Coflow: A Networking Abstraction for Distributed Data-Parallel Applications

by

N M Mosharaf Kabir Chowdhury

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Sylvia Ratnasamy
Professor John Chuang

Fall 2015

Coflow: A Networking Abstraction for Distributed Data-Parallel Applications

Abstract

Coflow: A Networking Abstraction for Distributed Data-Parallel Applications

by

N M Mosharaf Kabir Chowdhury

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Over the past decade, the confluence of an unprecedented growth in data volumes and the rapid rise of cloud computing has fundamentally transformed systems software and corresponding infrastructure. To deal with massive datasets, more and more applications today are scaling out to large datacenters. These *distributed data-parallel applications* run on tens to thousands of machines in parallel to exploit I/O parallelism, and they enable a wide variety of use cases, including interactive analysis, SQL queries, machine learning, and graph processing.

Communication between the distributed computation tasks of these applications often result in massive data transfers over the network. Consequently, concentrated efforts in both industry and academia have gone into building high-capacity, low-latency datacenter networks at scale. At the same time, researchers and practitioners have proposed a wide variety of solutions to minimize flow completion times or to ensure per-flow fairness based on the point-to-point *flow* abstraction that forms the basis of the TCP/IP stack.

We observe that despite rapid innovations in both applications and infrastructure, application- and network-level goals are moving further apart. Data-parallel applications care about *all* their flows, but today's networks treat each point-to-point flow independently. This fundamental mismatch has resulted in complex point solutions for application developers, a myriad of configuration options for end users, and an overall loss of performance.

The key contribution of this dissertation is bridging this gap between application-level performance and network-level optimizations through the *coflow* abstraction. Each multipoint-to-multipoint coflow represents a collection of flows with a common application-level performance objective, enabling application-aware decision making in the network. We describe complete solutions including architectures, algorithms, and implementations that apply coflows to multiple scenarios using central coordination, and we demonstrate through large-scale cloud deployments and trace-driven simulations that simply knowing *how flows relate to each other* is enough for better network scheduling, meeting more deadlines, and providing higher performance isolation than what is otherwise possible using today's application-agnostic solutions.

In addition to performance improvements, coflows allow us to consolidate communication optimizations across multiple applications, simplifying software development and relieving end users

from parameter tuning. On the theoretical front, we discover and characterize for the first time the *concurrent open shop scheduling with coupled resources* family of problems. Because any flow is also a coflow with just one flow, coflows and coflow-based solutions presented in this dissertation generalize a large body of work in both networking and scheduling literatures.

To my family

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I thank Allah, the Lord of the worlds, the Beneficent, and the Merciful, for enabling me to complete this task.

I am immensely grateful to my advisor, Ion Stoica, who pushed me to focus on one problem at a time, to go deep into the details of each one, and to find the nugget. He instilled in me a deep appreciation for simplicity in design and clarity in presentation. He also gave me ample freedom and the right amount of guidance throughout my graduate studies. I am especially thankful for his honest feedback that improved me as a researcher and a person. My research benefited greatly from Ion's insistence on doing great work.

This dissertation is the culmination of many successful collaborations: Chapters 2 and 3 build upon joint work with Ion Stoica [65]; Chapter 4 includes materials from joint work with Matei Zaharia, Justin Ma, Michael Jordan, and Ion Stoica [67] and with Yuan Zhong and Ion Stoica [68]; Chapter 5 was joint work with Yuan Zhong and Ion Stoica [68]; Chapter 6 was joint work with Ion Stoica [66]; and Chapter 7 was joint work with Zhenhua Liu, Ali Ghodsi, and Ion Stoica [64].

I am thankful to my dissertation committee and qualifying examination committee members Sylvia Ratnasamy, John Chuang, Marti Hearst, and Scott Shenker for useful feedback. I especially thank Randy Katz for the two conversations that bookended my time in Berkeley – the former convinced me to come to Berkeley, while the latter guided my departure. Outside Berkeley, I am thankful to Srikanth Kandula for being a great mentor and an enthusiastic collaborator.

I am grateful to many colleagues in Berkeley and elsewhere for supporting my research as collaborators and reviewers. Across many projects not covered in this dissertation, I had been fortunate to collaborate with Sameer Agarwal, Dilip Joseph, Michael Franklin, Scott Shenker, Tathagata Das (TD), Ankur Dave, Murphy McCauley, Gautam Kumar, Sylvia Ratnasamy, Lucian Popa, Arvind Krishnamurthy, Peter Bodik, Ishai Menache, David Maltz, Pradeepkumar Mani, Srikanth Kandula, Rachit Agarwal, and Vyas Sekar. I learned something useful from each one of them.

The AMPLab and RAD Lab were amazing communities, who gave constant feedback. Matei demonstrated how to build and maintain large systems; Ganesh Ananthanarayanan had been an invaluable sounding board; Ali had wikipedic knowledge on many things computer science; and David Zats had pointers to most related work. The CCN group was the toughest audience I presented my ideas to. My work became stronger because of the heated arguments in CCN meetings. I thank Justine Sherry, Shivaram Venkataraman, and Neil Conway for gently critiquing many of our drafts. Finally, the support staff had been excellent; on several occasions, Jon Kuroda managed impromptu backup machines right before important deadlines.

My internships at Microsoft Research Cambridge (UK) and Microsoft Research Redmond/Bing as well as my time at Facebook gave me wonderful opportunities to experience research and work at industry labs and the industry. I am thankful to my mentors and managers – Christos Gkantsidis, Hitesh Ballani, David Maltz, and Hairong Kuang – for enabling many learning experiences.

I am thankful for a UC Berkeley Graduate Fellowship, a Facebook Fellowship, and to the many federal and industry sponsors of the AMPLab and RAD Lab.

Many friends in Berkeley enriched my life outside work. Bill Marczak had been a constant presence for the first three years when we shared apartments. I had countless dinner outings and movie nights with Atif Rahman, Khalid Ashraf (Mishu), Muhammed Karim (Rumi), Bill, Sameer, and TD. The Bangladeshi community in Berkeley was the source of many fond memories. Special thanks to Asif Khan, Nadia Nusrat, Atif, Mishu, Rumi, Ulfat Obaidi (Rimi), Imran Khan, Yasser Khan (Tasif), Sifat Muin (Ishita), Suman Hossain, Arefa Hossain (Shumi), Ariful Azad, Rubaya Parveen (Ishita), Urmita Sikder, Iqbal Haider (Sumon), Nafisa Akbar, and Inaya. Thanks to my numerous friends and well-wishers all over the world.

Although my graduate studies have been long, it is only a part of a much longer narrative. I want to thank all my teachers in Arab Mission Public School, Dhaka College, BUET, and the University of Waterloo. Among many outstanding teachers, mentors, and role models that shaped and supported my education and career are: Shahida Akhter, Md Nezam Uddin, Abdul Latif Salampuri, ATM Shakhawat Hossain, Mohammad Kaykobad, Reaz Ahmed, and Raouf Boutaba. Mohammad walked me through my first publication. Raouf gave me free rein to work on problems I liked during my Master's and allowed me to discover the joy of research.

I would like to express my deepest gratitude to my parents and siblings for their love and support. My mother, Shahida Akhter, is my guru. None of my achievements would have been possible without her countless sacrifices, infinite patience, and steely determination to give me the best possible education. She raised me to aim for the moon and taught me to keep trying until I succeed. My father, Alamgir Chowdhury (Aalo), instilled the sense of duty and responsibility in me. He passed away too soon, but he would have been the proudest to see my achievements. Shafiul Chowdhury (Raki), my brother, is younger only in age and elder in practice. He weathers many storms back in Bangladesh to shield me from the real world and continuously encourages me to carry on. I am thankful to my parents-in-law, Muhammad Musharraf Hussain and Mahmuda Musharraf, who encourage and support my decision to be in the academia. Special thanks to Rubana Chowdhurys, Deena Hussain, Mahmud Hussain, Nusrah Hussain, and Aleena for being who they are. I am grateful for having a supportive extended family.

Last but not least, I am profoundly indebted to my wife Bushra Hussain for her love, understanding, support, and sacrifices. She provides a sense of normalcy and keeps me grounded regardless of all the ups and downs in life. This journey would have felt much longer and more stressful without having her beside me.

Chapter 1

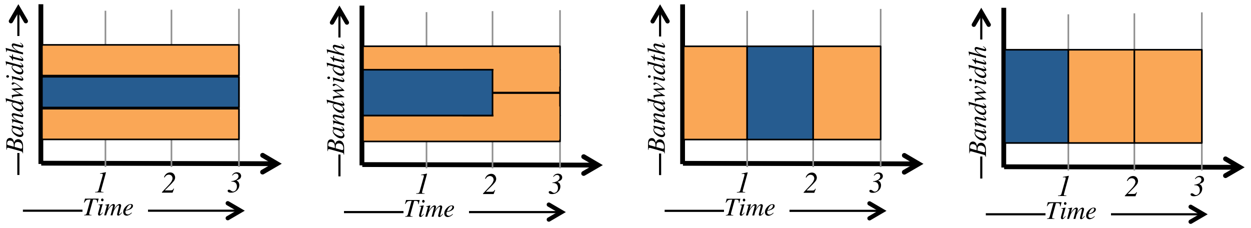
Introduction

The confluence of an unprecedented growth in data volumes and the rapid rise of cloud computing has fundamentally transformed systems software and corresponding infrastructure over the past decade. More and more applications today are dealing with large datasets from diverse sources. These sources range from user activities collected from traditional and mobile webs to data collected from connected equipment and scientific experiments. To make sense of all these datasets, researchers and practitioners in both academia and industry are building applications to enable, among others, SQL queries [8, 9, 22, 43, 135, 203], log analysis [6, 77, 208], machine learning activities [98, 136, 152], graph processing [101, 146, 147], approximation queries [22, 31], stream processing [23, 27, 32, 157, 209], and interactive analysis [22, 208].

Furthermore, to deal with growing data volumes in a fast and efficient manner, these applications are scaling out to large datacenters to exploit I/O parallelism. *Distributed data-parallel applications* such as these run on many machines in parallel by dividing the entire work – a *job* – into smaller pieces – individual *tasks*. They interact with cluster resource managers [105, 117, 195] to receive CPU, memory, and disk resources and use a variety of techniques to schedule tasks, handle failures, mitigate stragglers, and manage in-memory and on-disk datasets [39, 40, 55, 120, 145, 198, 207, 210]. For communication, however, each application implements its own library on top of TCP/IP to manage data transfers between tasks running on different machines.

To meet the growing bandwidth demand of distributed data-parallel applications, concentrated efforts have gone into building high-capacity, low-latency datacenter networks [69, 88, 106, 108, 110, 158, 186, 199]. To take advantage of these resources, networking researchers have also proposed a wide variety of solutions to minimize *flow completion times* or to ensure *per-flow fairness* [33, 35, 37, 45, 80, 87, 118, 121, 192, 200, 201] based on the point-to-point *flow* abstraction that forms the basis of the TCP/IP stack.

However, despite rapid innovations in both applications and infrastructure, application- and network-level goals are moving further apart. Data-parallel applications care about *all* their flows, but the network today treats each point-to-point flow independently. This fundamental mismatch has resulted in complex point solutions for application developers, a myriad of configuration options for end users, and an overall loss of performance. For example, per-flow fairness causes application-1 to receive more share in Figure 1.1a because it has more flows, instead of both appli-



(a) Per-flow fairness **(b)** Application-aware fairness **(c)** Per-flow prioritization **(d)** Application-aware prioritization

Figure 1.1: Bandwidth allocations of two flows from application-1 (orange/light) of size $(1 - \delta)$ and $(1 + \delta)$, and one flow from application-2 (blue/dark) of size 1 on a single bottleneck link of unit capacity. Both applications start at the same time.

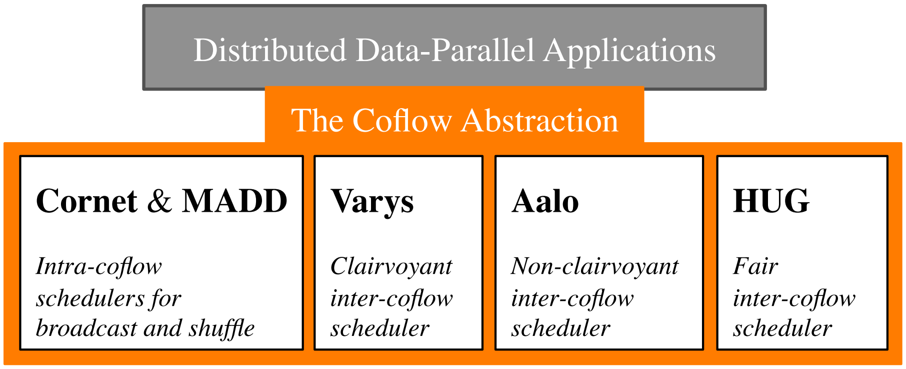

Distributed Data-Parallel Applications
The Coflow Abstraction
Cornet & MADD
Intra-coflow schedulers for broadcast and shuffle
Varys
Clairvoyant inter-coflow scheduler
Aalo
Non-clairvoyant inter-coflow scheduler
HUG
Fair inter-coflow scheduler

Figure 1.2: Components built in this dissertation to enable application-aware networking using the coflow abstraction. Any application can take advantage of coflows using the coflow API.

cations receiving the same (Figure 1.1b). Similarly, while the network minimizes the average flow completion time by prioritizing smaller flows in Figure 1.1c, application-2 completes in 2 time units instead of the optimal 1 time unit (Figure 1.1d). In both cases, flow-level optimal solutions are suboptimal at the application level.

How do we realign these goals? That is, *how do we enable application-network symbiosis to improve application-level performance and end user experience?* This dissertation argues that the answer lies in exposing the communication characteristics of applications to the network. Specifically, we present the *coflow* abstraction to capture the collective behavior of flows in distributed data-parallel applications. We apply coflows to multiple scenarios (Figure 1.2) to demonstrate that simply knowing *how flows relate to each other* is enough for minimizing job completion times, meeting deadlines, and providing performance isolation, and it can take us far beyond than what is otherwise possible today.

Not only do coflows improve application-level performance without additional resource usage, but they also allow us to consolidate communication optimizations, simplifying software development and relieving end users from parameter tuning. Additionally, coflows allow us to reason about complex dependencies in distributed data-parallel applications and can be composed together to capture those dependencies. Because any flow is also a coflow with just one flow, coflows and coflow-based solutions presented in this dissertation generalize a large body of work in both networking and scheduling literatures.

In the remainder of this chapter, we highlight the motivations for application-aware networking, overview the coflow abstraction and its applications, and summarize our results.

1.1 Problems Due to Application-Agnostic Networking

The flow abstraction used in today's networks was designed for point-to-point client-server applications. Consequently, solutions based on this abstraction [33, 35, 37, 45, 80, 87, 118, 121, 192, 200, 201] optimize the network assuming that today's distributed data-parallel applications have the same requirements and characteristics as their decades-old counterparts. To compensate for the lack of a proper abstraction, most of these applications [6, 8, 43, 101, 119, 146, 208, 209] implement customized communication libraries and expose a large number of tuning parameters to end users. The mismatch between application- and network-level objectives leads to several shortcomings:

1. **Optimization of Irrelevant Objectives:** Application-agnostic network-level solutions optimize metrics such as the average flow completion time and per-flow fairness that preferentially or equally treat flows *without* knowing their collective impact on application-level performance. They often do not lead to performance improvements and sometimes even hurt applications (Figure 1.1).

2. **Simplistic Models:** Given that point-to-point flows cannot even capture a single communication stage, flow-based solutions are unsuitable for reasoning about complex dependencies within multi-stage applications and between several of them in a data-processing pipeline, let alone optimizing their performance objectives.

3. **Point Solutions:** Most distributed data-parallel applications face a common set of underlying data transfer challenges. For example, the many-to-many communication pattern between two groups of tasks in successive computation stages – i.e., a *shuffle* – occurs in all these applications. The same goes for a *broadcast* or an *aggregation*. Without a common abstraction, we must address the same challenges anew for every application we build.

4. **Best-Effort Network Sharing:** Because every application implements its own set of optimization techniques, the network is shared between multiple applications in a best-effort manner. There is no notion of a global objective, be it minimizing the average completion time across all applications or providing performance isolation among them.

5. **Limited Usability:** For end users, too many parameters can be hard to understand and difficult to tune even for a single application. A shared environment further compounds this problem. Without knowing what others are tuning for, one cannot hope for better than best-effort, and the same parameter settings may behave unpredictably across multiple runs.

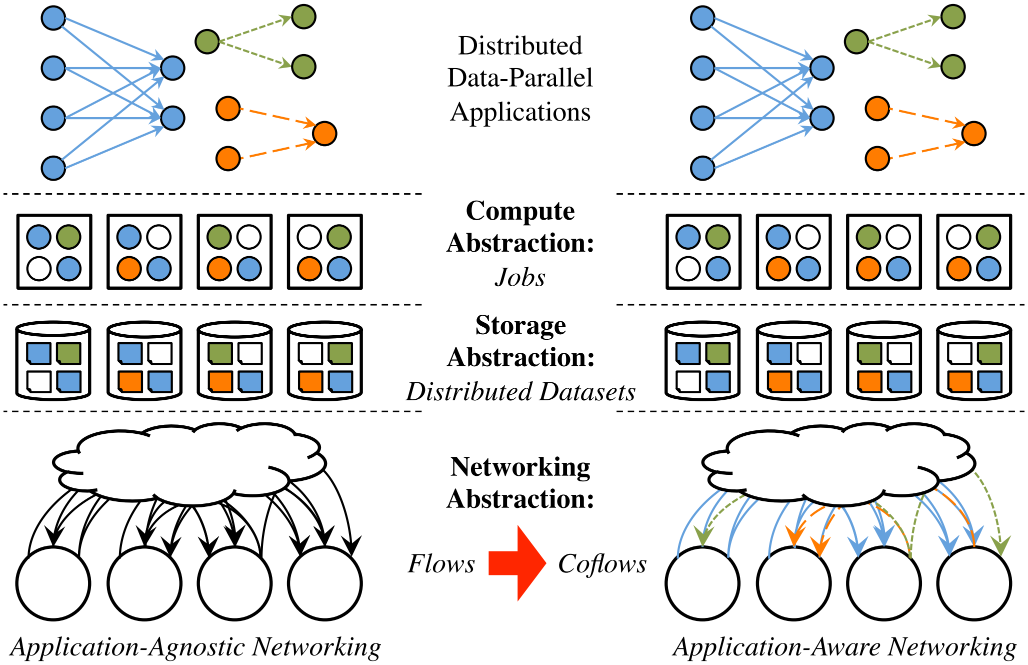


Figure 1.3: Coflows in the software stack of distributed data-parallel applications. Similar to compute and storage abstractions, coflows expose application-level relationships to the network. In contrast, traditional point-to-point flows from different applications are indistinguishable.

1.2 The Coflow Abstraction

We introduce the *coflow* abstraction to address these problems. The key insight behind coflows is one simple observation: *a communication stage in a distributed data-parallel application cannot complete until all its flows have completed*. Meaning, applications do not care about individual flows' completion times or fair sharing among them – the last flow of an application dictates its completion. A coflow captures this *all-or-nothing* characteristic of *a collection of flows*, and it is the first abstraction to expose this to the network (Figure 1.3).

The all-or-nothing observation had been captured for other resource types in data-parallel clusters – e.g., a job represents a set of tasks and a distributed file or dataset represents a collection of on-disk or in-memory data blocks – and are leveraged in almost all aspects of building systems, improving performance, scheduling resources, and providing fault tolerance [6, 31, 39, 40, 55, 77, 95, 101, 117, 119, 145, 147, 191, 208–210]. What makes coflows unique and more general is the *coupled nature* of the network – unlike independent resources such as CPU speed or disk/memory capacity, one must consider both senders and receivers to allocate network resources.

The key advantage of coflows is that they enable us to *reason about application-level objectives when optimizing network-level goals* such as performance, utilization, and isolation. This is crucial because application-agnostic solutions are often at odds with application-level metrics. For example, we show that improving a coflow's completion time (CCT) improves the completion time of its parent job, even when it hurts traditional metrics such as flow completion time or flow-level fairness. Furthermore, this improvement is achieved *without* any additional resources.

Coflows can be *extended and composed* to go beyond what is possible from just capturing the collective characteristic of some flows. For instance, when we have information about the total number of flows in a coflow or individual flow sizes, we can annotate coflows and use this information to improve performance or provide better isolation. Similarly, using the dependency information between the computation stages of complex jobs – i.e., the ones represented by directed acyclic graphs or DAGs – we can combine coflows while maintaining their dependencies in the DAG. Because any flow is also a coflow with just one flow, *coflows are general*.

Coflows also make a *qualitative difference in application development*. Despite diverse end goals, distributed data-parallel applications use a common set of multipoint-to-multipoint communication patterns – formed by a collection of point-to-point flows – with a handful of optimization objectives. Instead of reimplementing the same set of communication patterns in every application and trying to optimize them using tens of user-tunable parameters, coflows allow applications to offload this responsibility to a separate system such as Varys or Aalo presented in this dissertation. This reduces work duplication, simplifies application logic, and enables coherent optimizations.

Finally, we prove why *coordination cannot be avoided in coflow-based solutions*. We also show empirically that, for long-running coflows, it is better to incur small coordination overheads (e.g., 8 ms in our 100-machine cluster and 992 ms for a $100,000$-machine emulated datacenter) than avoiding it altogether. Our implementations carefully disregard coflows with durations shorter than coordination overheads, because they have smaller overall impact.

1.3 Scheduling Using Coflows

We used coflows to optimize individual communication patterns prevalent in distributed data-parallel applications and to improve performance across multiple applications for objectives such as minimizing the average completion time, ensuring coflow completions within deadlines, and providing isolation. In all cases, coflows enabled us to reason about the collective nature of communication in modern applications and go beyond what is possible using previous solutions.

1.3.1 Schedulers Proposed in this Dissertation

We discuss four applications of coflows to improve the communication performance for individual applications and across multiple applications in this dissertation.

1. **Intra-Coflow Scheduling:** Shuffle and broadcast are the two key communication patterns in modern distributed data-parallel applications. Traditional flow-based solutions improve flow-level fairness or flow completion times, but they ignore the collective nature of all the flows in a shuffle or a broadcast.

   For broadcasts, we proposed *Cornet* that implements a BitTorrent-like protocol [72]. It differs in one key aspect: Cornet attempts to accelerate the slowest participant instead of throttling it – the slowest one is the one that matters the most. For shuffles, we proposed an optimal algorithm called *Minimum Allocation for Desired Duration (MADD)* that explicitly sets data-proportional rates, ensuring that all the flows finish together with the slowest flow.

2. **Clairvoyant Inter-Coflow Scheduling:** Due to the high operating expense of large clusters, operators aim to maximize cluster utilization, while accommodating a variety of applications and workloads. This causes communication stages from multiple applications to coexist. Existing solutions suffer even more in shared environments. Smaller flows in each collection complete faster, but the larger ones lag behind and slow down entire communication stages.

   For many applications, coflow-level information – e.g., the number of flows in a coflow or individual flow sizes – are known a priori or can be estimated well [30, 77, 147, 208]. We designed *Varys* to capitalize on this observation and to perform clairvoyant inter-coflow scheduling to minimize the average coflow completion time and to ensure coflow completions within deadlines. Varys uses MADD as a building block and proposes *Smallest Effective Bottleneck First (SEBF)* to interpret the well-known Shortest-First heuristic in the context of coflows. In the process, we discovered and characterized the *concurrent open-shop scheduling with coupled resources* problem.

3. **Non-Clairvoyant Inter-Coflow Scheduling:** In many cases, coflow characteristics can be unknown a priori. Examples include applications with multiple coflows between successive computation stages [9, 73, 119, 181], when all tasks in a computation stage are not scheduled together [39], and when some tasks are restarted to mitigate failures or stragglers [40, 207, 210]. This raises a natural question: how to schedule coflows without complete knowledge?

   We proposed *Aalo* to perform non-clairvoyant coflow scheduling. Aalo interprets the classic Least-Attained Service (LAS) scheduling discipline in the context of coflows using *Discretized Coflow-Aware Least-Attained Service (D-CLAS)* and closely approximates Varys's performance without prior information. We showed that capturing how individual flows relate to each other using coflows is the key to extracting most of the benefits.

4. **Fair Inter-Coflow Scheduling:** Efficient coflow schedulers improve only the average-case performance. However, they cannot isolate coflows in multi-tenant environments such as public clouds, where tenants can be non-cooperative unlike those in private datacenters. Existing flow-level solutions perform even worse in terms of providing isolation. Algorithms for multi-resource fairness (e.g., DRF [97]) provide isolation, but they can arbitrarily decrease network utilization.

   We proposed *High Utilization with Guarantees (HUG)* to achieve the highest possible utilization while maximizing performance isolation between coflows from different tenants in non-cooperative environments. We proved that full utilization, i.e., work conservation, is not possible in such environments because work conservation incentivizes tenants to lie. In the process, we generalized single- [80, 121, 165] and multi-resource max-min fairness [83, 97, 112, 166] and multi-tenant network sharing solutions [124, 140, 171, 172, 179, 184].

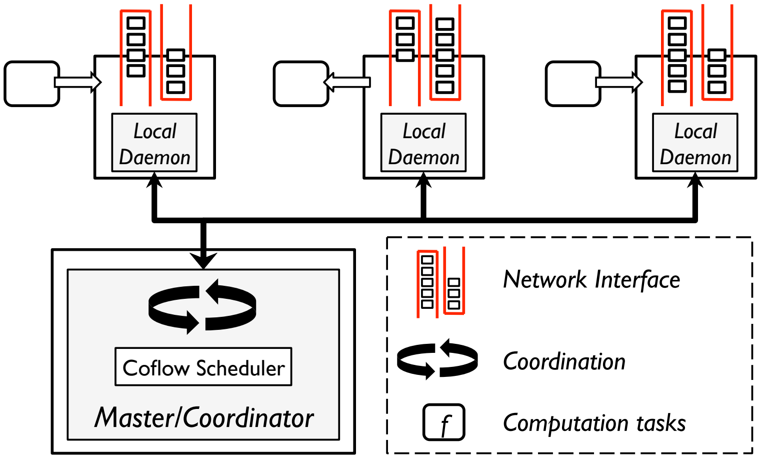


Figure 1.4: Centralized coordination architecture used in this dissertation.

Furthermore, HUG is work-conserving in cooperative environments such as private datacenters, where tenants are guaranteed not to lie.

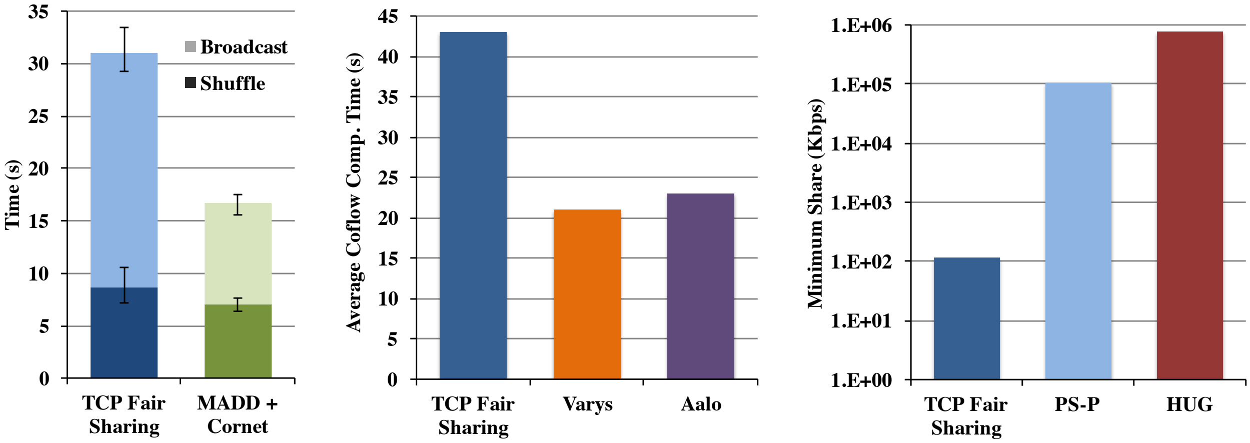
1.3.2 Scheduler Architecture

For each of the aforementioned coflow scheduling solutions, we built systems following the high-level architecture shown in Figure 1.4. The master or the coordinator orchestrates coflows using centralized coordination. Applications use the coflow API through a client library to register coflows with the master, send or receive data, and provide coflow-level information, when available. The master aggregates all interactions, creates a global view of the network, and determines new schedules. Each machine runs a daemon that interacts with local client library instances to enforce the global schedule determined by the master.

There are two primary ways to trigger coordination: (1) on coflow lifecycle events such as arrival, departure, and update; and (2) periodically. The former results in tight coordination, which is more accurate but adds coordination overheads to small coflows. The latter enables loose coordination, which scales better but loses efficiency due to local decisions in between coordination periods. Depending on information availability, we explored both directions in different solutions – MADD, Varys, and HUG coordinate on lifecycle events, while Cornet and Aalo use periodic coordination.

1.4 Summary of Results

We evaluated our solutions by deploying them on 100-machine Amazon EC2 clusters and through trace-driven simulations using workloads collected from a 3000-machine Facebook production cluster. Please refer to Appendix A for more details on the workload. We also performed micro-



(a) Intra-coflow scheduling **(b)** Efficient inter-coflow scheduling **(c)** Fair inter-coflow scheduling

Figure 1.5: [EC2] Performance comparison between coflow-based solutions and the state-of-the-art: (a) communication time in each iteration of a logistic regression application; (b) average CCT of a SQL (Hive) workload from Facebook trace; and (c) minimum network share across coflows in MapReduce applications. The Y-axis in (c) is in log scale.

benchmarks using individual machine learning applications that use communication-heavy algorithms such as logistic regression and collaborative filtering.

Figure 1.5 presents a quick summary of the benefits of coflow scheduling – both in improving the performance of individual applications and across multiple applications – by comparing our solutions against the existing flow-based mechanisms, namely TCP fair sharing [80, 121] as well as network sharing algorithms such as PS-P [124, 171, 172] in EC2 experiments.

In addition to quantitative results, we discuss challenges in scalability and fault-tolerance of centralized coordination-based solutions and present general techniques to address them. We also cover how the choice between a blocking and a non-blocking user-facing API – e.g., in Varys and Aalo, respectively – impact architectural design decisions and vice versa. Finally, throughout the dissertation, we enforce the generality of coflows in our assumptions and design decisions to ensure long-term adoption.

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides relevant background information. First, it surveys popular distributed data-parallel applications to identify common communication patterns. Next, it overviews advances in modern datacenter networks that enable us to model them as non-blocking fabrics. Finally, it summarizes existing solutions and emphasizes the necessity of application-aware networking.

Chapter 3 introduces coflows, categorizes them based on information availability, describes coflow-level objectives, and provides intuitions into why coflows would outperform flow-based solutions. Chapter 4 uses intra-coflow scheduling on two of the most common communication patterns. It presents a scalable algorithm for broadcast coflows as well as the optimal algorithm for shuffle coflows. Both improve performance by exploiting the fact that a coflow cannot complete until all its flows have completed.

Chapters 5, 6, and 7 address three inter-coflow scheduling problems using central coordination. Chapters 5 and 6 both focus on improving the average coflow completion time. However, the former uses complete information, while the latter performs almost as good even without complete information. Chapter 5 also describes how to support deadline-sensitive coflows, and introduces as well as characterizes the concurrent open shop scheduling with couple resources family of problems. Chapter 7 focuses on fair inter-coflow scheduling to provide performance isolation among coexisting coflows and generalizes single- and multi-resource fairness as well as multi-tenant network sharing under one unifying framework.

Finally, we summarize the broader impacts of this dissertation, discuss the lessons we have learned, and conclude by identifying directions for future work in Chapter 8.

Chapter 2

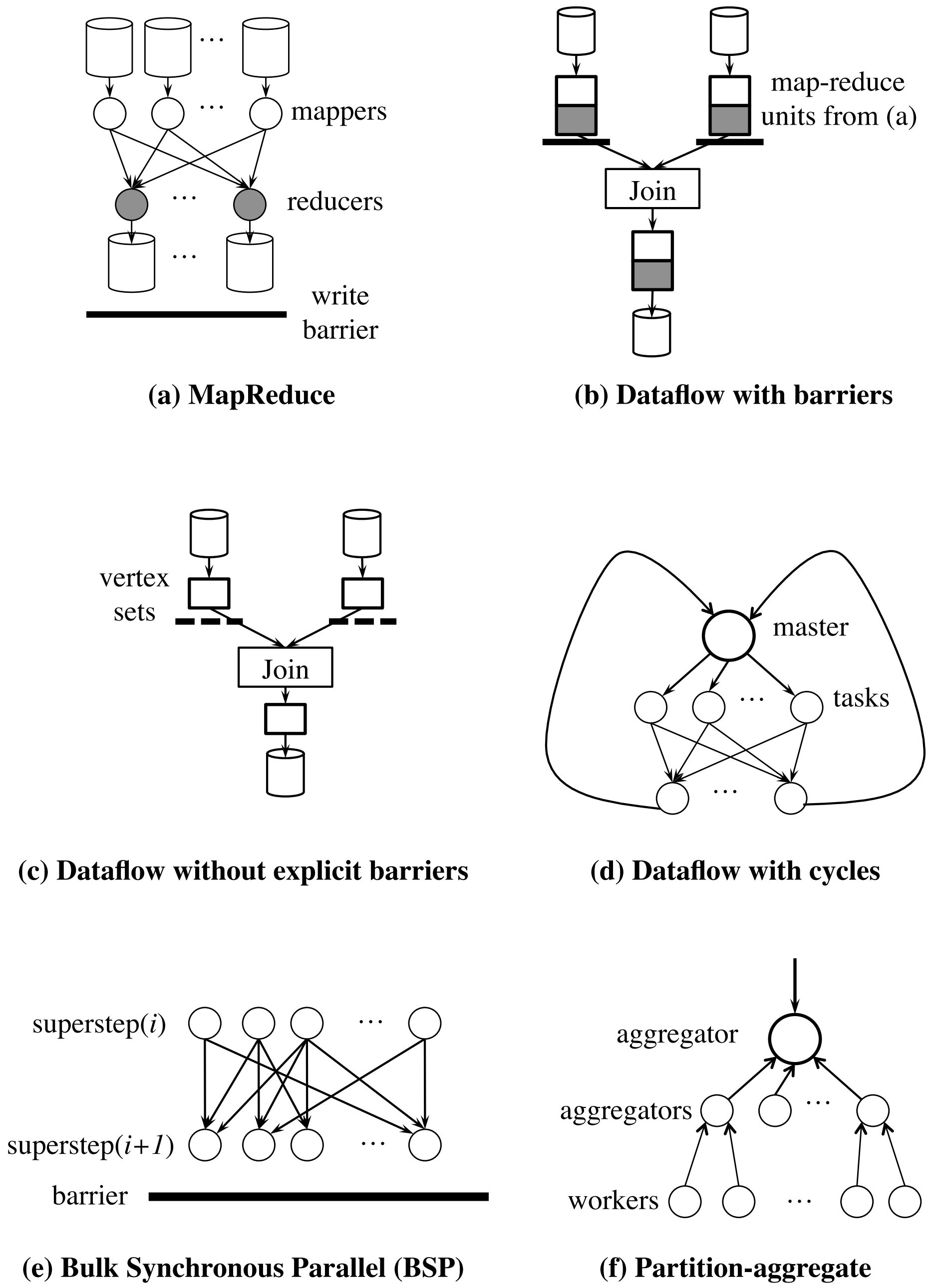
Background

Over the past decade, we have observed major transformations in how systems software and corresponding infrastructure are built. On the one hand, more and more applications (e.g., MapReduce) are scaling out to large clusters, taking advantage of I/O parallelism to deal with growing data volumes. On the other hand, concentrated efforts have gone into building high-capacity datacenters (e.g., Clos topologies) to support efficient execution of distributed data-parallel applications. In the process, the traditional point-to-point, client-server communication model has been superseded by higher-level communication patterns that involve multiple machines communicating with each other. In this chapter, we discuss the communication patterns that appear in modern scale-out applications, datacenter networks where these communications take place, and existing efforts in improving the performance of these patterns.

The remainder of this chapter is organized as follows. Section 2.1 discusses a wide variety of data-parallel applications – ranging from query processing and graph processing to search engines – and distills the commonalities among their communication requirements. Section 2.2 provides an overview of the infrastructure side. More specifically, it presents a taxonomy of modern datacenter network topologies along with forwarding, routing, and traffic engineering schemes used in those networks, and concludes by presenting the *non-blocking fabric* model of modern datacenter networks. Section 2.3 surveys the recent advances in optimizing the communication performance of distributed data-parallel applications and highlights the disconnect between application-level requirements and existing network-level optimizations. Section 2.4 concludes this chapter by summarizing our findings.

2.1 Communication in Distributed Data-Parallel Applications

Most distributed data-parallel applications are frameworks (e.g., MapReduce [77]) that take user-defined jobs and follow specific workflows enabled by corresponding programming models. Some others are user-facing pipelines, where user requests go through a multi-stage architecture to eventually send back corresponding responses (e.g., search results from Google or Bing, home feed in



(a) MapReduce

(b) Dataflow with barriers

(c) Dataflow without explicit barriers

(d) Dataflow with cycles

(e) Bulk Synchronous Parallel (BSP)

(f) Partition-aggregate

Figure 2.1: Communication patterns in distributed data-parallel applications: (a) Shuffle and CFS replication in MapReduce [77]; (b) Shuffle across multiple MapReduce jobs in dataflow pipelines that use MapReduce as a building block (e.g., Hive [8]); (c) Dataflow pipelines without explicit barriers (e.g., Dryad [119]); (d) Dataflow with cycles and broadcast support (e.g., Spark [208]); (e) Bulk Synchronous Parallel (e.g., Pregel [147]); (f) Aggregation during partition-aggregate communication in online services (e.g., user-facing backend of search engines and social networks).

Facebook). In this section, we study communication patterns of prevalent distributed data-parallel applications (Figure 2.1) and summarize their requirements.

We assume that tasks that participate in communication across computation stages are scheduled before communication takes place using one of the many available task schedulers for data-parallel frameworks [40, 120, 164, 207].

2.1.1 MapReduce

MapReduce [6, 77] is a well-known and widely used distributed data-parallel framework. In this model, each map tasks (mapper) reads its input from the cluster file system (CFS) [55,95], performs user-defined computations, and writes intermediate data to the disk. In the next stage, each reduce task (reducer) pulls intermediate data from different mappers, merges them, and writes its output to the CFS, which then replicates it to multiple destinations. The communication stage of MapReduce – i.e., the transferring of data from each mapper to each reducer – is known as the *shuffle*.

Given M mappers and R reducers, a MapReduce job will create a total of $M \times R$ flows for shuffle, and at least r flows for output replication in the CFS. The primary characteristic of communication in the MapReduce model is that a job will not finish until its last reducer has finished. Consequently, there is an explicit barrier at the end of the job.

2.1.2 Dataflow Pipelines

While MapReduce had been very popular throughout most of the past decade, it is not the most expressive of data-parallel frameworks. There exist a collection of general dataflow pipelines that address many deficiencies of MapReduce, and they have diverse communication characteristics. Typically, these frameworks exposes a SQL-like interface (e.g., Hive [8], DryadLINQ [206], SparkSQL [43]) to the users, and execute user queries using an underlying processing engine.

Dataflow with Barriers Some dataflow pipelines that support multiple stages use MapReduce as their building blocks (e.g., Sawzall [170], Pig [163], Hive [8]). Consequently, there are barriers at the end of each building block, and this paradigm is no different than MapReduce in terms of communication.

Dataflow without Explicit Barriers Several dataflow pipelines do not have explicit barriers and enable higher-level optimizations of the operators (e.g., Dryad [119], DryadLINQ [206], SCOPE [58], FlumeJava [59], MapReduce Online [73]); a stage can start as soon as some input is available. Because there is no explicit barrier, barrier-synchronized optimization techniques are not useful. Instead, researchers have focused on understanding the internals of the communication and optimizing for specific scenarios [111, 212].

Dataflow with Cycles Traditional dataflow pipelines unroll loops to support iterative computation requirements. Spark [208] obviates loop unrolling by keeping in-memory states across iterations. However, implicit barriers at the end of each iteration allow MapReduce-like communication

Model	Barrier Type	Barrier Reason	Loss Tolerance
MapReduce	Explicit	Write to CFS	None
Dataflow with Barriers	Explicit	Write to CFS	None
Dataflow w/o Explicit Barriers	Implicit	Input not ready	None
Dataflow with Cycles	Explicit	End of iteration	None
Bulk Synchronous Parallel	Explicit	End of superstep	None
Partition-Aggregate	Explicit	End of deadline	App. Dependent

Table 2.1: Summary of communication requirements in distributed data-parallel applications.

optimizations in cyclic dataflows [67]. These frameworks also explicitly support communication primitives such as broadcast, whereas, MapReduce-based frameworks rely of replicating to CFS and reading from the replicas to perform broadcast.

2.1.3 Bulk Synchronous Parallel (BSP)

Bulk Synchronous Parallel or BSP is another well-known model in distributed data-parallel computing. Examples of frameworks using this model include Pregel [147], Giraph [5], Hama [7], and GraphX [101] that focus on graph processing, matrix computation, and network algorithms. A BSP computation proceeds in a series of global supersteps, each containing three ordered stages: concurrent computation, communication between processes, and barrier synchronization. With explicit barriers at the end of each superstep, the communication stage can be globally optimized for the superstep.

However, sometimes complete information is not needed for reasonably good results; iterations can proceed with partial results. GraphLab [146] is such a framework for machine learning and data mining on graphs. Unlike BSP supersteps, iterations can proceed with whatever information is available as long as it converging; missing information can asynchronously arrive later.

2.1.4 Partition-Aggregate

User-facing online services (e.g., search results in Google or Bing, Facebook home feeds) receive requests from users and send it downward to the workers using an aggregation tree [54]. At each level of the tree, individual requests generate activities in different partitions. Ultimately, worker responses are aggregated and sent back to the user within strict deadlines – e.g., 200 to 300 milliseconds [35, 192, 200]. Responses that cannot make it within the deadline are either left behind [200] or sent later asynchronously (e.g., Facebook home feed).

2.1.5 Summary of Communication Requirements

Despite differences in programming models and execution mechanisms, most distributed data-parallel applications have one thing in common: *they run on a large number of machines that are* *organized into multiple stages or grouped by functionality*. Each of these groups communicate between themselves using a few common patterns (e.g., shuffle, broadcast, and aggregation) with a handful of objectives:

1. minimizing completion times,
2. meeting deadlines, and
3. fair allocation among coexisting entities.

While the former two objectives are prevalent in private datacenters, the latter is more common for cloud-hosted solutions such as Amazon Elastic MapReduce (EMR) [4].

Table 2.1 summarizes the key characteristics of the aforementioned distributed data-parallel applications in terms of the characteristics of barriers, their primary causes, and the ability of applications to withstand loss (i.e., whether all flows must complete or not) or delay. Note that approximation frameworks such as BlinkDB [31] perform sampling *before* running jobs on sampled datasets. Consequently, their communication stages also rely on all the flows completing.

2.2 Datacenter Networks

The proliferation of distributed data-parallel computing has coincided with and even enabled by the rise of public cloud computing [3,15,20] and private datacenters [48,106]. Consequently, communication in these applications invariably rely on underlying datacenter networks. In this section, we provide a quick overview of modern datacenter topologies and mechanisms used in different layers, and conclude by introducing the non-blocking fabric model of a datacenter network.

2.2.1 Topologies

Typical datacenters consist of tens to hundreds of thousands of machines [48] and require high-capacity networks between to them to ensure high-performance distributed applications [106,158]. While oversubscription was common in early three-tier, tree-based topologies, modern datacenter networks have shifted toward high and even full bisection bandwidth [88,106,108,110,158,199].[1]

Modern datacenter topologies can broadly be divided into four categories:

1. **Switch-Centric** topologies use switches as internal nodes of a datacenter network, which connect servers at the leaves. PortLand [158] and VL2 [106] are two well-known examples of this approach. In broad strokes, they use a special instance of a Clos topology [69] to interconnect commodity Ethernet switches to emulate a fat tree [143] and provide full bisection bandwidth. Switch-centric topologies are arguably the most widely-deployed datacenter topologies today, with deployments at Amazon, Microsoft, and Google [106,186].

[1] *Bisection bandwidth* is the maximum amount of bandwidth in a data center is measured by bisecting the network graph at any given point. In a full bisection bandwidth network, machines in one side of the bisection can use their entire bandwidth while communicating with the machines on the other side.

2. **Server-Centric** topologies use servers as relay nodes in addition to switches. BCube [108], DCell [110], and CamCube [29] are a few representative examples. While these topologies empirically provide high bisection bandwidth, determining their exact bisection bandwidth remain open in many cases [42].

3. **Hybrid Electrical/Optical** designs combine fast switching speeds of electrical switches with high-bandwidth optical circuits to build scale-out datacenter networks. Examples include c-Through [199], Helios [88], and OSA [61]. In these designs, top-of-the-rack (ToR) switches are simultaneously connected to both networks through an electrical and an optical port. Followups of hybrid designs (e.g., Mordia [173]) often focus on low-latency switching in the optical part to allow small-to-medium flows use the available bandwidth.

4. **Malleable** topologies rely on wireless and optics [114, 115, 214] to flexibly provide high-bandwidth in different paths instead of creating full bisection bandwidth networks. The key challenge in this case is again fast circuit creation in presence of dynamic load changes.

Apart from these, an interesting design point is random topologies – most notably Jellyfish [187] – that argue for randomly wiring components to achieve high bisection bandwidth.

In practice, however, Clos topology is the primary datacenter topology today: a recent report from Google (circa 2015) suggests that it is indeed possible to build full-bisection bandwidth networks with up to $100,000$ machines, each with 10 GbE NICs, for a total capacity of 1 Pbps [186].

2.2.2 Forwarding, Routing, Load Balancing, and Traffic Engineering

Regardless of design, modern datacenter topologies invariably provide multiple paths between each pair of machines for fault tolerance and better load balancing. Forwarding and routing in datacenters are topology-dependent, each with its own sets of pros and cons [60].

A more active area of research, however, is load balancing and traffic engineering across multiple paths. The traditional Equal Cost MultiPath (ECMP) load balancing suffers from significant imbalance due to hash collisions in presence of a few large flows [33, 34, 51]. This is primarily due to the fact that ECMP uses only local information. Hedera [33] and MicroTE [51] take a centralized traffic engineering approach to overcome ECMP's shortcomings. MPTCP [177] takes a host-based, transport layer approach for better load balancing, whereas CONGA [34] takes an in-network approach. There are many other proposals in between these extremes that we do not elaborate on for brevity.

The ideal goal of all these proposals is to restrict possibilities of congestion toward the edge of full bisection bandwidth networks. This decouples host-level network resource management from the network itself and allow concurrent innovation in both areas.

2.2.3 The Non-Blocking Fabric Model

Given the focus on building Clos-based full bisection bandwidth networks and restricting congestion to network edges, we can abstract out the entire datacenter network as one *non-blocking fabric*

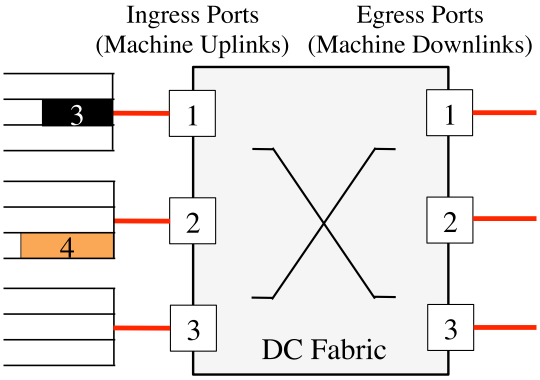


Figure 2.2: A 3×3 datacenter fabric with three ingress/egress ports corresponding to the three machines connected to it. Flows in ingress ports are organized by destinations. Links connecting machines to the network (highlighted in red/bold) are the only points of congestion; the network itself is a black-box.

or switch [34,37,46,68,86,129] and consider machine uplinks and downlinks, i.e., machine NICs, as the only sources of contention. It is also known as the *hose model* representation.

In this model, each ingress port (uplink) has some flows for various egress ports (downlinks). For ease of exposition, we organize them in Virtual Output Queues [151] at the ingress ports as shown in Figure 2.2. In this case, there are two flows transferring 3 and 4 units of data from machine-1 to machine-2 and from machine-2 to machine-3, respectively.

This model is attractive for its simplicity, and recent advances in datacenter fabrics [34, 106, 158, 186] make it practical as well. However, we use this abstraction only to simplify our analyses; we do not require nor enforce this in our evaluations.

2.3 Optimizing Communication of Data-Parallel Applications

Acknowledging the importance of communication on the performance of distributed data-parallel applications [35,67,200] – e.g., Facebook MapReduce jobs spends a quarter of their runtime in the shuffle stage on average (please refer to Chapter A for more details) – researchers and practitioners have proposed many solutions optimized for datacenter and cloud environments. All of them work on flows under the assumption that if tail-latency decreases or flows complete faster, applications will experience performance improvements. Research in this direction have looked at dropping flows [192,200], prioritization/preemption [37,118], and cutting long tails [35,211]; however, they do not exploit application-level information about the collective nature of such communication.

In this section, we summarize these approaches by dividing them into three broad categories based on the application-level information they do leverage. We also discuss why point-to-point flows are insufficient for capturing the collective communication requirements of data-parallel applications.

2.3.1 Traditional Size- and Deadline-Agnostic Approaches

Today's datacenter networks run primarily on the TCP/IP stack using point-to-point flows between two machines [35]. However, it is now well established that TCP is not the most suitable protocol for datacenters [35–37,118,192,200,201,211], mainly because TCP was designed for the Internet under a different set of assumptions than datacenters.

To make TCP more suitable for datacenters, a large body of solutions – mostly transport layer (e.g., DCTCP [35], ICTCP [201]) with a few cross-layer (e.g., DeTail [211], HULL [36]) – have been proposed in recent years. Most of them focus on making TCP faster in reacting to congestions – thus, reducing tail latency – while maintaining fair sharing across coexisting flows. Given that none of these proposals use any information about flow size or flow deadline, fairness is the obvious objective. PIAS [45] is unique in that, unlike the rest, it directly focuses on reducing flow completion time in a non-clairvoyant manner.

2.3.2 Size-Aware Approaches

Instead of indirectly impacting flow completion times through reductions in tail latency and faster reactions to congestions, some proposals just assume that flow sizes are known. Given flow size information, PDQ [118] and pFabric [37] enforce shortest-flow-first prioritization discipline throughout the datacenter. The intuition is the following: if flows complete faster, the communication stages will complete faster and communication with deadlines will complete within their deadlines. While decreasing flow completion times, they do not distinguish between the applications that are generating these flows.

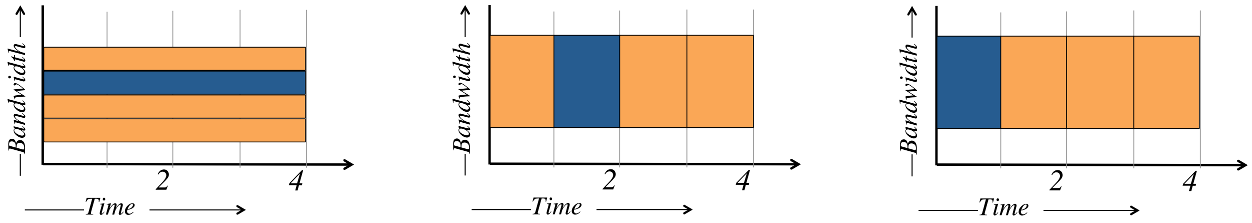
2.3.3 Deadline-Aware Approaches

Another approach for increasing predictability in datacenters is using explicit knowledge of deadlines. Deadlines can be known directly from the applications (e.g., partition-aggregate traffic from online services often have tens to hundreds of milliseconds deadlines [35,200]). Given these deadlines, most solutions reserve capacities throughout the entire path (e.g., PDQ [118], Silo [123]), while some use a combination of reservation, congestion notification, and end host rate limiting (e.g., D^3 [200], D2TCP [192]). When flows miss deadlines, most of these proposals de-prioritize them in the network; however, some have proposed completely quenching those flows [200].

2.3.4 The Need for Application-Awareness

Point-to-point flows are fundamentally independent. Below, we explain using a simple example why *independent flows inherently cannot capture the collective communication requirements of data-parallel applications*.

Consider two communication stages from two applications – application-1 with three flows and application-2 with one flow – on a single bottleneck link. The link has unit capacity, and each



(a) Size- and deadline-agnostic

(b) Size- or deadline-aware

(c) The optimal schedule

Figure 2.3: Three flows from application-1 (orange/light) and one flow from application-2 (blue/dark) on a single bottleneck link of unit capacity. Each flow has one unit of data to send and a deadline of one time unit. (a) Fair sharing of flows without size or deadline information. (b) One of the 4 possible smallest-flow-first or earliest-deadline-first schedules. (c) The optimal schedule with application-level information.

flow has one data unit to send and a deadline of one time unit. We see that the lowest average flow completion time is 2.5 time units, and at most one flow can complete within deadline.

Clearly, fair sharing will miss both goals (Figure 2.3a) by treating everyone equally. The average flow completion time is 4 time units and no flows complete within deadlines.

As we have discussed, recent proposals aim to circumvent these shortcomings by using additional information such as flow size and flow deadline. Figure 2.3b shows that they would think they've been successful. The average flow completion time is indeed 2.5 time units due to shortest-flow-first, and one flow has completed within the deadline of one time unit.

The question remains: *do applications see any benefit from all these improvements?* Unfortunately, no. Figure 2.3c is the only schedule where an application has completed within its deadline (i.e., application-2), and the average completion time of the communication stages of these two applications is 2.5 time units. In contrast, both applications failed to meet their deadlines in both Figure 2.3a and Figure 2.3b, and the average communication completion times are 4 and 3 time units, respectively.

2.4 Summary

There are three high-level takeaways from this chapter.

1. Despite diverse end goals, distributed data-parallel applications use a common set of multipoint-to-multipoint communication patterns with a handful of optimization objectives.

2. Modern datacenter networks actively attempt to push possibilities of contentions from inside the network fabric to end hosts at the edges, so much so that we can effectively consider them to be non-blocking fabrics.

3. Existing techniques for optimizing communication performance of datacenter applications are unaware of application-level requirements. Consequently, they rarely improve application-level performance and can even hurt applications.

In this dissertation, we develop a generalized abstraction for communication in datacenter applications, and we use it to enable application-aware network scheduling for *all* the application-level objectives we have identified. In the next chapter, we present the core concept of our solution – the coflow abstraction.

Chapter 3

Coflows and Their Applications

The key concept behind this dissertation is one simple observation: a communication stage in a distributed data-parallel application cannot complete until *all* its flows have completed. Surprisingly, the networking literature does not provide any construct to express such collective requirements. Specifically, the traditional point-to-point flow abstraction is fundamentally independent and cannot capture a semantics, where the collective behavior of all the flows between two groups of machines is more important than that of any individual flow. This lack of an abstraction has several consequences: (i) it promotes point solutions with limited applicability; (ii) it results in solutions that are not optimized for the appropriate objectives; and (iii) without an abstraction, it remains difficult to reason about the underlying principles and to anticipate problems that might arise in the future.

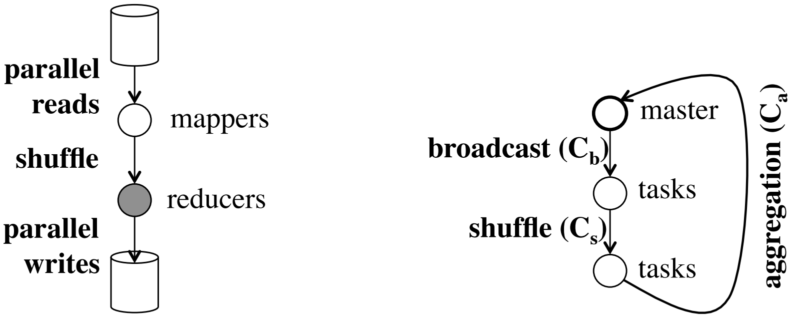
In this chapter, we introduce the *coflow* abstraction to capture diverse communication patterns and corresponding objectives that we observed in distributed data-parallel applications (§2.1). The chapter is organized in two parts. Section 3.1 defines the coflow abstraction, identifies different categories of coflows and their characteristics, and summarizes common coflow-level objectives that can be of interest to datacenter operators. Section 3.2 demonstrates with a simple example how coflows enable significant application-level improvements that traditional application-agnostic techniques simply cannot even for a single communication pattern. Finally, Section 3.3 summarizes our findings and presents an outline of how we use coflows to perform different optimizations in subsequent chapters.

3.1 The Coflow Abstraction

Although individual point-to-point flows are indistinguishable at the transport layer, as we have seen in Chapter 2, flows between groups of machines in a distributed data-parallel application have application-level semantics. For example, the last flow in a shuffle determines its completion time. Similarly, a delayed flow can cause an entire partition-aggregate communication pattern to miss its deadline. In this section, we introduce the coflow abstraction that captures such collective semantics and enables application-aware networking in datacenters.

Communication in Data-Parallel Apps	Coflow Structure
Communication in dataflow pipelines [6, 8, 119, 208]	Many-to-Many
Global communication barriers [147, 191]	All-to-All
Broadcast [6, 119, 208]	One-to-Many
Aggregation [6, 8, 77, 119, 208]	Many-to-One
Parallel read/write on distributed storage [55, 58, 63, 145]	Many One-to-One

Table 3.1: Coflows in distributed data-parallel applications.



(a) MapReduce ***(b) Spark***

Figure 3.1: Graphical representations of data-parallel applications using coflows. Circles represent parallel tasks and edges represent coflows.

3.1.1 What is (in) a Coflow?

A *coflow* is a collection of flows that share a common performance goal, e.g., minimizing the completion time of the latest flow or ensuring that flows meet a common deadline. The flows of a coflow are independent in that the input of a flow does not depend on the output of another in the same coflow, and the endpoints of these flows can be in one or more machines. Examples of coflows include the shuffle between the mappers and the reducers in MapReduce [77] and the communication stage in the bulk-synchronous parallel (BSP) model [191]. Coflows can express most communication patterns between successive computation stages of data-parallel applications (Table 3.1) [65]. Note that traditional point-to-point flow is still a coflow with just a single flow.

We define the *completion time of a coflow* or *CCT* as the time duration between the beginning of its first flow and the completion of its last.

Using coflows as building blocks, we can now represent any distributed data-parallel pipeline as a sequence of machine groups connected by coflows. Figure 3.1 depicts the MapReduce and Spark jobs in Figure 2.1(a) and Figure 2.1(d) using four different coflow patterns.

Formally, each coflow $C(\mathbf{D})$ is a collection of flows over the datacenter fabric (§2.2.3) with P ingress and P egress ports, where the $P \times P$ matrix $\mathbf{D} = [d_{ij}]_{P\times P}$ represents the structure of C.

For each non-zero element $d_{ij} \in \mathbf{D}$, a flow f_{ij} transfers d_{ij} amount of data from the ith ingress port (P_i^{in}) to the jth egress port (P_j^{out}).

3.1.2 Coflow Categories Based on Information Availability

Depending on the availability of information about a coflow's structure (e.g., the number of flows, their endpoints, flow sizes, start times etc.), there can be two primary types of coflows.

1. **Clairvoyant Coflows:** We refer to a coflow to be clairvoyant when $C(\mathbf{D})$ is known before the coflow starts and it does not change over time. Applications that write their data to disk before transferring to the next stage typically create coflows that fall in this category. A large body of solutions [37,67,68,82,118] work under this assumption. For clairvoyant coflows, a broadcast has only one non-zero row and an aggregation has one non-zero column in $C(\mathbf{D})$.

2. **Non-clairvoyant Coflows:** When a coflow's structure and relevant information are not completely known when it coflow starts or $C(\mathbf{D})$ can change dynamically, we consider it to be a non-clairvoyant coflow. This can happen due to multi-wave scheduling [39], in multi-level DAGs with push-forward pipelining [9, 73, 119, 181], or due to task failures, restarts, and speculative execution. Aalo [66] works on non-clairvoyant coflows.

3.1.3 Objectives

For an individual coflow, the objective is always to minimize its own CCT. However, data-parallel clusters are often shared, and cluster operators can have higher-level across-cluster objectives. Depending on the availability of information, there can be three primary objectives inter-coflow resource sharing in a shared cluster.

1. *Minimizing the average CCT* aims to finish coflows faster on average and, thus, decrease job completion times. This is a viable objective for both clairvoyant [68] and non-clairvoyant [66,67,82] coflows.

2. *Meeting deadlines* attempts to maximize the number of coflows that meet their deadlines and makes sure they succeed by employing admission control. Hard deadlines can be supported only for clairvoyant coflows.

3. *Fair sharing between coflows* enables sharing between coflows at the risk of increasing the average CCT. Unlike flow-level fair sharing on individual links, coflow-level fair sharing requires considering the entire fabric. It is viable for both types of coflows.

In this dissertation, we develop solutions for optimizing all three objectives independently. Note that some of the objectives can be combined as well. For example, assuming two priority levels with deadline-sensitive coflows having the higher priority, one can try to first ensure timely completion of deadline-sensitive coflows and then try to minimize the average CCT for coflows that do not have any deadlines.

3.1.4 Composability

One of the most important characteristic of coflows is that they are composable. Meaning, one can combine multiple coflows to create a new coflow or break one apart into many coflows, depending on their objectives. For example, one can consider a shuffle with M mapper and R reducers an $M \times R$ shuffle coflow or R $M \times 1$ aggregation coflows. The former would directly impact job completion times, whereas the latter would impact task completion times.

Dependencies The importance of composability becomes more prominent when consider coflow dependencies in a multi-stage dataflow pipeline.

Figure 3.1(b) shows an example. The aggregation coflow (C_a) of each iteration depends on the shuffle coflow (C_S), which, in turn, depends on the broadcast (C_b). In the absence of explicit barriers between task stages, all three can coexist. However, we can see that C_a cannot complete before C_s and C_s before C_b. If we consider them in isolation, we can end up with priority inversion.

Instead, we can compose another coflow C capturing the *finishes-before* dependencies between all the coflows from the same application (C_b, C_s, and C_a), and allow the operator to optimize C as a whole. Dependency information is used only to break internal ties. Indeed, not composing coflows into a larger one can hurt application-level performance instead of improving it [66].

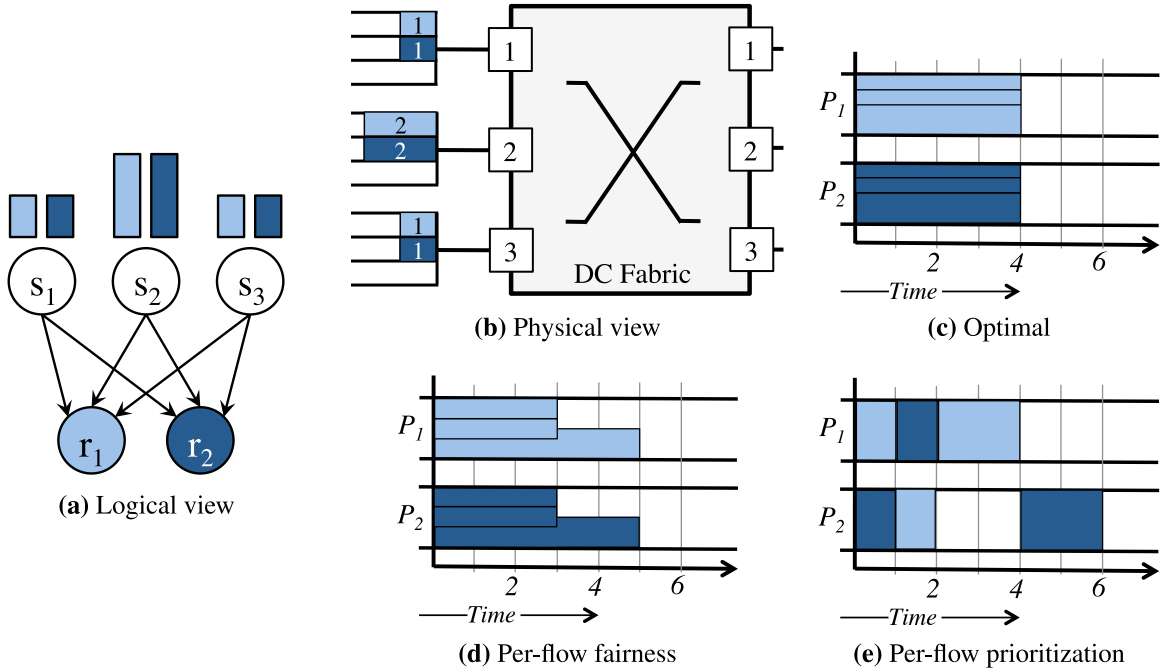
3.2 Benefits of Coflows

In this section, we demonstrate the benefits of coflows over the traditional flow abstraction using a simple example that involves just one communication pattern. Throughout the rest of this dissertation, we present more applications of coflows in diverse settings and objectives.

3.2.1 Comparison to Per-Flow Fairness

In current systems, the flows between senders and receivers experience unweighted fair sharing due to TCP. This can be suboptimal when the flows must transfer different amounts of data. For example, consider the shuffle in Figure 3.2, where senders s_1 and s_2 have one unit of data for each receiver and s_3 has two units for both. Under fair sharing (Figure 3.2d), each receiver starts fetching data at $1/3$ units/second from the three senders. After 3 seconds, the receivers exhaust the data on senders s_1 and s_2, and there is one unit of data left for each receiver on s_3. At this point, s_3 becomes a bottleneck, and the receivers take 2 more seconds to transfer the data off, completing the shuffle in 5 seconds. In contrast, in the optimal schedule (Figure 3.2c), the receivers would fetch data at a rate of $1/4$ units/second from s_1 and s_2 and $1/2$ units/second from s_3, finishing in 4 seconds (i.e., $1.25\times$ faster).

We present the algorithm to achieve the optimal schedule, *Minimum Allocation for Desired Duration (MADD)*, in Section 4.5.



(a) Logical view

(b) Physical view

(c) Optimal

(d) Per-flow fairness

(e) Per-flow prioritization

Figure 3.2: A 3×2 shuffle demonstrating the drawbacks of coflow-agnostic schemes. (a) The two receivers (at the bottom) need to fetch separate pieces of data from each sender, with the one sending twice as much as the rest. (b) The same shuffle on a 3×3 datacenter fabric with three ingress/egress ports. Flows in ingress ports are organized by destinations and color-coded by receivers. (c)–(e) Allocation of *egress* port capacities (vertical axis) using different mechanisms, where each port can transfer one unit of data in one time unit. Corresponding shuffle completion times for (c) the optimal schedule is 4 time units; (d) per-flow fairness is 5 time units; and (e) per-flow prioritization is 6 time units.

3.2.2 Comparison to Per-Flow Prioritization

A large body of recent work (e.g., pFabric [37], PDQ [118]) focus on flow-level prioritization and schedule smaller flows first to minimize flow completion times (FCT), regardless of application- or coflow-level objectives. Figure 3.2e demonstrates the drawback of these approaches. By focusing on individual flows' completion times, they lose the bigger picture: the shuffle completion time in our example using per-flow prioritization (6 seconds) is $1.5\times$ slower than the optimal schedule. This is despite the fact that the average FCT for per-flow prioritization (2.67 seconds) is smaller than both the optimal schedule (4 seconds) and per-flow fairness (3.67 seconds).

3.3 Summary

In this chapter, we have introduced, defined, categorized, and characterized the coflow abstraction. In addition, we have demonstrated how the already available application-level information can help in improving end-to-end communication performance despite an apparent loss of network-level performance in terms of application-agnostic metrics.

In the subsequent chapters, we show coflows in action; i.e., how coflows can improve the communication performance of individual applications (Chapter 4) and across multiple applications for clairvoyant (Chapter 5) as well as for non-clairvoyant coflows (Chapter 6). Finally, in Chapter 7, we demonstrate how to fairly divide the datacenter network among coexisting coflows.

Chapter 4

Intra-Coflow Scheduling

In this chapter, we present the first application of the coflow abstraction and demonstrate how coflow awareness can result in faster communication stages – and, in turn, faster end-to-end completion of jobs – for distributed data-parallel applications. Unlike traditional application-agnostic approaches, the proposed solutions leverage a simple observation that is embedded in the definition of a coflow – *a communication stage cannot complete until all its flows have completed*. We apply the observation to two of the most common coflows: shuffle and broadcast. Incidentally, the former is a clairvoyant coflow, where the communication matrix is known a priori, and the latter is a non-clairvoyant coflow, where flows are dynamically added because of its BitTorrent-like implementation. To the best of our knowledge, this is the first solution to exploit the *all-or-nothing* property of communication conveyed through coflows.

The rest of this chapter is organized as follows. The next section discusses the importance of communication in distributed data-parallel applications and the state-of-the-art in addressing relevant challenges. Section 4.2 provides an outline of our approach using coflows as building blocks. Section 4.3 presents two real-world examples to quantitatively illustrate the importance of communication in big data workloads. Section 4.4 presents our broadcast scheme, Cornet, and Section 4.5 presents the optimal shuffle scheduling algorithm, MADD. We then evaluate the proposed algorithms in Section 4.6, survey related work in Section 4.7, and summarize our findings in Section 4.8.

4.1 Background

The past decade has seen a rapid growth of distributed data-parallel applications to analyze and make sense of growing volumes of data collected and generated by user activities on web services such as Google, Facebook, and Yahoo!. With the advent of cloud computing, even small startups are dealing with very large datasets using distributed data-parallel applications. As discussed in Chapter 2, these frameworks (e.g., MapReduce [77], Dryad [119], and Spark [208]) typically implement a data flow computation model, where datasets pass through a sequence of processing stages.

Many of the jobs executed using these frameworks manipulate massive amounts of data and run on clusters consisting of as many as tens of thousands of machines. Due to the very high cost of these clusters, operators aim to maximize the cluster utilization, while accommodating a variety of applications, workloads, and user requirements. To achieve these goals, many solutions have been proposed to reduce job completion times [40, 120, 207, 210], accommodate interactive workloads [120,164,207], and increase utilization [78,79,117,195]. While in large part successful, these solutions primarily focus on scheduling and managing computation and storage resources, while mostly ignoring the network.

However, managing and optimizing network activity is critical for improving job performance. Indeed, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 25% of the running times of jobs with reduce stages (see Appendix A for more details). Section 4.3 presents two real-world machine learning applications – a spam classification algorithm and a collaborative filtering job – illustrating even bigger impacts because of their iterative nature.

Despite the impact of communication, until recently, researchers and practitioners have largely overlooked application-level requirements when improving network-level metrics such as flow-level fairness and flow completion time (FCT). Existing approaches toward improving communication performance can be categorized into two broad classes: (i) increasing datacenter bandwidth and (ii) decreasing flow completion times. However, proposals for full bisection bandwidth networks [106, 108, 110, 158] along with flow-level scheduling [33, 37, 45, 51, 118, 200] can only improve network-level performance, but they do not account for collective behaviors of flows due to the lack of job-level semantics. Worse, this mismatch often hurts application-level performance, even when network-oriented metrics such as FCT or fairness improve (§3.2).

The coflow abstraction bridges this gap by exposing application-level semantics to the network. It builds upon the *all-or-nothing* property observed in many aspects of data-parallel computing such as task scheduling [40, 210] and distributed cache allocation [39]; for the network, it means all flows must complete for the completion of a communication stage. In this chapter, we exploit coflow-level knowledge to improve the performance of two common communication patterns – shuffle and broadcast – that occur in virtually all distributed data-parallel applications and are responsible for a bulk of the network traffic in these clusters (see Appendix A for a breakdown of contributions the major traffic sources). Shuffle captures the many-to-many communication pattern between the map and reduce stages in MapReduce, and between Dryad's stages. Broadcast captures the one-to-many communication pattern employed by iterative optimization algorithms [215] as well as fragment-replicate joins in Hadoop [13].

4.2 Solution Outline

Our solutions for broadcast and shuffle are rooted in the all-or-nothing property of these coflows, and we rely on *centralized coordination* to exploit this property. For broadcasts, we propose an algorithm that implements a BitTorrent-like protocol optimized for datacenters. We refer to this as Cornet and augment it using an adaptive clustering algorithm to take advantage of the hierarchical network topology in many datacenters. The key difference between Cornet and traditional BitTorrent, however, is that whenever a participant is slow, Cornet attempts to accelerate its completion instead of throttling it – the slowest one is the one that matters the most. For shuffles, we propose an optimal algorithm called Minimum Allocation for Desired Duration (MADD) that explicitly sets data-proportional rates, ensuring that all the flows finish together with the slowest flow.

The proposed solutions can be implemented at the application layer and overlaid on top of diverse routing topologies [33, 106, 110, 158], access control schemes [56, 107], and virtualization layers [169, 171, 184]. We believe that this implementation approach is both appropriate and attractive for two primary reasons. First, algorithms are easier to implement into the central masters of the high-level programming frameworks (e.g., MapReduce). Second, it allows for faster deployment without modifying routers and switches, and even in the public cloud.

We built prototype implementations of the proposed algorithms in Apache Spark [208] and conducted experiments on DETERlab and Amazon EC2. Our experiments show that Cornet is up to $4.5\times$ faster than the default Hadoop implementation, while MADD can speed up shuffles by 29%. Moreover, they reduced communication times by up to $3.6\times$ and job completion times by up to $1.9\times$ for the aforementioned machine learning applications.

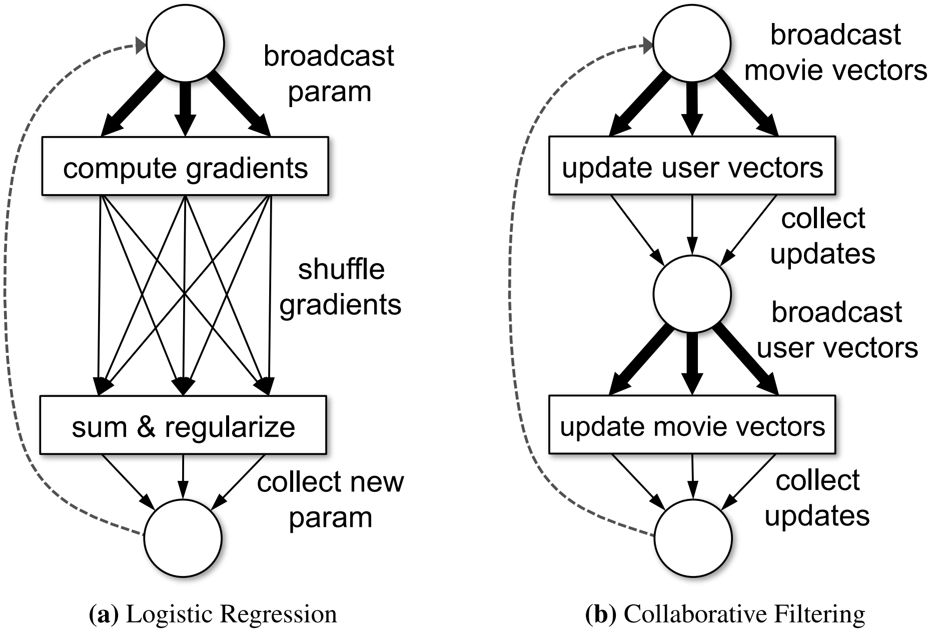
4.3 Motivating Applications

To motivate the importance of communication, in this section, we study two applications implemented using Spark that involve broadcast and shuffle coflows: a *logistic regression* implementation for identifying spam on Twitter [189] and a *collaborative filtering* algorithm for the Netflix Challenge [215].

4.3.1 Logistic Regression

As an example of an iterative MapReduce application in Spark, we consider Monarch [189], a system for identifying spam links on Twitter. The application processed 55 GB of data collected about $345,000$ tweets containing links. For each tweet, the group collected 1000-2000 features relating to the page linked to (e.g., domain name, IP address, and frequencies of words on the page). The dataset contained 20 million distinct features in total. The application identifies which features correlate with links to spam using logistic regression [116].

We depict the per-iteration workflow of this application in Figure 4.1a. Each iteration includes a large broadcast (300 MB) and a shuffle (190 MB per reducer) operation; it typically takes the application at least 100 iterations to converge. Each coflow acts as a barrier: the job is held up by the slowest machine to complete. In our initial implementation of Spark, which used the same broadcast and shuffle strategies as Hadoop, we found that communication accounted for 42% of the iteration time, with 30% spent in broadcast and 12% spent in shuffle on a 30-machine cluster. With such a large fraction of the running time spent on communication, optimizing the completion times of these coflows is critical.



(a) Logistic Regression **(b)** Collaborative Filtering

Figure 4.1: Per-iteration workflow diagrams for our motivating machine learning applications. The circle represents the master machine and the boxes represent the set of worker machines.

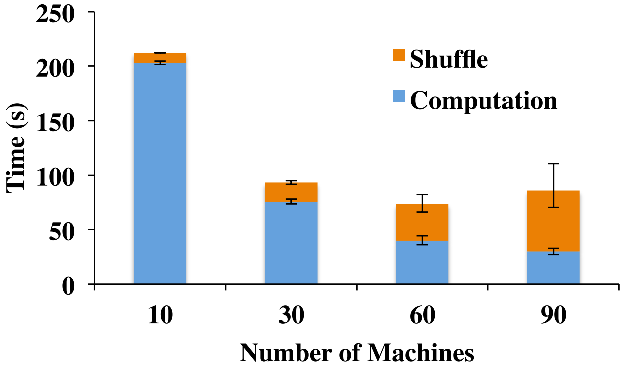


Figure 4.2: [EC2] Communication and computation times per iteration when scaling the collaborative filtering job using HDFS-based broadcast.

4.3.2 Collaborative Filtering

As a second example of an iterative algorithm, we discuss a collaborative filtering job used for the Netflix Challenge data. The goal is to predict users' ratings for movies they have not seen based on their ratings for other movies. The job uses an algorithm called alternating least squares (ALS) [215]. ALS models each user and each movie as having K features, such that a user's rating for a movie is the dot product of the user's feature vector and the movie's. It seeks to find these vectors through an iterative process.

Figure 4.1b shows the workflow of ALS. The algorithm alternately broadcasts the current user or movie feature vectors, to allow the machines to optimize the other set of vectors in parallel. Each broadcast is roughly 385 MB. These broadcasts limited the scalability of the job in our initial implementation of broadcast, which was through shared files in the Hadoop Distributed File System (HDFS) – the same strategy used in Hadoop. For example, Figure 4.2 plots the iteration times for the same problem size on various numbers of machines. Computation time goes down linearly with the number of machines, but communication time grows linearly. At 60 machines, the broadcasts cost 45% of the iteration time. Furthermore, the job stopped scaling past 60 machines, because the extra communication cost from adding machines outweighed the reduction in computation time (as can be seen at 90 machines).

4.4 The Broadcast Coflow

Data-intensive applications often need to send large pieces of data to multiple machines. For example, in the collaborative filtering algorithm in Section 4.3, broadcasting an $O(100\text{ MB})$ parameter vector quickly became a scaling bottleneck. In addition, distributing files to perform a fragment-replicate join[1] in Hadoop [13], rolling out software updates [21], and deploying VM images [19] are some other use cases where the same data must be sent to a large number of machines.

In this section, we discuss current mechanisms for implementing broadcast in datacenters and identify several of their limitations (§4.4.1). We then present Cornet, a BitTorrent-like protocol designed specifically for datacenters that can outperform the default Hadoop implementation by $4.5\times$ (§4.4.2). Lastly, we present a topology-aware variant of Cornet that leverages global control to further improve performance by up to $2\times$ (§4.4.3).

4.4.1 Existing Solutions

One of the most common broadcast solutions in existing data-intensive applications involves writing the data to a shared file system (e.g., HDFS [6], NFS) and reading it later from that centralized storage. In Hadoop, both Pig's fragment-replicate join implementation [13] and the DistributedCache API for deploying code and data files with a job use this solution. This is likely done out of a lack of other readily available options. Unfortunately, as the number of receivers grows, the centralized storage system can quickly become a bottleneck, as we observed in Section 4.3.

To eliminate the centralized bottleneck, some systems use d-ary distribution trees rooted at the source machine. Data is divided into blocks that are passed along the tree. As soon as a machine finishes receiving the complete data, it can become the root of a separate tree. d is sometimes set to 1 to form a chain instead of a tree (e.g., in LANTorrent [19] and in the protocol for writing blocks in HDFS [6]). Unfortunately, tree and chain schemes suffer from two limitations. First, in a tree with $d > 1$, the sending capacity of the leaf machines (which are at least half the machines) is not utilized. Second, a slow machine or link will slow down its entire subtree, which is problematic at large scales due to the prevalence of stragglers [77].

[1] This is a join between a small table and a large table where the small table is broadcasted to all the map tasks.

Unstructured data distribution mechanisms such as BitTorrent [72], traditionally used in the Internet, address these drawbacks by providing scalability, fault-tolerance, and high throughput in heterogeneous and dynamic networks. Recognizing these qualities, Twitter has built Murder [21], a wrapper over the BitTornado [10] implementation of BitTorrent, to deploy software to its servers.

4.4.2 Cornet: BitTorrent for Datacenters

Cornet is a BitTorrent-like protocol optimized for datacenters. In particular, Cornet takes advantage of the cooperative nature of a cluster: i.e., high-speed and low-latency connections, the absence of selfish peers, and the fact that there is no malicious data corruption. By leveraging these properties, Cornet can outperform BitTorrent implementations for the Internet by up to $4.5\times$.

Cornet differs from BitTorrent in three main aspects:

- Unlike BitTorrent, which splits files into blocks and subdivides blocks into small chunks with sizes of up to 256 KB, Cornet only splits data into large blocks (4 MB by default).
- While in BitTorrent some peers do not contribute to the broadcast and leave as soon as they finish the download, in Cornet, each machine contributes its full capacity over the full duration of the broadcast. Thus, Cornet does not include a tit-for-tat scheme or choking/unchoking mechanisms to incentivize machines [71].
- Cornet does not employ expensive SHA1 operations on each data block to ensure data integrity; instead, it performs a single integrity check over the whole data.

Cornet also employs a cap on the number of simultaneous connections to improve performance.[2] When a peer is sending to the maximum number of recipients, it puts further requests into a queue until one of the sending slots becomes available. This ensures faster service times for the small number of connected peers and allows them to finish quickly to join the session as the latest sources for the blocks they just received.

During broadcast, receivers explicitly request for specific blocks from their counterparts. However, during the initial stage, the source of a Cornet broadcast sends out at least one copy of each block in a round-robin fashion before duplicating any block.

Similar to a BitTorrent tracker, Cornet includes a controller that assigns a set of peers to each machine. However, unlike BitTorrent, each machine requests new peers every second. This coordination allows Cornet to adapt to network topologies and to optimize coflow completion times.

4.4.3 Topology-Aware Cornet

Many datacenters employ hierarchical network topologies with oversubscription ratios as high as 10 [38, 48], where transfer times between two machines on the same rack are significantly lower than between machines on different racks. To take network topology into account, we have developed two extensions to Cornet.

[2] The default limits for the number of receive and send slots per machine are 8 and 12, respectively.

CornetTopology In this case, we assume that the network topology is known in advance, which is appropriate, for example, in private datacenters. In CornetTopology, we use the configuration database that specifies locality groups, e.g., which rack each machine is in. When a receiver requests for a new set of peers, instead of choosing among all possible recipients (as in vanilla Cornet), we prioritize machines on the same rack as the receiver. Essentially, each rack forms its individual swarm with minimal cross-rack communication. The results in Section 4.6.2 show that CornetTopology can reduce broadcast time by 50%.

CornetClustering In cloud environments, users have no control over machine placements, and cloud providers do not disclose any information regarding network topology. Even if the initial placements were given out, VM migrations in the background could invalidate this information. For these cases, we have developed CornetClustering that automatically infers and exploits the underlying network topology.

It starts off without any topology information such as the vanilla Cornet. Throughout the course of an application's lifetime, as more and more broadcasts happen, it records block transfer times between different pairs of receivers and uses a learning algorithm to infer the rack-level topology. Once we infer the topology, we use the same mechanism as in CornetTopology. The controller keeps recalculating the inference periodically to keep an updated view of the network.

The inference procedure consists of the following four steps.

I First, we record machine-to-machine block transfer times. We use this data to construct an $n \times n$ distance matrix $\mathbf{D}$, where n is the number of receiver machines and the entries are the median block transfer times between a pair of machines.

II Next, we infer the missing entries in the distance matrix using a version of the nonnegative matrix factorization procedure of Mao and Saul [148].

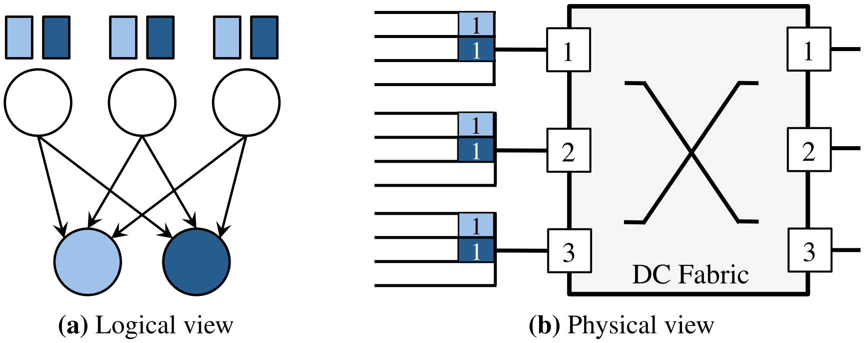
III After completing the matrix $\mathbf{D}$, we project the machines onto a two-dimensional space using non-metric multidimensional scaling [137].

IV Finally, we cluster using a mixture of spherical Gaussians with fixed variance σ^2 and automatically select the number of partitions based on the Bayesian information criterion score [93].

In operational use, one can set σ to the typical intra-rack block transfer time (in our experimental setup, we use $\sigma = 200$ ms). With enough training data, the procedure usually infers the exact topology and provides a similar speedup to CornetTopology, as we show in Section 4.6.2.

4.4.4 Size-Aware Broadcast Algorithm Selection

While Cornet achieves good performance for a variety of workloads and topologies, it does not always provide the best performance. For example, in our experiments we found that for a small number of receivers, a chain distribution topology usually performs better. In such a case, the



(a) Logical view **(b)** Physical view

Figure 4.3: A 3×2 shuffle. (a) The two receivers (at the bottom) need to fetch separate pieces of data, depicted as boxes of different colors, from each sender. (b) The same shuffle on 3×3 datacenter fabric with three ingress/egress ports. Flows in ingress ports are organized by destinations and color-coded by receivers.

Cornet controller can decide whether to employ one algorithm or another based on the number of receivers. In general, as new broadcast algorithms are developed, it can pick the best one to match a particular data size and topology. This ability illustrates the advantage of coordination, which enables the controller to make decisions based on global information.

4.5 The Shuffle Coflow

During the shuffle stage of a MapReduce job, each reducer is assigned a range of the key space produced by the mappers and must fetch some elements from every mapper. Consequently, shuffle coflows are some of the most common communication patterns in datacenters. Similar constructs exist most data-intensive applications [6, 119, 146, 147, 208]. In general, a shuffle consists of R receivers, $r_1, \ldots, r_R$, and M senders, $s_1, \ldots, s_M$, where the ith sender needs to send a distinct dataset d_{ij} to the jth receiver. Figure 4.3 depicts a typical shuffle with $R = 2$ and $M = 3$.

Because each piece of data goes from only one sender to one receiver, unlike in broadcasts, receivers cannot improve performance by sharing data. The main concern during a shuffle is, therefore, to keep bottleneck links fully utilized (§4.5.1). We find that the strategy used by systems such as Hadoop, where each receiver opens connections to multiple random senders and rely on TCP fair sharing among these flows, is close to optimal when data sizes are balanced (§4.5.2). There are cases with unbalanced data sizes in which this strategy can perform $1.5\times$ worse than optimal. Finally, we propose an optimal algorithm called Minimum Allocation for Desired Duration (MADD) to address these scenarios (§4.5.3).

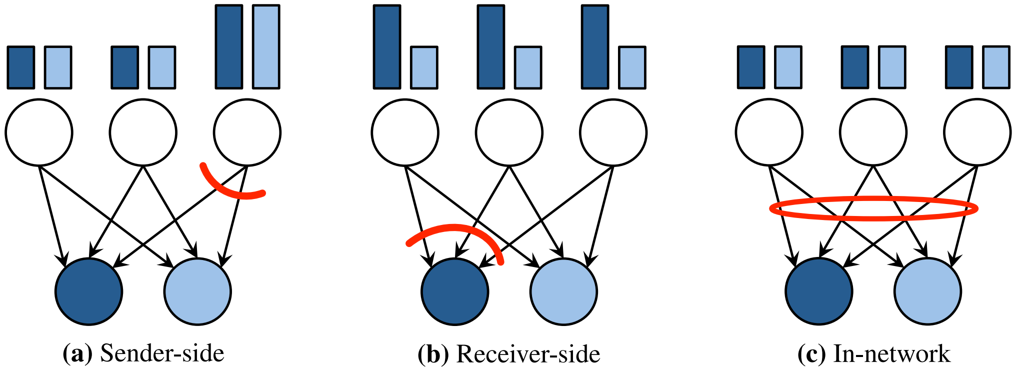


Figure 4.4: Different bottleneck locations dictating shuffle performance.

4.5.1 Bottlenecks and Optimality in Shuffle

Figure 4.4 shows three situations where a bottleneck limits shuffle performance and scheduling can have little impact on the overall completion time. In Figure 4.4a, one of the senders has more data to send than others (e.g., a map produced more output in a MapReduce job), so this machine's link to the network is the bottleneck. Even with the random scheduling scheme in current systems, this link is likely to stay fully utilized throughout the transfer, and because a fixed amount of data *must* flow along this link to finish the shuffle, the completion time of the shuffle will be the same regardless of the scheduling of other flows. Figure 4.4b shows an analogous situation where a receiver is the bottleneck. Finally, in Figure 4.4c, there is a bottleneck in the network – for example, if the network is not non-blocking – and again the order of data fetches will not affect the overall completion time as long as the contended links are kept fully utilized.

These examples suggest a simple optimality criterion for shuffle scheduling: *an optimal shuffle algorithm keeps the bottlenecks fully utilized throughout the transfer*. This condition is clearly necessary, because if there was a time period during which a shuffle schedule kept all ports less than 100% utilized, the completion time could be lowered by slightly increasing the rate of all flows during that period. The condition is also sufficient on a non-blocking fabric. In this case, the shuffle must transfer $\sum_j d_{ij}$ amount of data through each ingress port i (P_i^{in}) and $\sum_i d_{ij}$ through each egress port j (P_j^{out}), where d_{ij} is the amount of data to transfer from P_i^{in} to P_j^{out} at rate r_{ij}. The minimum completion time (Γ) becomes

$$\Gamma = \max\left(\max_i \frac{\sum_j d_{ij}}{Rem(P_i^{\text{in}})}, \max_j \frac{\sum_i d_{ij}}{Rem(P_j^{\text{out}})}\right) \tag{4.1}$$

where $Rem(.)$ denotes the remaining bandwidth of an ingress or egress port. If any port is fully utilized throughout the shuffle duration, then this lower bound has been reached, and the schedule is the optimal. The former argument of Equation (4.1) represents the minimum time to transfer $\sum_{ij} d_{ij}$ amount of data through the input ports, and the latter is for the output ports.

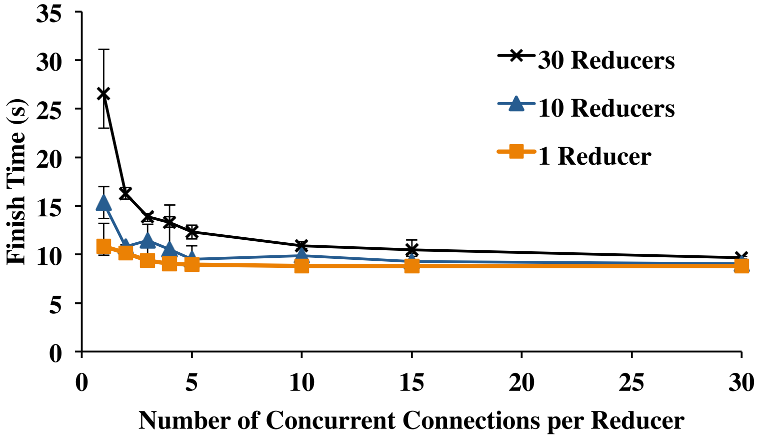


Figure 4.5: [EC2] Completion times for a shuffle with 30 senders and 1 to 30 receivers, as a function of the number of concurrent flows (from random senders) per receiver.

4.5.2 Load Balancing in Current Implementations

The optimality observation indicates that the links of both senders and receivers should be kept as highly utilized as possible. Indeed, if the amount of data per machine is balanced, which is often the case in large MapReduce jobs simply because many tasks have run on every machine, then all of the machines' outgoing links can potentially become bottlenecks. The biggest risk with the randomized data fetching scheme in current systems is that some senders get too few connections to them, underutilizing their links.[3] Our main finding is that having multiple connections per receiver drastically reduces this risk and yields near-optimal shuffle times. In particular, Hadoop's setting of 5 connections per receiver seems to work well, although more connections can improve performance slightly.

We conducted an experiment with 30 senders and 1 to 30 receivers on Amazon EC2, using extra large machines. Each receiver fetched 1 GB of data in total, balanced across the senders. We varied the number of parallel connections opened by each receiver from 1 to 30. We plot the average transfer times for five runs in Figure 4.5, with max/min error bars.

We note two trends in the data. First, using a single fetch connection per receiver leads to poor performance, but transfer times improve quickly with even two connections. Second, with enough concurrent connections, transfer times approach 8 seconds asymptotically, which is a lower bound on the time we can expect for machines with 1 Gbps links. Indeed, with 30 connections per receiver, the overall transfer rate per receiver was 790 Mbps for 30 receivers, 844 Mbps for 10 receivers, and 866 Mbps for 1 receiver, while the best transfer rate we got between any two machines in our cluster was 929 Mbps. This indicates that randomized selection of senders is within 15% of optimal, and may be even closer because there may be other traffic on EC2 interfering with our job, or a topology with less than full bisection bandwidth.

[3] Systems such as Hadoop cap the number of receiving connections per reduce task for pragmatic reasons, such as limiting the number of threads in the application. Having fewer connections per receiver can also mitigate incast [194].

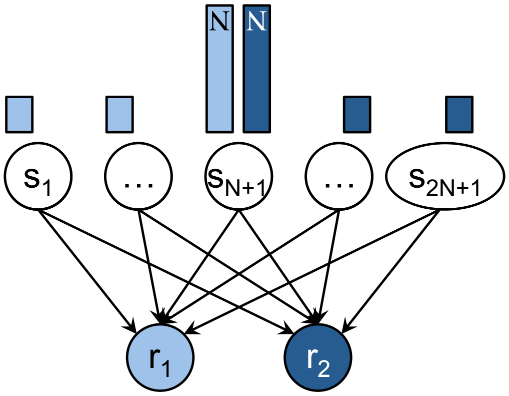


Figure 4.6: A shuffle configuration where MADD outperforms per-flow fairness. Senders s_1 to s_N sends one data unit *only* to receiver r_1, senders s_{N+1} to s_{2N} sends one data unit *only* to receiver r_2, and s_{N+1} sends N data units to *both* the receivers.

The improvement in transfer times with more connections happens for two reasons. First, with only one connection per receiver, collisions (when two receivers pick the same sender) can only lower performance, because some senders will be idle. In contrast, with even 2 threads, a collision that slows down some flows may speed up others (because some senders are now sending to only one receiver). Second, with more connections per receiver, the standard deviation of the number of flows to each sender decreases relative to its mean, reducing the effect of imbalances.

4.5.3 Minimum Allocation for Desired Duration (MADD)

We now consider how to optimally schedule a shuffle on a non-blocking network fabric, where the jth receiver needs to fetch d_{ij} units of data from the ith sender. We aim to minimize the completion time of the shuffle, i.e., the time when the last receiver finishes, under the assumption that data sizes between each sender-receiver pairs are known a priori.

We propose a simple algorithm called Minimum Allocation for Desired Duration (MADD) that *allocates rates to each flow for all flows to finish on or before a specified time*. To minimize a shuffle's completion time, we want all its flows to finish before its bottleneck, i.e., on or before Γ time units. We can guarantee this by ensuring that the rates (r_{ij}) of each flow is at least $\frac{d_{ij}}{\Gamma}$.

Comparison to Coflow-Agnostic Schemes

The optimal schedule shown in Figure 3.2 (§ 3.2) shows MADD's performance against per-flow fairness and per-flow prioritization schemes.

The improvements from MADD is due to data skew – i.e., the discrepancies in amounts of data each receiver has to receive from each sender. This discrepancy can be increased in certain shuffle configurations. Figure 4.6 shows one such example; for $N = 100$, MADD finishes $1.495\times$ faster than fair sharing. We found that MADD can outperform current TCP-based shuffle implementations by up to $1.5\times$.

Nevertheless, we found that configurations where MADD outperforms fair sharing are rare in practice. If the amounts of data to be transferred between each sender and each reducer are roughly balanced, i.e., in the absence of data skew, MADD reduces to fair sharing. In addition, if there is a single bottleneck sender or bottleneck receiver, then fair sharing will generally keep that machine's link fully utilized, resulting in an optimal schedule.

Per-flow prioritization schemes are likely to be suboptimal in most cases for coflow-level performance because they completely avoid flow-level sharing.

4.6 Evaluation

We have evaluated Cornet and MADD in the context of Spark and ran experiments in two environments: Amazon EC2 [3] and DETERlab [12]. On EC2, we used extra-large high-memory instances, which appear to occupy whole physical machines and had enough memory to perform the experiments without involving disk behavior (except for HDFS-based mechanisms). Although topology information is not provided by EC2, our tests revealed that machines were able to achieve 929 Mbps in each direction and 790 Mbps during 30 machines all-to-all communication (Figure 4.5), suggesting a near-full bisection bandwidth network. The DETERlab cluster spanned 3 racks and was used as ground-truth to verify the correctness of Cornet's clustering algorithm.

Our experiments show the following:

- Cornet performs $4.5\times$ better than the default Hadoop implementation and BitTornado (§4.6.1), and with topology awareness, Cornet can provide further $2\times$ improvement (§4.6.2).
- MADD can improve shuffle speeds by 29% (§4.6.3).
- Taken together, Cornet and MADD reduced communication times in the logistic regression and collaborative filtering applications in Section 4.3 by up to $3.6\times$ and sped up jobs by up to $1.9\times$ (§4.6.4).

Since coflows act as synchronization steps in many iterative and data-intensive frameworks, capturing the behavior of the slowest receiver is the most important metric for comparing alternatives. We, therefore, use the *completion time* of the entire coflow as our main performance metric.

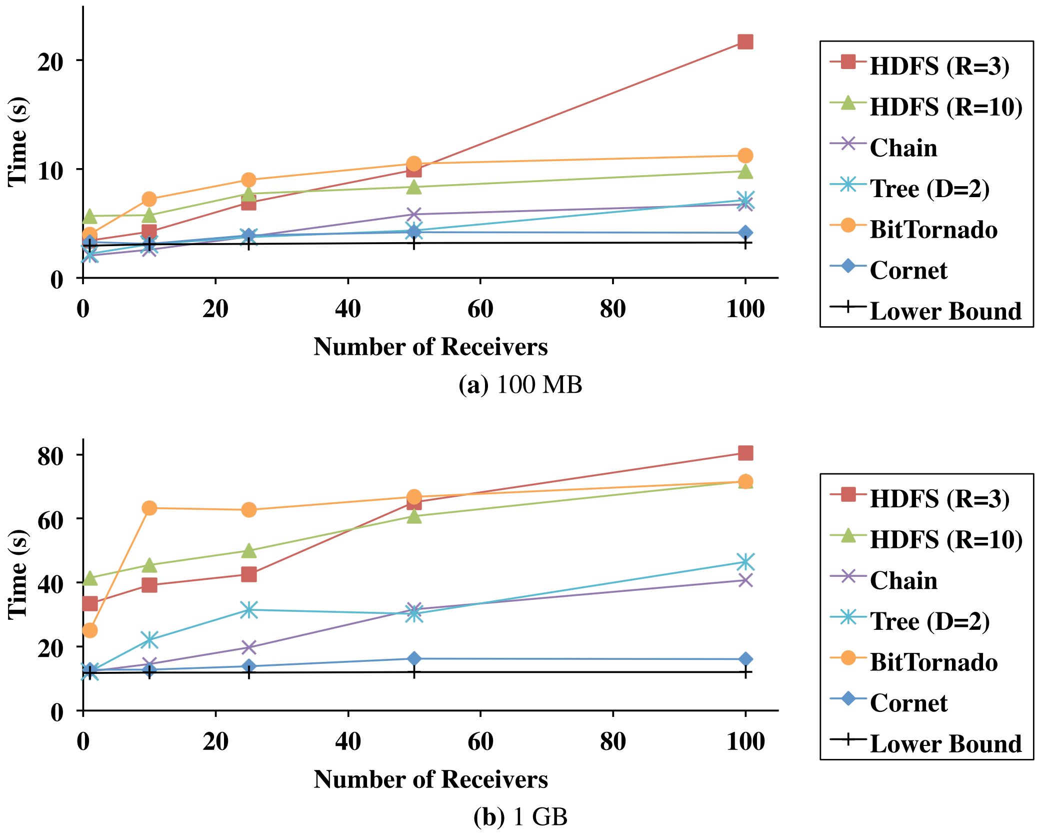
4.6.1 Comparison of Broadcast Mechanisms

Figure 4.7 shows the average completion times of different broadcast mechanisms (Table 4.1) to transfer 100 MB and 1 GB of data to multiple receivers from a single source. Error bars represent the minimum and the maximum observed values across five runs.

We see that the overheads of choking/unchoking, aggressive hashing, and allowing receivers to leave as soon as they are done, fail to take full advantage of the faster network in a datacenter

Algorithm	Description
HDFS (R=3)	Sender creates 3 replicas of the input in HDFS and receivers read from them
HDFS (R=10)	Same as before, but there are 10 replicas
Chain	A chain of receivers rooted at the sender
Tree (D=2)	Binary tree with sender as the root
BitTornado	BitTorrent implementation for the Internet
Cornet	Approach proposed in Section 4.4
Theoretical Lower Bound	Minimum broadcast time in the EC2 network (measured to have 1.5 Gbps pairwise bidirectional bandwidth) using pipelined binomial tree distribution mechanism [94]

Table 4.1: Broadcast mechanisms compared.



(a) 100 MB

(b) 1 GB

Figure 4.7: [EC2] Completion times of different broadcast mechanisms for varying data sizes.

environment and made BitTornado[4] as much as 4.5× slower than the streamlined Cornet implementation.

[4] We used Murder [21] with a modification that forced every peer to stay in the swarm until all of them had finished.

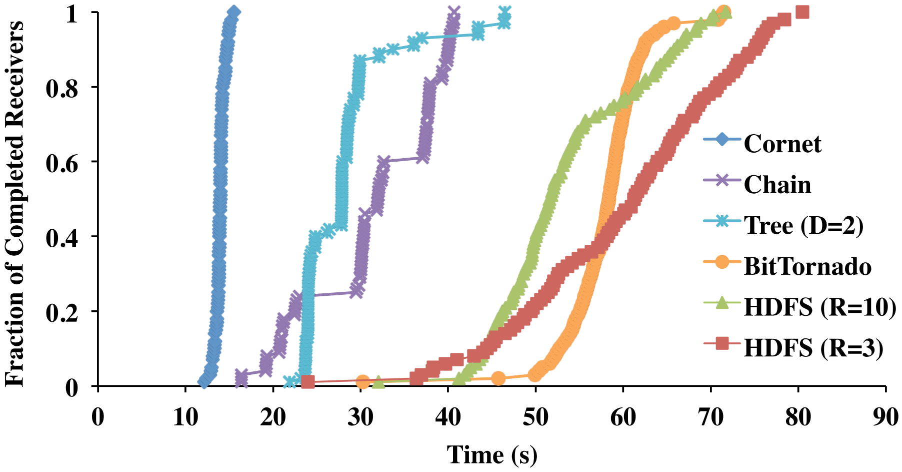


Figure 4.8: [EC2] CDF of completion times of individual receivers while transferring 1 GB to 100 receivers using different broadcast mechanisms. Legend names are ordered by the topmost points of each line, i.e., when *all* the receivers completed receiving.

Cornet scaled well up to 100 receivers for a wide range of data sizes in our experiments. For example, Cornet took as low as 15.4 seconds to complete broadcasting 1 GB data to 100 receivers and remained within 33% of the theoretical lower bound. If there were too few participants or the amount of data was small, Cornet could not fully utilize the available bandwidth. However, as the number of receivers increased, Cornet completion times increased in a much slower manner than its alternatives, which convinces us that Cornet can scale well beyond 100 receivers.

We found structured mechanisms to work well only for smaller scale. Any delay introduced by a straggling internal machine of a tree or a chain propagated and got magnified throughout the structure. Indeed, upon inspection, we found that the non-monotonicity of chain and tree completion times were due to this very reason in some experimental runs (e.g., completion time for 25 receivers using a tree structure is larger than that for 50 receivers in 4.7b).

As expected, HDFS-based mechanisms performed well only for small amounts of data. While increasing the number of replicas helps, there is a trade-off between time spent in creating replicas vs. time all the receivers would spend in reading from those replicas. In our experiments, HDFS with 3 replicas performed better than HDFS with 10 replicas when the total number of receivers was less than 50. Overall, HDFS with 3 and 10 replicas were up to 5× and 4.5× slower than Cornet, respectively.

A Closer Look at Per-machine Completion Times. We present the CDFs of completion times of individual receivers for each of the compared broadcast mechanisms in Figure 4.8.

Notice that almost all the receivers in Cornet finished simultaneously. The slight bends at the two endpoints illustrate the receivers ($<10\%$) that finished earlier or slower than the average receiver. Cornet tries to minimize such deviations (at most three seconds in this case) by forcing all the completed receivers to stay in the system. The CDF representing BitTornado reception times

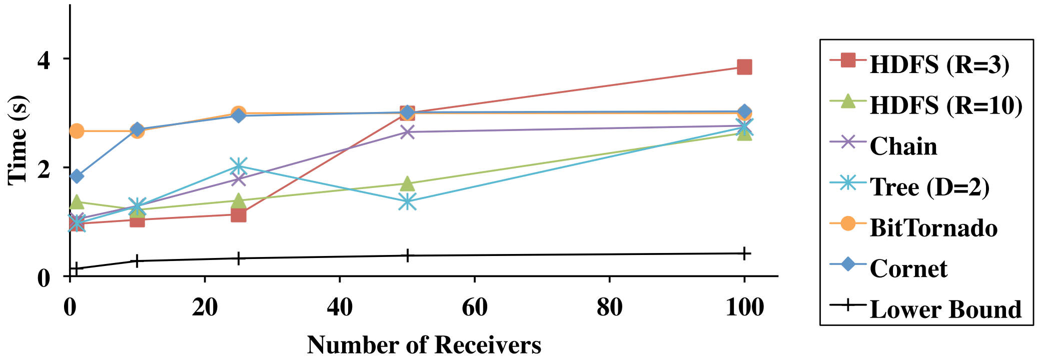


Figure 4.9: [EC2] Broadcast completion times for 10 MB data.

is similar to that of Cornet except that the variation in individual completion times is significantly higher and the average receiver is almost 4× slower.

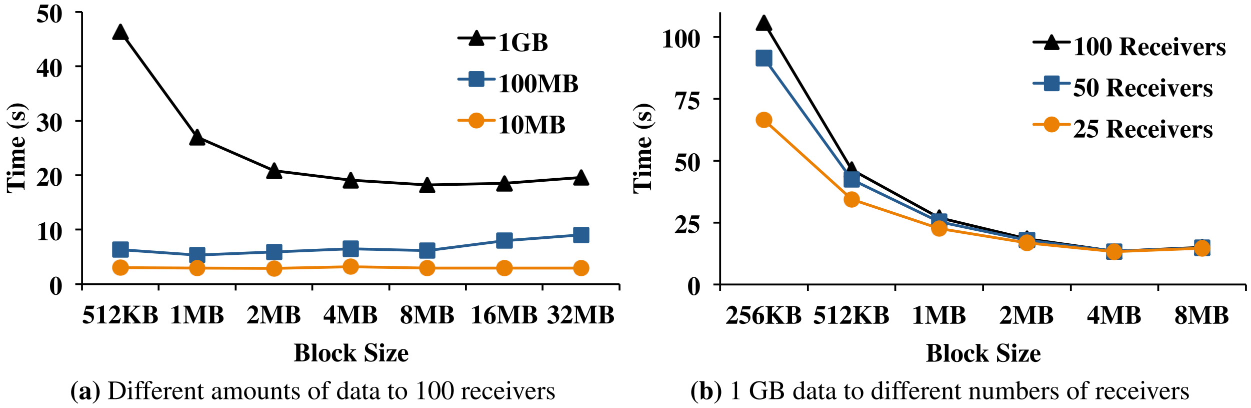
Next, the steps in the CDFs of chain and tree highlight how stragglers slow down all their children in the distribution structure. Each horizontal segment indicates a machine that was slow in finishing reception and the subsequent vertical segment indicates the receivers that experienced head-of-line blocking due to a slow ancestor.

Finally, receivers in HDFS-based transfer mechanism with 10 replicas start finishing slower than those with 3 replicas due to higher replication overhead. However, in the long run, receivers using 10 replicas finish faster because of less reading contention.

The Case for Coordination As evident from Figure 4.7, broadcast mechanisms have specific operating regimes. In particular, chain- and tree-based approaches are faster than Cornet for small numbers of machines and small data sizes, likely because the block sizes and polling intervals in Cornet prevent it from utilizing all the machines' bandwidth right away. We confirmed this by running another set of experiments with 10 MB (Figure 4.9), where tree and chain outperformed other approaches in many cases. The Cornet controller can pick the best transfer mechanism for a given data size and number of machines using its global knowledge. The advantages of coordination becomes more apparent later when we evaluate Cornet using adaptive clustering algorithms.

Impact of Block Size

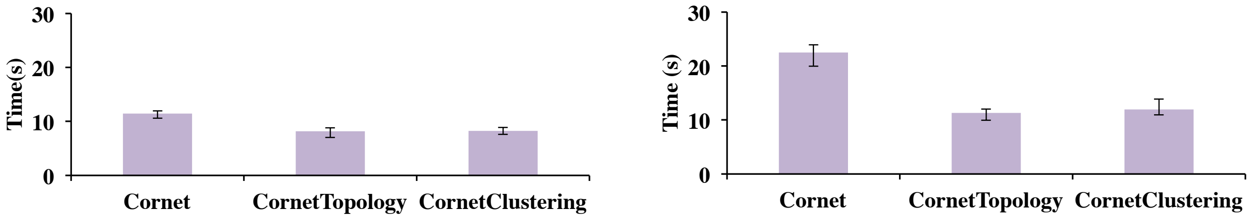
For a given data size, too large a block size limits sharing between peers. However, if block size is too small, overheads increase. Figure 4.10a presents the impact of block size on Cornet completion times for different data sizes. When the amount of data is too small, there is no significant change in completion times for varying block sizes because completion times are dominated by Cornet's overheads. However, broadcast completion times vary noticeably as the amount of data increases. In addition, the minimum completion time for a particular data size occur at different block sizes (e.g., 1 MB block size for 100 MB data and 4 MB for 1 GB). For a given amount of data, however, minimum completion times for different numbers of receivers can be found at the same block size



(a) Different amounts of data to 100 receivers

(b) 1 GB data to different numbers of receivers

Figure 4.10: [EC2] Cornet completion times for varying block sizes.



(a) 100 MB

(b) 200 MB

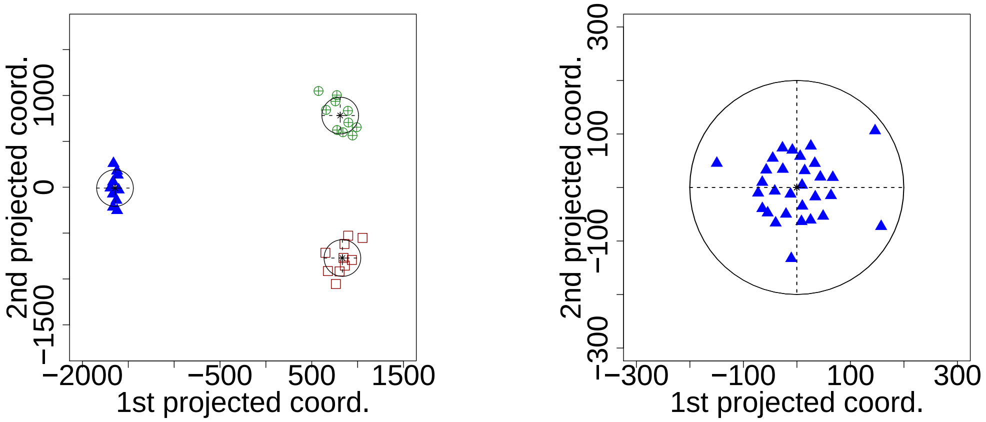
Figure 4.11: [DETERlab] Cornet completion times when the rack topology is unknown, given, and inferred using clustering.

(Figure 4.10b). While these experiments do not give the extent of dependency (or independence) of data size or the number of receivers with block size, they do strengthen our intuition that block size does affect broadcast completion times significantly and require dynamic treatment primarily with respect to the amount of data to be broadcasted.

4.6.2 Topology-Aware Cornet

In this section, we explore an extension of Cornet that exploits network topology information. We hypothesized that if there is a significant difference between block transfer times within a rack vs. between racks, then a topology-aware version of Cornet, which reduces cross-rack communication, will experience improved transfer times. To answer this question, we conducted an experiment on a 31 machine DETERlab testbed (1 controller and 30 receivers). The testbed topology was as follows: Rack A was connected to Rack B and Rack B to Rack C. Each rack had 10 receiver machines. The controller was in Rack B.

We ran the experiment with three configurations. The first was the default topology-oblivious Cornet that allowed any receiver to randomly contact any other receiver. The second was CornetTopology, where the controller partitioned the receivers according to Racks A, B, and C, and



(a) DETERlab

(b) Amazon EC2

Figure 4.12: Two-dimensional, non-metric projection of receiver machines based on a distance matrix of machine-to-machine block transfer times. The triangles, squares, and circles in (a) represent racks A, B, and C respectively in the DETERlab testbed.

disallowed communication across partitions. The last one was CornetClustering, where the controller dynamically inferred the partitioning of the machines based on the machine-to-machine block transfer times from 10 previous training runs.

The results in Figure 4.11 show the average completion times to transfer 100 MB and 200 MB of data to all 30 receivers over 10 runs with min-max error bars. Given the topology information (CornetTopology), completion times decreased by 50% compared to vanilla Cornet for the 200 MB broadcast. In 9 out of 10 runs for the 200 MB broadcast, CornetTopology inferred the exact topology (see Figure 4.12a for a typical partitioning). Only in one run did it infer 5 partitions (splitting two of the racks in half), though this only resulted in a 2.5 second slowdown compared to inferring the exact topology. With the ten runs averaged together, CornetClustering's reduction in completion time was 47%.

We also evaluated Cornet and CornetClustering on a 30 machine EC2 cluster. Evaluating CornetTopology was not possible because we could not obtain the ground-truth topology for EC2. The performance of Cornet using inferred topology did not improve over Cornet on EC2 – the algorithm found one cluster, likely due to EC2's high bisection bandwidth (Section 4.5.2). The projection in Figure 4.12b shows that with the exception of a few outliers (due to congestion), all the machines appeared to be relatively close to one another and could not be partitioned into well-separated groups.

Overall, the results on the DETERlab demonstrate that when there is a sizable gap between intra-rack and inter-rack transfer times, knowing the actual machine topology or inferring it can significantly improve broadcast times.

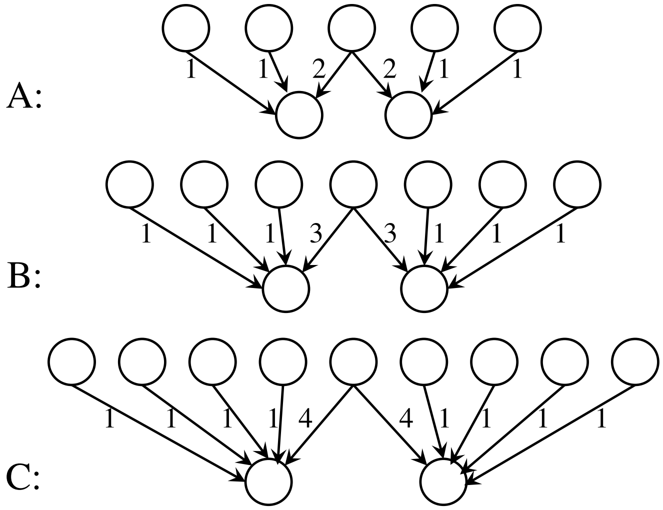


Figure 4.13: Shuffle configurations used in MADD evaluation. The arrows show the number of units of data sent from each mapper to each reducer.

Topology	Standard Shuffle	MADD	Speedup	Theoretical Speedup
A	83.3 (1.1)	70.6 (1.8)	18%	25%
B	131 (1.8)	105 (0.5)	24%	33%
C	183 (2.6)	142 (0.7)	29%	38%

Table 4.2: [EC2] Completion times in seconds for MADD compared to a standard shuffle implementation for the shuffles in Figure 4.13. Standard deviations are in parentheses.

4.6.3 Minimum Allocation for Desired Duration (MADD)

In this experiment, we evaluate the optimal Minimum Allocation for Desired Duration (MADD) algorithm discussed in Section 4.5.3 using three topologies on Amazon EC2. Figure 4.13 illustrates these topologies, with arrows showing the number of units of data sent between each pair of machines (one unit corresponded to 2 GB in our tests). All of them are different variations of the shuffle configuration in Figure 4.6.

We ran each scenario under both a standard implementation of shuffle (where each reducer simultaneously connects to at most 5 mappers) and under MADD. We implemented MADD by setting rates proportional to the amount of data each receiver had to fetch through coordinated, application-layer rate limiting on top of underlying TCP.

We present average results from five runs, as well as standard deviations, in Table 4.2. In all cases, MADD performs better than a standard implementation of shuffle, by 18%, 24%, and 29% for configurations A, B and C, respectively. In addition, we present the theoretical speedup predicted for each topology, which would be achieved in a full bisection bandwidth network with a perfect implementation of fair sharing between flows. The measured results are similar to those

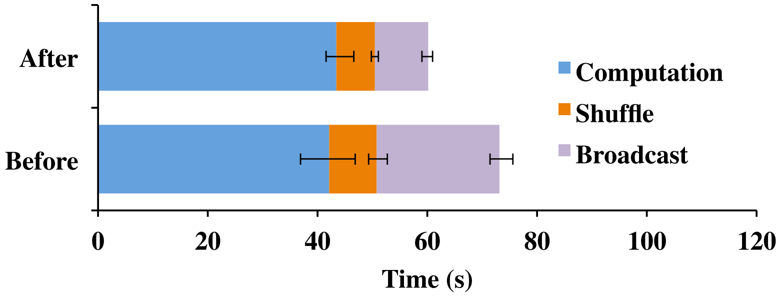
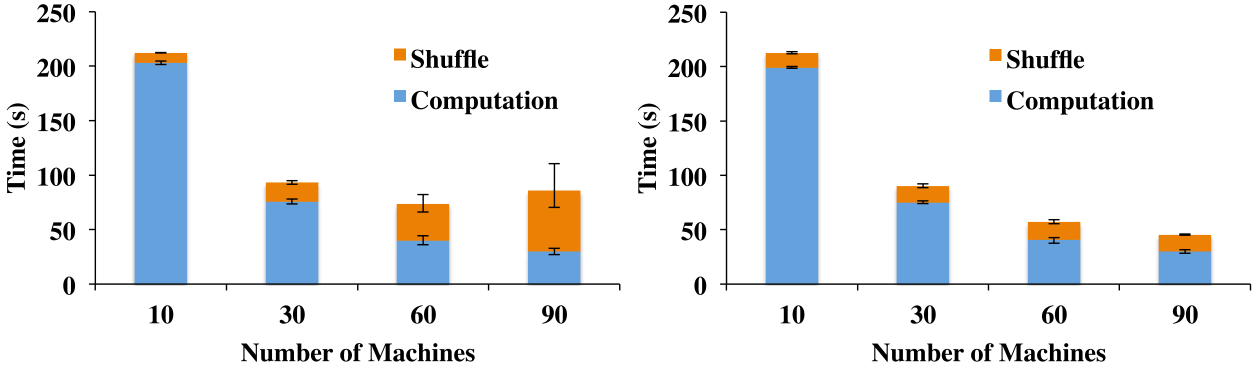


Figure 4.14: [EC2] Per-iteration completion times for the logistic regression application before and after using Cornet and MADD.



(a) Before

(b) After

Figure 4.15: [EC2] Per-iteration completion times when scaling the collaborative filtering application using MADD.

predicted but somewhat lower because fair sharing between TCP flows is not perfect (e.g., if a machine starts 2 GB transfers to several machines at the same time, these transfers can finish 10-15 seconds apart).

4.6.4 End-to-End Results on Full Applications

We revisit the motivating applications from Section 4.3 to examine the improvements in end-to-end run times after adopting the new broadcast and shuffling algorithms. In particular, "before" entails running with a HDFS-based broadcast implementation (with the default 3× replication) and a shuffle with 5 threads per receiver (the default in Hadoop). Meanwhile, "after" entails both Cornet and MADD.

Figure 4.14 illustrates the breakdown of time spent in different activities in each iteration of Monarch in a 30 machine EC2 cluster. We see that its communication overhead in each iteration decreased from 42% of the run time to 28%, and iterations finished 22% faster overall. There is a

$2.3\times$ speedup in broadcast and a $1.23\times$ speedup in shuffle. The improvements for both broadcast and shuffle are in line with the findings in Sections 4.6.1 and 4.5.2.

Figure 4.15b presents the per-iteration completion times for the collaborative filtering job while scaling it up to 90 machines using Cornet. Unlike the HDFS-based solution (Figure 4.15a), broadcast time increased from 13.4 to only 15.3 seconds using Cornet. As a result, the job could be scaled up to 90 machines with $1.9\times$ improvement in iteration times. The average time spent in broadcast decreased by $3.6\times$, from 55.8 to 15.3 seconds, for 90 machines. These results are in line with Section 4.6.1 given 385 MB broadcast per iteration.

4.7 Related Work

Centralized Network Controllers Centralized controllers for routing, access control, and load balancing in the network had been proposed by the 4D architecture [107] and projects such as Tesseract [204], Ethane [56], PLayer [126], Hedera [33], and FastPass [167]. While PLayer and Ethane focus on access control, our primary objective is application-level performance improvement. The scope of our work is limited to shared clusters and datacenters, whereas 4D, Tesseract, and Ethane are designed for wide-area and enterprise networks. However, unlike Hedera, Fastpass, or other proposals for centralized control planes, we work at the granularity of *coflows* to optimize overall application performance, and not at the packet or flow level.

Scheduling and Management in Data-intensive Applications A plethora of schemes exist to schedule and manage tasks of data-intensive applications. Examples include fair schedulers for Hadoop [207] and Dryad [120], and Mantri [40] for outlier detection. The core tenet of existing work in this area is achieving data locality to avoid network transfers as much as possible. Mesos [117] and YARN [195] provide thin management layers to allow diverse applications to efficiently share computation and storage resources, but leaves sharing of network resources to underlying transport mechanisms. The proposed mechanisms complement these systems.

One-to-Many Data Transfer Mechanisms Broadcast, multicast, and diverse group communication mechanisms in application and lower layers of the network stack have been studied extensively in the literature. Diot et al. provide a comprehensive survey and taxonomy of relevant protocols and mechanisms of distributed multi-point communication in [81]. Cornet is designed for transferring large amounts of data in high-speed datacenter networks.

SplitStream [57] improves network utilization and tackles the bottleneck problem observed in d-ary trees by creating multiple distribution trees with disjoint leave sets. However, it is designed primarily for multimedia streaming over the Internet, where frames can be dropped. Maintaining its structural constraints in presence of failure is complicated as well.

BitTorrent [72] is wildly popular for file-sharing. BitTorrent and similar peer-to-peer mechanisms are in use to distribute planet-scale software updates [99]. However, Murder [21] is one of the few BitTorrent deployments inside a datacenter. Antfarm [168] uses a central coordinator across multiple swarms to optimize content distribution over the Internet. Cornet is a BitTorrent-like system that is optimized for datacenters and uses adaptive clustering algorithm in the controller to infer and take advantage of network topologies.

Incast or Many-to-One Transfers TCP incast collapse is typically observed in barrier-synchronized request workloads where a receiver synchronously receives small amounts of data from a large number of senders [194]. However, incast collapse has been reported in MapReduce-like data-intensive workloads as well [62]. The latter case boils down to a special case of shuffle with only one reducer. With MADD, we can effectively limit how many senders are simultaneously sending and at what rate to alleviate this problem for data-intensive workloads.

Inferring Topology from Machine-to-Machine Latencies Inferring network topology in CornetClustering (Section 4.4.3) is similar in spirit to inferring network coordinates [84]. These methods could act as a substitute for the non-metric multidimensional scaling step in the CornetClustering procedure.

4.8 Summary

In this chapter, we focused on two common coflows, broadcasts and shuffles, and leveraged the all-or-nothing property of coflows to develop efficient algorithms. For broadcasts, we proposed a topology-aware BitTorrent-like scheme called Cornet that outperforms the status quo in Hadoop by $4.5\times$. For shuffles, we proposed the optimal algorithm called Minimum Allocation for Desired Duration (MADD). Overall, our schemes can increase application performance by up to $1.9\times$. Both Cornet and MADD can be implemented at the application layer using centralized coordination and does not require hardware changes to run in current datacenters and in the cloud.

Both of these algorithms are currently used in practice by several open-source systems. Most notably, Cornet was merged with Apache Spark [208] as the default broadcast mechanism in Spark release 1.1.0 and Varys [68] – described in more details in the next chapter – uses MADD as a building block.

Chapter 5

Clairvoyant Inter-Coflow Scheduling

In the previous chapter, we demonstrated the application of coflows to improve the communication performance of individual applications. In this chapter, we focus on improving communication performance of multiple coexisting applications using clairvoyant inter-coflow scheduling. We address inter-coflow scheduling for two different objectives: decreasing communication time of distributed data-parallel jobs and guaranteeing predictable communication time. In the process, we introduce the *concurrent open shop scheduling with coupled resources* problem, analyze its complexity, and propose effective heuristics to optimize either objective.

The rest of this chapter is organized as follows. The next section covers the state-of-the-art in optimizing communication performance of coexisting distributed data-parallel applications. Section 5.2 outlines the proposed approach, which is implemented in a system called Varys. Section 5.3 illustrates the possible benefits from inter-coflow scheduling over existing coflow-agnostic schemes. Section 5.4, Section 5.5, and Section 5.6, respectively, present a system-level overview of Varys, an analysis of the algorithms and heuristics used in Varys, and corresponding implementation details. We then evaluate Varys's performance in Section 5.7 through deployments and simulations. Next, we discuss in Section 5.8 its current limitations and future research directions, survey related work in Section 5.9, and summarize our findings in Section 5.10.

5.1 Background

Despite the differences among data-intensive frameworks [6, 8, 77, 119, 147, 191, 208], their communication is structured and takes place between groups of machines in successive computation stages [65]. Often a communication stage cannot finish until all its flows have completed [67, 82]. The coflow abstraction represents such collections of parallel flows to convey job-specific communication requirements – for example, minimizing completion time or meeting a deadline – to the network and enables application-aware network scheduling. Indeed, as shown in Chapter 4, optimizing a coflow's completion time (CCT) decreases the completion time of corresponding job.

However, jobs from one or more frameworks create multiple coflows in a shared cluster. Analysis of production traces shows wide variations in coflow characteristics in terms of total size, the number of parallel flows, and the size of individual flows (see §A.3 for more details). Simple scheduling mechanisms such as FIFO and its variants [67, 82], which are attractive for the ease of decentralization, do not perform well in such an environment – one large coflow can slow down many smaller ones or result in many missed deadlines.

Simply applying a shortest- or smallest-first heuristic, the predominant way to solve most scheduling problems, is not sufficient either. Inter-coflow scheduling is different from scheduling individual flows [37, 118], because each coflow involves multiple parallel flows. It also differs from related problems such as scheduling parallel tasks [40, 207] or caching parallel blocks [39]; unlike CPU or memory, the network involves *coupled resources* – each flow's progress depends on its rates at *both* source and destination. We show that these coupled constraints make *permutation schedules* – scheduling coflows one after another without interleaving their flows – suboptimal. Hence, centralized scheduling becomes impractical, because the scheduler needs to preempt flows or recalculate their rates at arbitrary points in time even when no new flows start or complete.

In this chapter, we show how to perform inter-coflow scheduling for arbitrary coflows to either to improve application-level performance by minimizing CCTs or to guarantee predictable completions within coflow deadlines. We prove this problem to be strongly NP-hard for either objective and focus on developing pragmatic heuristics. Furthermore, we show how make centralized coflow scheduling practical by rescheduling only on coflow arrivals and completions.

5.2 Solution Outline

Our key observation is the following: in the presence of coupled constraints, the bottleneck endpoints of a coflow determine its completion time. We propose the *Smallest Effective Bottleneck First* (SEBF) heuristic that greedily schedules a coflow based on its bottleneck's completion time. We then use the MADD algorithm introduced in Chapter 4 to allocate rates to the individual flows of each coflow. Because MADD *slows down all the flows in a coflow to match the completion time of the flow that will take the longest to finish*, other coexisting coflows can make progress and the average CCT decreases. While the combination of SEBF and MADD is not necessarily optimal, we have found it to work well in practice.

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

Online coflow scheduling introduces two additional challenges. First, one must allow coflow preemption to avoid head-of-line-blocking when minimizing CCTs, but preemption can cause starvation of some coflows. We avoid starvation by ensuring that all coflows receive non-zero bandwidth over fixed intervals. Note that there is no starvation in the deadline-sensitive scenario, because admitted coflows are scheduled in their arrival order. Second, letting resources idle can lead to increased CCTs or higher rejections depending on future coflows. To achieve work conservation we introduce a backfilling procedure on top of MADD.

The problem of minimizing the average CCT is, in fact, a generalization of the well-studied problem of minimizing the average flow completion time (FCT) for multiple flows. Likewise, the proposed two-step solution generalizes the shortest-remaining-time-first (SRTF) scheduling policy [37, 118].

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

We deployed Varys on a 100-machine EC2 cluster and evaluated it by replaying production traces from Facebook. Varys improved CCTs both on average (up to $3.16\times$) and at high percentiles ($3.84\times$ at the 95th percentile) in comparison to per-flow fair sharing. Hence, end-to-end completion times of jobs, especially the communication-heavy ones, decreased. The aggregate network utilization remained the same, and there was no starvation. In trace-driven simulations, we found Varys to be $2.16\times$ better than fair sharing, $3.26\times$ better than per-flow prioritization, and $3.33\times$ better than FIFO schedulers. Moreover, in EC2 experiments (simulations), Varys allowed up to $2\times$ ($1.44\times$) more coflows to meet their deadlines in comparison to per-flow schemes.

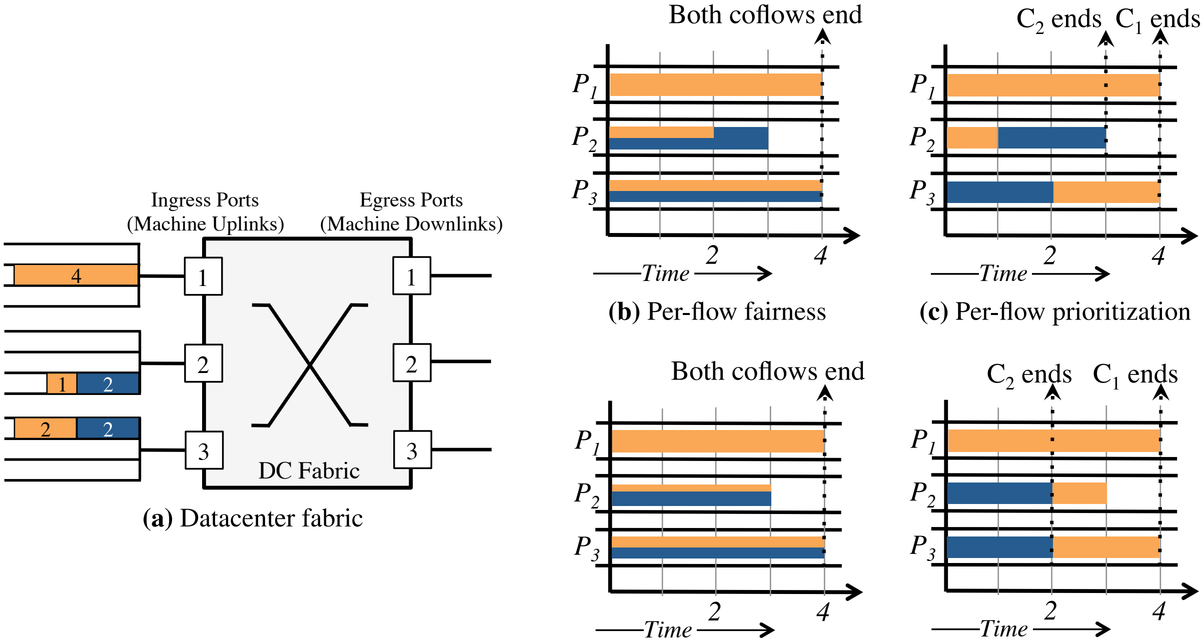
5.3 Potential Benefits of Inter-Coflow Scheduling

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

Consider Figure 5.1a. Assuming both coflows to arrive at the same time, Figure 5.1 compares four different schedules. Per-flow fairness (Figure 5.1b) ensures max-min fairness among flows in each link. However, fairness among flows of even the same coflow can increase CCT [67]. WSS (Figure 5.1d) – the optimal algorithm in homogeneous networks – is up to $1.5\times$ faster than per-flow fairness for individual coflows [67]; but for multiple coflows, it minimizes the completion time across *all* coflows and increases the average CCT. Recently proposed shortest-flow-first prioritization mechanisms [37, 118] (Figure 5.1c) decrease average FCT, but they increase the average CCT by interleaving flows from different coflows. Finally, the optimal schedule (Figure 5.1e) minimizes the average CCT by finishing flows in the coflow order (C_2 followed by C_1). The FIFO schedule [67, 82] would have been as good as the optimal if C_2 arrived before C_1, but it could be as bad as per-flow fair sharing or WSS if C_2 arrived later.

Deadline-Sensitive Communication Assume that C_1 and C_2 have the same deadline of 2 time units – C_1 would never meet its deadline as its minimum CCT is 4. Using per-flow fairness or WSS, both C_1 and C_2 miss their deadlines. Using earliest-deadline-first (EDF) across flows [118], C_2 meets its deadline only 25% of the time. However, the optimal coflow schedule does not admit C_1, and C_2 always succeeds.

[1] Pronounced \'vä-ris\.



(**a**) Datacenter fabric

(**b**) Per-flow fairness

(**c**) Per-flow prioritization

(**d**) WSS [67]

(**e**) The optimal schedule

Figure 5.1: Coflow scheduling over a 3 × 3 datacenter fabric with three ingress/egress ports (a). Flows in ingress ports are organized by destinations and color-coded by coflows – C_1 in orange/light and C_2 in blue/dark. Allocation of *ingress* port capacities (vertical axis) using different mechanisms for the coflows in Figure 5.1a. Each port can transfer one unit of data in one time unit. The average FCT and CCT for (b) per-flow fairness are 3.4 and 4 time units; (c) per-flow prioritization are 2.8 and 3.5 time units; (d) WSS are 3.6 and 4 time units; and (e) the optimal schedule are 3 and 3 time units.

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

5.4 Varys Overview

Varys is a coordinated coflow scheduler to optimize either the performance or the predictability of communication in data-intensive applications. In this section, we present a brief overview of Varys to help the reader follow the analysis and design of inter-coflow scheduling algorithms (§5.5) and Varys's design details (§5.6).

5.4.1 Problem Statement

When improving performance, given a coflow with information about its individual flows, their size, and endpoints, Varys must decide *when to start* its flows and *at what rate* to serve them to minimize the average CCT of the cluster. It can preempt existing coflows to avoid head-of-line blocking, but it must also avoid starvation. Information about a coflow is unknown prior to its arrival; however, once a coflow arrives, it's structure does not change over time.

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

Irrespective of the objective, the inter-coflow scheduling problem is NP-hard (§5.5). Varys implements a scheduler that exploits the variations in coflow characteristics (§A.3) to perform reasonably well in realistic settings.

5.4.2 Architectural Overview

Varys master schedules coflows from different frameworks using global coordination (Figure 5.2). It works in two modes: it either tries to minimize CCT or to meet deadlines. For the latter, it uses admission control, and rejected coflows must be resubmitted later. Frameworks use a client library to interact with Varys to register and define coflows (§5.6.1). The master aggregates all interactions to create a global view of the network and determines rates of flows in each coflow (§5.6.2) that are enforced by the client library.

Varys daemons, one on each machine, handle *time-decoupled* coflows, where senders and receivers are not simultaneously active. Instead of hogging the CPU, sender tasks (e.g., mappers) of data-intensive applications often complete after writing their output to the disk. Whenever corresponding receivers (e.g., reducers) are ready, Varys daemons serve them by coordinating with the master. Varys daemons use the same client library as other tasks. Additionally, these daemons send periodic measurements of the network usage at each machine to Varys master. The master aggregates them using existing techniques [63] to estimate current utilizations and use remaining bandwidth ($Rem(.)$) during scheduling (§5.5.3).

We have implemented Varys in the application layer out of practicality – it can readily be deployed in the cloud, while providing large improvements for both objectives we consider (§5.7).

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

Scalability Varys reschedules only on coflow arrival and completion events. We did not observe the typical number of concurrent coflows (tens to hundreds [67, 171]) to limit its scalability. Varys batches control messages at $O(100)$ milliseconds intervals to reduce coordination overheads, which affect small coflows (§5.7.2). Fortunately, most traffic in data-intensive clusters are from large coflows (§A.3). We do not use Varys for coflows with bottlenecks smaller than 25 MB in size.

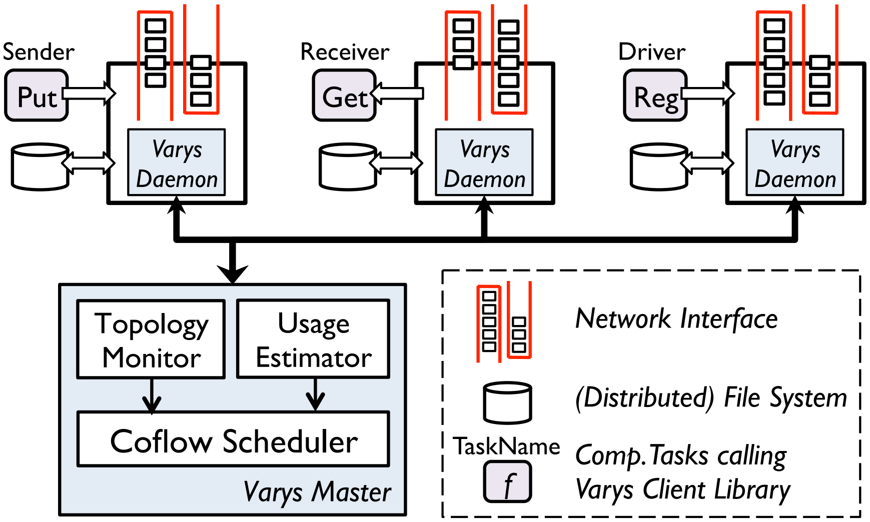


Figure 5.2: Varys architecture. Computation frameworks use the `VarysClient` library to interact with Varys.

5.5 Clairvoyant Coflow Scheduling: Analytical Results

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

Detailed analysis and proofs can be found in Appendix B.

5.5.1 Problem Formulation and Complexity

We consider two objectives for optimizing data-intensive communication: either *minimizing* the average CCT or improving predictability by *maximizing* the number of coflows that meet deadlines (§B.1). Achieving either objective is NP-hard, even when

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

We prove it by reducing the concurrent open-shop scheduling problem [180] to inter-coflow scheduling (**Theorem B.1.1**).

The online inter-coflow scheduling problem is even more difficult to solve because of the following reasons:

1. **Capacity Constraints:** Ingress and egress ports of the datacenter fabric have finite, possibly heterogeneous, capacities. Hence, the optimal solution must find the best *ordering* of flows to dispatch at each ingress port and simultaneously calculate the best *matching* at the egress ports. Furthermore, when optimizing for predictability, it must decide whether or not to *admit* a coflow.

2. **Lack of Future Knowledge:** Arrival times and characteristics of new coflows and their flows cannot be predicted.

Because the rate of any flow depends on its allocations at both ingress and egress ports, we refer to the inter-coflow scheduling problem as an instance of the concurrent open shop scheduling *with coupled resources* (**Remark B.1.2**). To the best of our knowledge, this variation of the problem – with ordering and matching requirements – has not appeared in the literature prior to this work.

5.5.2 Desirable Properties and Tradeoffs

Efficient scheduling (minimizing completion times) and predictable scheduling (guaranteeing coflow completions within their deadlines) are inherently conflicting. The former requires *preemptive* solutions to avoid *head-of-line* blocking. Shortest-remaining-time-first (SRTF) for optimally scheduling flows on a single link is an example [118]. Preemption, in the worst case, can lead to starvation; e.g., SRTF starves long flows. The latter, on the contrary, requires admission control to provide guarantees.

We expect an ideal scheduler to satisfy the following additional goals:

1. **Starvation Freedom:** Coflows, irrespective of their characteristics, should not starve for arbitrarily long periods.

2. **Work-conserving Allocation:** Available resources should be used as much as possible.

The former ensures eventual completion of coflows irrespective of system load. The latter avoids underutilization of the network, which intuitively should result in lower CCTs and higher admissions. However, both are at odds with our primary objectives (§B.2).

Predictable scheduling has an additional goal.

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

In the following, we present algorithms that achieve high network utilization, and ensure starvation freedom when minimizing CCT (§5.5.3) and guarantees completion of admitted coflows when maximizing predictability (§5.5.4).

5.5.3 Inter-Coflow Scheduling to Minimize CCT

Given the complexity, instead of finding an optimal algorithm, we focus on understanding what an offline optimal schedule might look like under simplifying assumptions. Next, we compute the minimum time to complete a single coflow. We use this result as a building block to devise a scheduling heuristic and an iterative bandwidth allocation algorithm. We conclude by presenting the necessary steps to transform our offline solution to an online one with guarantees for starvation freedom and work conservation.

Solution Approach

Consider the offline problem of scheduling $|\mathbb{C}|$ coflows ($\mathbb{C} = \{C_1, C_2, \ldots, C_{|\mathbb{C}|}\}$) that arrived at time 0. The optimality of the shortest-remaining-processing-time (SRTF) heuristic on a single link suggests that shortest- or smallest-first schedules are the most effective in minimizing completion times [37,118]. However, in the multi-link scenario, links can have different schedules. This makes the search space exponentially large – there are $((|\mathbb{C}|P)!)^P$ possible solutions when scheduling $|\mathbb{C}|$ coflows with P^2 flows each on a $P \times P$ fabric!

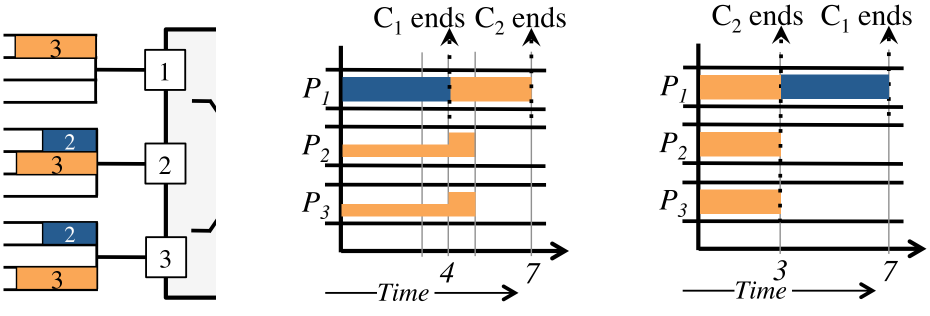
If we remove the capacity constraints from either ingress or egress ports, under the assumptions of Section 5.5.1, the coflow scheduling problem simplifies to the traditional concurrent open shop scheduling problem, which has optimal *permutation schedules* [149]; meaning, scheduling coflows one after another is sufficient, and searching within the $|\mathbb{C}|!$ possible schedules is enough. Unfortunately, permutation schedules can be suboptimal for coupled resources (**Theorem B.3.1**), which can lead to flow preemptions at arbitrary points in time – not just at coflow arrivals and completions (**Remark B.3.2**). To avoid incessant rescheduling, we restrict ourselves to permutation schedules.

The Smallest Effective Bottleneck First Heuristic

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

Given the optimality of the shortest- or smallest-first policy in minimizing the average FCT [37, 118], a natural choice for scheduling coflows would be to approximate that with a *Shortest-Coflow-First (SCF)* heuristic. However, SCF does not take into account the width of a coflow. A width-based alternative to SCF is the *Narrowest-Coflow-First (NCF)* heuristic, but NCF cannot differentiate between a short coflow from a long one. A *smallesT-Coflow-First (TCF)* heuristic is a better alternative than the two – while SCF can be influenced just by a single long flow (i.e., coflow length) and NCF relies only on coflow width, TCF responds to both.

However, the actual completion time of coflow (Γ) actually depends on its bottleneck (see Equation (4.1)). We propose the *Smallest Effective Bottleneck First (SEBF)* heuristic that considers a coflow's length, width, size, and skew to schedule it in the smallest-Γ-first order. Figure 5.3 shows an example: although C_2 (orange/light) is bigger than C_1 in length, width, and size, SEBF schedules it first to reduce the average CCT to 5 time units from 5.5. While no heuristic is perfect, we found SEBF to perform noticeably better than TCF, SCF, and NCF in both trace-driven and synthetic simulations. Additionally, SEBF performs more than $3\times$ better than non-preemptive coflow schedulers (§5.7.4).



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

Figure 5.3: Allocations of *egress* port capacities (vertical axis) for the coflows in (a) on a 3 × 3 fabric for different coflow scheduling heuristics.

Iterative MADD

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

Given Γ, *the minimum completion time of a coflow can be attained as long as all flows finish at time* Γ. We can ensure that by setting the rates (r_{ij}) of each flow to d_{ij}/Γ. We defined this algorithm (lines 7–10 in Pseudocode 1) as MADD. *It allocates the least amount of bandwidth to complete a coflow in minimum possible time.*

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

From Offline to Online

Transforming to an online scheduling environment from an offline one requires addressing at least two major challenges: *achieving work conservation* and *avoiding starvation*. We address both through minor adjustments to the proposed solutions.

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

Pseudocode 1 Coflow Scheduling to Minimize CCT

```
 1: procedure allocBandwidth(Coflows $\mathbb{C}$, $Rem(.)$, Bool $cct$)
 2:     for all $C \in \mathbb{C}$ do
 3:         $\tau = \Gamma^C$ (Calculated using Equation (4.1))
 4:         if not $cct$ then
 5:             $\tau = D^C$
 6:         end if
 7:         for all $d_{ij} \in C$ do                                  ▷ MADD
 8:             $r_{ij} = d_{ij}/\tau$
 9:             Update $Rem(P_i^{\text{in}})$ and $Rem(P_j^{\text{out}})$
10:         end for
11:     end for
12: end procedure

13: procedure minCCTOffline(Coflows $\mathbb{C}$, $C$, $Rem(.)$)
14:     $\mathbb{C}'$ = SORT_ASC ($\mathbb{C} \cup C$) using SEBF
15:     allocBandwidth($\mathbb{C}'$, $Rem(.)$, true)
16:     Distribute unused bandwidth to $C \in \mathbb{C}'$        ▷ Work conservation (§5.5.3)
17:     return $\mathbb{C}'$
18: end procedure

19: procedure minCCTOnline(Coflows $\mathbb{C}$, $C$, $Rem(.)$)
20:     if timeSinceLastDelta() $< T$ then                     ▷ $T$-interval: Decrease CCT
21:         $\mathbb{C}'$ = minCCTOffline($\mathbb{C}$, $C$, $Rem(.)$)
22:         Update $\mathbb{C}_{\text{zero}}$, the set of starved coflows
23:     else                                                ▷ $\delta$-interval: Starvation freedom
24:         $C^* = \bigcup C$ for all $C \in \mathbb{C}_{\text{zero}}$
25:         Apply MADD on $C^*$
26:         Schedule a call to minCCTOnline(.) after $\delta$ interval
27:     end if
28: end procedure
```

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

This ensures that *all* coflows receive non-zero service in every $(T + \delta)$ interval and eventually complete. This is similar to ensuring at least one ticket for each process in lottery scheduling [197].

Pseudocode 2 Coflow Scheduling to Guarantee Completion Within Deadline

```
 1: procedure meetDeadline(Coflows $\mathbb{C}$, $C$, $Rem(.)$)
 2:     allocBandwidth($\mathbb{C}$, $Rem(.)$, false)        ▷ Recalculate min rates for $C \in \mathbb{C}$
 3:     if $\Gamma^C \leq D^C$ then                           ▷ Admission control
 4:         $\mathbb{C}'$ = Enqueue $C$ to $\mathbb{C}$       ▷ Add $C$ in the arrival order
 5:         allocBandwidth($\mathbb{C}'$, $Rem(.)$, false)
 6:         Distribute unused bandwidth to $C \in \mathbb{C}'$   ▷ Work conservation
 7:         return true
 8:     end if
 9:     Distribute unused bandwidth to $C \in \mathbb{C}$
10:     return false
11: end procedure
```

Avoiding starvation comes at the cost of an increased average CCT. At every $(T + \delta)$ interval, the total CCT increases by at most $|\mathbb{C}|\delta$, and it depends on the ratio of T and δ.

5.5.4 Inter-Coflow Scheduling to Guarantee Deadline

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

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

5.6 Design Details

We have implemented Varys in about $5,000$ lines of Scala with extensive use of `Akka` [1] for messaging and the `Kryo` serialization library [18]. This section illustrates how frameworks interact with Varys and discusses how Varys schedules coflows.

5.6.1 Varys Client Library: The Coflow API

Varys client library provides an API similar to DOT [190] to abstract away the underlying scheduling and communication mechanisms. Cluster frameworks (e.g., Spark, Hadoop, or Dryad) must create `VarysClient` objects to invoke the API and interact with Varys. User jobs, however, do *not* require any modifications.

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

Table 5.1: The Coflow API

The coflow API has four primary methods (Table 5.1). Framework drivers initialize a coflow through `register()`, which returns a unique *coflowId* from Varys master. *numFlows* is a hint for the scheduler on when to consider the coflow READY to be scheduled. Additional information or hints (e.g., coflow deadline or dependencies) can be given through *options* – an optional list of key-value pairs. The matching `unregister()` signals coflow completion.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.6.2 Inter-Coflow Scheduling in Varys

Varys implements the algorithms in Section 5.5 for inter-coflow scheduling, which are called upon coflow arrival and completion events. $Rem(.)$ is calculated from the aggregated measurements collected by Varys daemons. Pseudocode 3 lists the key event handlers in Varys. Initially, all coflows are marked UNREADY (onCoflowRegister). The size of a coflow is updated as `VarysClient` instances periodically notify flow-related events using onFlowPut and onFlowGet. A coflow is considered READY to be scheduled after *numFlows* `get()` calls (onFlowGet). Varys master groups the new allocations calculated by the scheduler by respective `VarysClients` and sends the changes asynchronously.

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

Pseudocode 3 Message Handlers in Varys Scheduler

```
procedure onCoflowRegister(Coflow $C$)
    Mark $C$ as UNREADY
    $\mathbb{C}_{\text{cur}} = \mathbb{C}_{\text{cur}} \cup \{C\}$        ▷ $\mathbb{C}_{\text{cur}}$ is the set of all coflows
end procedure

procedure OnCoflowUnregister(Coflow $C$)
    $\mathbb{C}_{\text{cur}} = \mathbb{C}_{\text{cur}} \setminus \{C\}$
    $\mathbb{C}_{\text{ready}}$ = $\mathbb{C}$.filter(READY)
    Call appropriate scheduler from Section 5.5
end procedure

procedure onFlowPut(Flow $f$, Coflow $C$)
    Update $Size(C)$ and relevant data structures
end procedure

procedure onFlowGet(Flow $f$, Coflow $C$)
    Update relevant data structures
    Mark $C$ as READY after get() is called numFlows times
    if $C$ is READY then
        $\mathbb{C}_{\text{ready}}$ = $\mathbb{C}$.filter(READY)
        Call appropriate scheduler from Section 5.5
    end if
end procedure
```

5.7 Evaluation

We evaluated Varys through a set of experiments on 100-machine EC2 [3] clusters using a Hive/MapReduce trace collected from a large production cluster at Facebook. For a larger scale evaluation, we used a trace-driven simulator that performs a detailed replay of task logs from the same trace. The highlights of our evaluation are as follows.

- For communication-dominated jobs, Varys improves the average (95th percentile) CCT and job completion time by up to $3.16\times$ ($3.84\times$) and $2.5\times$ ($2.94\times$), respectively, over per-flow fairness. Across all jobs, the improvements are $1.85\times$ ($1.74\times$) and $1.25\times$ ($1.15\times$) (§5.7.2).

- Simulations show that Varys improves the average (95th percentile) CCT by $3.26\times$ ($5.19\times$) over per-flow prioritization mechanisms (§5.7.2).

- Varys enables almost $2\times$ more coflows to complete within their deadlines in EC2 experiments (§5.7.3).

- Varys does not cause starvation, and simulations show Varys to be $3.33\times$ faster than a non-preemptive solution ($6.9\times$ at the 95th percentile) (§5.7.4).

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

Table 5.2: Jobs binned by time spent in communication.

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

Table 5.3: Coflows binned by length (**S**hort and **L**ong) and width (**N**arrow and **W**ide).

5.7.1 Methodology

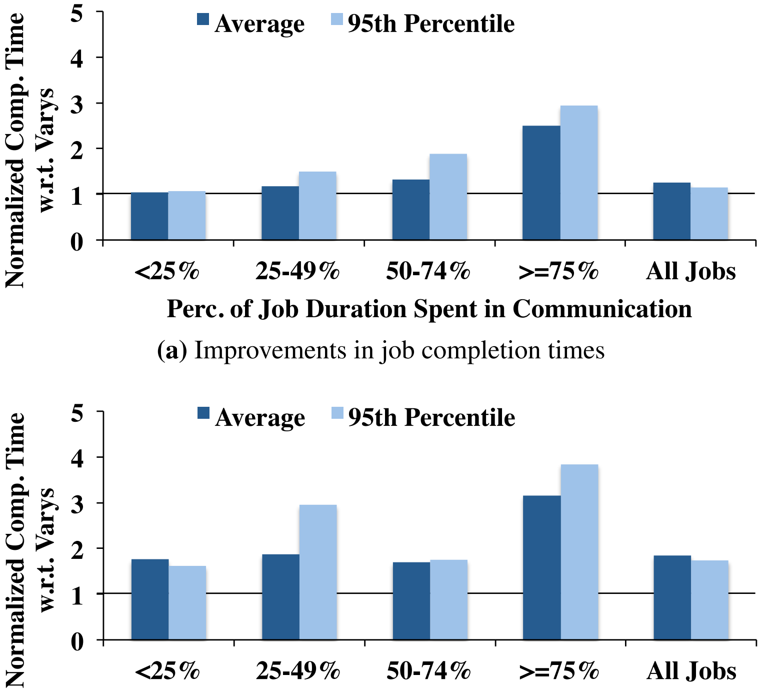
Workload Our workload is based on a Hive/MapReduce trace at Facebook that was collected on a 3000-machine cluster with 150 racks. The original cluster had a $10:1$ core-to-rack oversubscription ratio with a total bisection bandwidth of 300 Gbps. We scale down jobs accordingly to match the maximum possible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

We consider jobs with non-zero shuffle and divide them into bins (Table 5.2) based on the fraction of their durations spent in shuffle. Table 5.3 divides the coflows in these jobs into four categories based on their characteristics. We consider a coflow to be *short* if its longest flow is less than 5 MB and *narrow* if it involves at most 50 flows.

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

Cluster Our experiments use extra large high-memory EC2 instances, which appear to occupy entire physical machines and have enough memory to perform all experiments without introducing disk overheads. We observed bandwidths close to 800 Mbps per machine on clusters of 100 machines. We use a compute engine similar to Spark [208] that uses the coflow API. We use $\delta = 200$ milliseconds and $T = 2$ seconds as defaults.

Simulator We use a trace-driven simulator to gain more insights into Varys's performance at a larger scale. The simulator performs a detailed task-level replay of the Facebook trace. It preserves input-to-output ratios of tasks, locality constraints, and inter-arrival times between jobs. It runs at $10s$ decision intervals for faster completion.



(a) Improvements in job completion times

(b) Improvements in time spent in communication

Figure 5.4: [EC2] Average and 95th percentile improvements in job and communication completion times over per-flow fairness using Varys.

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

$$\text{Normalized Completion Time} = \frac{\text{Compared Duration}}{\text{Varys's Duration}}$$

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

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

The baseline for our deployment is TCP fair sharing. We compare the trace-driven simulator against per-flow fairness as well. Due to the lack of implementations of per-flow prioritization mechanisms [37, 118], we compare against them only in simulation.

5.7.2 Varys's Performance in Minimizing CCT

Figure 5.4a shows that inter-coflow scheduling reduced the average and 95th percentile completion times of communication-dominated jobs by up to $2.5\times$ and $2.94\times$, respectively, in EC2 experi-

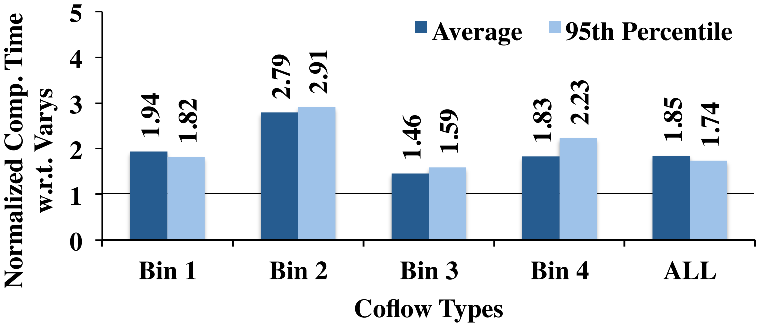


Figure 5.5: [EC2] Improvements in the average and 95th percentile CCTs using coflows w.r.t. the default per-flow fairness mechanism.

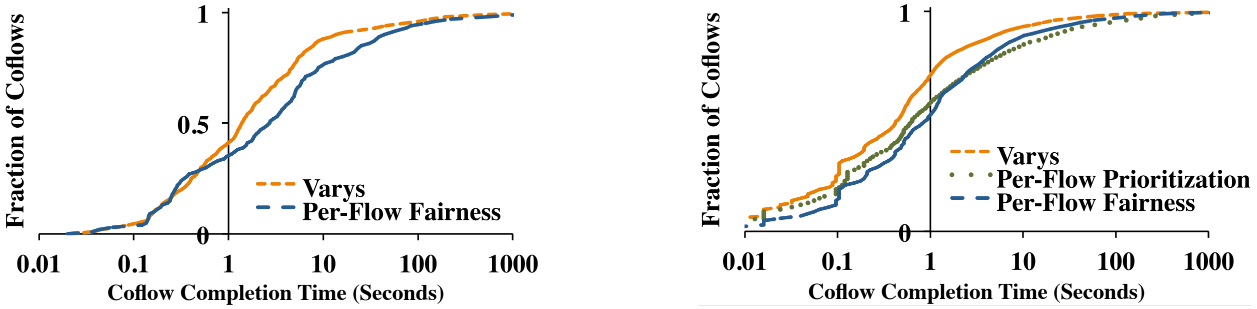
ments. Corresponding average and 95th percentile improvements in the average CCT (CommTime) were up to 3.16× and 3.84× (Figure 5.4b). Note that varying improvements in the average CCT in different bins are not correlated, because it depends more on coflow characteristics than that of jobs. However, as expected, jobs become increasingly faster as the communication represent a higher fraction of their completion times. Across all bins, the average end-to-end completion times improved by 1.25× and the average CCT improved by 1.85×; corresponding 95th percentile improvements were 1.15× and 1.74×.

Figure 5.5 shows that Varys improves CCT for diverse coflow characteristics. Because bottlenecks are not directly correlated with a coflow's length or width, pairwise comparisons across bins – especially those involving bin-2 and bin-3 – are harder. We do observe more improvements for coflows in bin-1 than bin-4 in terms of average CCT, even though their 95th percentile improvements contradict. This is due to coordination overheads in Varys – recall that Varys does not handle small coflows to avoid fixed overheads.

Figure 5.6a presents comparative CDFs of CCTs for all coflows. Per-flow fairness performs better – 1.08× on average and 1.25× at the 95th percentile – only for some of the tiny, sub-second ($<$ 500 milliseconds) coflows, which still use TCP fair sharing. As coflows become larger, the advantages of coflow scheduling becomes more prominent. We elaborate on Varys's overheads next; later, we show simulation results that shed more light on the performance of small coflows in the absence of coordination overheads.

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

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



(a) EC2

(b) Simulation

Figure 5.6: CCT distributions for Varys, per-flow fairness, and per-flow prioritization schemes (a) in EC2 deployment and (b) in simulation. The X-axes are in logarithmic scale.

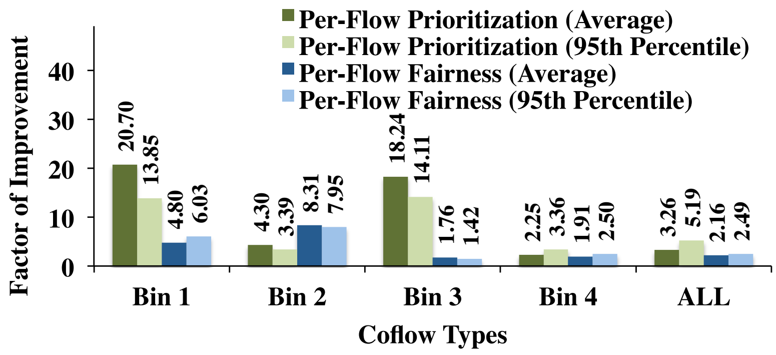


Figure 5.7: [Simulation] Improvements in the average and 95th percentile CCTs using inter-coflow scheduling.

A large fraction of these overheads could be avoided in the presence of in-network isolation of control plane messages [89].

Trace-Driven Simulation We compared the performance of inter-coflow scheduling against per-flow fairness and prioritization schemes in simulations. Without coordination overheads, the improvements are noticeably larger (Figure 5.7) – the average and 95th percentile CCTs improved by $2.16\times$ and $2.49\times$ over per-flow fairness and by $3.26\times$ and $5.19\times$ over per-flow prioritization.

Note that comparative improvements for bin-1 w.r.t. other bins are significantly larger than that in experiments because of the absence of scheduler coordination overheads. We observe larger absolute values of improvements in Figure 5.7 in comparison to the ones in Figure 5.5. Primary factors for this phenomenon include instant scheduling, zero-latency setup/cleanup/update of coflows, and perfectly timed flow arrivals (i.e., coflows are READY to be scheduled upon arrival) in the simulation. In the absence of these overheads, we see in Figure 5.6b that Varys can indeed outperform per-flow schemes even for sub-second coflows.

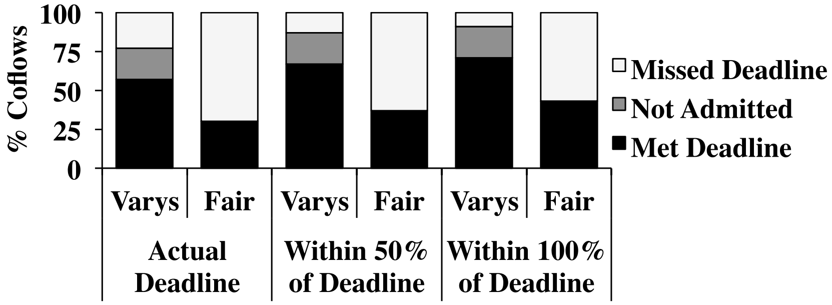


Figure 5.8: [EC2] Percentage of coflows that meet deadline using Varys in comparison to per-flow fairness. Increased deadlines improve performance.

What About Per-Flow Prioritization? Figure 5.7 highlights that per-flow prioritization mechanisms are even worse (by $1.51\times$) than per-flow fairness provided by TCP when optimizing CCTs. The primary reason is indiscriminate interleaving across coflows – while all flows make some progress using flow-level fairness, per-flow prioritization favors only the small flows irrespective of the progress of their parent coflows. However, as expected, flow-level prioritization is still $1.08\times$ faster than per-flow fairness in terms of the average FCT. Figure 5.6b presents the distribution of CCTs using per-flow prioritization in comparison to other approaches.

How Far are We From the Optimal? While finding the optimal schedule is infeasible, we tried to find an optimistic estimation of possible improvements by comparing against an *offline* 2-approximation combinatorial ordering heuristic for coflows *without* coupled resources [149]. We found that the average CCT did not change using the combinatorial approach. For bin-1 to bin-4, the changes were $0.78\times$, $1.03\times$, $0.92\times$, and $1.13\times$, respectively.

5.7.3 Varys's Performance for Deadline-Sensitive Coflows

Inter-coflow scheduling allowed almost $2\times$ more coflows to complete within corresponding deadlines in EC2 experiments (Figure 5.8) – 57% coflows met their deadlines using Varys as opposed to 30% using the default mechanism. Coflows across different bins experienced similar results, which is expected because Varys does not differentiate between coflows when optimizing for deadlines.

Recall that because the original trace did not contain coflow-specific deadlines, we introduced them based on the minimum CCT of coflows (§5.7.1). Hence, we did not expect 100% admission rate. However, a quarter of the admitted coflows failed to meet their deadlines. This goes back to the lack of network support in estimating utilizations and enforcing Varys-determined allocations: Varys admitted more coflows than it should have had, which themselves missed their deadlines and caused some others to miss as well. Trace-driven simulations later shed more light on this.

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

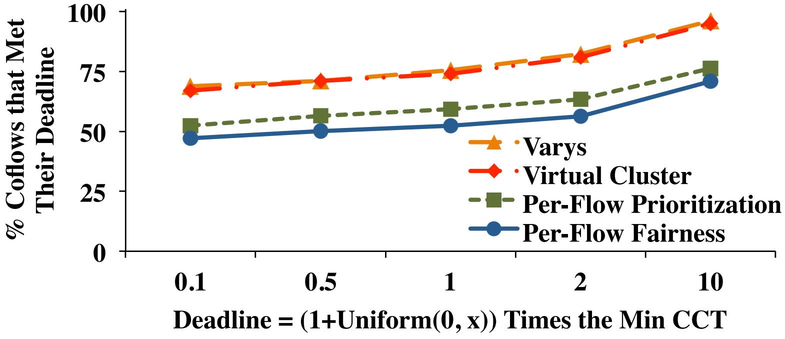


Figure 5.9: [Simulation] More coflows meet deadline using inter-coflow scheduling than using per-flow fairness and prioritization schemes.

Trace-Driven Simulation In trace-driven simulations, for the default case (x=1), Varys admitted 75% of the coflows and *all* of them met their deadlines (Figure 5.9). Note that the admission rate is lower than that in our experiments. Prioritization schemes fared better than per-flow fairness unlike when the objective was minimizing CCT: 59% coflows completed within their deadlines in comparison to 52% using fair sharing.

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

What About Reservation Schemes? Because the impact of admission control is similar to reserving resources, we compared our performance with that of the Virtual Cluster (VC) abstraction [46], where all machines can communicate at the same maximum rate through a virtual non-blocking switch. The VC abstraction admitted and completed slightly fewer coflows (73%) than Varys (75%), because reservation using VCs is more conservative.

5.7.4 Impact of Preemption

While minimizing CCT, preemption-based mechanisms can starve certain coflows when the system is overloaded. Varys takes precautions (§5.5.3) to avoid such scenarios. As expected, we did not observe any perpetual starvation during experiments or simulations.

What About a Non-Preemptive Scheduler? Processing coflows in their arrival order (i.e., FIFO) avoids starvation [67]. However, simulations confirmed that head-of-line blocking significantly hurts performance – especially, the short coflows in bin-1 and bin-3.

We found that processing coflows in the FIFO order can result in $31.63\times$, $6.06\times$, $21.72\times$, and $1.95\times$ slower completion times for bin-1 to bin-4. The average (95th percentile) CCT became $3.33\times$ ($6.9\times$) slower than that using Varys.

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

Table 5.4: Four extreme coflow mixes from the Facebook trace.

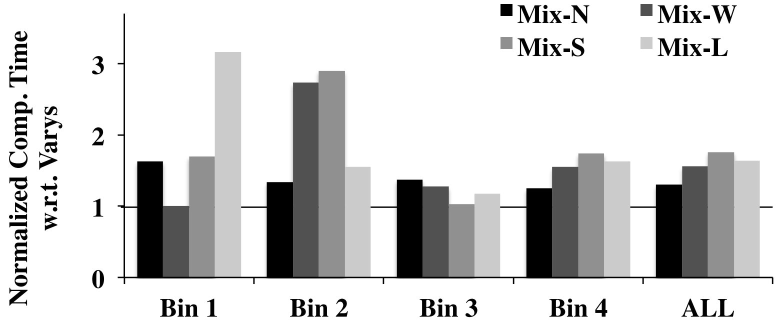


Figure 5.10: [EC2] Improvements in the average CCT using coflows for different coflow mixes from Table 5.4.

5.7.5 Impact on Network Utilization

To understand Varys's impact on network utilization, we compared the ratios of *makespans* in the original workload as well as the ones in Table 5.4. Given a fixed workload, a change in makespan means a change in aggregate network utilization.

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

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

5.7.6 Impact of Coflow Mix

To explore the impact of changes in the coflow mix, we selected four extreme hours (Table 5.4) from the trace and performed hour-long experiments on EC2. These hours were chosen based on the high percentage of certain types of coflows (e.g., narrow ones in Mix-N) during those periods.

Figure 5.10 shows that the average CCT improves irrespective of the mix, albeit in varying degrees. Observations made earlier (§5.7.2) still hold for each mix. However, identifying the exact reason(s) for different levels of improvements is difficult. This is due to the online nature of the experiments – the overall degree of improvement depends on the instantaneous interplay of con-

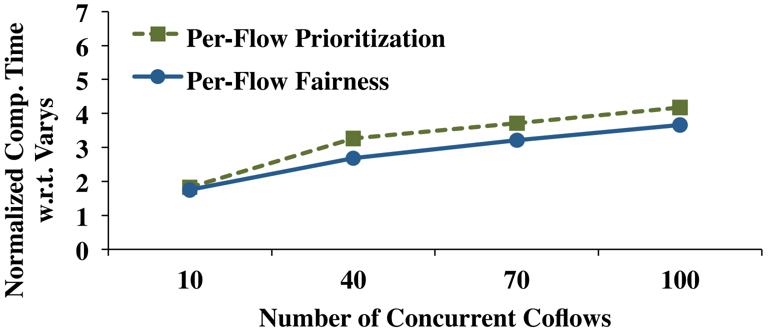


Figure 5.11: [Simulation] Improvements in the average CCT for varying numbers of concurrent coflows.

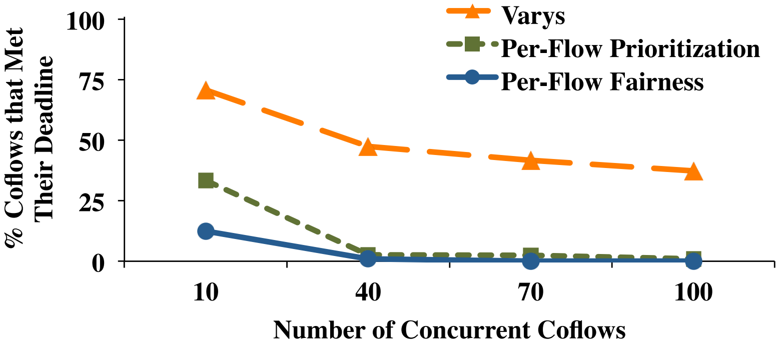


Figure 5.12: [Simulation] Changes in the percentage of coflows that meet deadlines for varying numbers of concurrent coflows.

current coflows. We also did not observe any clear correlation between the number of coflows or workload size and corresponding improvements.

5.7.7 Impact of Cluster/Network Load

So far we have evaluated Varys's improvements in online settings, where the number of concurrent coflows varied over time. To better understand the impact of network load, we used the same coflow mix as the original trace but varied the number of concurrent coflows in an offline setting. We see in Figure 5.11 that Varys's improvements increase with increased concurrency: per-flow mechanisms fall increasingly further behind as they ignore the structures of more coflows. Also, flow-level fairness consistently outperforms per-flow prioritization mechanisms in terms of the average CCT.

Deadline-Sensitive Coflows We performed a similar analysis for deadline-sensitive coflows. Because in this case Varys's performance depends on the arrival order, we randomized the coflow order across runs and present their average in Figure 5.12. We observe that as the number of coexisting coflows increases, a large number of coflows (37% for 100 concurrent coflows) meet their deadlines using Varys; per-flow mechanisms completely stop working even before. Also, per-flow prioritization outperforms (however marginally) flow-level fairness for coflows with deadlines.

5.8 Discussion

Scheduling With Unknown Flow Sizes Knowing or estimating exact flow sizes is difficult in frameworks that push data to the next stage as soon as possible [119], and without known flow sizes, preemption becomes impractical. FIFO scheduling can solve this problem, but it suffers from head-of-line blocking (§5.7.4). We believe that coflow-level fairness can be a nice compromise between these two extremes. However, the definition and associated properties of fairness at the coflow level is an open problem.

Avoiding Overheads for Small Coflows Varys's centralized design makes it less useful for small coflows (§5.4); however, small coflows contribute less than 1% of the traffic in data-intensive clusters (§A.3). Furthermore, in-network isolation of control plane messages [89] or faster signaling channels such as RDMA [85] can reduce Varys's application-layer signaling overheads (§5.7.2) to support even smaller coflows.

One possible approach toward mitigating this issue is designing decentralized approximations of our algorithms. This requires new algorithms and possible changes to network devices, unlike our application-layer design. Another approach would be to automatically avoiding coordination until it becomes absolutely necessary.

Handling Coflow Dependencies While most jobs require only a single coflow, dataflow pipelines (e.g., Dryad, Spark) can create multiple coflows with dependencies between them [65]. A simple approach to support coflow dependencies would be to order first by ancestry and then breaking ties using SEBF. Some variation of the Critical-Path Method [131] might perform even better. We leave it as a topic of future work. Note that dependencies can be passed along to the scheduler through *options* in the `register()` method.

Multi-Wave Coflows Large jobs often schedule mappers in multiple waves [39]. A job can create separate coflows for each wave. Alternatively, if the job uses its wave-width (i.e., the number of parallel mappers) as *numFlows* in `register()`, Varys can handle each wave separately. Applications can convey information about wave-induced coflows to the scheduler as dependencies.

In-Network Bottlenecks Varys performs well even when the network is not a non-blocking switch (§5.7). If likely bottleneck locations are known, e.g., rack-to-core links are typically oversubscribed [63], Varys can be extended to allocate rack-to-core bandwidth instead of NIC bandwidth. When bottlenecks are unknown, e.g., due to in-network failures, routing, or load imbalance, Varys can react based on bandwidth estimations collected by its daemons. Nonetheless, designing and deploying coflow-aware routing and load balancing techniques remain an open challenge.

5.9 Related Work

Coflow Schedulers Varys improves over Orchestra [67] in four major ways. First, Orchestra primarily optimizes individual coflows and uses FIFO among them; whereas, Varys uses an efficient coflow scheduler to significantly outperform FIFO. Second, Varys supports deadlines and ensures guaranteed coflow completion. Third, Varys uses a rate-based approach instead of manipulating the number of TCP flows, which breaks if all coflows do not share the same endpoints. Finally, Varys supports coflows from multiple frameworks such as Mesos [117] handles non-network resources.

Baraat [82] is a FIFO-based decentralized coflow scheduler focusing on small coflows. It uses fair sharing to avoid head-of-line blocking and does not support deadlines. Furthermore, we formulate the coflow scheduling problem and analyze its characteristics.

Datacenter Traffic Management Hedera [33] manages flows using a centralized scheduler to increase network throughput, and MicroTE [51] adapts to traffic variations by leveraging their short-term predictability. However, both work with flows and are unsuitable for optimizing CCTs. Sinbad [63] uses endpoint flexible transfers for load balancing. Once it makes network-aware placement decisions, Varys can optimize cross-rack write coflows.

High Capacity Networks Full bisection bandwidth topologies [106, 158] do not imply contention-free networks. In the presence of skewed data and hotspot distributions [63], managing edge bandwidth is still necessary. Inter-coflow scheduling improves performance and predictability even in these high capacity networks.

Traffic Reduction Techniques Data locality [77], both disk [40, 207] and memory [39], reduces network usage only during reads. The amount of network usage due to intermediate data communication can be reduced by pushing filters toward the sources [30, 111]. Our approach is complementary; i.e., it can be applied to whatever data traverses the network after applying those techniques.

Network Sharing Among Tenants Fair sharing of network resources between multiple tenants has received considerable attention [46, 171, 184, 202]. Our work is complementary; we focus on optimizing performance of concurrent coflows within a single administrative domain, instead of achieving fairness among competing entities. Moreover, we focus on performance and predictability as opposed to fairness.

Concurrent Open Shop Scheduling Inter-coflow scheduling has its roots in the concurrent open shop scheduling problem [180], which is strongly NP-hard for even two machines. Even in the offline scenario, the best known result is a 2-approximation algorithm [149], and it is inapproximable within a factor strictly less than $\frac{6}{5}$ if $P \neq NP$ [149]. Our setting is different as follows. First, machines are not independent; i.e., links are coupled because each flow involves a source and a destination. Second, jobs are not known a priori; i.e., coflows arrive in an online fashion.

5.10 Summary

In this chapter, we have implemented clairvoyant coflows – i.e., when a coflow's structure is known upon its arrival – in a system called Varys and introduced and analyzed the *concurrent open shop scheduling with coupled resources* problem. To minimize coflow completion times (CCT), we proposed the SEBF heuristic to schedule coflows and the MADD algorithm to allocate bandwidth to their flows. Together, they decrease the average CCT without starving any coflow and maintain high network utilization. Through EC2 deployments and trace-driven simulations, we showed that Varys outperforms per-flow mechanisms and non-preemptive coflow schedulers by more than $3\times$. Furthermore, by applying MADD in conjunction with admission control, Varys allowed up to $2\times$ more coflows to meet their deadlines in comparison to per-flow schemes.

Clairvoyant inter-coflow scheduling, however, is only a first step in understanding the possibilities of coflows in improving communication performance in shared clusters. It raises a variety of exciting research questions – including how to schedule without knowing flow sizes, what is fair among coflows, how to avoid/decrease coordination overheads, and how to handle coflow dependencies – that we address in subsequent chapters.

Chapter 6

Non-Clairvoyant Inter-Coflow Scheduling

In the previous chapter, we showed how to efficiently schedule clairvoyant coflows – i.e., coflows without any dynamic changes once they have started – in a shared cluster. In this chapter, we focus on approximating similar gains even when coflow characteristics are not known a priori and can change over time. By focusing on non-clairvoyant inter-coflow scheduling, we extend the applicability of the coflow abstraction to a wider variety of distributed data-parallel applications and to unforeseen network-, cluster-, and application-level dynamics.

The remainder of this chapter is organized as follows. The next section covers the state-of-the-art in optimizing communication performance of coexisting distributed data-parallel applications when communication characteristics are not known a priori. Section 6.2 outlines our proposal, which we have implemented in a system called Aalo, followed by the challenges and potential gains from non-clairvoyant inter-coflow scheduling in Section 6.3. Section 6.4 presents a system-level overview of Aalo, and in Section 6.5, we present the non-clairvoyant scheduling algorithm. Next, we extend our proposal to support coflow dependencies in Section 6.6 and present Aalo's implementation details in Section 6.7. We then evaluate Aalo's performance in Section 6.8 through deployments and simulations. Next, we discuss in Section 6.9 its current limitations and future research directions, survey related work in Section 6.10, and summarize our findings in Section 6.11.

6.1 Background

Inter-coflow scheduling to minimize the average coflow completion time (CCT) is NP-hard [68]. Existing FIFO-based solutions, e.g., Baraat [82] and Orchestra [67], compromise on performance by multiplexing coflows to avoid head-of-line blocking. Varys [68] improves performance using heuristics such as smallest-bottleneck-first and smallest-total-size-first, but it assumes *complete* prior knowledge of coflow characteristics such as the number of flows, their sizes, and endpoints.

Unfortunately, in many cases, coflow characteristics are unknown a priori. Multi-stage jobs use pipelining between successive computation stages [9, 73, 119, 181] – i.e., data is transferred as soon as it is generated – making it hard to know the size of each flow. Moreover, a single stage may consist of multiple waves [39],[1] preventing all flows within a coflow from starting together. Finally, task failures and speculation [77,119,208] result in redundant flows; meaning, the exact number of flows or their endpoints cannot be determined until a coflow has completed. Consequently, coflow schedulers that rely on prior knowledge remain inapplicable to a large number of use cases.

In this chapter, we present a coordinated inter-coflow scheduler – called *Coflow-Aware Least-Attained Service* (CLAS) – to minimize the average CCT *without* any prior knowledge of coflow characteristics.

6.2 Solution Outline

CLAS generalizes the classic least-attained service (LAS) scheduling discipline [176] to coflows. However, instead of independently considering the number of bytes sent by each flow, CLAS takes into account the *total* number of bytes sent by *all* the flows of a coflow. In particular, CLAS *assigns each coflow a priority that is decreasing in the total number of bytes the coflow has already sent.* As a result, smaller coflows have higher priorities than larger ones, which helps in reducing the average CCT. Note that for heavy-tailed distributions of coflow sizes, CLAS approximates the smallest-total-size-first heuristic,[2] which we have been shown to work well for realistic workloads in the previous chapter.

For light-tailed distributions of coflow sizes, however, a straightforward implementation of CLAS can lead to fine-grained sharing,[3] which is known to be suboptimal for minimizing the average CCT [67,68,82]. The optimal schedule in such cases is FIFO [82].

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

We implement D-CLAS using a multi-level scheduler, where each queue maintains all the coflows with the same priority. Within each queue, coflows follow the FIFO order. Across queues, we use weighted fair queuing at the coflow granularity, where weights are based on the queues' priorities. Using weighted sharing, instead of strict priorities, avoids starvation because each queue is guaranteed to receive some non-zero service. By approximating FIFO (as in Baraat [82]) for light-tailed coflows and smallest-coflow-first (as in Varys [68]) for heavy-tailed coflows, the proposed scheduler works well in practice.

We have implemented D-CLAS in Aalo,[4] a system that supports coflow dependencies and pipelines, and works well in presence of cluster dynamics such as multi-wave scheduling. Aalo

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

[2] Under the heavy-tailed distribution assumption, the number of bytes already sent is a good predictor of the actual coflow size [159].

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

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

requires *no* prior knowledge of coflow characteristics, e.g., coflow size, number of flows in the coflow, or its endpoints. While Aalo needs to track the total number of bytes sent by a coflow to update its priority,[5] this requires only loose coordination as priority thresholds are coarse. Moreover, coflows whose total sizes are smaller than the first priority threshold require no coordination. Aalo runs *without* any changes to the network or user jobs, and data-parallel applications require minimal changes to use it (§6.7).

We deployed Aalo on a 100-machine EC2 cluster and evaluated it by replaying production traces from Facebook and with TPC-DS [25] queries. Aalo improved CCTs both on average (up to $2.25\times$) and at high percentiles ($2.93\times$ at the 95th percentile) w.r.t. per-flow fair sharing, which decreased corresponding job completion times. Aalo's average improvements were within 12% of Varys for single-stage, single-wave coflows, and it outperformed Varys for multi-stage, multi-wave coflows by up to $3.7\times$ by removing artificial barriers and through dependency-aware scheduling. In trace-driven simulations, we found Aalo to perform $2.7\times$ better than per-flow fair sharing and up to $16\times$ better than fully decentralized solutions that suffer significantly due to the lack of coordination. Simulations show that Aalo performs well across a wide range of parameter space and coflow distributions.

6.3 Motivation

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

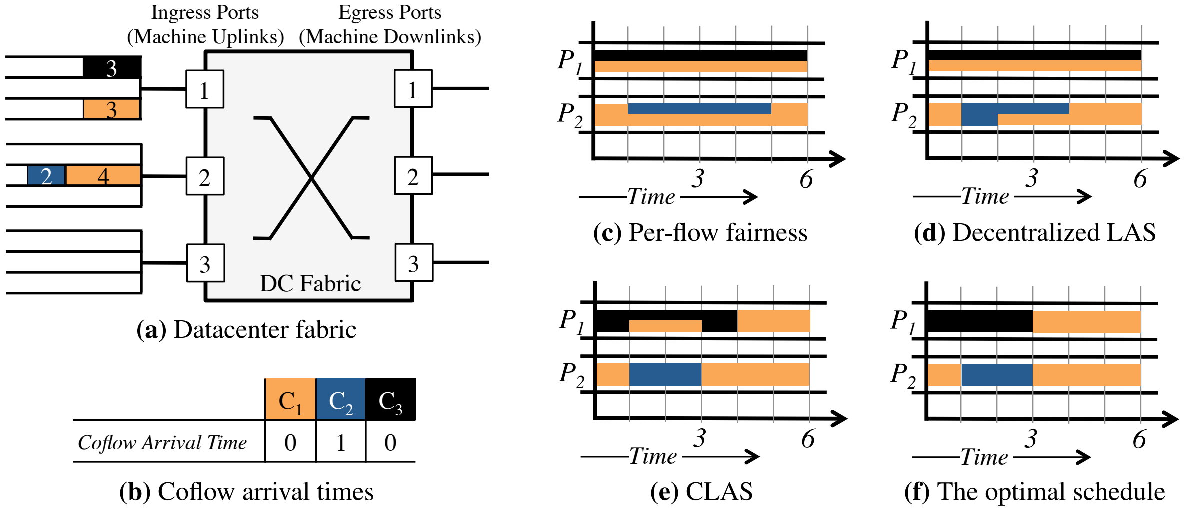
6.3.1 Challenges

An efficient non-clairvoyant [156] coflow scheduler must address two primary challenges:

1. **Scheduling Without Complete Knowledge:** Without a priori knowledge of coflows, heuristics such as smallest-bottleneck-first [68] are inapplicable – one cannot schedule coflows based on unknown bottlenecks. Worse, redundant flows from restarted and speculative tasks unpredictably affect a coflow's structure and bottlenecks. While FIFO-based schedulers (e.g., FIFO-LM in Baraat [82]) do not need complete knowledge, they multiplex to avoid head-of-line blocking, losing performance.

2. **Need for Coordination:** Coordination is the key to performance in coflow scheduling. We show analytically (**Theorem C.1.1**) and empirically (§6.8.2, §6.8.6) that fully decentralized schedulers such as Baraat [82] can perform poorly in data-parallel clusters because *local-only* observations are poor indicators of CCTs of large coflows. Fully centralized solutions such as Varys [68], on the contrary, introduce high overheads for small coflows.

[5] As stated by **Theorem C.1.1** in Appendix C.1, *any* coflow scheduler's performance can drop dramatically in the absence of coordination.



(a) Datacenter fabric

(b) Coflow arrival times

(c) Per-flow fairness

(d) Decentralized LAS

(e) CLAS

(f) The optimal schedule

Figure 6.1: Online coflow scheduling over a 3×3 datacenter fabric with three ingress/egress ports (a). Flows in ingress ports are organized by destinations and color-coded by coflows – C_1 in orange/light, C_2 in blue/dark, and C_3 in black. Coflows arrive online over time (b). Assuming each port can transfer one unit of data in one time unit, (c)–(f) depict the allocations of *ingress* port capacities (vertical axis) for different mechanisms: The average CCT for (c) per-flow fairness is 5.33 time units; (d) decentralized LAS is 5 time units; (e) CLAS with instant coordination is 4 time units; and (f) the optimal schedule is 3.67 time units.

6.3.2 Potential Gains

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

Consider the example in Figure 6.1 that compares three non-clairvoyant mechanisms against the optimal clairvoyant schedule. Per-flow fair sharing (Figure 6.1c) ensures max-min fairness in each link, but it suffers by ignoring coflows [67, 68]. Applying least-attained service (LAS) [45, 162, 176] in a decentralized manner (Figure 6.1d) does not help, because local observations cannot predict a coflow's actual size – e.g., it shares P_1 equally between C_1 and C_3, being oblivious to C_1's flow in P_2. The FIFO-LM schedule [82] would be at least as bad. Taking the total size of coflows into account through global coordination significantly decreases the average CCT (Figure 6.1e). The optimal solution (Figure 6.1f) exploits complete knowledge for the minimum average CCT. The FIFO schedule [67] would have resulted in a lower average CCT (4.67 time units) than decentralized LAS if C_3 was scheduled before C_1, and it would have been the same if C_1 was scheduled before C_3.

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

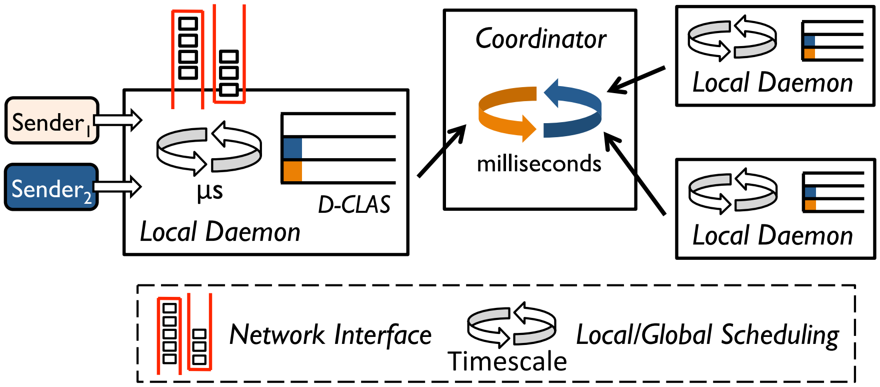


Figure 6.2: Aalo architecture. Computation frameworks interact with their local Aalo daemons using a client library, and the daemons periodically coordinate to determine the global ordering of coflows.

6.4 Aalo Overview

Aalo uses a non-clairvoyant coflow scheduler that optimizes the communication performance of data-intensive applications without a priori knowledge, while being resilient to the dynamics of job schedulers and data-parallel clusters. This section briefly overviews Aalo to help the reader follow the analysis and design of its scheduling algorithms (§6.5), mechanisms to handle dynamic events (§6.6), and design details (§6.7) presented in subsequent sections.

6.4.1 Problem Statement

Our goal is *dynamically prioritizing coflows without prior knowledge of their characteristics while respecting coflow dependencies*. This problem – *non-clairvoyant coflow scheduling with precedence constraints* – is NP-hard, because coflow scheduling with complete knowledge is NP-hard too [68]. In addition to minimizing CCTs, we must guarantee starvation freedom and work conservation.

6.4.2 Architectural Overview

Aalo uses a loosely-coordinated architecture (Figure 6.2), because full decentralization can render coflow scheduling pointless (**Theorem C.1.1**). It implements global and local controls at two time granularities:

- **Long-Term Global Coordination:** Aalo daemons send locally-observed coflow sizes to a central coordinator every $O(10)$ milliseconds. The coordinator determines the global coflow ordering using the D-CLAS framework (§6.5) and periodically sends out the updated schedule and globally-observed coflow sizes to all the daemons.

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

Frameworks use a client library to interact with the coordinator over the network to define coflows and their dependencies (§6.7). To send data, they must use the Aalo-provided `OutputStream`. The coordinator has an ID generator that creates unique CoflowIds while taking coflow dependencies into account (§6.6.1).

We have implemented Aalo in the application layer *without* any changes or support from the underlying network. We have deployed it in the cloud, and it performs well even for sub-second jobs (§6.8).

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

Scalability The faster Aalo daemons can coordinate, the better it performs. The number of coordination messages is linear with the number of daemons and *independent* of coflows. It is not a bottleneck for clusters with $O(100)$ machines, and our evaluation suggests that Aalo can scale up to $O(10,000)$ machines with minimal performance loss (§6.8.6). Most coflows are small and scheduled through local decisions; hence, unlike Varys, Aalo handles tiny coflows well.

6.5 Scheduling Without Prior Knowledge

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

[6] For data-intensive applications, 60% (85%) coflows are less than 100 MB (1 GB) in total size. Appendix A has the detailed distributions.

For brevity of exposition, we present the mechanisms in the context of single-stage, single-wave coflows. We extend them to handle multi-stage, multi-wave coflows as well as task failures and speculation in Section 6.6.

6.5.1 Complexity and Desirable Properties

The offline coflow scheduling problem – i.e., when all coflows arrive together and their characteristics are known a priori – is NP-hard [68]. Consequently, the non-clairvoyant coflow scheduling problem is NP-hard as well.

In the non-clairvoyant setting, smallest-bottleneck-first [68] – the best-performing clairvoyant heuristic – becomes inapplicable. This is because the bottleneck of a coflow is revealed only *after* it has completed. Instead, one must schedule coflows based on an attribute that

1. *can approximate its clairvoyant counterpart* using current observations, and

2. *involves all the flows* to avoid the drawbacks from the lack of coordination (**Theorem C.1.1**).

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

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

1. *Guarantee starvation freedom* for bounded CCTs;

2. *Ensure work conservation* to increase utilization; and

3. *Decrease coordination requirements* for scalability.

Coflow-Aware Least-Attained Service (CLAS) Although the smallest-total-size-first heuristic had been shown to perform marginally worse ($1.14\times$) than smallest-bottleneck-first in the clairvoyant setting [68], it becomes a viable option in the non-clairvoyant case. The *current size* of a coflow – i.e., how much it has already sent throughout the entire cluster – meets both criteria. This is because unlike a coflow's bottleneck, it *monotonically* increases with each flow regardless of start time or endpoints. As a result, setting a coflow's priority that decreases with it's current size can ensure that smaller coflows finish faster, which, in turn, minimizes the average CCT. Furthermore, it is a good indicator of actual size [159], because coflow size typically follows heavy-tailed distribution [39, 68].

We refer to this scheme as Coordinated or Coflow-Aware Least-Attained Service (CLAS). Note that CLAS reduces to the well-known Least-Attained Service (LAS) [162, 176] scheduling discipline in the case of a single link.

6.5.2 Priority Discretization

Unfortunately, using continuous priorities derived from coflow sizes can degenerate into fair sharing (§C.2), which increases the average CCT [67,68,82]. Coordination needed to find global coflow sizes poses an additional challenge. We must be able to preempt at opportune moments to decrease CCT without requiring excessive coordination.

In the following, we describe a priority *discretization* framework to eventually design an efficient, non-clairvoyant coflow scheduler. Unlike classic non-clairvoyant schedulers – least-attained service (LAS) in single links [162, 176] and multi-level feedback queues (MLFQ) in operating systems [44, 70, 74] – that perform fair sharing in presence of similar flows/tasks to provide interactivity, our solution improves the average CCT even in presence of identical coflows.

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

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

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

The first two ensure that coflows are prioritized based on their current sizes, while the last is for completeness.

6.5.3 Tradeoffs in Designing Coflow Schedulers

Given the multi-level framework, a coflow scheduler can be characterized by its information source, queue structure, and scheduling disciplines at different granularities. Tradeoffs made while navigating this solution space result in diverse algorithms, ranging from centralized shortest-first to decentralized FIFO [67, 68, 82] and many in between. We elaborate on the key tradeoffs below.

Information Source There are two primary categories of coflow schedulers: *clairvoyant* schedulers use a priori information and *non-clairvoyant* ones do not. Non-clairvoyant schedulers have one more decision to make: whether to use globally-coordinated coflow sizes or to rely on local information. The former is accurate but more time consuming. The latter diverges (**Theorem C.1.1**) for coflows with large skews, which is common in production clusters [63, 68].

Queue Structure A scheduler must also determine the number of queues it wants to use and their thresholds. On the one hand, FIFO-derived schemes (e.g., Orchestra, Baraat) use exactly one

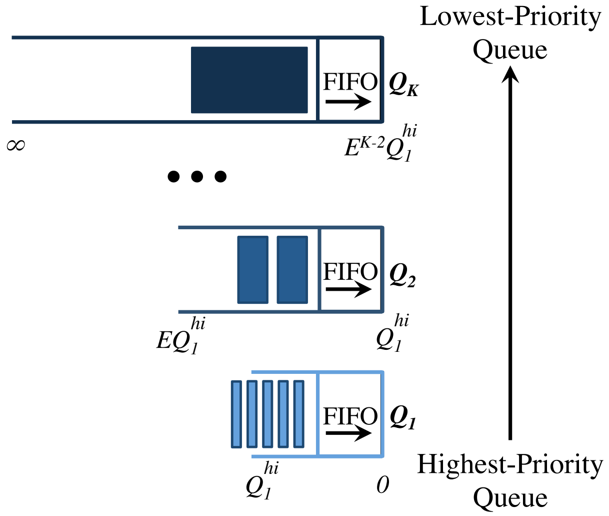


Figure 6.3: Discretized Coflow-Aware Least-Attained Service. Consecutive queues hold coflows with exponentially larger size.

queue.[7] FIFO works well when coflows follow light-tailed distributions [82]. Clairvoyant efficient schedulers (e.g., Varys), on the other hand, can be considered to have as many queues as there are coflows. They perform the best when coflow sizes are *known* and are heavy-tailed [68]. At both extremes, queue thresholds are irrelevant.

For solutions in between, determining an ideal number of queues and corresponding thresholds is difficult; even for tasks on a single machine, no optimal solution exists [44]. Increasing the number of levels/queues is appealing, but fine-grained prioritization can collapse to fair sharing when coflow sizes are unknown and hurt CCTs. More queues also generate more “queue-change” events and increase coordination requirements.

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

6.5.4 Discretized Coflow-Aware Least-Attained Service

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

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

	Orchestra [67]	**Varys [68]**	**Baraat [82]**	**Aalo**
On-Arrival Knowledge	Clairvoyant	Clairvoyant	Non-clairvoyant	Non-clairvoyant
Coflow Size Information	Global	Global	Local	Global Approx.
Number of Queues (K)	One	Num Coflows	One	$\log_E$(Max Size)
Queue Thresholds	N/A	Exact Size	N/A	$Q_{i+1}^{\text{hi}} = E \times Q_i^{\text{hi}}$
Queue Scheduling	N/A	Strict Priority	N/A	Weighted
Coflow Scheduling in Each Queue	FIFO	N/A	FIFO	FIFO
Flow Scheduling	WSS	MADD	Max-Min	Max-Min
Work Conservation	Next Coflow	Next Queue	Next Coflow	Weighted Among Queues
Starvation Avoidance	N/A	Promote to Q_1	N/A	N/A
HOL Blocking Avoidance	Multiplexing	N/A	Multiplexing	N/A

Table 6.1: Qualitative comparison of coflow scheduling algorithms.

Pseudocode 4 D-CLAS Scheduler to Minimize CCT

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

14: procedure D-CLAS
15:     $W = \sum Q_i.weight$
16:     for all $i \in [1, K]$ do
17:         $Q_i$.share = $Q_i$.weight / $W$                     ▷ Weighted sharing between queues
18:     end for
19:     reschedule($Q$, Max-Min among $Q_{j \neq i}$)
20: end procedure
```

To achieve our goals, each queue in D-CLAS contains *exponentially larger* coflows than its immediately higher-priority queue (Figure 6.3). Formally, $Q^{\text{hi}}_{i+1} = E \times Q^{\text{hi}}_{i}$, where the factor E determines how much bigger coflows in one queue are from that in another. Consequently, the number of queues remains small and can be expressed as an E-based logarithmic function of the maximum coflow size.

The final component in defining our queue structure is determining Q^{hi}_1. Because global coordination, irrespective of mechanism, has an associated time penalty depending on the scale of the cluster, we want coflows that are too small to be globally coordinated in Q_1. Larger coflows reside in increasingly more stable, lower-priority queues ($Q_2, \ldots, Q_K$).

While we typically use $E = 10$ and $Q^{\text{hi}}_1 = 10$ MB in our cluster, simulations show that for $K > 1$, a wide range of K, E, Q^{hi}_1 combinations work well (§6.8.5).

Non-Clairvoyant Efficient Schedulers D-CLAS clusters similar coflows together and allows us to implement different scheduling disciplines among queues and among coflows within each queue (Pseudocode 4). It uses weighted sharing among queues, where queue weights decrease with lowered priority; i.e., Q_i.weight $\geq Q_{i+1}$.weight at line 17 in Pseudocode 4. Excess share of any queue is divided among unsaturated queues in proportion to their weights using max-min fairness (line 19).

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

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

6.5.5 Summary

Table 6.1 summarizes the key characteristics of the schedulers discussed in this section. D-CLAS minimizes the average CCT by prioritizing significantly different coflows across queues and FIFO ordering similar coflows in the same queue. It does so without starvation, and it approximates FIFO schedulers for light-tailed and priority schedulers for heavy-tailed coflow distributions.

6.6 Handling Uncertainties

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

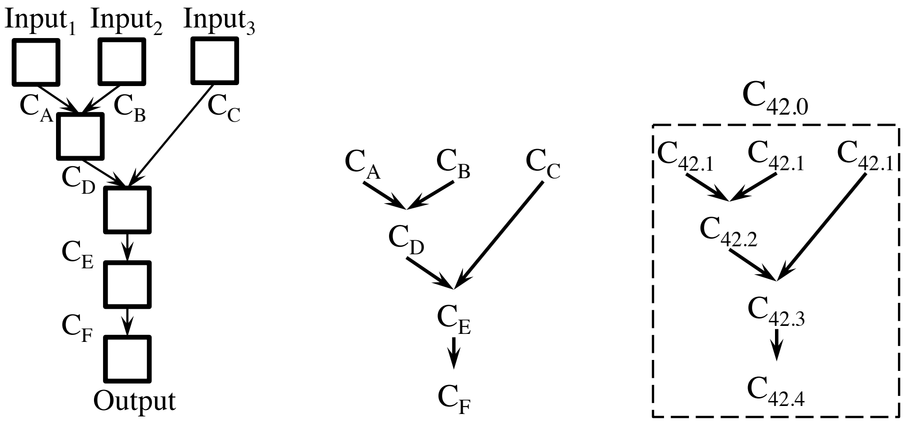
6.6.1 Multi-Stage Dataflow DAGs

The primary concern in coflow scheduling in the context of multi-stage jobs [8, 9, 119, 208] is the divergence of CCT and job completion time. Minimizing CCTs might not always result in faster jobs – one must carefully handle coflow *dependencies* within the same DAG (Figure 6.4).

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

1. **Starts-After (**$C_E \longmapsto C_F$**):** In presence of explicit barriers [8], C_F cannot start until C_E has finished.

2. **Finishes-Before (**$C_E \longrightarrow C_F$**):** With pipelining between successive stages [73, 119], C_F can coexist with C_E but it cannot finish until C_E has finished.



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

Figure 6.4: Coflow dependencies in TPC-DS query-42: (a) Query plan generated by Shark [203]; boxes and arrows respectively represent computation and communication stages. (b) Finishes-Before relationships between coflows are represented by arrows. (c) CoflowIds assigned by Aalo.

Pseudocode 5 Coflow ID Generation

```
NextCoflowID = 0                                          ▷ Initialization
procedure NEWCOFLOWID(CoflowId pId, Coflows $\mathcal{P}$)
    if pId == Nil then
        newId = NextCoflowID++                            ▷ Unique external id
        return newId.0
    else
        sId = 1 + $\max_{C \in \mathcal{P}}$ C.sId        ▷ Ordered internal id
        return pId.sId
    end if
end procedure
```

Note that coflows in different branches of a DAG can be unrelated to each other.

Job schedulers identify coflow dependencies while building query plans (Figure 6.4a). They can make Aalo aware of these dependencies all at once, or in a coflow-by-coflow basis. *Given coflow dependencies, we want to efficiently schedule them to minimize corresponding job completion times.*

We make two observations about coflow dependencies. First, coflows from the same job should be treated as a single entity. Second, within each entity, dependent coflows must be deprioritized during contention. The former ensures that minimizing CCTs directly affect job completion times, while the latter prevents circular dependencies. For example, all six coflows must complete in Figure 6.4a, and dependent coflows cannot complete without their parents in Figure 6.4b.

We simultaneously achieve both objectives by encoding the DAG identifier and internal coflow dependencies in the CoflowId. Specifically, we extend the CoflowId with an *internal* component in addition to its *external* component (Pseudocode 5). While the external part of a CoflowId uniquely identifies the DAG it belongs to, the internal part ensures ordering of coflows within the same DAG (Figure 6.4c). Our schedulers process coflows in each queue in the FIFO order based on their external components, and they break ties between coflows with the same external component using their internal CoflowIds (line 4 in Pseudocode 4).

Note that optimal DAG scheduling is NP-hard (§6.10). Our approach is similar to the Critical-Path Method [132] and resolves dependencies in each branch of a DAG, but it does not provide any guarantees for the entire DAG.

6.6.2 Dynamic Coflow Modifications

A flow can start only after its source and destination tasks have been scheduled. Tasks of large jobs are often scheduled in multiple waves depending on cluster capacity [39]. Hence, flows of such jobs are also created in batches, and waiting for all flows of a stage to start can only halt a job. Because the number of tasks in each wave can dynamically change, Aalo must react without a priori knowledge. The same is true for unpredictable cluster events such as failures and stragglers. Both result in restart or replication of some tasks and corresponding flows, and Aalo must efficiently handle them as well.

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

6.7 Design Details

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

6.7.1 Pipelined Coflow API

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

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

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

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

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

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

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

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

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

```
unregister(sId)
```

Defining Dependencies Coflows can specify their parent(s) during registration, and Aalo uses this information to generate CoflowIds (Pseudocode 5). In our running example, if the shuffle (`sId`) depended on an earlier broadcast (`bId`) – common in many Spark [208] jobs – the driver would have defined `bId` as a dependency during registration as follows.

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

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

6.7.2 Coflow Scheduling in Aalo

Aalo daemons resynchronize every Δ milliseconds. Each daemon sends the locally-observed coflow sizes to the coordinator every Δ interval. Similarly, the coordinator sends out the globally-coordinated coflow order and corresponding sizes every Δ interval. Furthermore, the coordinator sends out explicit ON/OFF signals for individual flows in order to avoid receiver-side contentions and to expedite sender-receiver rate convergence.

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

Choice of Δ Aalo daemons are more closely in sync as Δ decreases. We suggest Δ to be $O(10)$ milliseconds, and our evaluation shows that a 100-machine EC2 cluster can resynchronize within 8 milliseconds on average (§6.8.6).

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

Table 6.2: Jobs binned by time spent in communication.

6.8 Evaluation

We evaluated Aalo through a series of experiments on 100-machine EC2 [3] clusters using traces from production clusters and an industrial benchmark. For larger-scale evaluations, we used a trace-driven simulator that performs a detailed replay of task logs. The highlights are:

- For communication-dominated jobs, Aalo improves the average (95th percentile) CCT and job completion time by up to $2.25\times$ ($2.93\times$) and $1.57\times$ ($1.77\times$), respectively, over per-flow fairness. Aalo improvements are, on average, within 12% of Varys (§6.8.2).

- As suggested by our analysis, coordination is the key to performance – independent local decisions (e.g., in [82]) can lead to more than $16\times$ performance loss (§6.8.2).

- Aalo outperforms per-flow fairness and Varys for multi-wave (§6.8.3) and DAG (§6.8.4) workloads by up to $3.7\times$.

- Aalo's improvements are stable over a wide range of parameter combinations for any $K \geq 2$ (§6.8.5).

- Aalo coordinator can scale to $O(10,000)$ daemons with minimal performance loss (§6.8.6).

6.8.1 Methodology

Workload Our workload is based on a Hive/MapReduce trace collected by Chowdhury et al. [68, Figure 4] from a 3000-machine, 150-rack Facebook cluster. The original cluster had a $10 : 1$ core-to-rack oversubscription ratio and a total bisection bandwidth of 300 Gbps. We scale down jobs accordingly to match the maximum possible 100 Gbps bisection bandwidth of our deployment while preserving their communication characteristics.

Additionally, we use TPC-DS [25] queries from the Cloudera benchmark [17, 26] to evaluate Aalo on DAG workloads. The query plans were generated using Shark [203].

Job/Coflow Bins We present our results by categorizing jobs based on their time spent in communication (Table 6.2) and by distinguishing coflows based on their lengths and widths (Table 6.3). Specifically, we consider a coflow to be *short* if its longest flow is less than 5 MB and *narrow* if it has at most 50 flows. Note that coflow sizes, similar to jobs, follow heavy-tailed distributions in data-intensive clusters [68].

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

Table 6.3: Coflows binned by length (**S**hort and **L**ong) and width (**N**arrow and **W**ide).

Cluster Our experiments use extra-large high-memory (`m2.4xlarge`) EC2 instances. We observed bandwidths close to 900 Mbps per machine on clusters of 100 machines. We use a compute engine similar to Spark [208] that uses the coflow API (§6.7.1) and use $\Delta = 10$ milliseconds, $E = K = 10$, and $Q_1^{\text{hi}} = 10$ MB as defaults.

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

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

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

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

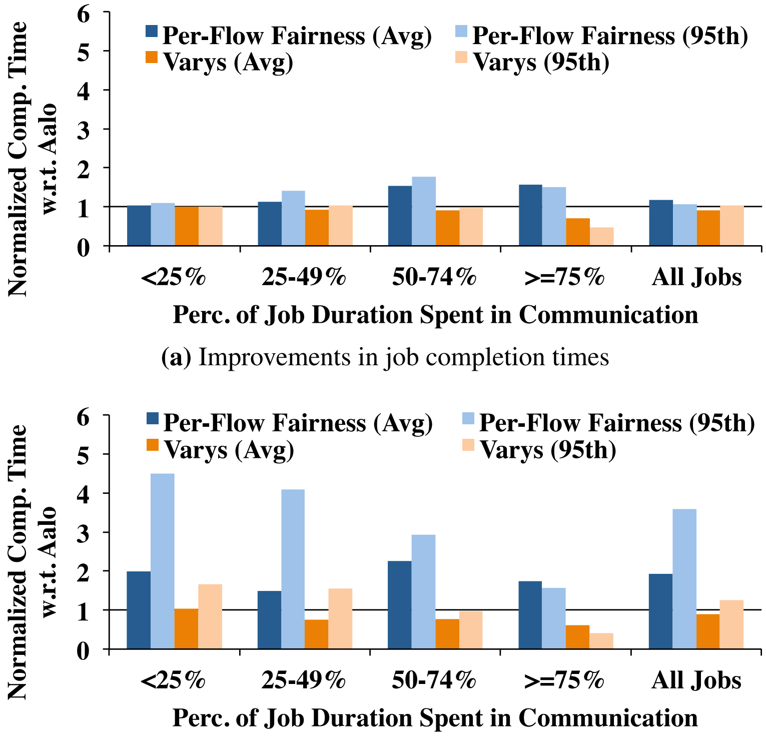
We contrast Aalo with TCP fair sharing and the open-source[8] implementation of Varys that uses a clairvoyant, smallest-bottleneck-first scheduler. Due to the lack of readily-deployable implementations of Baraat [82], we compare against it only in simulation. We present Aalo's results for D-CLAS with Q_i.weight = $K - i + 1$.

6.8.2 Aalo's Overall Improvements

Figure 6.5a shows that Aalo reduced the average and 95th percentile completion times of communication-dominated jobs by up to $1.57\times$ and $1.77\times$, respectively, in EC2 experiments in comparison to TCP-based per-flow fairness. Corresponding improvements in the average CCT (CommTime) were up to $2.25\times$ and $2.93\times$ (Figure 6.5b). As expected, jobs become increasingly faster as their time spent in communication increase. Across all bins, the average end-to-end completion times improved by $1.18\times$ and the average CCT improved by $1.93\times$; corresponding 95th percentile improvements were $1.06\times$ and $3.59\times$.

Varying improvements in the average CCT across bins in Figure 6.5b are not correlated, as it depends more on coflow characteristics than that of jobs. Figure 6.6 shows that Aalo improved the

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



(a) Improvements in job completion times

(b) Improvements in time spent in communication

Figure 6.5: [EC2] Average and 95th percentile improvements in job and communication completion times using Aalo over per-flow fairness and Varys.

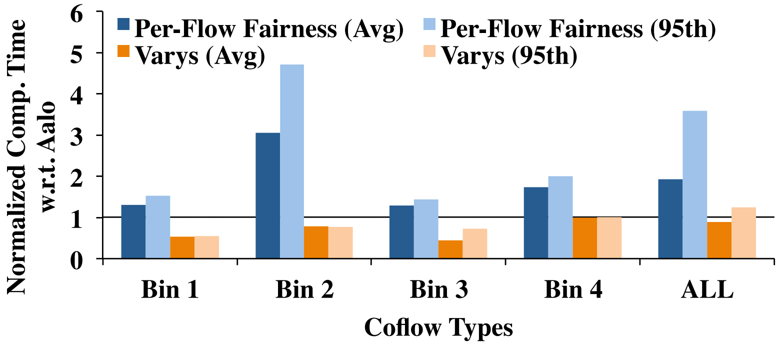


Figure 6.6: [EC2] Improvements in the average and 95th percentile CCTs using Aalo over per-flow fairness and Varys.

average CCT over per-flow fair sharing regardless of coflow width and length distributions. We observe more improvements in bin-2 and bin-4 over bin-1 and bin-3, respectively, because longer coflows give Aalo more opportunities for better estimation.

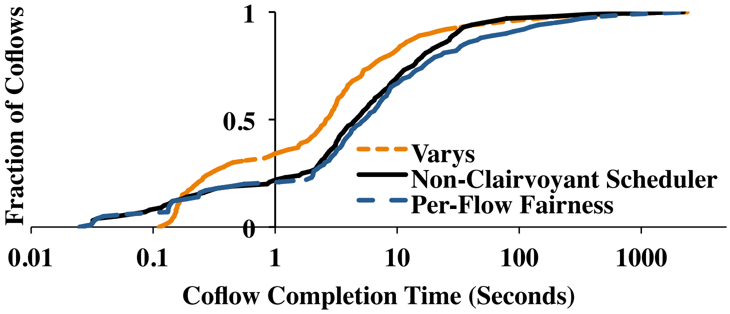


Figure 6.7: [EC2] CCT distributions for Aalo, Varys, and per-flow fairness mechanism. The X-axis is in log scale.

Finally, Figure 6.7 presents comparative CDFs of CCTs for all coflows. Across a wide range of coflow durations – milliseconds to hours – Aalo matches or outperforms TCP fair sharing. As mentioned earlier, Aalo's advantages keep increasing with longer coflows.

What About Clairvoyant Coflow Schedulers? To understand how far we are from clairvoyant solutions, we have compared Aalo against Varys, which uses *complete* knowledge of a coflow's individual flows. Figure 6.5 shows that across all jobs, the average job and coflow completion times using Aalo stay within 12% of Varys. At worst, Aalo is $1.43\times$ worse than Varys for 12% of the jobs.

Figure 6.6 presents a clearer picture of where Aalo is performing worse. For the largest coflows in bin-4 – sources of almost all the bytes – Aalo performs the same as Varys; it is only for the smaller coflows, especially the short ones in bin-1 and bin-3, Aalo suffers from its lack of foresight.

However, it still does not explain why Varys performs so much better than Aalo for coflows of durations between 200ms to 30s (Figure 6.7) given that Δ is only 10ms! Closer examination revealed this to be an isolation issue [138, 171]: Varys delays large coflows in presence of small ones and uses explicit rates for each flow. Because Aalo cannot explicitly control rates without a priori information, interference between coflows with few flows with very large coflows results in performance loss. Reliance on slow-to-react TCP for flow-level scheduling worsens the impact. We confirmed this by performing width-bounded experiments – we reduced the number of flows by $10\times$ while keeping same coflow sizes; this reduced the gap between the two CDFs from $\leq 6\times$ to $\leq 2\times$ in the worst case.

Scheduling Overheads Because coflows smaller than the first priority threshold are scheduled without coordination, Aalo easily outperforms Varys for sub-200ms coflows (Figure 6.7). For larger coflows, Aalo's average and 99th percentile coordination overheads were 8ms and 19ms, respectively, in our 100-machine cluster – an order of magnitude smaller than Varys due to Aalo' loose coordination requirements. Almost all of it were spent in communicating coordinated decisions. Impact of scheduling overheads on Aalo's performance is minimal, even at much larger scales (§6.8.6).

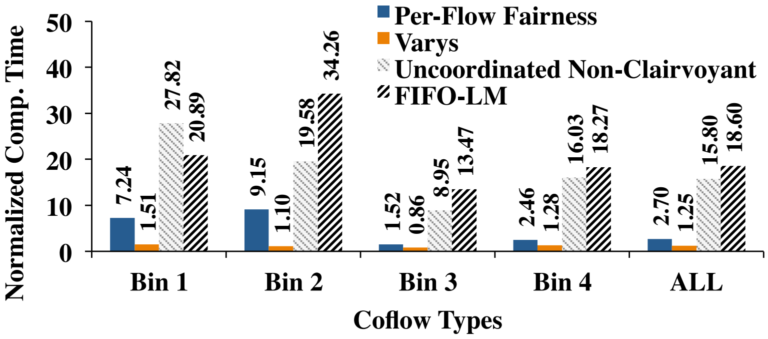


Figure 6.8: [Simulation] Average improvements in CCTs using Aalo. 95th percentile results are similar.

Trace-Driven Simulation

We compared Aalo against per-flow fairness, Varys, and non-clairvoyant scheduling *without* coordination in simulations (Figure 6.8). Similar to EC2 experiments, Aalo outperformed flow-level fairness with average and 95th percentile improvements being 2.7× and 2.18×.

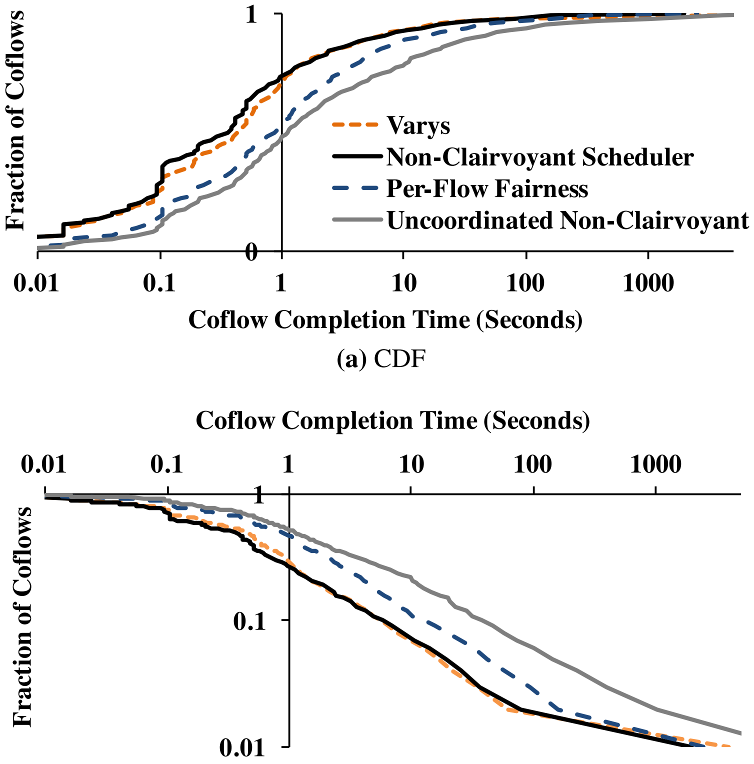
Figure 6.8 shows that Aalo outperforms Varys in most bins in the absence of coordination overheads. However, a closer observation reveals that Varys performed better than Aalo for coflows longer than 1.5 seconds except for at the 99th percentile (Figure 6.9).

What About Aalo *Without* Coordination? Given that Aalo takes few milliseconds to coordinate, we need to understand the importance of coordination. Simulations show that coflow scheduling without coordination can be significantly worse than even simple TCP fair sharing. On average, Aalo performed 15.8× better than its uncoordinated counterpart, bolstering our worst-case analysis (**Theorem C.1.1**). Experiments with increasing Δ suggest the same (§6.8.6).

What About FIFO with Limited Multiplexing in Baraat [82]? We found that FIFO-LM can be significantly worse than Aalo (18.6×) due to its lack of coordination: each switch takes locally-correct, but globally-inconsistent, scheduling decisions. Fair sharing among heavy coflows further worsens it. We had been careful – as the authors in [82] have pointed out – to select the threshold that each switch uses to consider a coflow heavy. Figure 6.8 shows the results for FIFO-LM's threshold set at the 80th percentile of the coflow size distribution; results for the threshold set to the 20th, 40th, 60th, 70th, and 90th percentiles were worse. Aalo and FIFO-LM performs similar for small coflows following light-tailed distributions (not shown).

6.8.3 Impact of Runtime Dynamics

So far we have only considered static coflows, where all flows of a coflow start together. However, operational events such as multi-wave scheduling, task failures, and speculative execution can dynamically change a coflow's structure in the runtime (§6.6.2). Because of their logical similarity



(**a**) CDF

(**b**) CCDF

Figure 6.9: [Simulation] CCT distributions for Aalo, Varys, per-flow fairness, and uncoordinated non-clairvoyant coflow scheduling. Both X-axes and the Y-axis of (b) are in log scale.

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

Table 6.4: Coflows binned by the number of waves.

– i.e., tasks start in batches and the number of active flows cannot be known a priori – we focus only on the multi-wave case.

The number of waves in a stage depends on the number of senders (e.g., mappers in MapReduce) [39]. In these experiments, we used the same coflow mix as the original trace but varied the maximum number of concurrent senders in each wave while keeping all the receivers active, essentially fixing the maximum number of waves in each coflow. Table 6.4 shows the fraction of coflows with different number of waves; e.g., all coflows had exactly one wave in Section 6.8.2.

Figure 6.10 shows the importance of leveraging coflow relationships across waves. As the number of multi-wave coflows increased, Aalo moved from trailing Varys by $0.94\times$ to outperforming

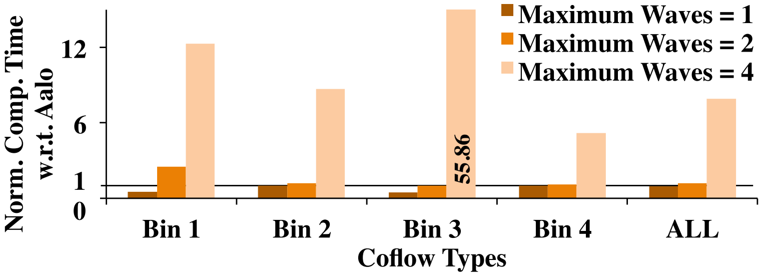


Figure 6.10: [EC2] Average improvements in CCTs w.r.t. Varys for multi-wave coflows.

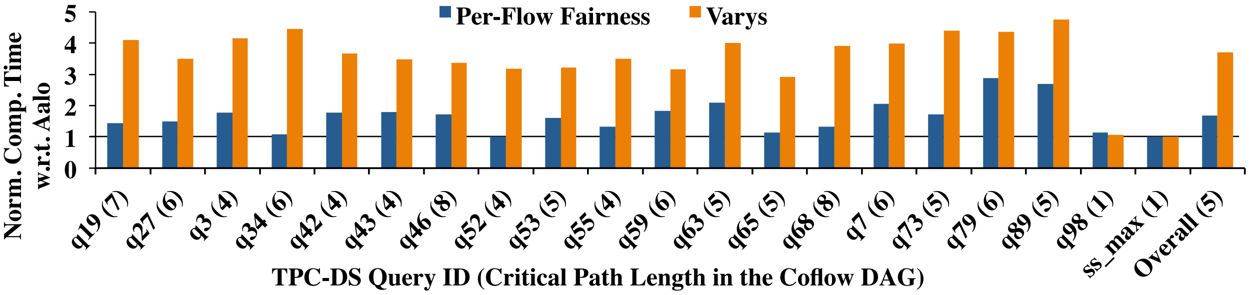


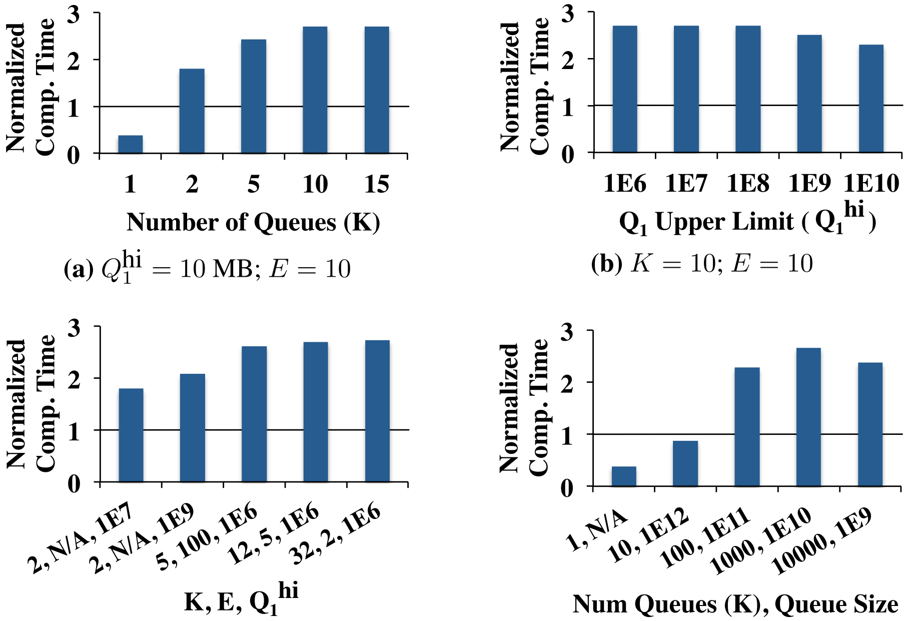
Figure 6.11: [EC2] Improvements in job-level communication times using Aalo for coflow DAGS in the Cloudera benchmark.

it by $1.21\times$ and $7.91\times$. Using Varys, one can take two approaches to handle multi-wave coflows – (i) creating separate coflows for each wave as they become available or (ii) introducing barriers to determine the bottleneck of the combined coflow – that both result in performance loss. In the former, Varys can efficiently schedule each wave but increases the stage-level CCT by ignoring the fact that all waves must finish for the stage to finish. The $56\times$ improvement in bin-3 presents an extreme example: one *straggler* coflow was scheduled much later than the rest, increasing the entire stage's runtime. In the latter, artificial barriers decrease parallelism and network utilization. Aalo circumvents the dilemma by creating exactly one coflow per stage for any number of waves and by avoiding barriers. Aalo's improvements over per-flow fairness remained similar to that in Section 6.8.2.

6.8.4 Impact on DAG Scheduling

In this section, we evaluate Aalo using multi-stage jobs. Because the Facebook trace consists of only single-coflow jobs, we used the Cloudera industrial benchmark [17,26] consisting of 20 TPC-DS queries. We ensured that each stage consists of a single wave, but multiple coflows from the same job can still run in parallel (Figure 6.4c).

Figure 6.11 shows that Aalo outperforms both per-flow fairness and Varys for DAGs that have more than one levels. Because Aalo does not introduce artificial barriers and can distinguish be-



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

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

(c) Exponentially-Spaced Queues

(d) Equal-Sized Queues

Figure 6.12: [Simulation] Aalo's sensitivity (measured as improvements over per-flow fairness) to (a) the number of queues, (b) the size of the highest-priority queue, and (c) exponential and (d) linear queue thresholds.

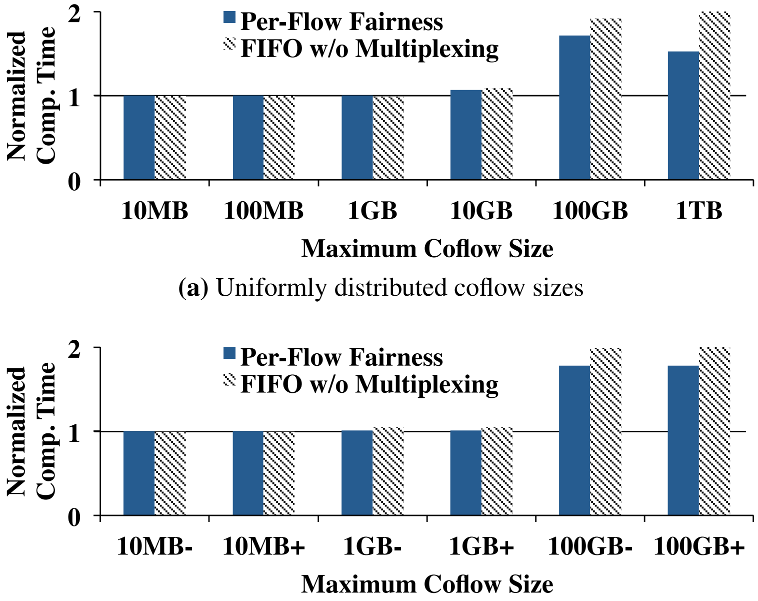
tween coflows from different levels of the critical path, improvements over Varys ($3.7\times$ on average) are higher than that over per-flow fairness ($1.7\times$ on average).

6.8.5 Sensitivity Analysis

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

The Number of Queues (K) Aalo performs increasingly better than per-flow fairness as we increase the number of queues (Figure 6.12a). However, we observe the largest jump as soon as Aalo starts avoiding head-of-line blocking for $K = 2$. Beyond that, we observe diminishing returns from more queues.

Queue Thresholds For more than one queues, Aalo must carefully determine their thresholds. Because we have defined queue thresholds as a function of the size of the initial queue Q_1^{hi} (§6.5.4), we focus on its impact on Aalo's performance. Recall that as we increase Q_1^{hi}, more coflows will be scheduled in the FIFO order in the highest-priority Q_1. Figure 6.12b shows that as we increase



(a) Uniformly distributed coflow sizes

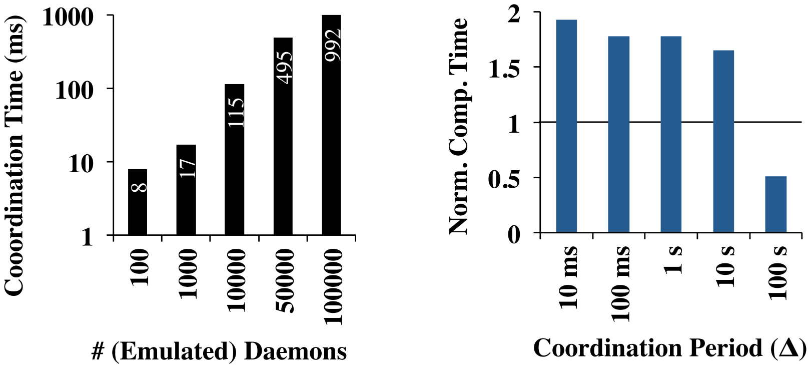
(b) Fixed-size coflows

Figure 6.13: [Simulation] Improvements in average CCTs using Aalo (a) when coflow sizes are uniformly distributed up to different maximum values and (b) when all coflows have the same size.

Q_1^{hi} up to 100 MB and schedule almost 60% of the coflows [68, Figure 4(e)] in the FIFO order, Aalo's performance remains steady. This is because all these coflows carry a tiny fraction of the total traffic ($\leq 0.1\%$). If we increase Q_1^{hi} further and start including increasingly larger coflows in the FIFO-scheduled Q_1, performance steadily deteriorates. Finally, Figure 6.12c demonstrates the interactions of E, the multiplicative factor used to determine queue thresholds, with K and Q_1^{hi}. We observe that for $K > 2$, Aalo's performance is steady for a wide range of (K, E, Q_1^{hi}) combinations.

What About Non-Exponential Queue Thresholds? Instead of creating exponentially larger queues, one can create equal-sized queues. Given the maximum coflow size of 10 TB, Figure 6.12d shows Aalo's performance for varying number of equal-sized queues – it requires orders of magnitude more queues to attain performance similar to exponential spacing. Although creating logical queues is inexpensive at end hosts, more queues generate more "queue-change" events and increase coordination costs.

Impact of Coflow Size Distributions So far we have evaluated Aalo on coflows that follow heavy-tailed distribution. Here, we compare Aalo against per-flow fairness and a non-preemptive FIFO scheduler on coflows with uniformly-distributed and fixed sizes. We present the average results of ten simulated runs for each scenario with 100 coflows, where coflow structures follow the distribution in Table 6.3.



(a) Overheads at Scale

(b) Impact of Δ

Figure 6.14: [EC2] Aalo scalability: (a) more daemons require longer coordination periods (Y-axis is in log scale), and (b) delayed coordination can hurt overall performance (measured as improvements over per-flow fairness).

In Figure 6.13a, coflow sizes follow uniform distributions $\mathsf{U}(0, x)$, where we vary x. In Figure 6.13b, all coflows have the same size, and we select sizes slightly smaller and bigger than Aalo's queue thresholds. We observe that in both cases, Aalo matched or outperformed the competition. Aalo emulates the FIFO scheduler when coflow sizes are smaller than Q_1^{hi}(=10 MB). As coflows become larger, Aalo performs better by emulating the efficient Varys scheduler.

6.8.6 Aalo's Scalability

To evaluate Aalo's scalability, we emulated running up to $100,000$ daemons on 100-machine EC2 clusters. Figure 6.14a presents the time to complete a coordination round averaged over 500 rounds for varying number of emulated daemons (e.g., $10,000$ emulated daemons refer to each machine emulating 100 daemons). During each experiment, the coordinator transferred scheduling information for 100 concurrent coflows on average to each of the emulated daemons.

Even though we might be able to coordinate $100,000$ daemons in 992ms, the coordination period (Δ) must be increased. To understand the impact of coordination on performance, we reran the earlier experiments (§6.8.2) for increasingly higher Δ (Figure 6.14b). For $\Delta = 1$s, Aalo's improvements over per-flow fairness dropped slightly from $1.93\times$ to $1.78\times$. For $\Delta > 1$s, performance started to drop faster and plummeted at $\Delta > 10$s. These trends hold across coflow bins and reinforce the need for coordination (**Theorem C.1.1**).

Because Δ must increase for Aalo to scale, sub-Δ coflows can further be improved if Aalo uses explicit switch/network support [82, 89]. However, we note that tiny coflows are still better off using Aalo than per-flow fairness schemes.

6.9 Discussion

Determining Optimal Queue Thresholds Finding the optimal number of queues and corresponding thresholds remains an open problem. Recent results in determining similar thresholds in the context of flows [45] do not immediately extend to coflows because of cross-flow dependencies. Dynamically changing these parameters based on online learning can be another direction of future work.

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

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

Approximate aggregation schemes such as Push-Sum [133] can be good starting points to develop solutions for the former within reasonable time and accuracy. The latter is perhaps more difficult, because it relies on fast propagation of receiver feedback throughout the entire network for quick convergence of sender- and receiver-side rates. Both can improve from in-network support from the fabric similar to that in CONGA [34].

Faster Interfaces and In-Network Bottlenecks As 10 GbE NICs become commonplace, a common concern is that scaling non-blocking fabrics might become cost prohibitive.[9] Aalo performs well even if the network is not non-blocking – for example, on the EC2 network used in the evaluation (§6.8). When bottleneck locations are known, e.g., rack-to-core links, Aalo can be modified to allocate rack-to-core bandwidth instead of NIC bandwidth [63]. For in-network bottlenecks, one can try enforcing coflows inside the network [213]. Nonetheless, designing, deploying, and enforcing distributed, coflow-aware routing and load balancing solutions remain largely unexplored.

6.10 Related Work

Coflow Schedulers Aalo's improvements over its clairvoyant predecessor Varys [68] are three-fold. First, it schedules coflows *without* any prior knowledge, making coflows practical in presence of task failures and straggler mitigation techniques. Second, it supports pipelining and dependencies in multi-stage DAGs and multi-wave stages through a simpler, non-blocking API. Finally, unlike Varys, Aalo performs well even for tiny coflows by avoiding coordination. For larger coflows, however, Varys marginally outperforms Aalo by exploiting complete knowledge.

Aalo outperforms existing non-clairvoyant coflow schedulers, namely Orchestra [67] and Baraat [82], by avoiding head-of-line blocking unlike the former and by using global information unlike the latter. While Baraat's fully decentralized approach is effective for light-tailed coflow

[9] A recent report from Google [186] suggests that it is indeed possible to build full-bisection bandwidth networks with up to $100,000$ machines, each with 10 GbE NICs, for a total capacity of 1 Pbps.

distributions, we prove in **Theorem C.1.1** that the lack coordination can be arbitrarily bad in the general case.

Qiu et al. have recently provided the first approximation algorithm for the clairvoyant coflow scheduling problem [175]. Similar results do not exist for the non-clairvoyant variation.

Flow Schedulers Coflows generalize traditional point-to-point flows by capturing the multipoint-to-multipoint aspect of data-parallel communication. While traffic managers such as Hedera [33] and MicroTE [51] cannot directly be used to optimize coflows, they can be extended to perform coflow-aware throughput maximization and load balancing.

Transport-level mechanisms to minimize FCTs, both clairvoyant (e.g., PDQ [118], pFabric [37], and D^3 [200]) and non-clairvoyant (e.g., PIAS [45]), fall short in minimizing CCTs as well [68].

Non-Clairvoyant Schedulers Scheduling without prior knowledge is known as non-clairvoyant scheduling [156]. To address this problem in time-sharing systems, Corbató et al. proposed the multi-level feedback queue (MLFQ) algorithm [74], which was later analyzed by Coffman and Kleinrock [70]. Many variations of this approach exist in the literature [162,176], e.g., foreground-background or least-attained service (LAS). In single machine (link), LAS performs almost as good as SRPT for heavy-tailed distributions of task (flow) sizes [176]. We prove that simply applying LAS throughout the fabric can be ineffective in the context of coflows (**Theorem C.1.1**). The closest instance of addressing a problem similar to ours is ATLAS [134], which controls concurrent accesses to multiple memory controllers in chip multiprocessor systems using coordinated LAS. However, ATLAS does not discretize LAS to ensure interactivity, and it does not consider coupled resources such as the network.

DAG and Workflow Schedulers When the entire DAG and completion times of each stage are known, the Critical Path Method (CPM) [131, 132] is the best known algorithm to minimize end-to-end completion times. Without prior knowledge, several dynamic heuristics have been proposed with varying results [205]. Most data-parallel computation frameworks use breadth-first traversal of DAGs to determine the scheduling order of each stage [8, 9, 208]. Aalo's heuristic enforces the *finishes-before* relationship between dependent coflows, but it cannot differentiate between independent coflows.

6.11 Summary

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

Chapter 7

Fair Inter-Coflow Scheduling

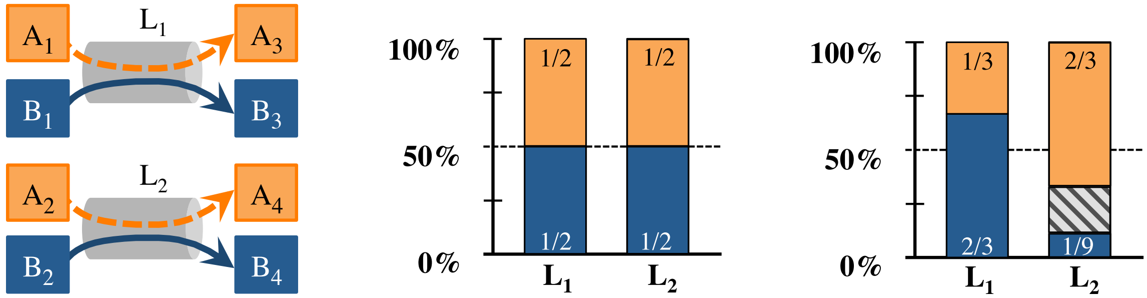
So far we have considered how to efficiently schedule clairvoyant and non-clairvoyant coflows in a shared cluster. However, efficient schedulers focus primarily on improving the average performance (i.e., the average CCT) and do not provide performance isolation. In this chapter, we consider the latter objective. By focusing on fair inter-coflow scheduling, we extend the applicability of the coflow abstraction to an even wider setting, where coflows can be generated by *non-cooperative* tenants in multi-tenant environments such as public clouds. In the process, we generalize single- [80, 121, 165] and multi-resource max-min fairness [83, 97, 112, 166] and multi-tenant network sharing solutions [124, 140, 171, 172, 179, 184] under a unifying framework.

The remainder of this chapter is organized as follows. The next section covers the state-of-the-art in single- and multi-resource fairness as well as multi-tenant network sharing. Section 7.2 outlines our proposed algorithm, called High Utilization with Guarantees or HUG. Section 7.3 presents the requirements of a fair inter-coflow scheduling algorithm, followed by a discussion on the inefficiencies of existing solutions in Section 7.4. Next, we present HUG and discuss its properties in Section 7.5, and we present HUG's implementation details in Section 7.6. We then evaluate HUG's performance in Section 7.7, survey related work in Section 7.8, and summarize our findings in Section 7.9.

7.1 Background

In shared, multi-tenant environments such as public clouds [3, 15, 16, 20, 117], the need for predictability and the means to achieve it remain a constant source of discussion [47, 123, 124, 139, 140, 171, 172, 179, 184]. The general consensus – recently summarized by Mogul and Popa [153] – is that tenants expect *isolation guarantee* for performance predictability, while network operators strive for *work conservation* to achieve high utilization and *strategy-proofness* to ensure isolation.

Max-min fairness [121] – a widely-used [49, 80, 100, 102, 165, 185, 188] allocation policy – achieves all three in the context of a single link. It provides *optimal* isolation guarantee by maximizing the minimum amount of bandwidth allocated to each flow. The bandwidth allocation of a user (tenant) determines its progress – i.e., how fast she can complete its data transfer. It is work-



(a) **(b)** Max-min fairness **(c)** DRF [97]

Figure 7.1: Bandwidth allocations in two independent links (a) for C_A (orange) with correlation vector $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ and C_B (dark blue) with $\overrightarrow{d_B} = \langle 1, \frac{1}{6} \rangle$. Shaded portions are unallocated.

conserving, because, given enough demand, it allocates the entire bandwidth of the link. Finally, max-min fairness is strategyproof, because tenants cannot get more bandwidth by lying about their demands (e.g., by sending more traffic).

However, a datacenter network involves many links, and tenants' demands on different links are often *correlated*. Informally, we say that the demands of a tenant on two links i and j are correlated, if for every bit the tenant sends on link-i, she sends at least α bits on link-j. More formally, with every tenant-k, we associate a correlation vector $\overrightarrow{d_k} = \langle d_k^1, d_k^2, \ldots, d_k^n \rangle$, where $d_k^i \leq 1$, which captures the fact that for every d_k^i bits tenant-k sends on link-i, she sends at least d_k^j bits on link-j.

Examples of applications with *correlated demands* long-running services [54,141], multi-tiered enterprise applications [113], and realtime streaming applications [27, 209]. However, optimal coflow schedules using MADD [67, 68] are the most common example. Consider the example in Figure 7.1a with two independent links and two coflows C_A and C_B. The correlation vector $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ means that (i) link-2 is C_A's bottleneck, (ii) for every $\mathcal{P}_A$ rate C_A is allocated on the bottleneck link, it requires *at least* $\mathcal{P}_A/2$ rate on link-1, resulting in a progress of $\mathcal{P}_A$, and (iii) except for the bottleneck link, coflows' demands are elastic, meaning C_A can use more than $\mathcal{P}_A/2$ rate on link-1.[1] Similarly, C_B requires *at least* $\mathcal{P}_B/6$ on link-2 for $\mathcal{P}_B$ on link-1. If we denote the rate allocated to C_k on link-i by a_k^i, then $\mathcal{P}_k = \min_i \left\{ \frac{a_i^k}{d_i^k} \right\}$, the minimum demand-normalized rate allocation over all links, captures its progress.

Given this setting, one would want *to generalize max-min fairness to coflows with correlated and elastic demands while maintaining its desirable properties: optimal isolation guarantee, high utilization, and strategy-proofness.*

Intuitively, one would want to maximize the minimum progress over all coflows, i.e., **maximize** $\min_k \mathcal{P}_k$, where $\min_k \mathcal{P}_k$ corresponds to the isolation guarantee of an allocation algorithm. We make three observations. First, when there is a single link in the system, this model trivially reduces to max-min fairness. Second, getting more aggregate bandwidth is not

[1] This does not increase the instantaneous progress, but may be beneficial for C_A as the system is dynamic and for network operators to increase total utilization.

always better. For C_A in the example, $\langle 50\text{Mbps}, 100\text{Mbps}\rangle$ is better than $\langle 90\text{Mbps}, 90\text{Mbps}\rangle$ or $\langle 25\text{Mbps}, 200\text{Mbps}\rangle$, even though the latter ones have more bandwidth in total. Third, simply applying max-min fairness to individual links is not enough. In our example, max-min fairness allocates equal resources to both coflows on both links, resulting in allocations $\langle \frac{1}{2}, \frac{1}{2}\rangle$ on both links (Figure 7.1b). Corresponding progress ($\mathcal{P}_A = \mathcal{P}_B = \frac{1}{2}$) result in a suboptimal isolation guarantee ($\min\{\mathcal{P}_A, \mathcal{P}_B\} = \frac{1}{2}$).

Dominant Resource Fairness (DRF) [97] extends max-min fairness to multiple resources and prevents such sub-optimality. It equalizes the shares of dominant resources – link-2 (link-1) for C_A (C_B) – across *all* tenants (coflows) with correlated demands and maximizes the isolation guarantee in a strategyproof manner. As shown in Figure 7.1c, using DRF, both coflows have the *same* progress – $\mathcal{P}_A = \mathcal{P}_B = \frac{2}{3}$, 50% higher than using max-min fairness on individual links. Moreover, DRF's isolation guarantee ($\min\{\mathcal{P}_A, \mathcal{P}_B\} = \frac{2}{3}$) is optimal across all possible allocations and is strategyproof.

However, DRF allocations were originally designed for *inelastic* demands [117]. Hence, they are not work-conserving. For example, bandwidth on link-2 in shades is not allocated to either coflow. In fact, we show that DRF can result in *arbitrarily low utilization*. This is wasteful in a transient and online environment, because unused bandwidth cannot be recovered.

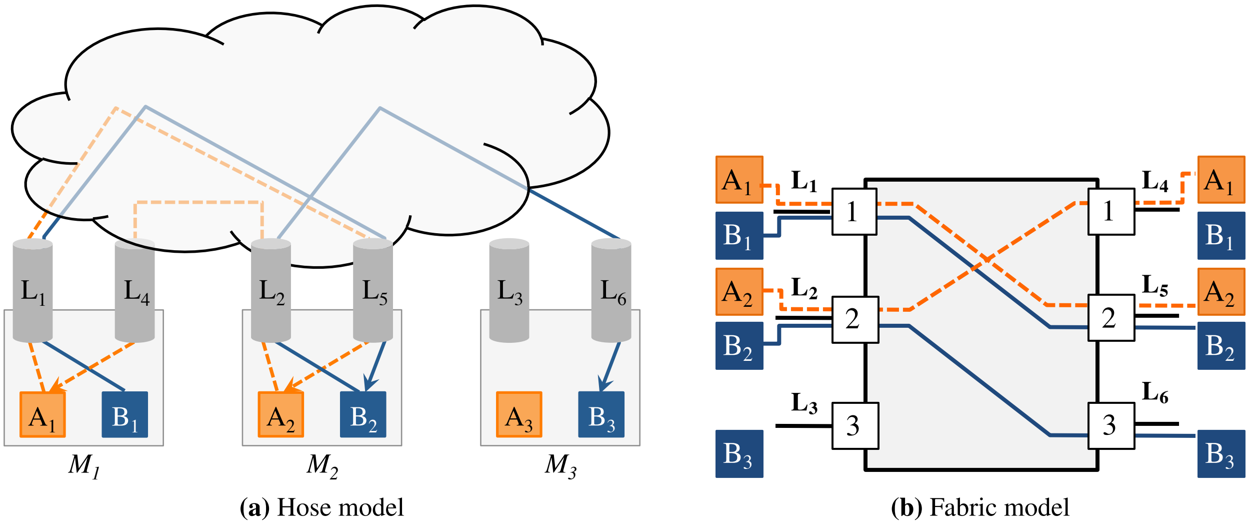
7.2 Solution Outline

We start by showing that strategy-proofness is a *necessary* condition for providing the optimal isolation guarantee – i.e., to **maximize** $\min_k \mathcal{P}_k$ – in non-cooperative environments. Next, we prove that work conservation – i.e., when coflows are allowed to use unallocated resources, such as the shaded area in Figure 7.1c, without constraints – spurs a race to the bottom. It incentivizes each coflow to continuously lie about its demand correlations, and in the process, it decreases the amount of useful work done by all coflows! Meaning, simply making DRF work-conserving can do more harm than good.

We propose a two-stage algorithm, *High Utilization with Guarantees* (HUG), to achieve our goals. HUG provides the highest utilization while maintaining the optimal isolation guarantee and strategy-proofness in non-cooperative environments. It allows coflows to elastically increase bandwidth usage as long as they cannot compromise strategy-proofness. In cooperative environments, where strategy-proofness might be a non-requirement, HUG simultaneously ensures work conservation and the optimal isolation guarantee.

HUG is easy to implement and scales well. Even with $100,000$ machines, new allocations can be centrally calculated and distributed throughout the network in less than a second – faster than that suggested in the literature [41]. Moreover, each machine can locally enforce HUG-calculated allocations using existing traffic control tools without any changes to the network.

We demonstrate the effectiveness of our proposal using EC2 experiments and trace-driven simulations. In non-cooperative environments, HUG provides the optimal minimum guarantee, which is $7.4\times$ higher than existing network sharing solutions such as PS-P [124, 171, 172] and $7000\times$ higher than traditional per-flow fairness, and $1.4\times$ better utilization than DRF for the given trace.



(a) Hose model (b) Fabric model

Figure 7.2: Coflows C_A (orange) and C_B (dark blue) and their communication patterns over a 3×3 datacenter fabric. The network fabric has three uplinks (L_1–L_3) and three downlinks (L_4–L_6) corresponding to the three physical machines.

In cooperative environments, HUG outperforms PS-P and per-flow fairness by $1.49\times$ and $45.75\times$ in terms of the maximum slowdown of job communication stages, and 70% jobs experience lower slowdown w.r.t. DRF. Finally, HUG provides $1.77\times$ better maximum CCT than Varys, although Varys outperforms it by $1.45\times$ in terms of average CCT.

7.3 Preliminaries

In this section, we elaborate on the assumptions and notations used in this chapter and summarize the desirable requirements for fair scheduling across multiple coflows.

7.3.1 Assumptions and Notations

We denote the correlation vector of the kth coflow ($k \in \{1, ..., M\}$) as $\overrightarrow{d_k} = \langle d_k^1, d_k^2, \ldots d_k^{2P} \rangle$, where d_k^i and d_k^{P+i} $(1 \le i \le P)$ respectively denote the uplink and downlink demands normalized[2] by link capacities (C^i) and $\sum_{i=1}^{P} d_k^i = \sum_{i=1}^{P} d_k^{P+i}$.

For the example in Figure 7.2, consider coflow correlation vectors:

$$\overrightarrow{d_A} = \langle \frac{1}{2}, 1, 0, 1, \frac{1}{2}, 0 \rangle$$
$$\overrightarrow{d_B} = \langle 1, \frac{1}{6}, 0, 0, 1, \frac{1}{6} \rangle$$

[2] Normalization helps us consider heterogeneous capacities. By default we normalize the correlation vector such that the largest component equals to 1 unless otherwise specified.

where $d_k^i = 0$ indicates the absence of any source or destination, and $d_k^i = 1$ indicates the bottleneck link(s) of a coflow.

Correlation vectors depend on coflows, that can be from *elastic-demand* batch jobs [6,77,119, 208] or from realtime streaming applications [23,209] with *inelastic demands*.

7.3.2 Intra-Coflow Fair Sharing Requirements

Given correlation vectors of M coflows, we must use an allocation algorithm $\mathcal{A}$ to determine the allocations of each coflow:

$$\mathcal{A}(\{\overrightarrow{d_1}, \overrightarrow{d_2}, \ldots, \overrightarrow{d_M}\}) = \{\overrightarrow{a_1}, \overrightarrow{a_2}, \ldots, \overrightarrow{a_M}\}$$

where $\overrightarrow{a_k} = \langle a_k^1, a_k^2, \ldots a_k^{2P} \rangle$ and a_k^i is the fraction of link-i guaranteed to the kth coflow.

As identified in previous work on multi-tenant fairness [47,171], any allocation policy $\mathcal{A}$ must meet three requirements – (optimal) isolation guarantee, high utilization, and proportionality – to fairly share the cloud network:

1. **Isolation Guarantee:** Each source or destination VM should receive minimum bandwidth guarantees *proportional* to their correlation vectors so that coflows can estimate *worst-case* performance. Formally, *progress* of C_k ($\mathcal{P}_k$) is defined as its minimum demand satisfaction ratio across the entire fabric:

   $$\mathcal{P}_k = \min_{1 \leq i \leq 2P} \left\{ \frac{a_k^i}{d_k^i} \right\}$$

   For example, progress of coflows C_A and C_B in Figure 7.3 are $\mathcal{P}_A = \mathcal{P}_B = \frac{1}{2}$.[3] Note that $\mathcal{P}_k = \frac{1}{M}$ if $\overrightarrow{d_k} = \langle 1, 1, \ldots, 1 \rangle$ for all coflows (generalizing PS-P), and $\mathcal{P}_k = \frac{1}{M}$ for flows on a single link (generalizing per-flow fairness).

   Isolation guarantee is defined as the lowest progress across all coflows, i.e., $\min\limits_k \mathcal{P}_k$.

2. **High Utilization:** Spare network capacities should be utilized by coflows with elastic demands to ensure high utilization as long as it does not decrease anyone's progress.

   A related concept is *work conservation*, which ensures that either a link is fully utilized or demands from all flows traversing the link have been satisfied [121,171]. Although existing research equalizes the two [46, 47, 124, 140, 171, 172, 179, 184, 202], we show in the next section why that is not the case.

3. **Proportionality:** A tenant's (coflow's) bandwidth allocation should be proportional to its number of VMs similar to resources such as CPU and memory. We further discuss this requirement in more details in Section 7.5.3.

[3] We are continuing the example in Figure 7.2 but omitted the rest of $\overrightarrow{a_k}$, because there is either no contention or they are symmetric.

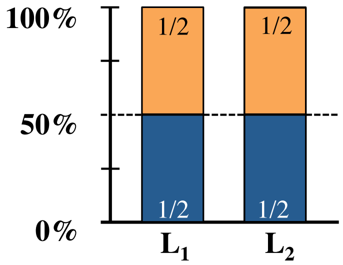


Figure 7.3: Bandwidth consumptions of C_A (orange) and C_B (dark blue) with correlation vectors $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ and $\overrightarrow{d_B} = \langle 1, \frac{1}{6} \rangle$ using PS-P [124, 171, 172]. Both coflows have elastic demands.

7.4 Challenges and Inefficiencies of Existing Solutions

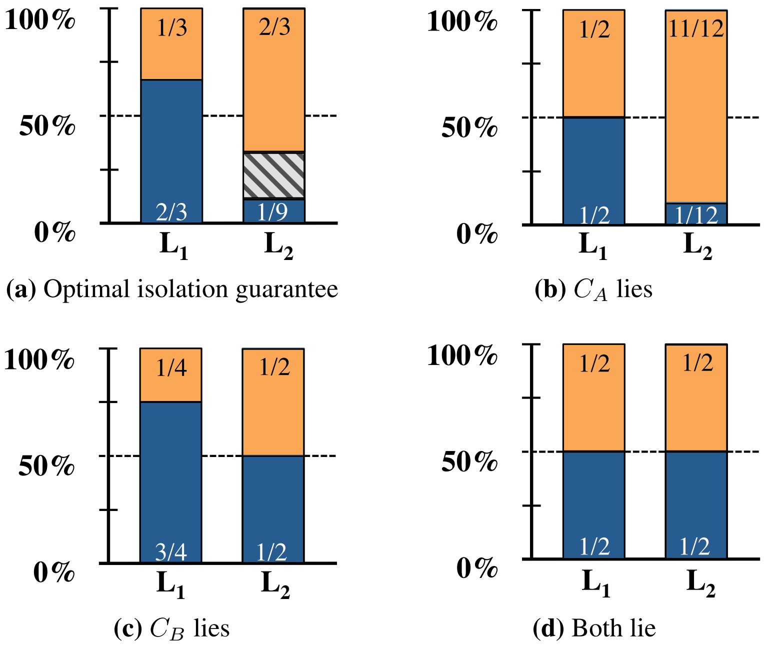
Prior work on multi-tenant fairness also identified two tradeoffs: *isolation guarantee vs. proportionality* and *high utilization vs. proportionality*. However, it has been implicitly assumed that tenant-level (coflow-level) *optimal* isolation guarantee[4] and network-level work conservation can coexist. Although optimal isolation guarantee and network-level work conservation can coexist for a single link – max-min fairness is an example – *optimal isolation guarantee and work conservation can be at odds when we consider the network as a whole.* This has several implications on both isolation guarantee and network utilization. In particular, we can (1) either optimize utilization, *then* maximize the isolation guarantee with best effort; or (2) optimize the isolation guarantee, *then* maximize utilization with best effort. Appendix D.1 has more details.

7.4.1 Full Utilization but Suboptimal Isolation Guarantee

As shown in prior work [171, Section 2.5], flow-level and VM-level mechanisms – e.g., per-flow, per source-destination pair [171], and per-endpoint fairness [179, 184] – can easily be manipulated by creating more flows or by using denser communication patterns. To avoid such manipulations, many allocation mechanisms [124, 171, 172] equally divide link capacities at the coflow level and allow work conservation for coflows with unmet demand. Figure 7.3 shows an allocation using PS-P [171] with isolation guarantee $\frac{1}{2}$. If both coflows have elastic-demand applications, they will consume entire allocations; i.e., $\overrightarrow{c_A} = \overrightarrow{c_B} = \overrightarrow{a_A} = \overrightarrow{a_B} = \langle \frac{1}{2}, \frac{1}{2} \rangle$, where $\overrightarrow{c_k} = \langle c_k^1, c_k^2, \ldots c_k^{2P} \rangle$ and c_k^i is the fraction of link-i *consumed* by C_k. Recall that a_k^i is the guaranteed allocation of link-i to C_k.

However, PS-P and similar mechanisms are also sub-optimal. For the ongoing example, Figure 7.4a shows the optimal isolation guarantee of $\frac{2}{3}$, which is higher than that provided by PS-P. In short, full utilization does not necessarily imply optimal isolation guarantee!

[4] Optimality means that the allocation maximizes the isolation guarantee across all coflows, i.e., **maximize** $\left\{ \min_k \mathcal{P}_k \right\}$.



(a) Optimal isolation guarantee

(b) C_A lies

(c) C_B lies

(d) Both lie

Figure 7.4: Bandwidth consumptions of C_A (orange) and C_B (dark blue) with correlation vectors $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ and $\overrightarrow{d_B} = \langle 1, \frac{1}{6} \rangle$, when both have elastic demands. (a) Optimal isolation guarantee in the absence of work conservation. With work conservation, (b) C_A increases its progress at the expense of C_B, and (c) C_B can do the same, which results in (d) a prisoner's dilemma.

7.4.2 Optimal Isolation Guarantee but Low Utilization

In contrast, optimal isolation guarantee does not necessarily mean full utilization. In general, optimal isolation guarantees can be calculated using DRF [97], which generalizes max-min fairness to multiple resources. In the example of Figure 7.4a, each uplink and downlink of the fabric is an independent resource – $2P$ in total.

Given this premise, it seems promising and straightforward to keep the DRF-component for optimal isolation guarantee and strategy-proofness, and try to ensure full utilization by allocating *all* remaining resources.In the following two subsections, we show that work conservation may render isolation guarantee no longer optimal, and even worse, may reduce useful network utilization.

7.4.3 Naive Work Conservation Reduces Optimal Isolation Guarantee

We first illustrate that even the optimal isolation guarantee allocation degenerates into the classic prisoner's dilemma problem [90] in the presence of work conservation. As a consequence, optimal isolation guarantees decrease (Figure 7.5).

If C_A can use the spare bandwidth in link-2, it can increase its progress at the expense of C_B by changing its correlation vector to $\overrightarrow{d'_A} = \langle 1, 1 \rangle$. With an unmodified $\overrightarrow{d_B} = \langle 1, \frac{1}{6} \rangle$, the new

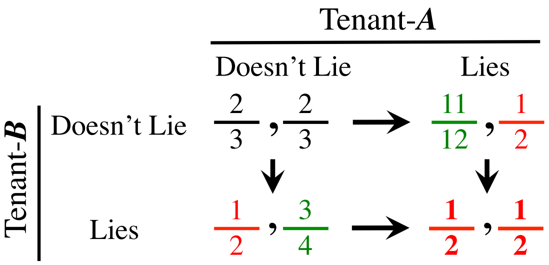


Figure 7.5: Payoff matrix for the example in Section 7.4. Each cell shows progress of C_A and C_B.

allocation would be $\overrightarrow{a_A} = \langle \frac{1}{2}, \frac{1}{2} \rangle$ and $\overrightarrow{a_B} = \langle \frac{1}{2}, \frac{1}{12} \rangle$. However, work conservation would increase it to $\overrightarrow{a_A} = \langle \frac{1}{2}, \frac{11}{12} \rangle$ (Figure 7.4b). Overall, progress of C_A would increase to $\frac{11}{12}$, while decreasing it to $\frac{1}{2}$ for C_B. As a result, the isolation guarantee decreases from $\frac{2}{3}$ to $\frac{1}{2}$.

The same is true for C_B as well. Consider again that only C_B reports a falsified correlation vector $\overrightarrow{d'_B} = \langle 1, 1 \rangle$ to receive a favorable allocation: $\overrightarrow{a_A} = \langle \frac{1}{4}, \frac{1}{2} \rangle$ and $\overrightarrow{a_B} = \langle \frac{1}{2}, \frac{1}{2} \rangle$. Work conservation would increase it to $\overrightarrow{a_B} = \langle \frac{3}{4}, \frac{1}{2} \rangle$ (Figure 7.4c). Overall, progress of C_B would increase to $\frac{3}{4}$, while decreasing it to $\frac{1}{2}$ for C_A, resulting in the same suboptimal isolation guarantee $\frac{1}{2}$.

Since both coflows gain by lying, they would both simultaneously lie: $\overrightarrow{d'_A} = \overrightarrow{d'_B} = \langle 1, 1 \rangle$, resulting in a lower isolation guarantee $\frac{1}{2}$ (Figure 7.4d). Both are worse off!

In this example, the inefficiency arises due to allocating all spare resources to the coflow who demands more. We show in Appendix D.2 that intuitive allocation policies of all spare resources – e.g., allocating all to who demands the least, allocating equally to all coflows with non-zero demands, and allocating proportionally to coflows' demands – do not work as well.

7.4.4 Naive Work Conservation can Even Decrease Utilization

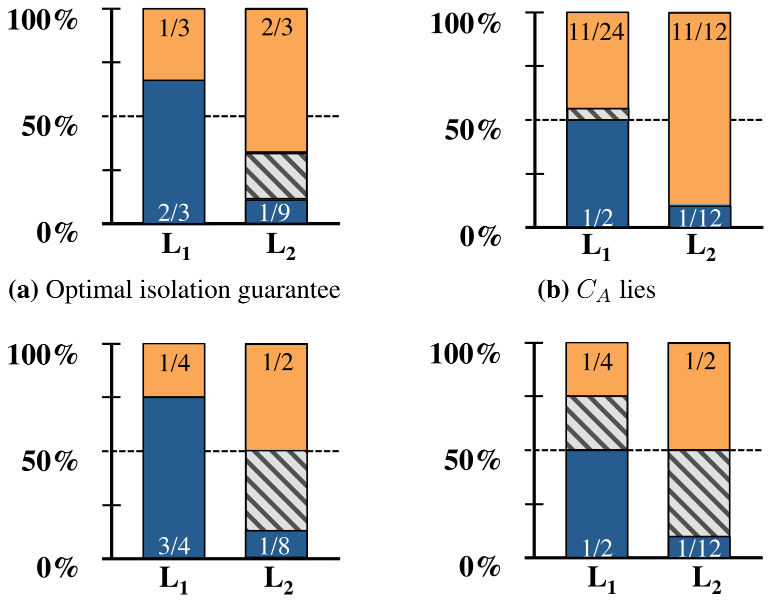
Now consider that *neither* coflow has elastic demands; i.e., they can only consume bandwidth proportional to their correlation vectors. A similar prisoner's dilemma unfolds (Figure 7.5), but this time, network utilization decreases as well.

Given the optimal isolation guarantee allocation, $\overrightarrow{a_A} = \overrightarrow{c_A} = \langle \frac{1}{3}, \frac{2}{3} \rangle$ and $\overrightarrow{a_B} = \overrightarrow{c_B} = \langle \frac{2}{3}, \frac{1}{9} \rangle$, both coflows have the same optimal isolation guarantee: $\frac{2}{3}$, and $\frac{2}{9}$th of link-2 remain unused (Figure 7.6a). One would expect work conservation to utilize this spare capacity.

Same as before, if C_A changes its correlation vector to $\overrightarrow{d'_A} = \langle 1, 1 \rangle$, it can receive an allocation $\overrightarrow{a_A} = \langle \frac{1}{2}, \frac{11}{12} \rangle$ and consume $\overrightarrow{c_A} = \langle \frac{11}{24}, \frac{11}{12} \rangle$. This increases its isolation guarantee to $\frac{11}{12}$ and total network utilization increases (Figure 7.6b).

Similarly, C_B can receive an allocation $\overrightarrow{a_B} = \langle \frac{3}{4}, \frac{1}{2} \rangle$ and consume $\overrightarrow{c_B} = \langle \frac{3}{4}, \frac{1}{8} \rangle$ to increase its isolation guarantee to $\frac{3}{4}$. Utilization decreases (Figure 7.6c).

Consequently, both coflows lie and consume $\overrightarrow{c_A} = \langle \frac{1}{4}, \frac{1}{2} \rangle$ and $\overrightarrow{c_B} = \langle \frac{1}{2}, \frac{1}{12} \rangle$ (Figure 7.6d). Instead of increasing, work conservation decreases network utilization!



(a) Optimal isolation guarantee

(b) C_A lies

(c) C_B lies

(d) Both lie

Figure 7.6: Bandwidth consumptions of C_A (orange) and C_B (dark blue) with correlation vectors $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$ and $\overrightarrow{d_B} = \langle 1, \frac{1}{6} \rangle$, when neither has elastic demands. (a) Optimal isolation guarantee allocation is not work-conserving. With work conservation, (b) utilization can increase or (c) decrease, based on which coflow lies. (d) However, ultimately it lowers utilization. Shaded means unallocated.

7.4.5 Summary

The primary takeaways of this section are the following:

- Existing mechanisms provide either suboptimal isolation guarantees or low network utilization, but never both.

- There exists a strong tradeoff between optimal isolation guarantee and high utilization. The key lies in strategy-proofness: optimal isolation guarantee requires it, while work conservation nullifies it.

- Unlike in the case of flows on single links, work conservation can decrease network utilization instead of increasing it in the case of coflows.

7.5 HUG: Analytical Results

In this section, we show that despite the tradeoff between optimal isolation guarantee and work conservation, it is possible to increase utilization to some extent. Moreover, we present HUG, the

d_k	Correlation vector of C_k's demand
a_k	Guaranteed allocation to C_k
$\mathcal{P}_k$	Progress of C_k; $\mathcal{P}_k := \min\limits_{1 \le i \le 2P} \left\{ \frac{a_k^i}{d_k^i} \right\}$, where subscript i stands for link-i
c_k	Resource consumption of C_k

Isolation Guarantee	$\min\limits_k \mathcal{P}_k$
Optimal Isolation Guarantee	$\max \left\{ \min\limits_k \mathcal{P}_k \right\}$
(Network) Utilization	$\sum\limits_i \sum\limits_k c_k^i$

Table 7.1: Notations and definitions in HUG.

optimal algorithm to ensure maximum achievable utilization *without* sacrificing optimal isolation guarantees and strategy-proofness of DRF.

We defer the proofs from this section to Appendix D.3.

7.5.1 Root Cause of the Tradeoff: Unrestricted Sharing of Spare Resources

Going back to Figure 7.4, both coflows were incentivized to lie because they were receiving spare resources without any restriction in the pursuit of work conservation.

After C_A lied in Figure 7.4b, both $\mathcal{P}_A$ and $\mathcal{P}_B$ decreased to $\frac{1}{2}$. However, by cheating, C_A managed to increase its allocation in link-1 to $\frac{1}{2}$ from $\frac{1}{3}$. Next, it relied on indiscriminate work conservation to increase its allocation in link-2 to $\frac{11}{12}$ from the initial $\frac{1}{2}$, effectively increasing $\mathcal{P}_A$ to $\frac{11}{12}$. Similarly in Figure 7.4c, C_B first increased its allocation in link-2 to $\frac{1}{2}$ from $\frac{1}{9}$ and then work conservation increased its allocation in link-1 to $\frac{3}{4}$ from the initial $\frac{1}{2}$.

Consequently, we must eliminate a coflow's incentive to gain too much spare resources by lying; i.e., *a coflow should never be able to manipulate and increase its progress due to work conservation* (**Lemma D.3.1**). Furthermore, algorithms that provide optimal isolation guarantee cannot always be work-conserving even in the presence of elastic-demand applications (**Corollary D.3.2**).

7.5.2 The Optimal Algorithm: HUG

Given the tradeoff, our goal is to design an algorithm that can achieve the highest utilization while keeping the optimal isolation guarantee and strategy-proofness. Formally, we want to

$$\begin{aligned} \text{Maximize} \quad & \sum_{i \in [1,2P]} \sum_{k \in [1,M]} c_k^i \\ \text{subject to} \quad & \min_{k \in [1,M]} \mathcal{P}_k = \mathcal{P}^*, \end{aligned} \tag{7.1}$$

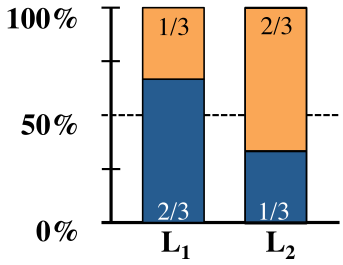


Figure 7.7: Allocations with maximum achievable utilizations and optimal isolation guarantees for C_A (orange) and C_B (dark blue).

```
Algorithm 6 High Utilization with Guarantees (HUG)
    Input: $\{\overrightarrow{d_k}\}$: reported correlation vector of $C_k$, $\forall k$
    Output: $\{\overrightarrow{a_k}\}$: guaranteed resource allocated to $C_k$, $\forall k$

    Stage 1: Calculate optimal isolation guarantee ($\mathcal{P}^*$) and minimum allocations $\overrightarrow{a_k} = \mathcal{P}^* \overrightarrow{d_k}, \ \forall k$

    Stage 2: Restrict maximum utilization for each of the $2P$ links, such that $c_k^i \leq \mathcal{P}^*, \ \forall i, \forall k$
```

where c_k^i is C_k's actual consumption[5] on link-i given a_k^i, and $\mathcal{P}^*$ is the optimal isolation guarantee.

We observe that an optimal algorithm would have restricted C_A's progress in Figure 7.4b and C_B's progress in Figure 7.4c to $\frac{2}{3}$. Consequently, they would not have been incentivized to lie and the prisoner's dilemma could have been avoided. Algorithm 6 – referred to as *High Utilization with Guarantees* (HUG) – is such a two-stage allocation mechanism that guarantees maximum utilization while optimizing the isolation guarantees across coflows and is strategyproof.

In the first stage, HUG allocates resources to maximize isolation guarantees across coflows. To achieve this, we pose our problem as a $2P$-resource fair sharing problem and use DRF [97, 166] to calculate $\mathcal{P}^*$. By reserving these allocations, HUG ensures isolation. Moreover, because DRF is strategy-proof, coflows are guaranteed to use these allocations (i.e., $c_k^i \geq a_k^i$).

While DRF maximizes the isolation guarantees (a.k.a. dominant shares), it results in low network utilization. In some cases, *DRF may even have utilization arbitrarily close to zero, and HUG can increase that to 100%* (**Lemma D.3.6**).

To achieve this, the second stage of HUG maximizes utilization while still keeping the allocation strategyproof. In this stage, we calculate upper bounds to restrict how much of the spare capacity a coflow can use in each link, with the constraint that the largest share across all links cannot increase (**Lemma D.3.1**). As a result, Algorithm 6 remains strategy-proofness across both stages. Because bandwidth restrictions can be applied locally, HUG can be enforced in individual machines.

Illustrated in Figure 7.7, the upper-bound is set at $\frac{2}{3}$ for both coflows, and C_B can use its elastic demand on link-2's spare resource, while C_A cannot as it has reached its bound on link-2.

[5] Can differ from allocation when coflows are lying.

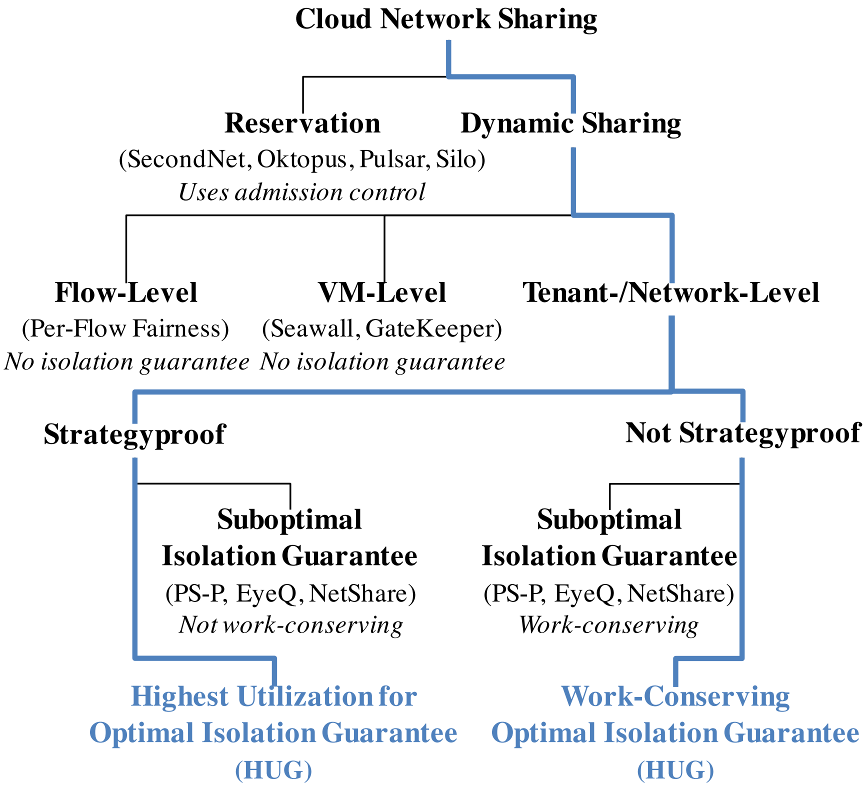


Figure 7.8: Design space for cloud network sharing.

7.5.3 HUG Properties

We list the main properties of HUG in the following.

1. In non-cooperative cloud environments, HUG is strategyproof (**Theorem D.3.3**), maximizes isolation guarantees (**Corollary D.3.4**), and ensures the highest utilization possible for an optimal isolation guarantee allocation (**Theorem D.3.5**).

2. In cooperative environments such as private datacenters, HUG maximizes isolation guarantees and is work-conserving. Work conservation is achievable because strategy-proofness is a non-requirement in this case.

3. Because HUG provides the optimal isolation guarantee, it provides min-cut proportionality (§ 7.5.3) in both non-cooperative and cooperative environments.

Figure 7.8 surveys the design space for cloud network sharing and places HUG in context by following the thick red lines through the design tree. At the highest level, unlike many alternatives [41, 46, 109, 123], HUG is a dynamic allocation algorithm. Next, HUG enforces its allocations at

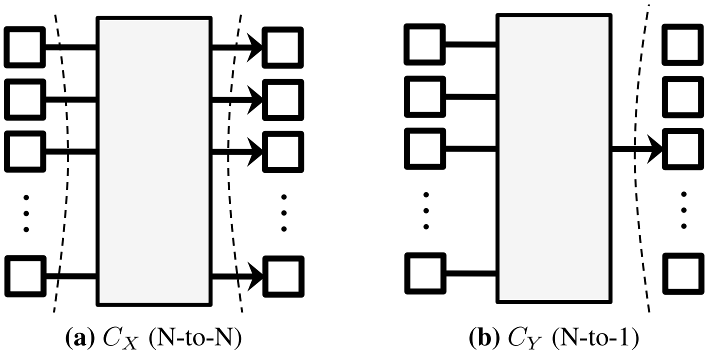

(a) C_X (N-to-N) **(b)** C_Y (N-to-1)

Figure 7.9: Communication patterns of C_X and C_Y with (a) two minimum cuts of size P, where P is the number of fabric ports, and (b) one minimum cut of size 1. The size of the minimum cut of a communication pattern determines its effective bandwidth even if it were placed alone.

the coflow-/tenant-/network-level, because flow- or VM-/machine-level allocations [179, 184] do not provide isolation guarantee.[6]

Due to the hard tradeoff between optimal isolation guarantee and work conservation in non-cooperative environments, *HUG ensures the highest utilization possible while maintaining the optimal isolation guarantee*. It incentivizes coflows to expose their true demands, ensuring that they actually consume their allocations instead of causing collateral damages. In cooperative environments, where strategy-proofness might be a non-requirement, *HUG simultaneously ensures both work conservation and the optimal isolation guarantee*. In contrast, existing solutions [124, 140, 171, 172] are suboptimal in both environments.

Min-Cut Proportionality

Prior work promoted the notion of *proportionality* [171], where tenants (coflows) would expect to receive total allocations proportional to their number of VMs *regardless* of communication patterns. Meaning, two coflows, each with N VMs, should receive equal bandwidth even if C_X has an all-to-all communication pattern (i.e., $\overrightarrow{d_X} = \langle 1, 1, \ldots, 1 \rangle$) and C_Y has an N-to-1 pattern (i.e., exactly one 1 in $\overrightarrow{d_Y}$ and the rest are zeros). Figure 7.9 shows an example. Clearly, C_Y will be bottlenecked at its only receiver; trying to equalize them will only result in low utilization. As expected, FairCloud proved that such proportionality is not achievable as it decreases both isolation guarantee and utilization [171]. None of the existing algorithms provide proportionality.

Instead, we consider a relaxed notion of proportionality, called *min-cut proportionality*, that depends on communication patterns and ties proportionality with a coflow's progress. Specifically, each coflow receives minimum bandwidth proportional to the size of the *minimum cut* [92] of their communication patterns. Meaning, in the earlier example, C_X would receive P times more total bandwidth than C_Y, but they would have the optimal isolation guarantee ($\mathcal{P}_X = \mathcal{P}_Y = \frac{1}{2}$).

[6] HUG can be applied in those settings with minor adaptations.

Min-cut proportionality and optimal isolation guarantee can coexist, but they both have tradeoffs with work conservation.

7.6 Design Details

This section discusses how to implement, enforce, and expose HUG (§7.6.1), how to exploit placement to further improve HUG's performance (§7.6.2), and how HUG can handle weighted, heterogeneous scenarios (§7.6.3).

7.6.1 Architectural Overview

In public clouds, where tenants generate individual coflows, HUG can easily be implemented atop existing monitoring infrastructure (e.g., Amazon CloudWatch [2]). Tenants would periodically update the correlation vectors of their coflows through a public API, and the operator would compute new allocations and update enforcing agents within milliseconds.

HUG API The user-facing API simply transfers a coflow's correlation vector ($\overrightarrow{d_k}$) to the operator. $\overrightarrow{d_k} = \langle 1, 1, \ldots, 1\rangle$ is used as the default correlation vector. By design, HUG incentivizes tenants to report and maintain accurate correlation vectors. This is because the more accurate it is – instead of the default $\overrightarrow{d_k} = \langle 1, 1, \ldots, 1\rangle$ – the higher are its progress and performance.

Calculating the correlation vector of a coflow is simple. For clairvoyant coflows, one can use MADD. For non-clairvoyant coflows, one can just report the current rate estimations. Existing techniques in traffic engineering can provide good accuracy in estimating and predicting demand matrices for coarse time granularities [50, 51, 63, 127, 128].

Centralized Computation For any update, the operator must run Algorithm 6. Although Stage-1 requires solving a linear program to determine the optimal isolation guarantee (i.e., the DRF allocation) [97], it can also be rewritten as a closed-form equation [166] when coflows can scale up and down following their normalized correlation vectors. The progress of all coflows after Stage-1 of Algorithm 6 – i.e., the optimal isolation guarantee – is:

$$\mathcal{P}^* = \frac{1}{\max\limits_{1 \le i \le 2P} \sum\limits_{k=1}^{M} d_k^i} \tag{7.2}$$

The guaranteed minimum allocations of C_k can be calculated as $a_k^i = \mathcal{P}^* d_k^i$ for all $1 \le i \le 2P$.

Equation (7.2) is computationally inexpensive. For our 100-machine cluster, calculating $\mathcal{P}^*$ takes about 5 microseconds. Communicating the decision to all 100 machines takes just 8 milliseconds and to $100,000$ (emulated) machines takes less than 1 second (§7.7.2).

Local Enforcement Enforcement in Stage-2 of Algorithm 6 is simple as well. After reserving the minimum uplink and downlink allocations for each coflow, each machine needs to ensure that no coflow can consume more than $\mathcal{P}^*$ fraction of the machine's up or down link capacities (C^i) to the network; i.e., $a_k^i \leq c_k^i \leq \mathcal{P}^*$. The spare is allocated among coflows using local max-min fairness *subject to* coflow-specific upper-bounds.

7.6.2 VM Placement and Re-Placement/Migration

While $\mathcal{P}^*$ is optimal for a *given* placement, it can be improved by changing the placement of a coflow's VMs based on its correlation vector. One must perform load balancing across all machines to minimize the denominator of Equation (7.2). Cloud operators can employ optimization frameworks such as [54] to perform initial VM placement and periodic migrations with an additional load balancing constraint. However, VM placement is a notoriously difficult problem because of often-incompatible constraints such as fault-tolerance and collocation [54], and we consider a detailed study a future work. It is worth noting that with *any* VM placement, HUG provides the highest attainable utilization without sacrificing optimal isolation guarantee and strategy-proofness.

7.6.3 Additional Constraints

Weighted Coflows Giving preferential treatment to coflows is simple. Just using $w_k\overrightarrow{d_k}$ instead of $\overrightarrow{d_k}$ in Equation (7.2) would account for coflow weights (w_k for C_k) in calculating $\mathcal{P}^*$.

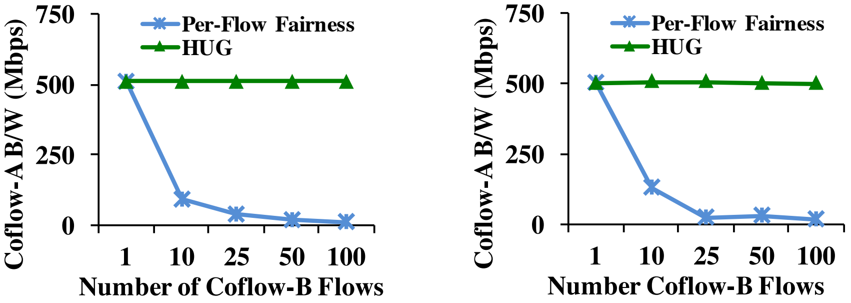
Heterogeneous Capacities Because allocations are calculated independently in each machine based on $\mathcal{P}^*$ and local capacities (C^i), HUG supports heterogeneous link capacities.

Bounded Demands So far we have considered only elastic coflows. If C_k has bounded demands, i.e., $d_k^i < 1$ for all $i \in [1, 2P]$, calculating a common $\mathcal{P}^*$ and corresponding $\overrightarrow{a_k}$ in *one round* using Equation (7.2) will be inefficient. This is because C_k might require less than the calculated allocation, and being bounded, it cannot elastically scale up to use it. Instead, we must use the *multi-round* DRF algorithm [166, Algorithm 1] in Stage-1 of HUG; Stage-2 will remain the same. Note that this is similar to max-min fairness in a single link when one flow has a smaller demand than its $\frac{1}{n}$th share.

7.7 Evaluation

We evaluate HUG using simulations and EC2 experiments under controlled settings as well as using production traces. The highlights of our findings are as follows.

- HUG isolates coflows at both host and network levels, and it can scale up to $100,000$ machine with less than one second overhead (§7.7.2).



(a) Simulation

(b) EC2

Figure 7.10: Bandwidth allocation of C_A at a particular VM, while C_B connects to increasingly larger number of destinations from that same VM.

- HUG ensures the optimal minimum guarantee – almost $7000\times$ more than per-flow fairness and about $7.4\times$ more than PS-P in production traces – while providing $1.4\times$ higher utilization than DRF (§7.7.3).
- HUG outperforms per-flow fairness (PS-P) by $45.75\times$ $(1.25\times)$ in terms of maximum slowdown and by $1.49\times$ $(1.14\times)$ in minimizing the average CCT (§7.7.4).
- HUG outperforms Varys in terms of maximum CCT by $1.77\times$, even though Varys is $1.55\times$ better in terms of maximum slowdown and $1.45\times$ better in minimizing the average CCT. This highlights the tradeoff between optimal isolation guarantee and utilization even in cooperative environments (§7.7.4).

7.7.1 Methodology

We ran our experiments on 100 `m2.4xlarge` Amazon EC2 [3] instances running on Linux kernel $3.4.37$ and used the default `htb` and `tc` implementations. While there had been proposals for more accurate `qdisc` implementations (e.g., EyeQ [124] and FastPass [167]), the default `htb` worked sufficiently well for our purposes. Each of the machines had 1 Gbps NICs, and we could use the full 100 Gbps bandwidth simultaneously.

For the simulations, we used a flow-level simulator written in Java.

Before each set of experiments, we separately describe corresponding methodology, experimental settings, metrics, and workload in more details.

7.7.2 Benchmarks

We start our evaluation by presenting simulation and experimental results from benchmarks performed on controlled communication patterns and correlation vectors.

Host-Level Scenarios

We first focus on HUG's ability to ensure coflow isolation at individual machines. We consider a scenario with two coflows C_A and C_B that have two sources (A_1 and B_1) collocated on the same physical machine. A_1 communicates with destination A_2, whereas B_1 communicates with up to 100 destinations. Assuming both coflows demand the entire capacity from the host machine (i.e., $d_A^1 = d_B^1 = 1$), the outgoing link's bandwidth must be divided *equally* between them.

Figure 7.10 presents the allocations of A_1 as B_1 communicates with increasingly larger number of destinations. As B_1 communicates with more and more destinations, A_1's share keep decreasing using per-flow fairness. On the contrary, HUG isolates C_A from C_B and ensures maximum isolation guarantee in both simulation and experiment. Note that PS-P would have provided the same allocation [171, Figure 8], if we had an available implementation with switch support.

Dynamic Network-Level Scenarios

We now extend our scope to an entire cluster with 100 EC2 machines. In this scenario, we have three coflows C_A, C_B, and C_C that arrive over time, each with 100 VMs; i.e., VMs A_i, B_i, and C_i are collocated on the ith physical machine. However, they have different communication patterns: coflows C_A and C_C have pairwise one-to-one communication patterns (100 flows each), whereas C_B follows an all-to-all pattern using $10,000$ flows. Specifically, A_i communicates with $A_{(i+50)\%100}$, C_j communicates with $C_{(j+25)\%100}$, and any B_k communicates with all B_l, where $i, j, k, l \in \{1, ..., 100\}$. Assume that each coflow demands the entire capacity at each machine; hence, the entire capacity of the cluster should be equally divided among the active coflows to maximize isolation guarantees.

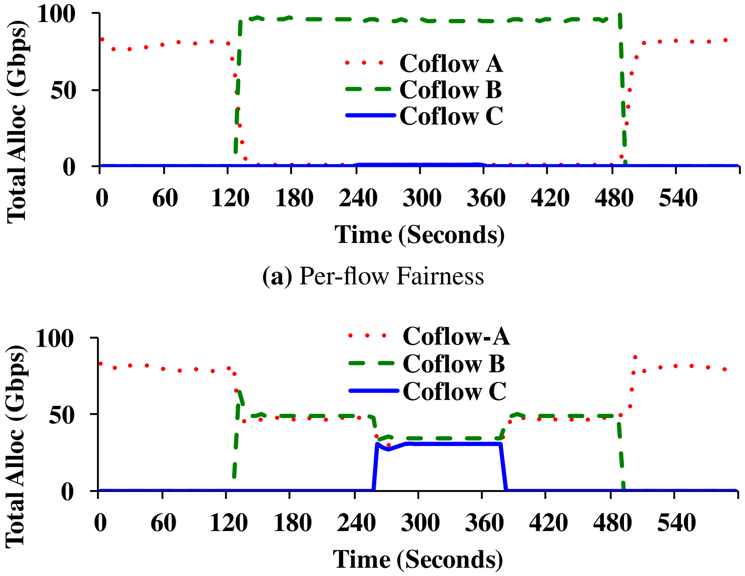
Figure 7.11a shows that as soon as C_B arrives, it takes up the entire capacity in the absence of isolation guarantee. C_C receives only marginal share as it arrives after C_B and leaves before it. Note that C_A (when alone) uses only about 80% of the available capacity; this is simply because just one TCP flow often cannot saturate the link.

Figure 7.11b shows HUG in action. As coflows arrive and depart, their shares are dynamically calculated, propagated, and enforced in each machine of the cluster.

Scalability

The time for calculating new allocations upon coflow arrival or departure using HUG is less than 5 microseconds in our 100 machine cluster. In fact, a recomputation due to a coflow's arrival, departure, or change of correlation vector would take about 8.6 milliseconds on average for a $100,000$-machine datacenter.

Communicating a new allocation takes less than 10 milliseconds to 100 machines and around 1 second for $100,000$ *emulated* machines (i.e., sending the same message 1000 times to each of the 100 machines).



(**a**) Per-flow Fairness

(**b**) HUG

Figure 7.11: [EC2] Bandwidth consumptions of three coflows arriving over time in a 100-machine EC2 cluster. Each coflow has 100 VMs, but each uses a different communication pattern (§7.7.2). (b) HUG isolates coflows C_A and C_C from coflow C_B.

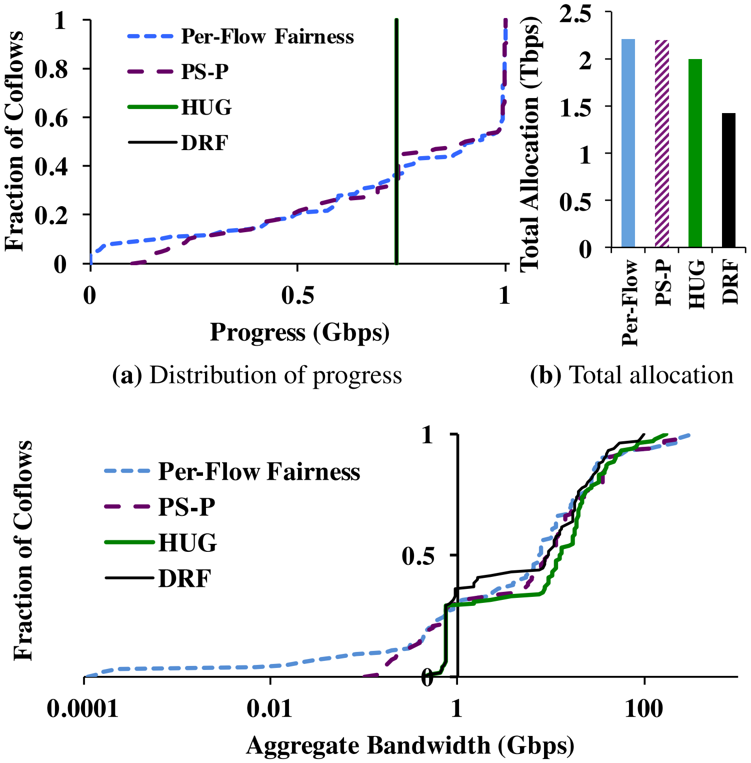
7.7.3 Instantaneous Fairness

While Section 7.7.2 evaluated HUG in controlled, synthetic scenarios, this section focuses on HUG's instantaneous characteristics in the context of a large-scale cluster. Due to the lack of implementations of DRF [97] and PS-P [171], we compare against them only in simulation.

Methodology We use a one-hour snapshot of a MapReduce trace extracted from a 3200-machine Facebook cluster with 100 concurrent jobs collected by Popa et al. [171, Section 5.3]. Machines are connected to the network using 1 Gbps NICs. In the trace, a job with M mappers and R reducers – hence, the corresponding $M \times R$ shuffle – is described as a matrix with the amount of data to transfer between each M-R pair. We calculated the correlation vectors of individual shuffles/coflows from their communication matrices using MADD.

Given the workload, we calculate the progress of each coflow using different mechanisms and cross-examine characteristics such as isolation guarantee, utilization, and proportionality.

Impact on Progress Figure 7.12a presents the distribution of progress of each coflow. Recall that the progress of a coflow is the amount of bandwidth it is receiving in its bottleneck up or downlink (i.e., progress can be at most 1 Gbps). Both HUG and DRF (overlapping vertical lines in Figure 7.12a) ensure the same progress (0.74 Gbps) for all coflows. Note that despite having same progress, coflows will finish at different times based on how much data each one has to send (§7.7.4). Per-flow fairness and PS-P provide very wide ranges: 112 Kbps to 1 Gbps for the



(a) Distribution of progress

(b) Total allocation

(c) Distribution of allocations

Figure 7.12: [Simulation] HUG ensures higher isolation guarantee than high-utilization schemes such as per-flow fairness and PS-P, and provides higher utilization than multi-resource fairness schemes such as DRF.

former and 0.1 Gbps to 1 Gbps for the latter. Coflows with many flows crowd out the ones with fewer flows under per-flow fairness, and PS-P suffers by ignoring correlation vectors and through indiscriminate work conservation.

Impact on Utilization By favoring large coflows, per-flow fairness and PS-P do succeed in their goals of increasing network utilization (Figure 7.12b). The former utilizes 69% of 3.2 Tbps total capacity across all machines and the latter utilizes 68.6%. In contrast, DRF utilizes only 45%. HUG provides a common ground by extending utilization to 62.4% without breaking strategy-proofness and providing optimal isolation guarantee.

Figure 7.12c breaks down total bandwidth of each coflow, demonstrating two high-level points:

1. HUG ensures overall higher utilization ($1.4\times$ on average) than DRF by ensuring equal progress for smaller coflows and by using up additional bandwidth for larger coflows. It does so while ensuring the same optimal, isolation guarantee as DRF.

Bin	**1 (SN)**	**2 (LN)**	**3 (SW)**	**4 (LW)**
% of Coflows	52%	16%	15%	17%

Table 7.2: Coflows binned by length (**S**hort and **L**ong) and width (**N**arrow and **W**ide).

2. Per-flow fairness crosses HUG at the 90th percentile; i.e., the top 10% coflows receive more bandwidth than they do under HUG, while the other 90% receive less than they do using HUG. PS-P crosses over at the 76th percentile.

Impact on Proportionality A collateral benefit of HUG is that coflows receive allocations proportional to their bottleneck demands. Consequently, despite the same progress across all coflows (Figure 7.12a), their total allocations vary (Figure 7.12c) based on the size of minimum cuts in their communication patterns.

7.7.4 Long-Term Characteristics

In this section, we compare HUG's long-term impact on performance.

Methodology For these simulations, we use a MapReduce/Hive trace from a 3000-machine production Facebook cluster – the same trace used in the earlier chapters. Coflows in this trace have diverse length (i.e., size of the longest flow) and width (i.e., the number of flows) characteristics (Table 7.2). We consider a coflow to be *short* if its longest flow is less than 5 MB and *narrow* if it has at most 50 flows. We calculated the correlation vector of each coflow as we did before (§ 7.7.3).

Metrics We consider two metrics: *maximum slowdown* and *average CCT* to respectively measure long-term progress and performance characteristics.

We measure long-term progress of a coflow as its CCT due to a scheme normalized by the minimum CCT if it were running alone; i.e.,

$$\text{Slowdown} = \frac{\text{Compared Duration}}{\text{Minimum Duration}}$$

The minimum value to slowdown is one – the smaller the better.

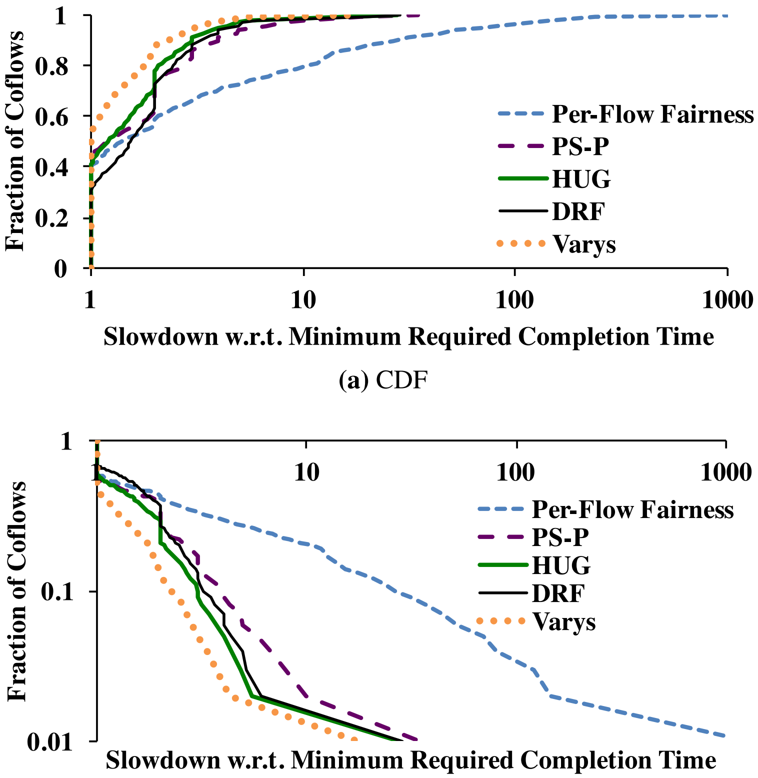
We measure performance as the CCT of a scheme normalized by HUG's CCT; i.e.,

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

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

	Min	Max	AVG	STDEV
Per-Flow Fairness	1	1281	15.52	65.54
PS-P	1	35	2.22	2.97
HUG	1	28	1.86	2.25
DRF	1	29	2.11	2.66
Varys	1	18	1.43	0.99

Table 7.3: [Simulation] Slowdowns using different mechanisms w.r.t. the minimum CCTs.



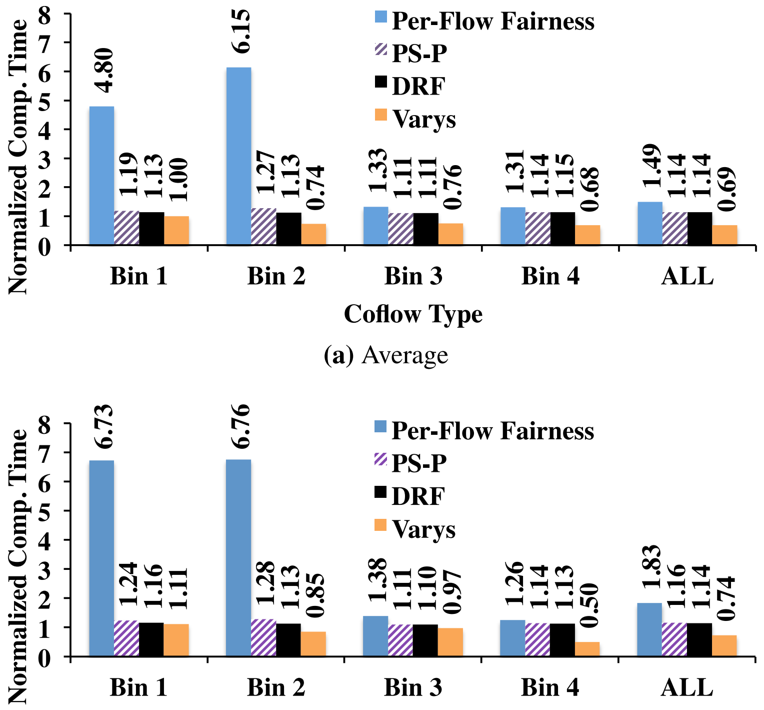
(a) CDF

(b) CCDF

Figure 7.13: [Simulation] Slowdowns of coflows using different mechanisms w.r.t. the minimum CCTs. Both X-axes and the Y-axis of (b) are in log scale.

Improvements Over Per-Flow Fairness HUG improves over per-flow fairness both in terms of slowdown and performance. The maximum slowdown using HUG is $45.75\times$ better than that of per-flow fairness (Table 7.3). Overall, HUG provides better slowdown across the board (Figure 7.13) – 61% are better off using HUG and the rest remain the same.

HUG improves the average and 95th percentile CCT by $1.49\times$ (Figure 7.14). The biggest wins comes from bin-1 and bin-2 that include the so-called narrow coflows with less than 50 flows.



(a) Average

(b) 95th percentile

Figure 7.14: [Simulation] Average and 95th percentile improvements in CCT using HUG.

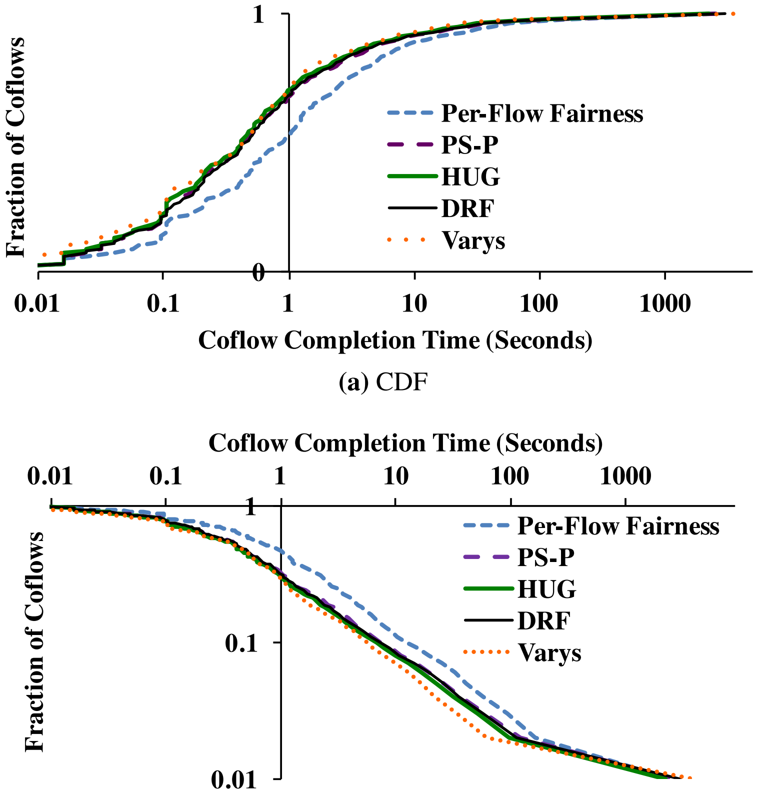
This reinforces the fact that HUG isolates coflows with fewer flows from those with many flows. Overall, HUG pays off across all bins. Finally, it provides $1.12\times$ better maximum CCT than per-flow fairness (Figure 7.15).

Improvements Over PS-P HUG improves over PS-P in terms of maximum slowdown by $1.25\times$. 45% of the coflows are better off using HUG. HUG also providers better average CCT than PS-P for an overall improvement of $1.14\times$. Large improvements again come in bin-1 and bin-2 because PS-P also favors coflows with more flows. HUG also provides $1.21\times$ better maximum CCT than PS-P.

Note that instantaneous high utilization of per-flow fairness and PS-P (§7.7.3) does not help in the long run due to lower isolation guarantee.

Improvements Over DRF While HUG and DRF has the same maximum slowdown, 70% coflows are better off using HUG. HUG also providers better average CCT than DRF for an overall improvement of $1.14\times$. Furthermore, it provides $1.34\times$ better maximum CCT than DRF.

Comparison to Varys Varys outperforms HUG by $1.55\times$ in terms of the maximum slowdown and by $1.45\times$ in terms of average CCT. However, because Varys attempts to improve the average



(a) CDF

(b) CCDF

Figure 7.15: [Simulation] CCTs using different mechanisms w.r.t. the minimum completion times. Both X-axes and the Y-axis of (b) are in log scale.

CCT by prioritization, it risks in terms of maximum CCT. More precisely, HUG outperforms Varys by $1.77\times$ in terms of maximum CCT.

7.8 Related Work

Single-Resource Fairness The concept of max-min fairness was first proposed by Jaffe [121] to ensure at least $\frac{1}{n}$th of a link's capacity to each flow. Thereafter, many mechanisms have been proposed to achieve it, including weighted fair queueing (WFQ) [80, 165] and approaches similar to or extending WFQ [49, 100, 102, 185, 188]. We generalize max-min fairness to parallel communication activities observed in datacenter and cloud applications, and show that unlike in the single-link scenario, optimal isolation guarantee, strategy-proofness, and work conservation cannot coexist.

Multi-Resource Fairness Dominant Resource Fairness (DRF) [97] by Ghodsi et al. maximizes the dominant share of each user while being strategyproof. A large body of work – both before [160, 193] and after [83, 112, 166] DRF – have attempted to improve the system-level efficiency of multi-resource allocation; however, all come at the cost of strategy-proofness. We have proven that work-conserving allocation without strategy-proofness can hurt utilization instead of improving it.

Dominant Resource Fair Queueing (DRFQ) [96] maintains DRF properties over time within *individual* middleboxes. HUG generalizes DRF to environments with elastic demands to increase utilization across the entire network.

Joe-Wong et al. [125] have a presented a unifying framework to capture fairness-efficiency tradeoffs in multi-resource environments. This framework assumes a *cooperative* environment; meaning, tenants do not lie about their demands. HUG falls under their FDS family of mechanisms. In non-cooperative environments, however, we have shown that the interplay between work conservation and strategy-proofness comes to the forefront, and our work complements the framework of [125].

Network-Wide / Tenant-Level Fairness Proposals for sharing cloud networks range from static allocation [41, 46, 123] and VM-level guarantees [179, 184] to variations of network-wide sharing mechanisms [124, 140, 171, 172, 202]. We refer the reader to the survey by Mogul and Popa [153] for an overview.

FairCloud [171] stands out by systematically discussing the tradeoffs and addresses several limitations of other approaches. Our work generalizes FairCloud [171] and many proposals similar to FairCloud's PS-P policy [124, 172, 179]. When all tenants (coflows) have elastic demands, i.e., all correlation vectors have all elements as 1, we give the same allocation; for all other cases, we provide higher isolation guarantee and utilization.

Efficient Schedulers Researchers have also focused on efficient scheduling and/or packing of datacenter resources to minimize job and communication completion times [37, 66–68, 82, 105, 118]. Our work is complementary to these pieces of work in that we focus on coflow isolation instead of average-case performance.

7.9 Summary

HUG ensures highest network utilization *with* optimal isolation guarantee across multiple coflows in non-cooperative environments. It outperforms efficient coflow schedulers such as Varys in terms of maximum CCT, while sacrificing the average CCT. In cooperative environments, where strategy-proofness might be a non-requirement, HUG simultaneously ensures both work conservation and the optimal isolation guarantee.

Results in this chapter are not specific to coflows either. We have proved that there is a strong tradeoff between optimal isolation guarantees and high utilization in non-cooperative public clouds, and work conservation can decrease utilization instead of improving it, because no network sharing algorithm remains strategyproof in its presence.

HUG generalizes single-resource max-min fairness to multi-resource environments whenever tenants' demands on different resources are correlated and elastic. In particular, it provides optimal isolation guarantee, which is significantly higher than that provided by existing multi-tenant network sharing algorithms. HUG also complements DRF with provably highest utilization without sacrificing other useful properties of DRF. Regardless of resource types, the identified tradeoff exists in general multi-resource allocation problems, and all those scenarios can employ HUG.

Chapter 8

Conclusions and Future Work

How do we improve the communication performance of distributed data-parallel applications in the era of massively parallel clusters? This dissertation shows that the answer lies in the applications themselves: by realigning application-level performance objectives with network-level optimizations, the coflow abstraction can improve the performance of coexisting applications, and when necessary, it can be used to provide performance isolation as well.

Apart from the performance gains enabled by the coflow abstraction by generalizing point-to-point flows to multipoint-to-multipoint communication, a core contribution of this work is the generalized *all-or-nothing* characteristic captured by each coflow that is pervasive across different types of resources in data-parallel clusters. The generalization is due to the coupled nature of the network – unlike CPU speed or disk/memory capacity, one must allocate resources on both senders and receivers in the case of the network. Additionally, we discovered the *concurrent open shop scheduling with coupled resources* problem and characterized it for the first time, generalizing concurrent open shop scheduling that applies to parallel jobs, distributed files, and distributed in-memory datasets. Consequently, many of the algorithms and principles derived from this dissertation can potentially be applied to contexts beyond the network.

In the rest of this chapter, we summarize some of the lessons that influenced this work in Section 8.1, highlight broader impacts of this dissertation in Section 8.2, propose several directions for future work in Section 8.3, and finally, conclude.

8.1 Lessons Learned

Leverage Application-Level Knowledge The genesis of the coflow abstraction traces back to one simple observation – *a communication stage of a distributed data-parallel application cannot complete until all its flows have completed.* Instead of just improving tail latencies of flows in datacenters, we examined the applications that create the flows and how individual flows impact application-level performance.

We observed that the decades-old contract between the network and the applications using it – a point-to-point flow's performance is directly aligned with that of a client-server application – has since changed. Applications today are increasingly more distributed by design, and their communication takes place between groups of tasks in successive computation stages. Such applications can make progress only after *all* the flows have completed. While this phenomenon has become common over the past decade, none of the previous work questioned the flow abstraction itself when optimizing flow-level performance.

The coflow abstraction bridges the gap by generalizing point-to-point flows to multipoint-to-multipoint communication. As applications evolve, we must always be prepared to revisit and question our assumptions and make adjustments to accommodate application-aware changes in network-level abstractions to keep the two aligned.

Exploit the Capabilities of Underlying Environments In addition to applications, the network itself – especially in the datacenter context – has gone through a major revolution. Datacenter networks today have orders-of-magnitude higher bandwidth and lower latency than traditional local-area networks or the Internet. Our decision to extensively exploit the characteristics of datacenters have had several implications on this work.

First, low-latency datacenter networks made it possible for us to even consider the coflow abstraction that fundamentally requires coordination to be effective.

Second, in terms of building coflow-enabling systems, we opted for a centralized design based on the collective community experience of building compute frameworks or file systems for datacenters.[1] The simplicity allowed us to focus on different aspects of intra- and inter-coflow scheduling without being encumbered by the challenges of decentralized designs. Although a centralized design for managing the network does not seem controversial anymore with the rise of software-defined networking (SDN) in last few years, it was indeed one when we started.

Finally, assuming full bisection bandwidth in datacenters allowed us to have a simplified network model for analyses and algorithm design. We found empirically that the resulting algorithms work well even when datacenters have some oversubscription. Recent reports from the industry [186] suggest that building full bisection bandwidth networks at large scale is feasible today.

Design for Shared Environments While it is appealing to optimize for a specific application or environment, another lesson from our work is that real-world deployments are more complex, have more constraints, and often shared among a variety of applications. Because we designed to accomodate as many distributed data-parallel applications as possible, we had to iteratively simplify our abstraction and interfaces to leverage it. Over time, we found that this very simplicity allowed us to apply the coflow abstraction to a diverse set of applications and objectives.

Another key decision was to aim for public clouds that are shared between multiple tenants. While this may have restricted us from exploiting the full potential of coflow scheduling from a performance perspective, it forced us to extract as much as possible from the application layer without making any changes to the underlying operating system or network equipment. In the long

[1] Although not covered in this dissertation, the author has also worked on building and deploying other datacenter-scale systems including compute frameworks, file systems, and resource allocators [54, 63, 208].

run, it increased the portability of our artifacts, and allowed us and others to deploy, run, evaluate, build on top of, and compare against them on diverse environments.

Perhaps the most surprising outcome of designing for shared environments is that there are more opportunities to improve over existing solutions. For example, the optimal clairvoyant intra-coflow algorithm (MADD) can improve over traditional per-flow fairness by up to $1.5\times$. In the presence of coflows from multiple applications, Varys gains significantly more over traditional techniques because they make increasingly more mistakes. Similarly, HUG provides orders-of-magnitude better isolation guarantee among coflows in public clouds than per-flow fairness.

Simplicity Increases Applicability Finally, part of what made the coflow abstraction increasingly more general and applicable over time is its simplicity. We started with a collection of parallel flows belonging to one application with complete knowledge of individual flows under the assumption that the application has a single stage, all tasks in that stage are scheduled at the same time, and tasks do not fail. As we simplified the abstraction by removing each of these assumptions, coflows became applicable to multi-stage jobs, where each stage can have multiple waves of tasks, and there can be arbitrary failures, restarts, and speculative executions of those tasks. By focusing on the instantaneous share of a coflow instead of its completion time, we showed that one can also provide performance isolation.

Individual coflow-based solutions compose as well. For example, an operator can enforce isolation across tenant-level coflows in a shared cloud, and each tenant can perform efficient coflow scheduling inside its isolated share. Similarly, for clairvoyant coflows, one can support coflow deadlines alongside minimizing the average completion time. Finally, separating coflow scheduling from transport layer protocols enabled us to build solutions that run well on today's networks and are likely to be able to leverage future innovations in the underlying layers.

8.2 Impact

This dissertation has already impacted several practical and theoretical aspects of networking in distributed data-parallel applications and big data clusters in both academic and industrial settings.

8.2.1 Theoretical Impact

Scheduling Perhaps the biggest theoretical contribution of this work is simply identifying and characterizing the "concurrent open shop scheduling with coupled resources" family of problems, generalizing the concurrent open shop scheduling problem [144, 149, 155, 180]. In addition, we showed that unlike many traditional scheduling problems, Graham's list scheduling heuristic [103, 104] – and permutation scheduling in general – is not guaranteed to result in good solutions due to the matching constraints from sender-receiver coupling in the network.

The first approximation algorithm for clairvoyant inter-coflow scheduling to minimize the average CCT is due to Qiu et al. [175]. They presented a polynomial-time deterministic approximation algorithm with an approximation ratio of $\frac{67}{3}$ and a randomized version of the algorithm, with a ratio $9 + \frac{16\sqrt{2}}{3}$. Similar results do not yet exist for the non-clairvoyant variation.

Routing RAPIER [213] makes a first attempt toward co-optimizing scheduling and routing in datacenters using coflows as building blocks. By exploiting coflow-encapsulated information during routing, RAPIER selects routes for individual flows in oversubscribed, multi-path datacenter networks, while scheduling coflows at the same time. The joint solution can perform better than scheduling or routing independently.

Placement Corral [122] performs network-aware task placement in the context of distributed data-parallel applications. The authors showed that using clairvoyant inter-coflow scheduling (i.e., Varys) on top of Corral can significantly improve the median job completion time [122, Figure 14]. Even when not using Corral, Varys decreased the median job completion time by 45% over TCP in their evaluation. This reaffirms our findings about the impact of CCT on job completion times on a completely different set of workloads.

8.2.2 Real-World Impact

Open-Source Software Software artifacts developed throughout this dissertation are available as open-source software, enabling several groups to work on coflow-aware scheduling in optical as well as hybrid networks. A simplified version of Cornet now ships as the default broadcast mechanism for Apache Spark since release-1.1.0 and used by thousands of users and hundreds of companies for different applications.

Another vote of confidence for our work comes from the recent move toward separating communication layers from individual applications and consolidating them under a separate subsystem. Key examples include the pluggable Shuffle Service exposed by Apache YARN [195] that is used by Hadoop, Spark, and other systems, as well as the communication layer design of TensorFlow, Google's latest machine learning system [28]. This is a crucial step toward the generalized architecture for cross-application communication optimization proposed in this dissertation.

Network Support for Low-Latency Coflows Baraat [82] performs decentralized coflow-aware scheduling that works well for small coflows. In the process, the authors demonstrated that coflows can be implemented at the packet level through cooperation between network equipment and lower layers of the network stack in end hosts.

On a similar note, Ferguson et al. [89] outlined how coflows can exploit in-network isolation mechanisms for avoiding some of the performance issues faced by our application-layer implementations.

Going forward, we envision pervasive use of coflows not only within datacenters but in other contexts as well.

8.3 Future Work

In this section, we identify several avenues for future work. Whenever possible, we highlight the primary challenges and possible solutions to address them.

8.3.1 Scheduling-Routing Co-Design for Coflows Over General Graphs

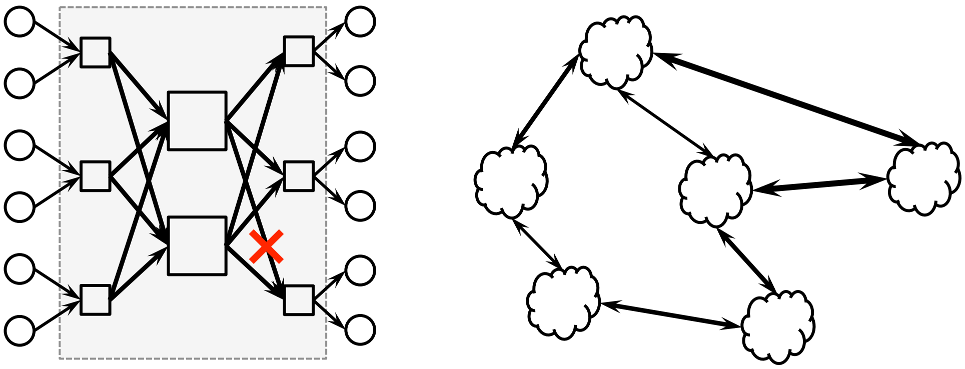
Throughout this dissertation, we have assumed datacenter networks to have full bisection bandwidth and network edges to be the only bottlenecks. While this is often true in practice [106, 158, 186] and simplifies the designs and analyses of our solutions, there are three natural scenarios where this assumption may not hold.

1. Oversubscribed datacenter networks;
2. Link failures in full bisection bandwidth networks; and
3. Geo-distributed data-parallel applications.

The first is simpler because oversubscribed datacenter networks often have certain structure. For example, rack-to-core links are typical points of oversubscription in many cases [63]. Modifying our solutions to schedule the oversubscribed links instead of machine-to-rack links can potentially address this scenario. The second increases difficulty by allowing internal links to become bottlenecks. Datacenter networks have multiple paths inside the fabric, and link failures can break the illusion of full bisection bandwidth, causing irregular oversubscription. Finally, geo-distributed applications deals with data across multiple datacenters throughout the planet [174, 196], and any internal node (i.e., a datacenter) of the network can also be a sender or receiver of a coflow.

In all cases, there are many paths – possibly with heterogeneous capacities – for each of the flows in a coflow (Figure 8.1), and bottlenecks are *not* guaranteed to appear only at network edges. Meaning, we must simultaneously consider scheduling over time and routing across paths to avoid internal bottlenecks.

Even in this case, there are two steps to coflow scheduling: (1) scheduling one coflow, and (2) scheduling multiple coflows to optimize different objectives. To address the first problem in this new context, one can consider a multi-commodity max-flow min-cut based approach [142] that would take into account the details of a coflow, network topology, and link bandwidths. The intuition is to identify the bottlenecks within a coflow to identify appropriate multipath bandwidth allocations to minimize its CCT. Next, to minimize the average CCT, one can try two alternatives. One approach would be formulating and solving a joint optimization problem. Another would be to examine variations of smallest-first heuristics similar to the ones proposed in this dissertation. In summary, research in this direction will make coflow scheduling more robust to network-level variations.



(a) Multi-path datacenter networks

(b) Inter-datacenter networks

Figure 8.1: Networks with multiple paths between sender-receiver pairs or with link failures can have in-network bottlenecks.

8.3.2 Coflow Identification Without Application Modification

Another key assumption throughout this dissertation is that developers are willing to make changes to their code to use coflow APIs, and they can *correctly* identify and introduce coflows in their applications. However, despite the best of efforts, both keeping APIs up-to-date with underlying libraries and learning new APIs to use them correctly are uphill battles [178, 182]. This raises a fundamental question: *can we automatically identify coflows without requiring any application-level modifications?*

To address this, one can use machine learning to identify coflows based on features such as communication patterns and flow characteristics. While Internet traffic classification has a rich literature [52,130,150,154,161], some intrinsic differences – e.g., mutual dependencies of flows in a coflow – prevents their direct adoption. Moreover, timeliness is paramount in coflow identification because its result would be the input for coflow scheduling algorithms.

Even the best identification algorithm will misidentify some coflows. The challenge is then designing *error-tolerant* scheduling algorithms. This is different from the non-clairvoyant scheduler in Aalo, because we might not even know which flows belong to a coflow and which ones do not.

8.3.3 Decentralized Coflow Schedulers

While coordination is critical for good scheduling performance (**Theorem C.1.1**), it has its overheads. Small coflows suffer the most, especially in Varys that requires tight coordination. Aalo avoids this by not scheduling coflows until they cross a certain threshold. Even then, small coflows do not perform better than using per-flow fairness or TCP; they are just not worse off using Aalo. It raises a natural question: *can we do better?*

One possible direction is changing switches and network components to perform FIFO scheduling based on CoflowIds [82]. This requires significant changes and can cause head-of-line blocking; however, for coflows smaller than a certain threshold, application-level solutions cannot compete with hardware-based ones.

On the other hand, for a large class of coflows, one can consider decentralizing Aalo. It will involve two key factors:

1. Decentralized calculation of coflow sizes, and

2. Avoiding receiver-side contentions without coordination.

Approximate aggregation schemes such as Push-Sum [133] can be good starting points to develop solutions for the former within reasonable time and accuracy. The latter is more difficult, because it relies on fast propagation of receiver feedback throughout the entire network for quick convergence of sender- and receiver-side rates. Designing distributed protocols to achieve these goals with bounded convergence guarantees is an important direction for future work.

8.3.4 Decoupling Bandwidth and Latency for Hierarchical Coflow Scheduling

We have considered three inter-coflow objectives in this dissertation: minimizing the average coflow completion time, meeting coflow deadlines, and providing instantaneous fair shares across coexisting coflows. However, by considering each one independently, we have made an implicit assumption: all applications put coflows in the *same* queue, and we must process them to optimize for one specific objective for that single queue.

We observe two major trends – both enabled by state-of-the-art cloud platforms [11, 14, 24, 117, 195] – that point toward an increasingly more complex setting in practice. First, the traditional model of offline batch processing [77,119,208] is increasingly being complemented by near-realtime, online stream processing systems [27,32,157,209]. To enable data sharing and seamless transition between the two models, both coexist on shared clusters. The key difference between the two computation models lies in their performance goals. While the *average* completion time can sufficiently capture the performance of batch jobs [68, 82, 105], each streaming application is a continuous string of small batch jobs, where the completion time of each *discrete* job, i.e., their responsiveness, is of utmost importance. The second trend is the use of hierarchical scheduling in many organizations and cloud platforms that allocate cluster resources among diverse computing models with an aim to impose organizational structures and priorities [53]. Taken together, they raise one fundamental question: *how to optimize different performance objectives for coflows in multiple queues that are organized in a hierarchical fashion*?

If we momentarily ignore the data-parallel context, the fundamental problem is not new. Supporting multiple classes of traffic while maintaining organizational hierarchies is a classic networking problem that gave rise to the hierarchical link-sharing model [91,183]. We can think of extending it to the entire fabric so that throughput-sensitive queues of batch jobs and latency-sensitive streaming applications can coexist following organizational preferences to create a hierarchical cluster-sharing model. Specifically, we can take the coflow abstraction to its logical extreme by considering it to be a *multi-port* packet, map sequences of coflows – streaming applications generate periodic coflows and batch jobs create queues of them – to point-to-point flows, and compare the entire network fabric of the cluster to a single link. Given this setting, we envision building upon, extending, and in fact, generalizing the concepts of network calculus [75, 76] and hierarchical fair service curves [188] to coflows over network fabrics.

8.3.5 Coflow-Aware Infrastructure Design

We have a made conscious effort in this dissertation to develop application-level solutions that do not rely on any support from underlying operating systems, hypervisors, or network components. This had two benefits. First, it allowed us to deploy and evaluate our solutions on the cloud. Second, it allowed us to iterate faster and try out a variety of solutions before settling for a good one. Given that we now have more confidence in the power of the coflow abstraction and the usefulness of coflow-based solutions, a natural direction is introducing pervasive support for coflows throughout the infrastructure stack.

The benefits of infrastructure-level support are not difficult to imagine. For example, operating systems, hypervisors, and middleboxes are the perfect places to implement automated coflow identification mechanisms, switch-level support for CoflowId can allow faster scheduling, distributed summing inside the fabric can enable decentralizing the schedulers, and in-network isolation can separate out control plane messages related to coflows to decrease coordination latencies. We envision a flurry of research activity in coflow-enabled infrastructure design in near future.

8.3.6 Analyses of Concurrent Open Shop Scheduling With Coupled Resources

One of the core contributions of this dissertation is identifying and characterizing the novel *concurrent open ship scheduling with coupled resources* problem. While we have focused on designing fast heuristics for practical deployments, understanding the theoretical properties of different variations of this problem remains relatively unexplored and provides a broad research agenda for the operations research and theory community. To this date, the only known theoretical result is due to Qiu et al., who have provided the first approximation algorithm for the offline clairvoyant coflow scheduling problem [175]. However, similar results do not exist for the non-clairvoyant or the fair variations. Furthermore, there exist no competitive ratio analyses of the online inter-coflow scheduling problems.

8.3.7 Generalization to Multiple Resource Types

Although we restrict ourselves only to network resources, nothing stops us, at least conceptually, from including compute, memory, and disks, if we consider end-to-end coflow scheduling. This is because, unlike the network, these resources do not pose matching constraints, and their bandwidths can be considered as *sequential* constraints before network-level senders and receivers in coflow scheduling formulations.

However, building such a system in practice can be non-trivial, simply because each resource has its own unique characteristics. In each machine, there are many parallel processors that communicate between themselves directly or indirectly, and there are many layers of memory and storage hierarchies with multiple parallel units. Coflows can appear within each resource type at very diverse time granularities, ranging from microseconds to seconds, requiring different approaches. Leveraging coflows at each level and stitching them all together to build a practical and balanced end-to-end solution remains an open problem.

8.3.8 Communication-Driven Application Design

Finally, we want to point out a completely opposite direction of research. This entire dissertation is built upon the assumption that application frameworks have certain communication characteristics, and we must try our best to optimize under application-imposed constraints. *Why don't we rearchitect applications with communication as the driving force?*

The time is especially ripe for communication-driven application design because we stand at the confluence of several hardware trends that all put communication at the center stage. First, *solid-state drives* strike a good balance between memory capacity and disk throughput, but a few of them can easily saturate even 10 GbE NICs. Second, *optical networks* are being proposed to achieve higher bisection bandwidth, but applications must exploit *wavelength locality* to fully utilize their broadcast and multicast capabilities. Third, technologies such as *RDMA* promise ultra-low latency in modern datacenters, but we must rethink our RPC and coordination mechanisms to exploit the latency improvements. Finally, *resource disaggregation* decouples the growth of compute and storage capacities, but it increases network utilization between consolidated resource banks. Given these trends, instead of retrofitting communication optimizations into current applications, we must consider designing new ones whose data ingestion, storage, and computation are all designed for better exploiting new hardware capabilities.

8.4 Final Remarks

In this dissertation, we have presented the coflow abstraction – the first, generalized solution that enables one to reason about and take advantage of multipoint-to-multipoint communication common in distributed data-parallel applications in the same way as traditional point-to-point communication observed in client-server applications. We cannot claim an abstraction to be the optimal one, but we have illustrated qualitatively and quantitatively that the expressiveness and power of coflows easily outperform that of its alternatives. More precisely, we have applied coflows to optimize the performance of individual communication patterns as well as to drastically improve performance and isolation in the presence of multiple, coexisting applications, with or without the complete knowledge of coflow characteristics. Although it is difficult to predict the exact course of any research area, the coflow abstraction has already opened the door to many new and exciting problems of practical importance and theoretical interest. We believe that coflows will have applications well beyond the confines of distributed data-parallel applications.

Appendix A

Coflows in Production

A.1 The Facebook Trace

The Facebook cluster consisted of about 3000 machines across 150 racks. Each machine had 16 cores and 48-to-64 GB RAM, and they were connected through a network with $10:1$ core-to-rack oversubscription ratio and a total bisection bandwidth of 300 Gbps.

Jobs were submitted using the Hive [8] query engine that composed a query in to a series of Hadoop jobs, which were then executed using the Hadoop MapReduce framework [6, 77]. Each MapReduce job consisted of two computation stages: map and reduce, and each stage had one or more parallel tasks. The number of map tasks (mappers) was determined by the number of partitions in the input file, and the number of reduce tasks (reducers) was predetermined by the users or the Hive query engine. Jobs were executed using a centralized, slot-based Fair Scheduler [207]. Finally, jobs used the Hadoop Distributed File System or HDFS [55] stored the data in the cluster, three-way replicated for reliability.

The trace consisted of about 790 thousand jobs that can broadly be divided into two categories: MapReduce jobs *with* the shuffle stage (320 thousand) and map-only or reduce-only jobs *without* any shuffle that were used for data ingestion and pre-processing [63]. The MapReduce jobs consisted of production jobs as well as experimental ones, with significant impact on productivity and revenue of Facebook.

Table A.1 summarizes the details.

A.2 Impact of Shuffle Coflows

The primary communication stage or coflow of MapReduce is known as the *shuffle* that transfers the output of each map task to the input of each reduce task, creating a many-to-many communication pattern. Note that 40.5% of the jobs in the Facebook trace involves a shuffle, and they contribute to 43% of the 10 PB expensive cross-rack traffic for the month-long period. Shuffle also create considerable hotspots in the network [63].

Date	Oct 2010
Duration	One month
Number of Machines	$3,000$
Number of Racks	150
Core:Rack Oversubscription	$10:1$
Framework	Hadoop [6]
Query Engine	Hive [8]
Cluster File System	HDFS [55]
File Block Size	256 MB

(a) Infrastructure and software

Total Jobs	$790,000$
Total Tasks	205 million
MapReduce Jobs	$320,000$
Tasks in MapReduce Jobs	150 million

(b) Jobs

Remote HDFS reads	17%
Shuffle	43%
HDFS replication	40%

(c) Sources of cross-rack traffic

Table A.1: Summary of Facebook traces.

The time complete an entire shuffle, i.e., the *shuffle completion time*, has a more direct – and often significant – impact on job completion time depending on what fraction of a job's end-to-end lifetime has been spent in shuffle. However, calculating the shuffle completion time of a job – consequently, the impact of shuffles on job completion times – is non-trivial.

Consider a MapReduce job J with M mappers, R reducers, and a shuffle with $M \times R$ flows. Determining the *shuffle finish time* is straightforward; it is simply when the last reducer has completed receiving its input from all the mappers.

$$t^{J}_{\text{s.f.t.}} = \max_{r} t^{r}_{\text{s.f.t.}}$$

where, $t^{J}_{\text{s.f.t.}}$ and $t^{r}_{\text{s.f.t.}}$ refer to the shuffle finish time of the entire job and a reducer r. On the contrary, one can use either of the following two approaches to determine the *shuffle start time*:

- **Conservative Approach:** when *all* the mappers have completed and *all* the reducers have started their shuffle.

  $$t^{J}_{\text{s.s.t. (conservative)}} = \max\left(\max_{m} t^{m}_{\text{t.f.t.}}, \max_{r} t^{r}_{\text{s.s.t.}}\right)$$

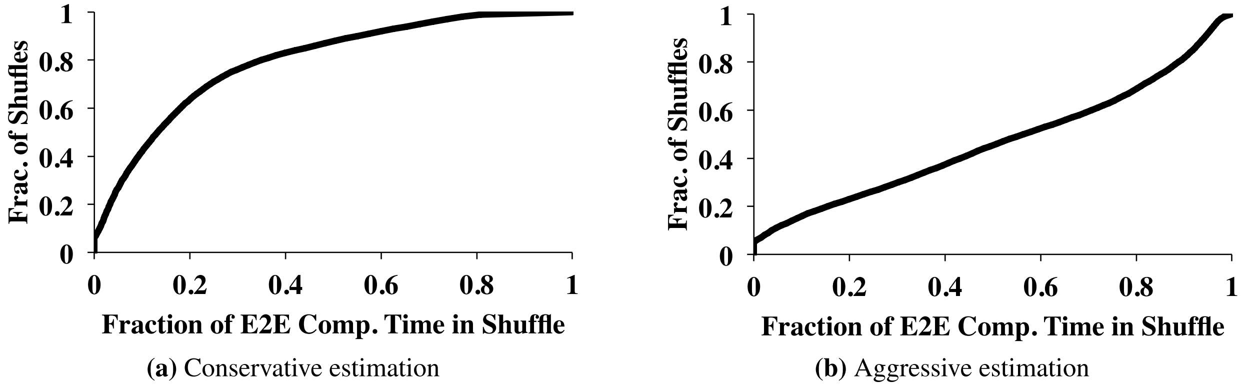
  where, $t^{J}_{\text{s.s.t.}}$ and $t^{r}_{\text{s.s.t.}}$ refer to the shuffle start time of the entire job and a reducer r, and $t^{m}_{\text{t.f.t.}}$ refers to the task finish time of a mapper m. This is a conservative estimate because reducers can also start fetching map outputs before all the mappers have finished.

- **Aggressive Approach:** when *at least one* reducer has started its shuffle.

  $$t^{J}_{\text{s.s.t. (aggressive)}} = \min_{r} t^{r}_{\text{s.s.t.}}$$

The shuffle completion time of a job is the difference between shuffle finish and start times.

$$t^{J}_{\text{s.c.t.}} = t^{J}_{\text{s.f.t.}} - t^{J}_{\text{s.s.t.}}$$



(a) Conservative estimation

(b) Aggressive estimation

Figure A.1: CDFs of the fraction of end-to-end job completion time spent in the shuffle stage in $320,000$ Facebook MapReduce jobs with reduce stages.

Figure A.1 presents two CDFs of the fraction of time spent in the shuffle stage (as defined above) for the entire trace. For the conservative estimation, 13% of jobs with shuffle stages spent more than 50% of their time in shuffle; on average, jobs spent 22% of their runtime in shuffle. For the aggressive estimation, 52% of jobs with shuffle stages spent more than 50% of their time in shuffle; on average, jobs spent 56% of their runtime in shuffle.

The actual impact lies somewhere in between for this particular trace, and it can significantly vary based on workload (e.g., filter-heavy vs. join-heavy). For example, in one particular week of the same trace, 26% of jobs with shuffle stages spent more than 50% of their time in shuffle; on average, jobs spend 33% of their runtime in shuffle [67].

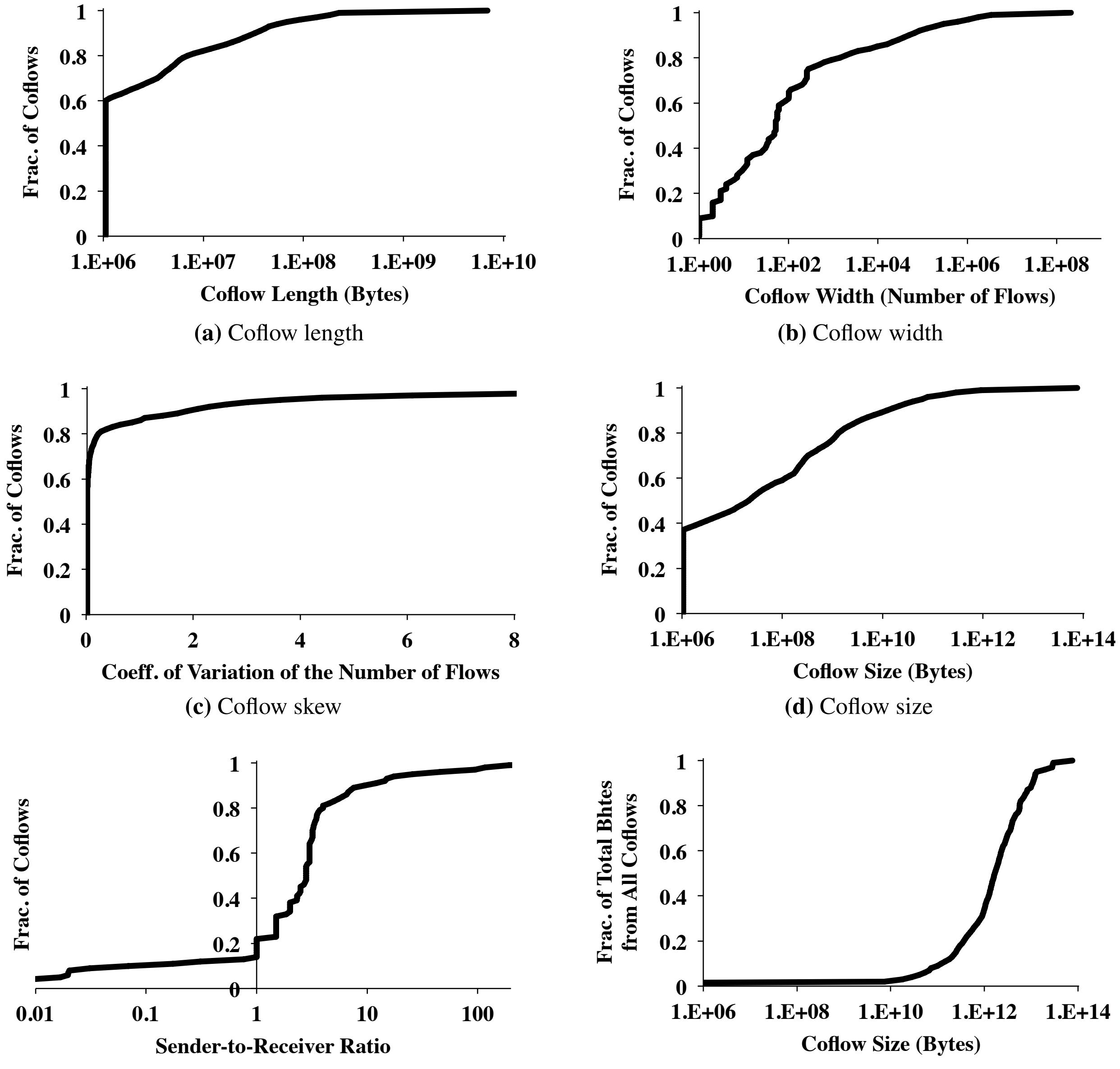
A.3 Coflow Characteristics

In this section, we focus on understanding the structural characteristics of coflows by highlighting two key attributes – wide variety in coflow structures and disproportionate footprint of few large coflows – that play crucial role in designing coflow scheduling algorithms.

A.3.1 Diversity of Coflow Structures

Because a coflow consists of multiple parallel flows, it cannot be characterized just by its size. Instead, we define four attributes of a coflow for better characterization:

1. **Length:** the size of its largest flow in bytes;
2. **Width:** the total number of flows in a coflow;
3. **Skew:** the coefficient of variation of flow sizes.
4. **Size:** the sum of flow sizes in bytes; and



(a) Coflow length

(b) Coflow width

(c) Coflow skew

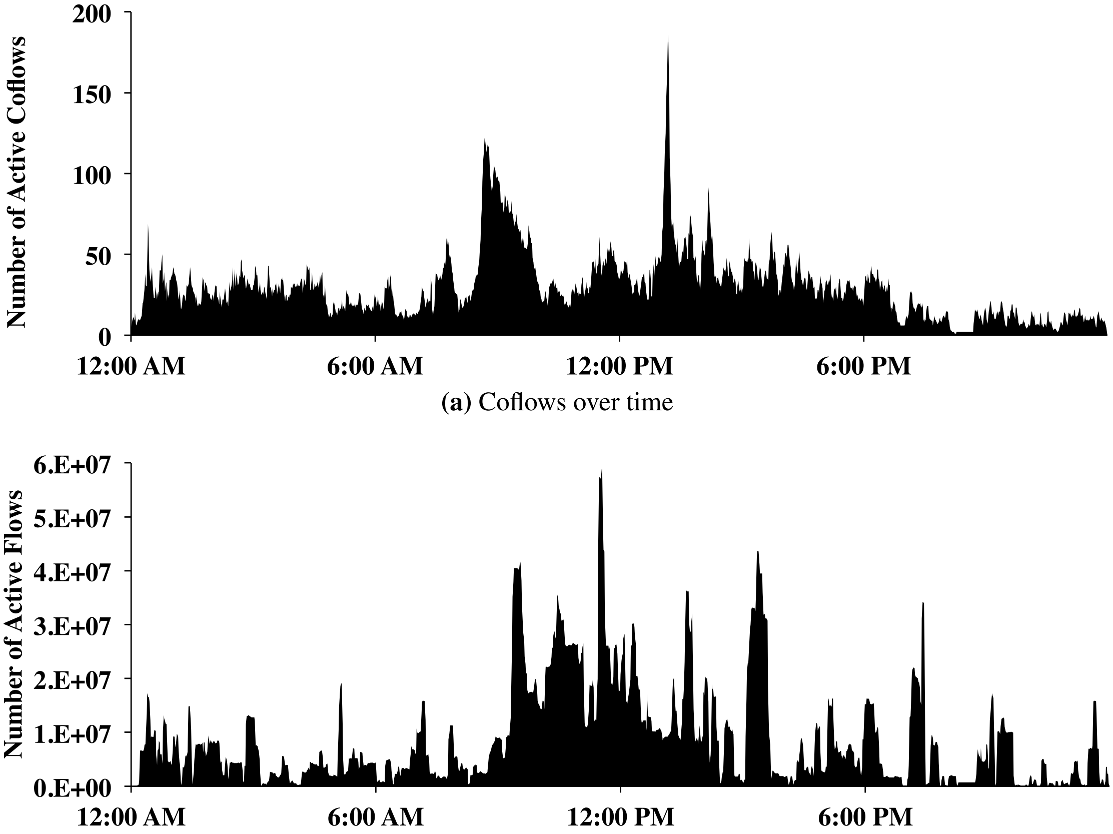
(d) Coflow size

(e) Coflow bottlenecks

(f) Coflow footprint

Figure A.2: Coflows in production vary widely in their (a) lengths, (b) widths, (c) skews of flow sizes, (d) total sizes, and (e) bottleneck locations. Moreover, (f) numerous small coflows contribute to a small fraction of network footprint.

The key takeaway from the CDF plots in Figure A.2 is that coflows vary widely in all four characteristics. We observe that while almost 60% coflows are short ($\leq$1 MB in length), flows in some coflows can be very large. Similarly, more than 50% coflows are narrow (with at most 50 flows), but they reside with coflows that consist of millions of flows. Furthermore, we see



(a) Coflows over time

(b) Flows over time

Figure A.3: Upper bounds on the numbers of concurrent coflows and flows throughout one day. Time is in GMT.

variations of flow sizes within the same coflow (Figure A.2c), which underpins the improvements from inter-coflow scheduling – the skew or slack in one coflow allows scheduling another coflow without impacting the performance of the prior one. Note that we rounded up flow sizes to 1 MB to calculate skew in Figure A.2c to ignore small variations.

Identifying bottlenecks and exploiting them is the key to improvements. Figure A.2e shows that the ratio of sender and receiver ports/machines[1] can be very different across coflows, and senders can be bottlenecks in almost a quarter of the coflows.

We found that length and width of a coflow have little correlation; a coflow can have many small flows as well as few large flows. However, as Figure A.2b and Figure A.2c suggest, width and skew are indeed correlated – as width increases, the probability of variations among flows increases as well. We also observed large variation in coflow size (Figure A.2d).

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

A.3.2 Heavy-Tailed Distribution of Coflow Size

Data-intensive jobs in production clusters are known to follow heavy-tailed distributions in terms of their number of tasks, size of input, and output size [39, 63]. We observe the same for coflow size as well. Figure A.2f presents the fraction of total coflows contributed by coflows of different size. Comparing it with Figure A.2d, we see that almost all traffic are generated by a handful of large coflows – 98% (99.6%) of the relevant bytes belong to only 8% (15%) of the coflows that are more than 10 GB (1 GB) in size.

A.4 Coflow Arrival Over Time and Concurrency

Coflows arrive over time as users submit new jobs, and inter-coflow scheduling is useful only in the presence of multiple coflows at the same time. As the number of coexisting coflows increases, opportunities for improving the average CCT increase as well.

Figure A.3 shows upper-bounds on the number of coflows (shuffles) and flows (generated from those shuffles) over time. There are upper-bounds because we count the total number of coflows (or flows) in the entire cluster, even though there can be groups of coflows (or flows) without any spatially overlap at all. Because of the diversity in coflow structures, we see that the number of coflows and flows are not necessarily correlated. Furthermore, throughout the trace we observed only a few hundred shuffles active at the same time.

Appendix B

Clairvoyant Inter-Coflow Scheduling

B.1 Problem Formulation and Complexity in the Offline Case

Each coflow $C(\mathbf{D})$ is a collection of flows over the datacenter backplane with P ingress and P egress ports, where the $P \times P$ matrix $\mathbf{D} = [d_{ij}]_{P \times P}$ represents the structure of C. For each non-zero element $d_{ij} \in \mathbf{D}$, a flow f_{ij} transfers d_{ij} amount of data from the ith ingress port (P_i^{in}) to the jth egress port (P_j^{out}) across the backplane at rate r_{ij}, which is determined by the scheduling algorithm. Assume that there are no other constraints besides physical limits such as port capacity.

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

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

$$\text{Subject to} \sum_{j',k} r_{ij'}^k(t) \leq 1 \qquad \forall t, i \tag{B.2}$$

$$\sum_{i',k} r_{i'j}^k(t) \leq 1 \qquad \forall t, j \tag{B.3}$$

$$\sum_{t=1}^{C_k} r_{ij}^k(t) \geq d_{ij}^k \qquad \forall i, j, k \tag{B.4}$$

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

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

in the offline case as follows.

$$\text{Maximize} \sum_{k=1}^{K} U_k \tag{B.5}$$

$$\text{Subject to inequalities (B.2), (B.3), and (B.4);}$$

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

The non-trivial nature of inequality (B.4) and the time-dependent formulation make these optimizations less amenable to standard approaches (e.g., relaxation and rounding). Indeed, optimizing either objective (O or Z) is NP-hard.

Theorem B.1.1 *Even under the assumptions of Section 5.5.1, optimizing* O *or* Z *in the offline case is NP-hard for all $P \geq 2$.*

Proof Sketch We reduce the NP-hard concurrent open shop scheduling problem [180] to the coflow scheduling problem. Consider a network fabric with only 2 ingress and egress ports ($P = 2$) and all links have the same capacity (without loss of generality, we can let this capacity be 1). Since there are only 2 ports, all coflows are of the form $C(\mathbf{D})$, where $\mathbf{D} = (d_{ij})_{i,j=1}^{2}$ is a 2×2 data matrix. Suppose that n coflows arrive at time 0, and let $\mathbf{D}^k = (d_{ij}^k)_{i,j=1}^{2}$ be the matrix of the kth coflow. Moreover, assume for all k, $d_{ij}^k = 0$ if $i = j$. In other words, every coflow only consists of 2 flows, one sending data from ingress port P_1^{in} to egress port P_2^{out}, and the other sending from ingress port P_2^{in} to egress port P_1^{out}.

Consider now an equivalent concurrent open shop scheduling problem with 2 identical machines (hence the same capacity). Suppose n jobs arrive at time 0, and the kth job has d_{12}^k amount of work for machine 1 and d_{21}^k for machine 2. Since this is NP-hard [180], the coflow scheduling problem described above is NP-hard as well. ■

Remark B.1.2 Given the close relationship between concurrent open shop scheduling and coflow scheduling, it is natural to expect that techniques to express concurrent open shop scheduling as a mixed-integer program and using standard LP relaxation techniques to derive approximation algorithms [149, 180] would readily extend to our case. However, they do not, because the coupled constraints (B.2) and (B.3) make permutation schedules sub-optimal (**Theorem B.3.1**).

B.2 Tradeoffs in Optimizing CCT

With Work Conservation Consider Figure B.1a. Coflows C_1 and C_2 arrive at time 0 with one and two flows, respectively. Each flow transfers unit data. C_3 arrives one time unit later and uses a single flow to send 0.5 data unit. Figure B.1b shows the work-conserving solution, which finishes in 2 time units for an average CCT of 1.67 time units. The optimal solution (Figure B.1c), however, takes 2.5 time units for the *same* amount of data (i.e., it lowers utilization); still, it has a 1.11× lower average CCT (1.5 time units).

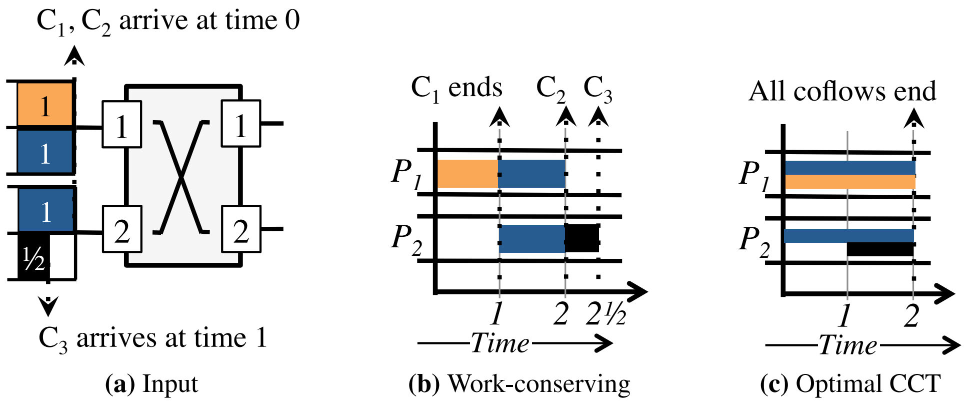


Figure B.1: Allocation of *ingress* port capacities (vertical axis) for the coflows in (a) on a 2 × 2 datacenter fabric for (b) a work-conserving and (c) a CCT-optimized schedule. Although the former schedule is work-conserving and achieves higher utilization, the latter has a lower average CCT.

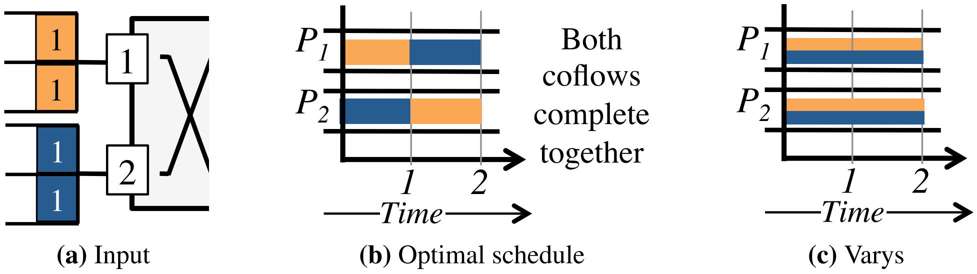


Figure B.2: Flow-interleaved allocation of *egress* port capacities (vertical axis) for the coflows in (a) for CCT-optimality (b).

With Avoiding Starvation The tradeoff between minimum completion time and starvation is well-known for flows (tasks) on individual links (machines) – longer flows starve if a continuous stream of short flows keep arriving. Without preemption, however, short flows would suffer head-of-line blocking, and the average FCT will increase. The same tradeoff holds for coflows, because the datacenter fabric and coflows generalize links and flows, respectively.

B.3 Ordering Properties of Coflow Schedules

Theorem B.3.1 *Permutation schedule is not optimal for minimizing the average CCT.*

Proof Sketch Both permutation schedules – C_1 before C_2 and C_2 before C_1 – would be suboptimal for the example in Figure B.2a. ■

Remark B.3.2 In Varys, SEBF would schedule C_1 before C_2 (arbitrarily breaking the tie); iterative MADD will allocate the minimum bandwidth to quickly finish C_1, and then give the remaining bandwidth to C_2 (Figure B.2c). The average CCT will be the same as the optimal for this example.

Appendix C

Non-Clairvoyant Inter-Coflow Scheduling

C.1 Coflow Scheduling with Local Knowledge

Theorem C.1.1 *Any coflow scheduling algorithm where schedulers do not coordinate, has a worst-case approximation ratio of $\Omega(\sqrt{n})$ for n concurrent coflows.*

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

For each input and output port, consider one coflow with just one flow that starts from that input port or is destined for that output port; i.e., for all coflows $C_k, k \in [1, m]$, let $d^k_{k,m-k+1} = 1$ and $d^k_{i,j} = 0$ for all $i \neq k$ and $j \neq m - k + 1$. Next, consider the rest of the coflows to have exactly k flows that engage all input and output ports of the fabric; i.e., for all coflows $C_k, k \in [m+1, n]$, let $d^k_{i,m-i+1} = 1$ for all $i \in [1, m]$ and $d^k_{l,j} = 0$ for all $l \neq i$ and $j \neq m - i + 1$. We have constructed an instance of *distributed order scheduling*, where n orders must be scheduled on m facilities [155]. The proof follows from [155, Theorem 5.1 on page 3]. ■

C.2 Continuous vs. Discretized Prioritization

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

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

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

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

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

Note that the above holds only when a coflow's size accurately predicts it's completion time, which might not always be the case [68, §5.3.2]. Deriving a closed-form expression for the general case remains an open problem.

Appendix D

Fair Inter-Coflow Scheduling

D.1 Dual Objectives for Fair Inter-Coflow Scheduling

The two conflicting requirements of fair coflow scheduling can be defined as follows.

1. Utilization: $\sum_{i \in [1,2P]} \sum_{k \in [1,M]} c_k^i$
2. Isolation guarantee: $\min_{k \in [1,M]} \mathcal{P}_k$

Given the tradeoff between the two, one can consider one of the two possible optimizations:

O1 Ensure highest utilization, *then* maximize the isolation guarantee with best effort;

O2 Ensure optimal isolation guarantee, *then* maximize utilization with best effort.[1]

O1: Utilization-First In this case, the optimization attempts to maximize the isolation guarantee across all coflows while keeping the highest network utilization.

$$\begin{aligned} \text{Max} \quad & \min_{k \in [1,M]} \mathcal{P}_k \\ \text{s.t.} \quad & \sum_{i \in [1,2P]} \sum_{k \in [1,M]} c_k^i \in \arg\max \sum_{i \in [1,2P]} \sum_{k \in [1,M]} c_k^i \end{aligned} \tag{D.1}$$

Although this ensures maximum network utilization, isolation guarantee to individual coflows can be arbitrarily low. This formulation can still be useful in private datacenters [105].

[1] Maximizing a combination of these two is also an interesting future direction.

To ensure some isolation guarantee, existing cloud network sharing approaches [46, 124, 140, 171, 172, 179, 184, 202] use a similar formulation:

$$\begin{aligned} \text{Maximize} \quad & \sum_{1 \le i \le 2P} \sum_{k \in [1,M]} c_k^i \\ \text{subject to} \quad & \mathcal{P}_k \ge \frac{1}{M},\ k \in [1, M] \end{aligned} \tag{D.2}$$

The objective here is to maximize utilization while ensuring at least $\frac{1}{M}$th of each link to C_k. However, this approach has two drawbacks: suboptimal isolation guarantee and lower utilization (§ 7.4):

O2: Isolation-Guarantee-First Instead, we have formulated the problem as follows:

$$\begin{aligned} \text{Maximize} \quad & \sum_{i \in [1,2P]} \sum_{k \in [1,M]} c_k^i \\ \text{subject to} \quad & \min_{k \in [1,M]} \mathcal{P}_k = \mathcal{P}_k^* \\ & c_k^i \ge a_k^i,\ i \in [1, 2P], k \in [1, M] \end{aligned} \tag{D.3}$$

Here, we maximize resource consumption while keeping the optimal isolation guarantee across all coflows, denoted by $\mathcal{P}_k^*$. Meanwhile, the constraint on consumption being at least guaranteed minimum allocation ensures strategy-proofness; thus, ensuring that guaranteed allocated resources will be utilized.

Because c_k^i values have no upper bounds except for physical capacity constraints, optimization **O2** may result in suboptimal isolation guarantee in non-cooperative environments (§7.4.3). HUG introduces the following additional constraint to avoid this issue only in non-cooperative environments:

$$c_k^i \le \mathcal{P}^*,\ i \in [1, 2P], k \in [1, M]$$

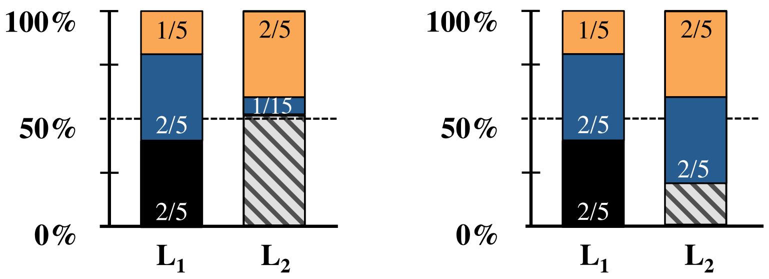
This constraint is not necessary when strategy-proofness is a non-requirement – e.g., in private datacenters.

D.2 Work Conservation VS. Strategy-proofness Tradeoff

We demonstrate the tradeoff between work conservation and strategy-proofness (thus isolation guarantee) by extending our running example from Section 7.3.

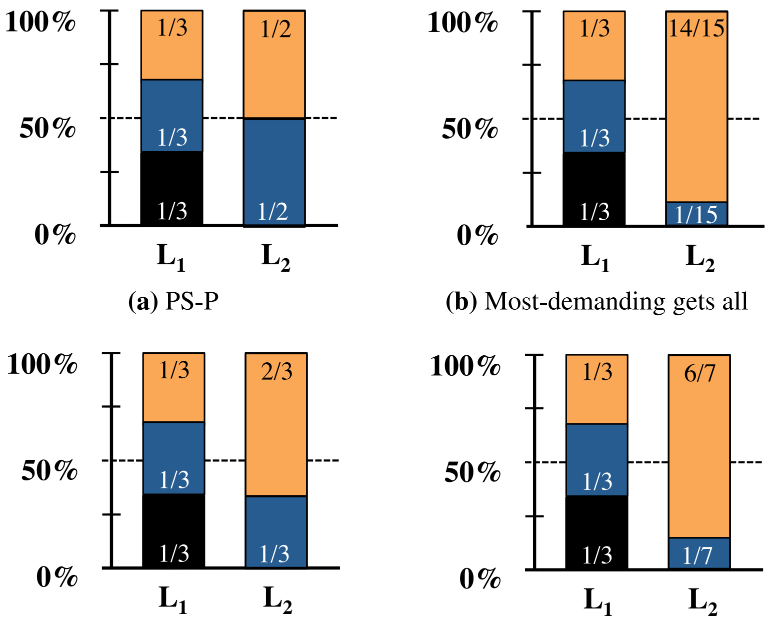
Consider another coflow (C_C) with correlation vector $\overrightarrow{d_C} = \langle 1, 0 \rangle$ in addition to the two coflows present earlier. The key distinction between C_C and either of the earlier two is that it does not demand any bandwidth on link-2. Given the three correlation vectors, we can use DRF to calculate the optimal isolation guarantee (Figure D.1a), where C_k has $\mathcal{P}_k = \frac{2}{5}$, link-1 is completely utilized, and $\frac{7}{15}$th of link-2 is proportionally divided between C_A and C_B.

This leaves us with two questions:



(a) DRF **(b)** HUG

Figure D.1: Hard tradeoff between work conservation and strategy-proofness. Adding one more coflow (C_C in black) to Figure 7.4 with correlation vector $\langle 1, 0 \rangle$ makes simultaneously achieving work conservation and optimal isolation guarantee impossible, even when all three coflows have elastic demands.



(a) PS-P **(b)** Most-demanding gets all

(c) Equal division **(d)** Proportional division

Figure D.2: Allocations after applying different work-conserving policies to divide spare bandwidth in link-2 for the example in Figure D.1.

1. *How do we completely allocate the remaining $\frac{8}{15}$th bandwidth of link-2?*
2. *Is it even possible without sacrificing optimal isolation guarantee and strategy-proofness?*

We show in the following that it is indeed not possible to allocate more than $\frac{4}{5}$th of link-2 (Figure D.1b) without sacrificing the optimal isolation guarantee.

Let us consider three primary categories of work conservation policies: *demand-agnostic*, *unfair*, and *locally fair*. All three will result in lower isolation guarantee, lower utilization, or both.

D.2.1 Demand-Agnostic Policies

Demand-agnostic policies equally divide the resource between the number of coflows independently in each link, irrespective of coflow demands, and provide isolation. Although strategyproof, this allocation (Figure D.2a) has lower isolation guarantee ($\mathcal{P}_A = \frac{1}{2}$ and $\mathcal{P}_B = \mathcal{P}_C = \frac{1}{3}$, therefore isolation guarantee is $\frac{1}{3}$) than the optimal isolation guarantee allocation shown in Figure D.1a ($\mathcal{P}_A = \mathcal{P}_B = \mathcal{P}_C = \frac{2}{5}$, therefore isolation guarantee is $\frac{2}{5}$). PS-P [124,171,172] fall in this category.

Worse, when coflows do not have elastic-demand applications, demand-agnostic policies are not even work-conserving (similar to the example in §7.4.4).

Lemma D.2.1 *When coflows do not have elastic demands, per-resource equal sharing is not work-conserving.*

Proof Sketch Only $\frac{11}{12}$th of link-1 and $\frac{5}{9}$th of link-2 will be consumed; i.e., none of the links will be saturated! ■

To be work-conserving, PS-P suggests dividing spare resources based on whoever wants it.

Lemma D.2.2 *When coflows do not have elastic demands, PS-P is not work-conserving.*

Proof Sketch If C_B gives up its spare allocation in link-2, C_A can increase its progress to $\mathcal{P}_A = \frac{2}{3}$ and saturate link-1; however, C_B and C_C will remain at $\mathcal{P}_B = \mathcal{P}_C = \frac{1}{3}$. If C_A gives up its spare allocation in link-1, C_B and C_C can increase their progress to $\mathcal{P}_B = \mathcal{P}_C = \frac{3}{8}$ and saturate link-1, but C_A will remain at $\mathcal{P}_A = \frac{1}{2}$. Because both C_A and C_B have chances of increasing their progress, both will hold off to their allocations even with useless traffic – another instance of Prisoner's dilemma. ■

D.2.2 Unfair Policies

Instead of demand-agnostic policies, one can also consider simpler, unfair policies; e.g., allocating all the resources to the coflow with the least or the most demand.

Lemma D.2.3 *Allocating spare resource to the coflow with the least demand can result in zero spare allocation.*

Proof Sketch Although this strategy provides the optimal allocation for Figure 7.4, when at least one coflow in a link has zero demand, it can trivially result in no additional utilization; e.g., C_C in Figure D.1. ■

Lemma D.2.4 *Allocating spare resource to the coflow with the least demand is not strategyproof.*

Proof Sketch Consider C_A lied and changed its correlation vector to $\overrightarrow{d'_A} = \langle 1, \frac{1}{10} \rangle$. The new optimal isolation guarantee allocation for unchanged C_B and C_C correlation vectors would be: $\overrightarrow{a_A} = \langle \frac{1}{3}, \frac{1}{30} \rangle$, $\overrightarrow{a_B} = \langle \frac{1}{3}, \frac{1}{15} \rangle$, and $\overrightarrow{a_C} = \langle \frac{1}{3}, 0 \rangle$. Now the spare resource in link-2 will be allocated to C_A because it asked for the least amount, and its final allocation would be $\overrightarrow{a'_A} = \langle \frac{1}{3}, \frac{14}{15} \rangle$. As a result, its progress improved from $\mathcal{P}_A = \frac{2}{5}$ to $\mathcal{P}'_A = \frac{2}{3}$, while the others' decreased to $\mathcal{P}_B = \mathcal{P}_C = \frac{1}{3}$. ■

Corollary D.2.5 (of Lemma D.2.4) *In presence of work conservation, coflows can lie both by increasing and decreasing their demands, or a combination of both.*

Lemma D.2.6 *Allocating spare resource to the coflow with the highest demand is not strategyproof.*

Proof Sketch If C_A changes its demand vector to $\overrightarrow{d'_A} = \langle 1, 1 \rangle$, the eventual allocation (Figure D.2b) will again result in lower progress ($\mathcal{P}_B = \mathcal{P}_C = \frac{1}{3}$). Because C_B is still receiving more than $\frac{1}{6}$th of its allocation in link-1 in link-2, it does not need to lie. ■

Corollary D.2.7 (of Lemmas D.2.4, D.2.6) *Allocating spare resource randomly to coflows is not strategyproof.*

D.2.3 Locally Fair Policies

Finally, one can also consider equally or proportionally dividing the spare resource on link-2 between C_A and C_B. Unfortunately, these strategies are not strategyproof either.

Lemma D.2.8 *Allocating spare resource equally to coflows is not strategyproof.*

Proof Sketch If the remaining $\frac{8}{15}$th of link-2 is equally divided, the share of C_A will increase to $\frac{2}{3}$-rd and incentivize its to lie. Again, the isolation guarantee will be smaller (Figure D.2c). ■

Lemma D.2.9 *Allocating spare resource proportionally to coflows' demands is not strategyproof.*

Proof Sketch If one divides the spare in proportion to coflow demands, the allocation is different (Figure D.2d) than equal division; yet, C_A can still increase its progress at the expense of others.■

D.3 Properties of HUG

Lemma D.3.1 *Any work conservation policy that assigns spare resources such that the allocation of a coflow in any link including the spare can be more than its allocated resource on the bottleneck link based on its reported correlation vector is not strategyproof.*

Proof Sketch (of Lemma D.3.1) Consider C_A from the example in Figure 7.4. Assume that instead of reporting its true correlation vector $\overrightarrow{d_A} = \langle \frac{1}{2}, 1 \rangle$, it reports $\overrightarrow{d'_A} = \langle \frac{1}{2} + \epsilon, 1 \rangle$, where $\epsilon > 0$. As a result, its allocation will change to $\overrightarrow{a'_A} = \langle \frac{1/2+\epsilon}{3/2+\epsilon}, \frac{1}{3/2+\epsilon} \rangle$. Its allocation in link-1 $\left(\frac{1/2+\epsilon}{3/2+\epsilon}\right)$ is already larger than before $\left(\frac{1}{3}\right)$. If the work conservation policy allocates the spare resource in link-2 by δ (δ may be small but a positive value), its progress will change to $\mathcal{P}'_A = \min\left(\frac{a'^1_A}{d^1_A}, \frac{a'^2_A}{d^2_A}\right) = \min\left(\frac{1+2\epsilon}{3/2+\epsilon}, \frac{1}{3/2+\epsilon} + \delta\right)$. As long as $\epsilon < \frac{3/2\delta}{2/3-\delta}$ (if $\delta \geq \frac{2}{3}$, we have no constraint on ϵ), its progress will be better than when it was telling the truth, which makes the policy not strategyproof. The operator cannot prevent this because it knows neither a coflow's true correlation vector nor ϵ, the extent of the coflow's lie. ■

Corollary D.3.2 (of Lemma D.3.1) *Optimal isolation guarantee allocations cannot always be work-conserving even in the presence of elastic-demand applications.*

Theorem D.3.3 *Algorithm 6 is strategyproof.*

Proof Sketch (of Theorem D.3.3) Because DRF is strategyproof, the first stage of Algorithm 6 is strategyproof as well. We show that adding the second stage does not violate strategy-proofness of the combination.

Assume that link-b is a system bottleneck – the link DRF saturated to maximize isolation guarantee in the first stage. Meaning, $b = \arg\max\limits_{i} \sum\limits_{k=1}^{M} d^i_k$. We use $D^b = \sum\limits_{k=1}^{M} d^b_k$ to denote the total demand in link-b ($D^b \geq 1$), and $\mathcal{P}^b_k = 1/D^b$ for corresponding progress for all C_k ($k \in \{1, ..., M\}$) when link-b is the system bottleneck. In Figure 7.4, $b = 1$. The following arguments hold even for multiple bottlenecks.

Any C_k can attempt to increase its progress ($\mathcal{P}_k$) only by lying about its correlation vector ($\overrightarrow{d_k}$). Formally, its action space consists of all possible correlation vectors. It includes increasing and/or decreasing demands of individual resources to report a different vector, $\overrightarrow{d'_k}$ and obtain a new progress, $\mathcal{P}'_k (> \mathcal{P}_k)$. C_k can attempt one of the two alternatives when reporting $\overrightarrow{d'_k}$: either keep link-b still the system bottleneck or change it. We show that Algorithm 6 is strategyproof in both cases; i.e., $\mathcal{P}'_k \leq \mathcal{P}_k$.

Case 1: link-b is still the system bottleneck.

Its progress cannot improve because

- if $d'^b_k \leq d^b_k$, its share on the system bottleneck will decrease in the first stage; so will its progress. There is no spare resource to allocate in link-b.

For example, if C_A changes $d'^1_A = \frac{1}{4}$ instead of $d^1_A = \frac{1}{2}$ in Figure 7.4, its allocation will decrease to $\frac{1}{5}$th of link-1; hence, $\mathcal{P}'_A = \frac{2}{5}$ instead of $\mathcal{P}_A = \frac{2}{3}$.

- if $d'^b_k > d^b_k$, its share on the system bottleneck will increase. However, because $D'^b > D^b$ as $d'^b_k > d^b_k$, everyone's progress including its own will decrease in the first stage ($\mathcal{P}'^b_k \leq \mathcal{P}^b_k$). The second stage will ensure that its maximum consumption in any link-i $c_k^{\prime i} \leq \max_j \left\{a_k^{\prime j}\right\}$. Therefore its progress will be smaller than that when it tells the truth ($\mathcal{P}_k^{\prime b} < \mathcal{P}_k^b$).

For example, if C_A changes $d_A^{\prime 1} = 1$ instead of $d_A^1 = \frac{1}{2}$ in Figure 7.4, its allocation will increase to $\frac{1}{2}$ of link-1. However, progress of both coflows will decrease: $\mathcal{P}_A = \mathcal{P}_B = \frac{1}{2}$. The second stage will restrict its usage in link-2 to $\frac{1}{2}$ as well; hence, $\mathcal{P}_A' = \frac{1}{2}$ instead of $\mathcal{P}_A = \frac{2}{3}$.

Case 2: link-b is no longer a system bottleneck; instead, link-b' ($\neq b$) is now one of the system bottlenecks.

We need to consider the following two sub-cases.

- If $D^{\prime b'} \leq D^b$, the progress in the first stage will increase; i.e., $\mathcal{P}_k^{\prime b'} \geq \mathcal{P}_k^b$. However, C_k's allocation in link-b will be no larger than if it had told the truth, making its progress no better. To see this, consider the allocations of all other coflows in link-b before and after it lies. Denote by c_{-k}^b and $c_{-k}^{\prime b}$ the resource consumption of all other coflows in link-b when C_k was telling the truth and lying, respectively. We also have $c_{-k}^b = a_{-k}^b$ and $a_{-k}^b + a_k^b = 1$ because link-b was the bottleneck, and there was no spare resource to allocate for this link. When C_k lies, $a_{-k}^{\prime b} \geq a_{-k}^b$ because $\mathcal{P}_k^{\prime b'} \geq \mathcal{P}_k^b$. We also have $c_{-k}^{\prime b} \geq a_{-k}^{\prime b}$ and $c_{-k}^{\prime b} + c_k^{\prime b} \leq 1$. This implies $c_k^{\prime b} \leq 1 - c_{-k}^{\prime b} \leq 1 - a_{-k}^{\prime b} \leq 1 - a_{-k}^b = a_k^b = c_k^b$. Meaning, C_k's progress is no larger than that when it was telling the truth.

- If $D^{\prime b'} > D^b$, everyone's progress including its own decreases in the first stage ($\mathcal{P}_k^{\prime b'} < \mathcal{P}_k^b$). Similar to the second scenario in Case 1, the second stage will restrict C_k to the lowered progress.

Regardless of C_k's approaches – keeping the same system bottleneck or not – its progress using Algorithm 6 will not increase. ■

Corollary D.3.4 (of Theorem D.3.3) *Algorithm 6 maximizes isolation guarantee, i.e., the minimum progress across coflows.*

Theorem D.3.5 *Algorithm 6 achieves the highest resource utilization among all strategyproof algorithms that provide optimal isolation guarantee among coflows.*

Proof Sketch (of Theorem D.3.5) Follows from Lemma D.3.1 and Theorem D.3.3. ■

Lemma D.3.6 *Under some cases, DRF may have utilization arbitrarily close to 0, and HUG helps improve the utilization to 1.*

Proof Sketch (of Lemma D.3.6) Construct the cases with K links and N coflows, and each coflow has demand 1 on link-1 and ϵ on other links.

DRF will allocate to each coflow $\frac{1}{N}$ on link-1 and $\frac{\epsilon}{N}$ on all other links, resulting in a total utilization of $\frac{1+(K-1)\epsilon}{K} \to 0$ when $K \to \infty, \epsilon \to 0$ for any N.

HUG will allocate to each coflow $\frac{1}{N}$ on every link and achieve 100% utilization. ■

Bibliography

[1] Akka. http://akka.io.

[2] Amazon CloudWatch. http://aws.amazon.com/cloudwatch.

[3] Amazon EC2. http://aws.amazon.com/ec2.

[4] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

[5] Apache Giraph. http://giraph.apache.org.

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

[7] Apache Hama. http://hama.apache.org.

[8] Apache Hive. http://hive.apache.org.

[9] Apache Tez. http://tez.apache.org.

[10] BitTornado. http://www.bittornado.com.

[11] Databricks Cloud. http://databricks.com/cloud.

[12] DETERlab. http://www.isi.deterlab.net.

[13] Fragment Replicate Join – Pig wiki. http://wiki.apache.org/pig/PigFRJoin.

[14] Google Cloud Dataflow. https://cloud.google.com/dataflow.

[15] Google Compute Engine. https://cloud.google.com/compute.

[16] Google Container Engine. http://kubernetes.io.

[17] Impala performance update: Now reaching DBMS-class speed. http://blog.cloudera.com/blog/2014/01/impala-performance-dbms-class-speed.

[18] Kryo serialization library. https://github.com/EsotericSoftware/kryo.

[19] LANTorrent. http://www.nimbusproject.org.

[20] Microsoft Azure. `http://azure.microsoft.com`.

[21] Murder. `http://github.com/lg/murder`.

[22] Presto. `https://prestodb.io`.

[23] Storm: Distributed and fault-tolerant realtime computation. `http://storm-project.net`.

[24] The Berkeley Data Analytics Stack (BDAS). `https://amplab.cs.berkeley.edu/software/`.

[25] TPC Benchmark DS (TPC-DS). `http://www.tpc.org/tpcds`.

[26] TPC-DS kit for Impala. `https://github.com/cloudera/impala-tpcds-kit`.

[27] Trident: Stateful stream processing on Storm. `http://storm.apache.org/documentation/Trident-tutorial.html`.

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[29] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. In *SIGCOMM*, 2010.

[30] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing data parallel computing. In *NSDI*, 2012.

[31] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[32] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at Internet scale. *VLDB*, 2013.

[33] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[34] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.

[35] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[36] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI*, 2012.

[37] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. Mckeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*, 2013.

[38] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, 2011.

[39] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[40] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[41] S. Angel, H. Ballani, T. Karagiannis, G. OShea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *OSDI*, 2014.

[42] J. Arjona Aroca and A. Fernández Anta. Bisection (band)width of product networks with application to data centers. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):570–580, 2014.

[43] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[44] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Scheduling: The multi-level feedback queue. In *Operating Systems: Three Easy Pieces*. 2014.

[45] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI*, 2015.

[46] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[47] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. OShea. Chatty tenants and the cloud network sharing problem. In *USENIX NSDI*, pages 171–184, 2013.

[48] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[49] J. Bennett and H. Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. In *INFOCOM*, pages 120–128, 1996.

[50] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[51] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

[52] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian. Traffic classification on the fly. *SIGCOMM CCR*, 36(2):23–26, 2006.

[53] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SoCC*, 2013.

[54] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[55] D. Borthakur. The Hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007.

[56] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.

[57] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *SOSP*, 2003.

[58] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.

[59] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[60] K. Chen, C. Hu, X. Zhang, K. Zheng, Y. Chen, and A. V. Vasilakos. Survey on routing in data centers: Insights and future directions. *IEEE Network*, 25(4):6–10, 2011.

[61] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen. OSA: An optical switching architecture for data center networks with unprecedented flexibility. In *NSDI*, 2012.

[62] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN*, 2009.

[63] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*, 2013.

[64] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[65] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *HotNets*, 2012.

[66] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[67] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[68] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[69] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[70] E. G. Coffman and L. Kleinrock. Feedback queueing models for time-shared systems. *Journal of the ACM*, 15(4):549–576, 1968.

[71] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, 2003.

[72] B. Cohen. The BitTorrent protocol specification, 2008.

[73] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In *NSDI*, 2010.

[74] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Spring Joint Computer Conference*, pages 335–344, 1962.

[75] R. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[76] R. Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[77] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[78] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[79] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *ASPLOS*, 2014.

[80] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[81] C. Diot, W. Dabbous, and J. Crowcroft. Multipoint communication: A survey of protocols, functions, and mechanisms. *IEEE Journal on Selected Areas in Communications*, 15(3):277–290, 1997.

[82] F. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *SIGCOMM*, 2014.

[83] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial. No justified complaints: On fair sharing of multiple resources. In *ITCS*, 2012.

[84] B. Donnet, B. Gueye, and M. A. Kaafar. A Survey on Network Coordinates Systems, Design, and Security. *IEEE Communication Surveys and Tutorials*, 12(4), Oct. 2010.

[85] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[86] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.

[87] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, 2007.

[88] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM*, 2011.

[89] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, 2013.

[90] M. M. Flood. Some experimental games. *Management Science*, 5(1):5–26, 1958.

[91] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.

[92] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[93] C. Fraley and A. Raftery. MCLUST Version 3 for R: Normal mixture modeling and model-based clustering. Technical Report 504, Department of Statistics, University of Washington, Sept. 2006.

[94] P. Ganesan and M. Seshadri. On cooperative content distribution and the price of barter. In *ICDCS*, 2005.

[95] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[96] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. *SIGCOMM*, 2012.

[97] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[98] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, 2011.

[99] C. Gkantsidis, T. Karagiannis, and M. VojnoviC. Planet scale software updates. In *SIGCOMM*, 2006.

[100] S. J. Golestani. Network delay analysis of a class of fair queueing algorithms. *IEEE JSAC*, 13(6):1057–1070, 1995.

[101] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[102] P. Goyal, H. M. Vin, and H. Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *SIGCOMM*, 1996.

[103] R. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[104] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[105] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[106] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[107] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM CCR*, 35:41–54, 2005.

[108] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, 2009.

[109] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *CoNEXT*, 2010.

[110] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[111] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *OSDI*, 2012.

[112] A. Gutman and N. Nisan. Fair allocation without trade. In *AAMAS*, 2012.

[113] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In *SIGCOMM*, 2010.

[114] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *SIGCOMM*, 2011.

[115] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer. FireFly: A reconfigurable wireless data center fabric using free-space optics. In *SIGCOMM*, 2014.

[116] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2009.

[117] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

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

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

[120] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[121] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[122] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *SIGCOMM*, 2015.

[123] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message completion time in the cloud. In *SIGCOMM*, 2015.

[124] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical network performance isolation at the edge. In *NSDI*, 2013.

[125] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multiresource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.

[126] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, 2008.

[127] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *SIGCOMM*, 2005.

[128] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

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

[130] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. In *SIGCOMM*, 2005.

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

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

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

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

[135] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[136] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, 2013.

[137] J. B. Kruskal and M. Wish. Multidimensional Scaling. *Sage University Paper series on Quantitative Applications in the Social Sciences*, 07-001, 1978.

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

[139] K. LaCurts, J. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *HotCloud*, 2014.

[140] V. T. Lam, S. Radhakrishnan, A. Vahdat, G. Varghese, and R. Pan. NetShare and Stochastic NetShare: Predictable bandwidth allocation for data centers. *SIGCOMM CCR*, 42(3):5–11, 2012.

[141] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *SIGCOMM*, 2014.

[142] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, Nov. 1999.

[143] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 100(10):892–901, 1985.

[144] J. Y.-T. Leung, H. Li, and M. Pinedo. Order scheduling in an environment with dedicated resources in parallel. *Journal of Scheduling*, 8(5):355–386, 2005.

[145] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC*, 2014.

[146] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[147] 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 *SIGMOD*, 2010.

[148] Y. Mao and L. K. Saul. Modeling Distances in Large-Scale Networks by Matrix Factorization. In *IMC*, 2004.

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

[150] A. McGregor, M. Hall, P. Lorier, and J. Brunskill. Flow clustering using machine learning techniques. In *PAM*. 2004.

[151] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switch. *IEEE Transactions on Communications*, 47(8):1260 – 1267, 1999.

[152] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[153] J. C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *SIGCOMM CCR*, 42(5):44–48, 2012.

[154] A. W. Moore and D. Zuev. Internet traffic classification using Bayesian analysis techniques. 33(1):50–60, 2005.

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

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

[157] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataow system. In *SOSP*, 2013.

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

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

[160] J. F. Nash Jr. The bargaining problem. *Econometrica: Journal of the Econometric Society*, pages 155–162, 1950.

[161] T. T. Nguyen and G. Armitage. A survey of techniques for Internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

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

[163] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[164] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[165] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM ToN*, 1(3):344–357, 1993.

[166] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *EC*, 2012.

[167] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM*, 2014.

[168] R. Peterson and E. G. Sirer. Antfarm: Efficient content distribution with managed swarms. In *NSDI*, 2009.

[169] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *HotNets 2009*.

[170] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.

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

[172] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *SIGCOMM*, 2013.

[173] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating microsecond circuit switching into the data center. In *SIGCOMM*, 2013.

[174] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

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

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

[177] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *SIGCOMM*, 2011.

[178] M. P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[179] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *USENIX WIOV*, 2011.

[180] T. A. Roemer. A note on the complexity of the concurrent open shop problem. *Journal of Scheduling*, 9(4):389–396, 2006.

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

[182] C. Scaffidi. Why are APIs difficult to learn and use? *Crossroads*, 12(4):4–4, 2006.

[183] S. Shenker, D. D. Clark, and L. Zhang. A scheduling service model and a scheduling architecture for an integrated services packet network. Technical report, Xerox PARC, 1993.

[184] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Sharing the data center network. In *NSDI*, 2011.

[185] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM*, 1995.

[186] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. SIGCOMM, 2015.

[187] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.

[188] I. Stoica, H. Zhang, and T. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *SIGCOMM*, 1997.

[189] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy*, 2011.

[190] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *NSDI*, 2006.

[191] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[192] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter TCP (D2TCP). In *SIGCOMM*, 2012.

[193] H. R. Varian. Equity, envy, and efficiency. *Journal of economic theory*, 9(1):63–91, 1974.

[194] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM*, 2009.

[195] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[196] A. Vulimiri, C. Curino, B. Godfrey, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[197] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*, 1994.

[198] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *SoCC*, 2012.

[199] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time optics in data centers. In *SIGCOMM*, 2011.

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

[201] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in datacenter networks. In *CoNEXT*, 2010.

[202] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *SIGCOMM*, 2012.

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

[204] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4D network control plane. In *NSDI*, 2007.

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

[206] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[207] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

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

[209] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

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

[211] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM*, 2012.

[212] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI*, 2012.

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

[214] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. In *SIGCOMM*, 2012.

[215] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *AAIM*, 2008.