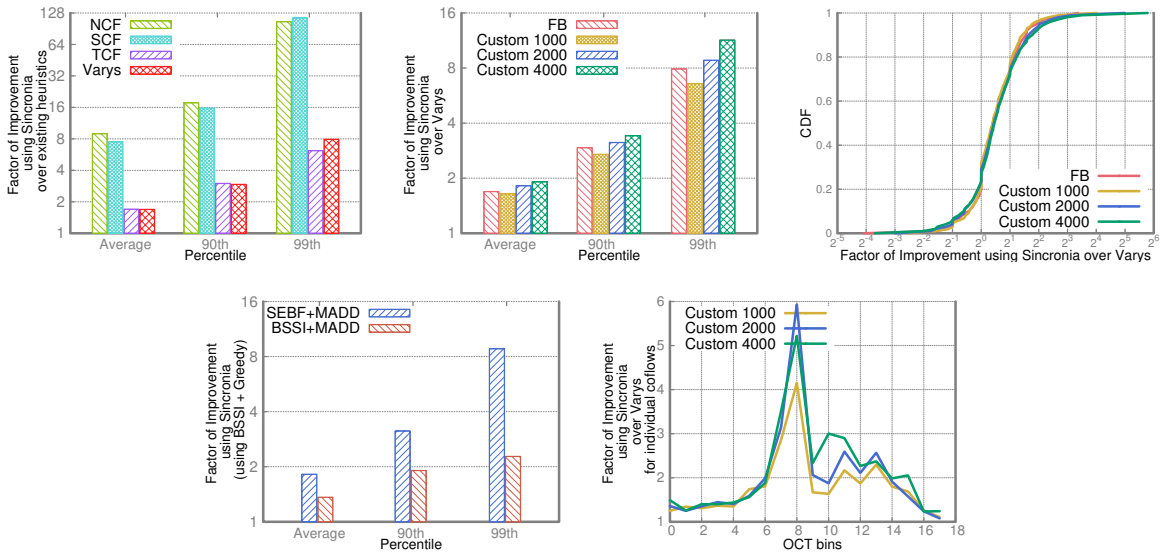


Figure 5: Sincronia evaluation results for the offline case. Sincronia achieves consistently better performance compared to existing heuristics for the original Facebook trace (top left) and maintains these improvements across a variety of workloads (top center, top right). Most of Sincronia's improvements are due to its BSSI scheduling algorithm from §3 (bottom left, bottom right). More discussion in §5.2.

about BSSI's performance. The figure plots the average factor of improvement measured across coflows binned by their OCTs normalized by the smallest OCT; that is, an OCT bin i contains the set of coflows whose OCT is within $[2^i, 2^{i+1})$ of the OCT of the smallest coflow. The figure shows that, when compared to Varys, Sincronia results in CCT improvements by making better scheduling decisions for coflows that are neither too small (low OCTs) nor too large (high OCTs). Indeed, scheduling decisions for small and large coflows are rather straightforward, and both Sincronia and Varys end up making similar decisions; it is precisely the medium size coflows where the weight scaling step in BSSI helps Sincronia in making the "right" decisions.

Online Algorithm. We now evaluate Sincronia for the case when coflows arrive over time. We focus on custom traces only since the Facebook trace has very low load and does not provide meaningful insights. As discussed in §5.1, we use the absolute best possible baseline here — ratio of coflow CCT and its OCT. Figure 6 shows that Sincronia handles online arrival of coflows very well; when compared to an unloaded network (slowdown = 1), Sincronia at load 0.9 performs within 1.7× on an average and within 5.7× at high percentiles for the Custom 4000 trace.

Figure 6(b)-6(c) provide more insights into Sincronia's performance. The former shows that around 60% of the coflows achieving slowdown 1, that is, are optimally scheduled by Sincronia. As the size of the trace increases, contention for network resources increases resulting in higher slowdowns. Figure 6(c) shows that larger slowdowns are caused mainly by larger coflows; this is not only because smaller coflows should indeed be prioritized but also that the work conservation heuristic in Sincronia (§4.2) prioritizes smaller coflows until long coflows have been waiting for a long time.

Sensitivity Analysis. Figure 5 and Figure 6 already show impact of trace sizes on Sincronia performance. We perform additional sensitivity analysis of Sincronia performance against network load, number of epochs before resetting the time horizon, and various epoch mechanisms in Figure 7. Figure 7(a) shows that, as expected, Sincronia performance improves as the network load decreases from 0.9 to 0.7. We see in Figure 7(b) that Sincronia performance varies with number of epochs before resetting the time horizon — as the number of epochs increase, larger fraction of coflows observe completion times close to their OCTs but the tail slowdown also worsens. Intuitively, as the number of epochs increase, the largest epoch size also increases; this, in turn, increases the chance of a small coflow arriving between epoch boundaries and being blocked by a large "ordered" coflow, one that was ordered at the start of the epoch and hence has higher priority than the small coflow. Finally, we compare the performance of Sincronia with varying epoch mechanisms. We use exponentially-increasing epoch sizes, fixed-epoch sizes (with size equal to the longest epoch in exponentially-increasing epoch size mechanism) and the "immediate recomputation" mechanism that executes the BSSI algorithm upon each coflow arrival and departure. Figure 7(c)

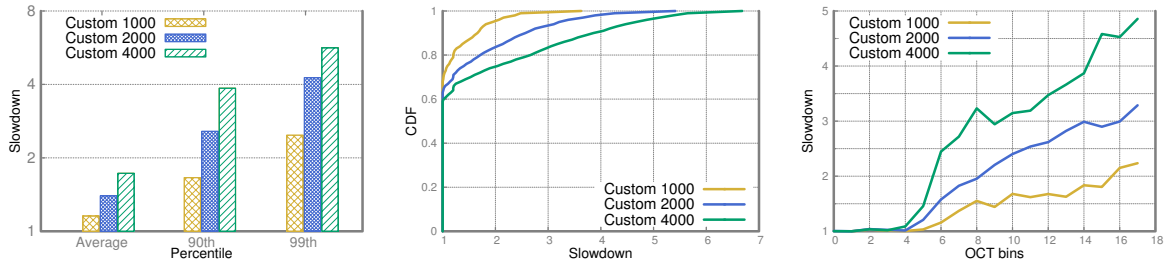


Figure 6: Sincronia evaluation results for the online case. (left, center) For the evaluated workloads, Sincronia achieves performance within 1.7× on an average and 5.7× at high percentiles when compared to an unloaded network; (right) Coflows with larger sizes observe larger slowdown in online version of Sincronia. More discussion in §5.2.

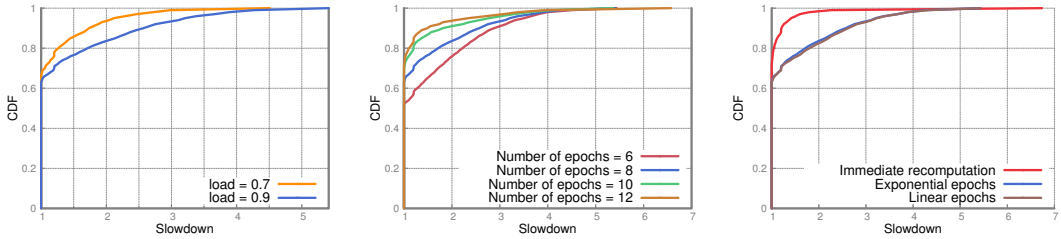


Figure 7: Sensitivity analysis of Sincronia performance over varying network load (left), varying number of epochs before resetting the time horizon (center) and varying epoch mechanisms (right). More discussion in §5.2.

shows that, over the evaluated workloads, the first two mechanisms perform comparably and are only marginally worse when compared to the (impractical) immediate recomputation mechanism.

5.3 Implementation Results

We now present evaluation results for Sincronia implementation on top of TCP (using Diffserv [6] for priority scheduling). Our implementation runs on a topology shown in Figure 9. We compare Sincronia performance against coflow-agnostic TCP implementation[4].

The main challenge in running these experiments is that TCP does not handle incast traffic very well, precisely the scenario for coflows (multiple sources sending data to the same destination). Thus, we compute the expected load on a destination (by using network load along with expected number of sources sending data to a destination in the Facebook trace), and refer to it as Maximum Sustainable Load (MSL) for the cluster. We then generate the workloads for the cluster using the workload generator for 16 ingress/egress ports, with 1Gbps access link bandwidth and varying loads (0.7×MSL, 1.4×MSL and 2.1×MSL).

[4]Despite trying for several weeks, we were not able to run Varys [10] over our topology; the code for Varys 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 [10] to share their implementation with us and helping us to try and run their code.

Figure 8 shows that for a workload comprising 526 coflows at load 0.7×MSL, Sincronia implementation on top of TCP improves the CCT by 18.61× on an average, by 46.95× at 90-th percentile, and 149.05× at 99-th percentile when compared to coflow-agnostic TCP.

As expected, the improvements are even more significant for larger loads. Figure 8(b) shows that Sincronia achieves these improvements by slowing down less than 10% of the coflows by more than 2× (and even smaller fraction for higher loads). Finally, Figure 8(c) shows that Sincronia achieves similar benefits for larger traces. This evaluation is a bit unfair — TCP is neither designed for coflows nor for minimizing average completion times. However, these results give us some indication on performance of Sincronia against existing network designs; for instance, Varys [10] reports to improve upon TCP by a factor of 1.85× on an average, and roughly by the same number at 95 percentiles.

6 RELATED WORK & OPEN PROBLEMS

There has been tremendous recent effort on network design for coflows, both in networking and in theory communities. In this section, we compare and contrast Sincronia against related work from these communities, and discuss a number of problems that remain open in the context of network design for coflows.

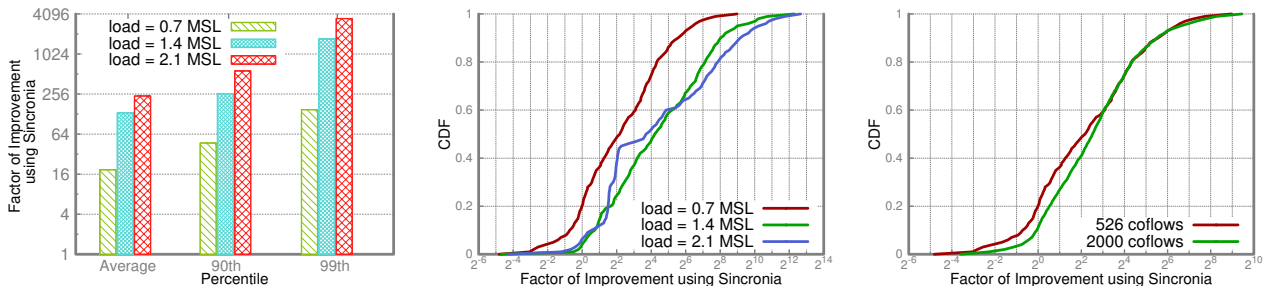


Figure 8: Evaluation results for Sincronia implementation on top of TCP over our testbed in Figure 9. Sincronia when implemented on top of TCP/DiffServ significantly improves coflow performance when compared to coflow-agnostic TCP both on an average and at high percentiles (left), and maintains these improvements across varying network loads (center) and trace sizes (right). More discussion in §5.3.

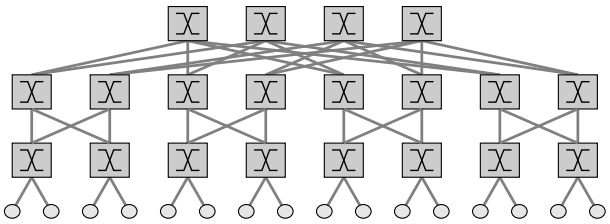


Figure 9: Testbed used in our experiments has 16 servers, each with 1Gbps access link bandwidth, interconnected with a FatTree topology comprising 20 PICA8 switches. Each switch supports priority queueing with 8 distinct levels.

Gap between lower and upper bounds. Existing network designs for coflows from networking community [7, 9, 10, 12] can perform arbitrarily worse than the optimal in terms of average CCT [2]. Qiu et. al. [22] provided the first coflow scheduling algorithm that achieves approximation guarantees of 64/3 for the offline case with zero release dates. This result was improved by Ahmadi et. al. [3] to 4-approximation for zero release dates and to 5-approximation with release dates. However, all these results require a central coordinator to either solving complex linear programs or to perform complex per-flow rate allocations. Sincronia achieves state-of-the-art approximation guarantees, both for zero release dates and, using a simple extension of the BSSI algorithm [2], for the case of release dates. However, Sincronia shows that these guarantees can be achieved without any complex per-flow rate allocation, thus overcoming the limitations of all existing results from the networking and the theory community. Despite all the above results, there is a gap between known lower bounds for coflow scheduling (impossibility of better than 2-approximation, §2) and the known upper bounds (4-approximation for offline problem with zero release dates) [2, 3]. It remains an intriguing problem to bridge this gap.

Improved competitive ratio for online coflow scheduling. Khuller et. al. [19] were the first to present an online algorithm that achieves a competitive ratio of 12. Sincronia achieves similar guarantees. Is it possible to design a coflow scheduling algorithm with a better competitive ratio? Note that an offline algorithm that achieves a better approximation factor for the case of zero release dates will already lead to improvements in the online algorithm by using the framework from Khuller et. al. [19].

Necessity of centralized solutions. As discussed in §2, there is a strong lower bound of $\Omega(\sqrt{n})$ on achievable approximation for average CCT using algorithms that do not use any coordination [8]. However, the "amount of coordination" required to mitigate this lower bound is unclear. For instance, similar to [10, 28, 29], Sincronia requires a centralized controller to implement BSSI. 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 understand the amount of computation needed to be done at the centralized controller.

Coflows with paths. Sincronia, similar to [10, 12], assumes that routing of flows within and across coflows are decided by the network layer (either via specific routing mechanisms, or via packet spraying). However, it is a priori conceivable that better performance may be achievable by co-designing routing along with scheduling of coflows. This problem has been studied in a few recent papers [18, 29]; however, designing optimal algorithms for this problem remains open.

Extensions to Non-clairvoyant scheduler. Sincronia, similar to [10, 12, 29], 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 [10] 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 [28]. However, designing a near-optimal non-clairvoyant scheduler remains an interesting future direction.

Extensions to other performance metrics. Finally, similar to most of the existing literature on coflows [8, 10, 28, 29], Sincronia is designed to optimize for average (weighted) completion time. It remains an interesting question to extend our results to the case of deadline-sensitive coflows and for minimizing other metrics such as tail coflow completion time. An intriguing question in this direction is also to define a notion of fairness for coflows.

7 PROOF OUTLINE FOR THEOREM 1

In this section, we provide a high-level idea for our results in §3. The detailed proofs for these results are in the technical report [2].

At a high-level, we use a linear programming relaxation to obtain a lower bound on the optimal value of the average CCT, and obtain our approximation guarantee by comparing the average CCT of our algorithm to this lower bound. Our algorithm is purely combinatorial, in that it does not require solving 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; hence, by our lower bound argument, 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.

8 CONCLUSION

We have presented Sincronia, a network design for coflows that provides near-optimal performance and can be implemented on top on any transport layer mechanism for flows that supports priority scheduling. Sincronia achieves this using a key technical result — we show that given a "right" ordering of coflows, any per-flow rate allocation mechanism achieves average coflow completion time within 4× of the optimal as long as (co)flows are prioritized with respect to the ordering. 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 transport mechanism that supports priority scheduling (without any per-flow rate allocation mechanism). 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.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1526067, CMMI-1537394, CCF- 1522054, and CCF-1740822.

REFERENCES

[1] 2018. Sincronia Repository. https://github.com/sincronia-coflow.

[2] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-Optimal Network Design for Coflows. In *Tech Report*.

[3] Saba Ahmadi, Samir Khuller, Manish Purohit, and Sheng Yang. 2017. On scheduling coflows. In *MOS IPCO*.

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

[5] Nikhil Bansal and Subhash Khot. 2010. Inapproximability of hypergraph vertex cover and applications to scheduling problems. In *EATCS ICALP*.

[6] Kwok Ho Chan, Jozef Babiarz, and Fred Baker. 2006. Configuration Guidelines for DiffServ Service Classes. https://tools.ietf.org/html/rfc4594.

[7] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: A networking abstraction for cluster applications. In *ACM HotNets*.

[8] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM*.

[9] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. 2011. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM*.

[10] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *ACM SIGCOMM*.

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

[12] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. 2014. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM*.

[13] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT*.

[14] Naveen Garg, Amit Kumar, and Vinayaka Pandit. 2007. Order scheduling models: hardness and algorithms. In *IARCS FSTTCS*.

[15] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew Moore, Gianni Antichi, and Marcin Wojcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*.

[16] Chi-Yao Hong, Matthew Caesar, and P Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*.

[17] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM EuroSys*.

[18] Hamidreza Jahanjou, Erez Kantor, and Rajmohan Rajaraman. 2017. Asymptotically Optimal Approximation Algorithms for Coflow Scheduling. In *ACM SPAA*.

[19] Samir Khuller, Jingling Li, Pascal Sturmfels, Kevin Sun, and Prayaag Venkat. 2018. Select and Permute: An Improved Online Framework for Scheduling to Minimize Weighted Completion Time. In *LATIN*.

[20] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment, 5(8): 716-727*.

[21] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*.

[22] Zhen Qiu, Cliff Stein, and Yuan Zhong. 2015. Minimizing the total weighted completion time of coflows in datacenter networks. In *ACM SPAA*.

[23] Thomas A. Roemer. 2006. A note on the complexity of the concurrent open shop problem. In *Journal of Scheduling, 9(4): 389-396*. Springer.

[24] Sushant Sachdeva and Rishi Saket. 2013. Optimal inapproximability for scheduling problems via structural hardness for hypergraph vertex cover. In *IEEE CCC*.

[25] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM*.

[26] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.. In *USENIX OSDI*.

[27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*.

[28] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. 2016. CODA: Toward automatically identifying and scheduling coflows in the dark. In *ACM SIGCOMM*.

[29] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. 2015. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *IEEE INFOCOM*.