Automating Network Resource Allocation for Coflows with Deadlines

by

Asif Hasnain

Submitted to the
Electrical Engineering, Computer Science, and Mathematics
in partial fulfillment of the requirements for the degree of
Doctor rerum naturalium (Dr. rer. nat.)
at the
Paderborn University
September 2021

Author .....

Asif Hasnain
Electrical Engineering, Computer Science, and Mathematics

Thesis Supervisor .....

Prof. Dr. Holger Karl
Electrical Engineering, Computer Science, and Mathematics

Second Supervisor .....

Prof. Dr. Friedhelm Meyer auf der Heide
Electrical Engineering, Computer Science, and Mathematics

Abstract

The coflow abstraction is used for specifying network resource requirements of data-parallel applications. It represents correlated flows in data flow models like MapReduce and partition-aggregate. In this dissertation, I mainly demonstrate that hand-crafted online coflow schedulers — to allocate data rates to correlated flows — can be replaced with a reinforcement learning (RL) based coflow scheduler to automate network resource allocation of coflows for data-parallel applications. Specifically, an RL-based coflow scheduler learns scheduling policies to optimize for a high-level performance objective of coflows, for example, maximize coflow admissions while meeting their deadlines.

In this dissertation, I have made three main contributions: I first present a new coflow heuristic that leverages released network resources of active coflows finishing before the deadline of a new coflow request. I then show that flow or coflow heuristics, in general, are prone to under-perform in maximizing (co)flow admissions because of stochastic flow or coflow arrivals in data traffic. Next, I demonstrate that an online flow scheduler can learn a scheduling policy to maximize flow admissions using reinforcement learning. Finally, I show how a coflow scheduler can learn policies in the presence of stochastic coflow arrivals to maximize coflow admissions while meeting their deadlines.

Kurzfassung

Die Coflow-Abstraktion wird zur Spezifikation der Netzressourcenanforderungen von datenparallelen Anwendungen verwendet. Sie repräsentiert korrelierte Flüsse in Datenflussmodellen wie MapReduce oder Partition-Aggregate. In dieser Dissertation zeige ich, dass manuell entworfene Online-Coflow-Scheduler zur Zuweisung von Datenraten an solche Flüsse durch einen auf Reinforcement Learning (RL) basierenden Coflow-Scheduler ersetzt werden können. Dies automatisiert die Zuweisung von Netzressourcen an Coflows über den Entwurf eines Verfahrens hinaus. Konkret lernt ein RL-basierter Coflow-Scheduler Planungsrichtlinien, um ein Leistungsziel von Coflows zu optimieren, z. B. die Maximierung der Coflow-Zulassungen bei gleichzeitiger Einhaltung ihrer Fristen.

In dieser Dissertation habe ich drei Hauptbeiträge geleistet: Zunächst stelle ich eine neue Coflow-Heuristik vor, die freigegebene Netzressourcen von Coflows nutzt, die vor Fristablauf einer anderen Coflow-Anforderung enden und diese an andere, ggf. neue Coflows zuweist. Dann zeige ich, dass Flow- oder Coflow-Heuristiken im Allgemeinen dazu neigen, bei der Maximierung der (Co)Flow-Zulassungen schlecht abzuschneiden, wenn im Datenverkehr stochastische Flows oder Coflows ankommen. Als Nächstes zeige ich, dass ein Online-Flow-Scheduler mit Hilfe von Reinforcement Learning eine Scheduling-Politik zur Maximierung der Flow-Zulassungen erlernen kann. Schließlich zeige ich, wie ein Coflow-Scheduler bei stochastischen Coflow-Ankünften Strategien erlernen kann, um die Coflow-Zulassungen zu maximieren und gleichzeitig ihre Fristen einhalten.

Acknowledgements

I would like to thank my advisor Prof. Dr. Holger Karl first for giving me the opportunity for which I will always remain indebted to you. This dissertation would not be possible without your advice, faith, and continuous support throughout my research work. I thoroughly enjoyed and learnt from our discussions on different research ideas.

I am profoundly grateful to Prof. Dr. Friedhelm Meyer auf der Heide for providing feedback on my thesis. I also want to thank my supportive colleagues at Paderborn University: Musa, Haitham, and Stefan. I am especially thankful to Musa and Haitham for having discussions and providing me with feedback on my papers. Likewise, I am grateful to the German Research Foundation (DFG) for supporting my research through the Collaborative Research Center “On-The-Fly Computing" (SFB 901).

Most importanly, I would like to express my gratitude towards for my parents, Asia and Ghulam, my wife, Annam, my children, Ibrahim and Fajr, my sisters and brothers-in-law for their continuous support and love. I am especially indebted to my parents for their unconditional love and motivation: thank you for believing in me.

Previously Published Papers

Chapter 3 revises the published paper [37]: A. Hasnain and H. Karl. Coflow scheduling with performance guarantees for data center applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 850–856, 2020

Chapter 4 revises the published paper [39]: Asif Hasnain and Holger Karl. Learning flow scheduling. In *2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–6, 2021

Chapter 5 revises the published paper [38]: Asif Hasnain and Holger Karl. Learning coflow admissions. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data-parallel applications run large-scale data processing on cluster of machines in a compute center, where huge volumes of data are generated and stored. Since multiple machines are involved in parallel data processing, common tasks of an application trigger different data flows between groups of machines; for example, shuffle in MapReduce [25, 24, 1] or the partition-aggregate model [19, 5], as shown in Fig. 1.1 and Fig. 1.2, respectively. In addition, data-parallel applications have data flows without explicit barriers (for example, in Dryad [44]); data flows with cycles (for example, in Spark [111]); and Bulk Synchronous Parallel data flow models (for example, in Pregel [66]) that do create barriers.

In this dissertation, I mainly consider the partition-aggregate model because the model is usually associated with the performance objective to *meet strict deadlines*, which is quite important in time-sensitive, interactive applications [119, 118, 18, 34, 47, 77, 120, 6, 23, 28, 96, 26] offered by companies like Amazon and Google. The model as such is used, for example, in a backend query of a search engine or news feeds of a social network. Specifically, the query is partitioned into multiple sub-queries for processing data on different worker machines and their result (from workers) is timely aggregated back to a frontend server. The aggregation task here creates multiple correlated flows between groups of machines.

However, despite rapid innovations [32, 14, 39] by network reseachers to minimize flow completion times based on the flow abstraction, the goal of optimizing individual flows differs from that of jointly optimizating *correlated* flows of data-parallel applications because data-parallel applications only care about the completion of *correlated* flows in one or more

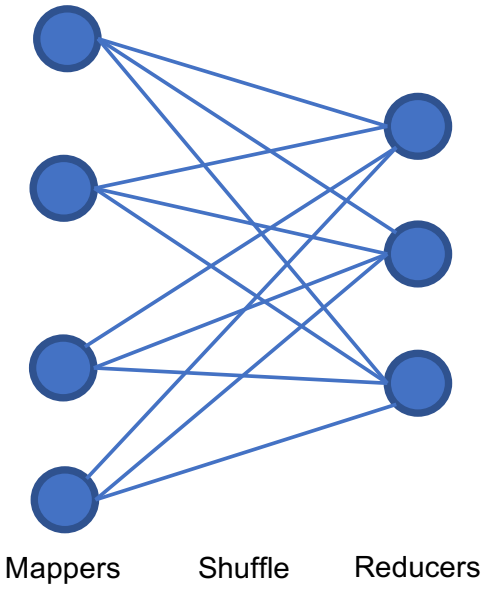

Figure 1.1: MapReduce [25] data model

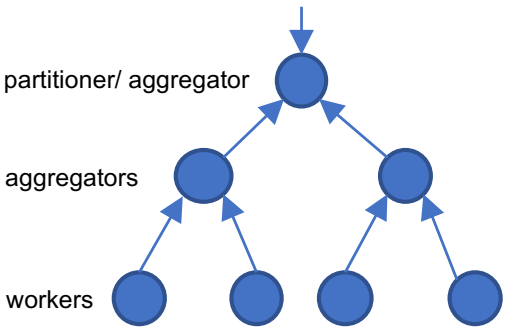

Figure 1.2: Partition-aggregate [19] data model

communication stages [86, 92]. Therefore, data-parallel applications should be able to specify such diverse network resource requirements to networks. In literature, one way to explicitly specify such resource requirements is the coflow network abstraction [19], which enables application-aware network resource allocation.

1.1 Coflows as an Abstraction in Networks

A coflow [19] represents correlated flows in data models between groups of machines during different computation stages [56, 86, 92]. A *communication stage is considered complete once all flows within a coflow have finished their data transimission.* Since the network performance of data-parallel applications depends on completion of all flows within a coflow, these flows are optimized together for a common performance objective. For example, the coflow scheduler — to allocate network resources like data rate to correlated flows — *minimizes average coflow completion times (CCTs)* for shuffle in MapReduce [25, 24, 1] or *meets strict coflow deadlines* in the partition-aggregate model [19, 5]. In this dissertation, I present different approaches to optimize coflow scheduling for *maximizing coflow admissions while meeting their deadlines*.

1.2 Identified Problems and Contribution

This thesis primarily explores the use of reinforcement learning (RL) to *automate network resource management for coflows* of data-parallel applications. My approaches leverage the coflow abstraction to optimize a common performace objective of correlated flows. In this thesis, I mainly present a new coflow heuristic and RL-based flow and coflow schedulers — to learn desired scheduling policies such that they make informed scheduling decisions.

Specifically, I design a new coflow heuristic that leverages released resources after completion of active coflows before the deadline of a new coflow request and show that it admits more coflows while meeting their deadlines.

However, coflow heuristics, in general, face different challenges (Section 2.3); for example, they are susceptible to stochastic coflow arrivals, that is, they admit less coflows for the desired performance objective to *maximize coflow admissions while meeting their deadlines*. Therefore, I further design RL-based flow and coflow schedulers to learn scheduling policies from arrival patterns in data traffic and show that they admit more (co)flows while meeting their deadlines in the presence of shochastic data traffic.

One of key advantages of the RL-based coflow scheduler, namely learning coflow scheduling (LCS) in Chapter 5, is that it does not require hand-crafted features as input and it automatically learns coflow scheduling policies without human instruction beyond a high-level specification of a performance objective. In addition, LCS scheduler has a flexible design that enables us to reformulate the reward function and retrain the scheduling agent for a different performance objective. For example, LCS can learn to minimize average CCT instead of maximizing coflow admissions. Even though in this thesis I focus on only one performance objective, our RL-based (co)flow scheduling techniques are applicable to different performance objectives as well as information-agnostic coflow schedulers.

In the next Chapter 2, I will elaborate the background, related work, and technical challenges of coflow scheduling in detail. In the following paragraphs, I highlight the research contributions in chronological order of their publication. The following Chapters 3, 4, and 5 are based on these published papers and contain verbatim content from them. Even though I am the main author of these papers, I use terms like “we" and “our" in these chapters of the thesis to indicate that the presented results are joint work with my colleagues. In Chapter 6, I then conclude my thesis and describe future research directions in network resource allocation.

Heuristic to Meet Coflow Deadlines: A. Hasnain and H. Karl. Coflow scheduling with performance guarantees for data center applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 850–856, 2020
This thesis first presents a new heuristic to maximize coflow admissions while meeting their deadlines. The primary challenge for the heuristic is to identify key features in network data traffic for higher coflow admissions. To overcome this challenge, our key insight is to leverage released resources

on completion of active coflows before the deadline of a new coflow request. This insight enables us to admit more coflows. In Chapter 3, we demonstrate a motivating example and further elaborate on both the heuristic and the online linear program. In addition, we show that these algorithms admit more coflows than the competing heuristic Varys [22] through large-scale trace-driven simulation on production traces.

Learning Flow Scheduling: Asif Hasnain and Holger Karl. Learning flow scheduling. In *2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–6, 2021

This dissertation then demonstrates the usage of deep reinforcement learning (DRL) to learn network scheduling policies for diverse workloads. In Chapter 4, we present a new flow scheduler, namely learning flow scheduling (LFS), which learns to make decisions. The key challenge in designing LFS is that stochastic flow arrivals make it difficult for the flow scheduling agent to optimize the desired performance objective. In our proposed approach, the LFS scheduler leverages a new training algorithm to perform end-to-end training on different network states and learns a flow scheduling policy through the formulated reward function. We further show in Chapter 4 that the LFS scheduler admits more coflows than greedy heuristics under varying network load.

Learning Coflow Admissions: Asif Hasnain and Holger Karl. Learning coflow admissions. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2021

Finally, this dissertation demonstrates that RL-based flow scheduling (which is discussed in Chapter 4) can be generalized to the big-switch network model with coflows. Specifically, we propose a new coflow scheduler, namely learning coflow admissions (LCS), to *maximize coflow admissions while meeting their deadlines* in Chapter 5. The key challenges in designing LCS are formulation of a reward function and learning scheduling policies on stochastic data traffic. To overcome the first challenge, we formulate the reward function on key features of network states, that is, CCTs and number of active coflows. The second challenge is addressed by using a separate baseline [101] for every set of coflow arrival sequences to reduce variance in policy gradient from learning on different coflow arrival sequences. Chapter 5 further elaborates the reward function and the training algorithm. It also shows that LCS learns a reasonable scheduling policy to admit higher number of coflows than the Varys heuristic [22] while meeting their deadlines.

Chapter 2

Background and Related Work

The last two decades have seen rapid growth of cluster computating systems (for instance, Hadoop [25, 1], Spark [112], and Storm [93]) to process massive amount of data generated by data-parallel applications. These systems often exhibit different communication patterns (for example, MapReduce [25, 1] and partition-aggregate [5, 19]), which are depicted by the coflow abstraction [19, 21] to optimize for common performance objectives of correlated flows.

In literature, the coflow abstraction has been readily used for improving network performance of data-parallel applications. Specifically, multiple coflow heuristics [109, 59, 117, 108, 98, 45, 63, 116, 16, 22, 21] are proposed to optimize for different performance objectives of coflows. However, these coflow heuristics face diverse technical challenges (Section 2.3), which degrades their overall performance.

The research work presented in this dissertation builds on the past literature and addresses these technical challenges, which are further elaborated in the remainder of the chapter. Specifically, the past literature on coflow scheduling for different performance objectives is presented in Section 2.1. In Section 2.2, I then discuss Markov decision process, reinforcement learning, and their usage in *automating network resource allocation of (co)flows* for data-parallel applications. In last Section 2.3, I conclude the chapter with the discussion on new technical challenges for coflow scheduling.

2.1 Performance Objectives for Coflow Schedulers

Minimize Average Coflow Completion Times: This performance objective is common for efficient resource allocation of throughput-sensitive applications [25, 44, 24], which need to finish data transmission as soon as possible. In literature, some papers [17, 3, 84, 57] proposed optimization algorithms to minimize average CCT while other papers [59, 117, 98, 45, 116, 16, 22, 21] proposed, more practical, heuristics to improve network performance of data-parallel applications. For example, Sincronia [3] optimizes for average CCT by employing the primal-dual method while the Stretch [17] algorithm achieved 2-approximation. In heuristics, Varys [22] employed shortest-effective-bottleneck-first (SEBF) heuristic to efficiently reduce the average CCT. Later, RAPIER [116] considered coflow routing and scheduling simulataneously to minimize the average CCT. Recently, Swallow [117] compresses network traffic to further reduce average CCT than Varys [22]. In addition, Aalo [16] minimized average CCT without prior knowledge of coflows. Specifically, it separated coflows into multiple priority queues and demoted them from the highest priority queue to lower priority queues when coflows sent more data than the pre-defined threshold queue values. However, Aalo [16] allocates excessive resources to non-bottleneck flows, which leads to wastage of resources and impacts average CCT. Fai [59] addressed this problem by identifying the bottleneck flow in a coflow and limiting data rate of non-bottleneck flows to the data rate of the bottleneck flow.

Meet Coflow Deadlines: In this performance objective, a coflow completion is only useful when all of its constituent flows have finished their data transmission within its deadlines. This performance objective is desired for predictable coflow completions for data models like partition-aggregate of time-sensitive applications [119, 118, 34, 47, 77, 120, 6, 23, 96, 26]. Varys [22] was the first coflow scheduler to maximize coflow admissions while meeting their deadlines and guaranteed coflow completions. Soon afterwards, Chronos [63] combined priority scheduling with limited multiplexing to meet coflow deadlines while mixCoflow [109] reduced footprint of network resources used by deadline coflows. In contrast, our heuristic [37] (in Chapter 3) utilizes released resources on CCTs of active coflows before the deadline of a new coflow request and successfully admits more coflows

than Varys [22].

However, coflow heuristics are suspectible to stochastic coflow arrivals (Section 2.3) in different network states, which make it difficult for handcrafted heuristics to make optimal scheduling decisions. Therefore, in this dissertation, we solve the coflow scheduling problem using reinforcement learning [91, 51] in which a coflow scheduler directly learns coflow scheduling policies for the specified, high-level performance objective. In the next section, we first formulate the coflow scheduling problem as a Markov decision process (MDP) and then discuss the proposed approaches [15, 61, 90, 97, 89, 14] to solve the problem using reinforcement learning.

2.2 Machine Learning

2.2.1 Markov Decision Process

Coflow scheduling is a sequential decision process in which scheduling decisions (that is, actions) impact not just the immediate reward from the network environment but also the future network states and their rewards. Therefore, the problem is formulated as a discrete-time Markov decision process (MDP). In MDP, the coflow scheduling agent directly interacts with the network environment at discrete timesteps t_e, where t_e is the time of the coflow scheduling event $e \in [1, \cdots, E]$ (see notation in Table 2.1 and Table 5.1). At each timestep t_e, the coflow scheduling agent takes an action a_e in network state s_e. For every action a_e, the network environment produces a reward R_e and transitions to the next network state s_{e+1}. The successive coflow scheduling decisions form a *sequence* or *trajectory* $j \in [1, \cdots, N]$ of state s_e^j, action a_e^j, and reward R_e^j, where N is the total number of trajectories. Specifically, in this dissertation, we assume that the state transition in network environment satisfies the *Markov* property, that is, the current network state s_e depends only on the last, preceding state s_{e-1} and the action a_{e-1} (taken in state s_{e-1}).

Table 2.1: Notation for coflow and system model

t_e	Time of the coflow scheduling event $e \in [1, \cdots, E]$
C	Set of arrived, online coflows
$\mathbb{C}$	Set of admitted coflows $\mathbb{C} \subseteq C$, which comprises active and completed coflows
$\bar{\mathbb{C}}_e$	Set of active coflows $\bar{\mathbb{C}}_e \subseteq \mathbb{C}$ at time t_e, not including a new coflow arriving at t_e
$\lvert\bar{\mathbb{C}}_e\rvert \leq u$	The number of concurrent, active coflows $\lvert\bar{\mathbb{C}}_e\rvert$ are limited to a maximum value u
F_c	Set of flows within a coflow $c \in C$
α_c	Arrival time of a coflow $c \in C$
d_c	Relative deadline of a coflow $c \in C$
$\alpha_c + d_c$	Absolute deadline of a coflow $c \in C$
l_f	A new flow f request on link l (the event e causing that new flow will be clear from context)
$f_c^{i,j}$	A flow in F_c from source i to destination j
v_f	Total data volume of flow f
$\bar{v}_f$	Remaining data volume of flow f
r_f	Assigned data rate to flow f
Γ_f	Flow completion time (FCT) of completed flow f
Γ_c	Coflow completion time (CCT) of completed coflow c
M	Set of servers
K_m	Capacity of link that connects server $m \in M$ to the non-blocking switch

Time-Average Reward: Since coflow scheduling is a continuous task in which the coflow scheduling agent has to make decisions forever without termination, we consider the *time-average reward* $\bar{R}_e$ in MDP, that is, there is no discounting — because both immediate and delayed rewards are important to optimize the time-average reward [91]. Therefore, the network environment produces a *differential* reward $R_e - \bar{R}_e$ for every action a_e in network state s_e.
Differential Return: In average-reward setting, the objective of a coflow scheduling agent is to maximize the differential return G_e over time, that is, sum of *differential* rewards from state s_e onwards, formally, $G_e = R_e - \bar{R}_e + R_{e+1} - \bar{R}_{e+1} + \cdots$ or $\lim_{\tau \to \infty} 1/\tau \sum_{e=0}^{\tau} (R_e - \bar{R}_e)$ [91], where τ is the finite training episode length.

2.2.2 Reinforcement Learning

To solve the formulated MDP (of the coflow scheduling problem), we employ reinforcement learning (RL). In RL, an agent learns to take actions through experience of interacting with the environment. Specifically, it learns a *policy* $\pi_\theta(a_e|s_e)$ to optimize a performance objective $J(\theta)$, for example, optimize variety of tasks in robotics [29, 83, 82, 58, 55], playing games [73], control tasks [35, 42], or driving autonomous systems [54]. RL is different from unsupervised learning because it optimizes a performance objective (for example, maximize rewards through its actions) while unsupervised learning finds hidden structures in an unlabeled dataset. Importantly, it maximizes rewards by *exploring* new actions and *exploiting* prior experience from interaction with the network environment.

Like other applications, the coflow scheduling agent learns a *policy* through end-to-end RL training on different network states without manual feature engineering. Specifically, the coflow scheduling policy learns to map network states to actions, for example, to admit or reject a coflow request. Formally, the policy $\pi_\theta(a_e|s_e)$ is defined as the probability distribution over actions. Since coflow scheduling is a continuous task with a large number of network states, storing these states in memory is computationally expensive. Therefore, RL is combined with function approximators like

neural network [7, 36] to *generalize* learning from prior scheduling experience. Importantly, in this dissertation, we have developed new RL training algorithms to successfully learn coflow scheduling policies.

2.2.3 Reinforcement Learning for Networking

Reinforcement learning (RL) has been applied to various aspects of networking, for example, congestion control [49, 64, 76, 113, 106], traffic engineering [110], routing [107, 60, 95], scaling and placement of virtual network functions (VNFs) [80, 81]. However, in this section, we only focus on RL-based proposals for flow and coflow scheduling [15, 61, 90, 97, 89, 14].

RL for Flow scheduling: Although there are many non-RL proposals [43, 96, 105] for flow scheduling to meet their deadlines, we consider only RL proposals (for flow scheduling) here. Recently, few papers [32, 31, 14, 98, 13] apply RL to maximize flow admissions, for example, [32] and [31] apply DRL to calculate data rates of active coflows such that they finish within deadline. However, the policy network outputs discrete data rates, which can impede convergance of the scheduling policy in a large network. In addition, it uses Q-learning [100], which is memory-intensive and not scalable for the continuous flow scheduling task. In contrast, our LFS [39] *maximizes* successful flow admissions using a modified Monte-Carlo actor critic algorithm [104], in which the policy network is based on a scalable neural network and it outputs a discrete action $a_e \in \{0, 1\}$ to either admit $a_e = 1$ or reject $a_e = 0$ the flow request f in state s_e (Chapter 4). Unlike LFS [39], AuTO [14] learns a flow scheduling policy using DRL to minimize average FCT in a datacenter.
However, unlike coflows, these flow schedulers lack knowledge of the correlated resource requirements in network, which can degrade performance of data-parallel applications.
RL for Coflow Scheduling: In literature, DeepAalo [97] applies DRL to coflow scheduling without prior knowledge of coflows; specifically, they automatically adjust threshold of priority queues to minimize average CCT. In addition, M-DRL [15] also learns demotion threshold values for multiple priority queues per port. However, both schedulers use FIFO within priority queues to avoid starvation of coflows but they miss the opportunity to

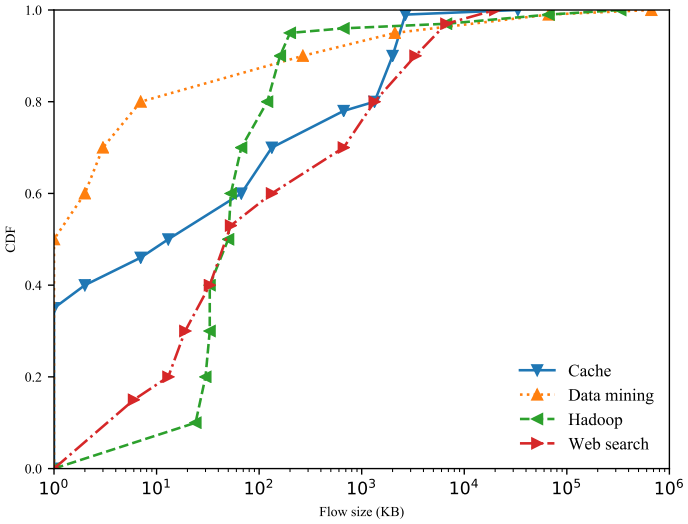

Figure 2.1: Flow Size Distributions

further reduce average CCT by first scheduling narrow coflows — which have flows on a few ports — in the highest priority queue [45]. In contrast, we employ DRL in LCS [38] for a different performance objective, that is, *maximize* coflow admissions while *meeting* their deadlines. DeepWeave [90] accelerates job scheduling using RL-based coflow scheduling, however, it assume that the dependency information between coflows of multi-stage jobs — which are usually depicted by directed-acyclic graph (DAG) — is known beforehand. In contrast, we only consider single-stage, single-wave coflows in our network model [38].

The research work presented in this dissertation builds on this literature and desire for automating network resource allocation for time-sensitive data-parallel applications. However, achieving this performance objective (that is, *maximize coflow admissions while meeting their deadlines to optimize predictable coflow completion times*) raises new challenges (Section 2.3), which are addressed in this thesis.

2.3 Practical Challenges

Prior Knowledge: Online inter-coflow scheduling for predictable communication times is NP-hard [22] even if we have prior information about coflows, that is, their arrival times, set of constituent flows, and their relative deadlines are known. Therefore, many coflow heuristics [109, 63, 22] are proposed for guaranteeing timely completion of successfully admitted coflow. However, state-of-the-art Varys [22] heuristic — to meet coflow deadlines — does not forsee and use released resources on CCTs of active coflows before the deadline of a new coflow request. Therefore, in Chapter 3 of this dissertation, we propose a new heuristic NH [37], which admits more coflows than Varys [22] while still meeting their deadlines.

Stochastic Network Traffic: The impact of stochastic input process is quite common in robotics control [29, 83, 82, 58, 55], manufacturing systems [99, 65], autonomous driving [54], and queueing systems [69, 68, 70, 67]. Similarly, stochastic input process in network (that is, flow or coflow arrivals in different network states) poses the second challenge for network resource allocation. Specifically, the stochastic (co)flow arrivals make it difficult for hand-crafted heuristics to make optimal scheduling decisions. For example, the coflow heuristics Varys [22] or our NH [37] (as described in Chapter 3) — to maximize successful coflow deadlines — are suspectible to stochastic coflow arrivals, which usually cause head-of-line blocking for coflows, that is, a reasonably large, admitted coflow (by either size, width, length, deadline, or completion time) blocks admission of upcoming smaller coflows. Thus, it is important for coflow schedulers to learn policies from stochastic coflow arrivals. Therefore, in this dissertation, we introduce deep reinforcement learning (DRL)-based flow and coflow schedulers in Chapter 4 and Chapter 5, respectively. These schedulers enable us to learn policies and make informed scheduling decisions in the presence of stochastic (co)flow arrivals in network traffic.

Unknown Flow Size Distribution: The flow size of different production workloads [78, 5, 33] is usually drawn from an unknown distribution, as shown in Fig. 2.1. As a result, coflow heuristics [37, 22] find it difficult to achieve the desired performance objective, that is, maximize coflow admissions while meeting their deadlines.

Impact of Preemption: A preemptive scheduler can avoid head-of-line

blocking of coflows to minimize average CCT [22] but, in the worst case, it can starve certain coflows under higher network loads. Therefore, in this dissertation, we learn from coflow arrival patterns to reject a large coflow (by any criteria) for smaller coflows — to avoid head-of-line blocking of coflows — and once a coflow is admitted, it is never preempted for its guaranteed completion within the deadline.

Varying Network Load: Since the network load may increase over time in production networks, coflow heuristics are usually not able to adapt to higher network loads [37]. Therefore, in this dissertation, we analyse the impact of varying network loads and show promising results of our heuristic [37] as well as DRL-based schedulers [38, 39].

Diverse Data Models: Since data-parallel applications have diverse resource requirements from the network, a *universal* coflow scheduler (similar to the UPS [72, 88]), is a desired solution to emulate any coflow scheduling algorithm. Although there is no *universal* coflow scheduler yet, in Chapter 5 of the dissertation, we argue that the LCS [38] design is flexible enough to meet different performance objectives of coflow scheduling algorithms. For instance, the reward function to *maximize successful coflow admissions* can be reformulated to, for example, *minimize average CCT* and then the coflow scheduler can be retrained on the reformulated reward function — for different resource requirements of a data model — to learn a new scheduling policy.

Formulating the Reward Function: For a given performance objective, the formation of a reward function is quite challenging because a naive solution to reward (or penalize) the coflow scheduling agent, for example, +1 and -1 on a coflow admission or rejection, respectively, can easily be impacted by admission of a coflow with large completion time (or coflow size, width, length, or deadline). In contrast, in this dissertation, we show that reward functions based on key features of network states, for example, CCTs and/or number of active coflows (in a network state), perform better than the naive solution.

Since stochastic network traffic adds noise to rewards [70, 69], it impedes effective training of a coflow scheduler to converge to a reasonable policy. For example, let us assume that the coflow scheduling agent receives three rewards 90, 100, and 107 for an action in each of the three different network states. At first glance, it is difficult to quantify from rewards that whether those actions are good or bad. One way [101, 67] to reduce the noise in

the reward is to subtract the expected, average reward in a network state. In this example, the differential rewards (Section 2.2) -9, 1, and 8 — after subtracting the time-average reward 99 — have lower noise. In addition, input-dependent baseline [70] effectively reduces the noise in rewards from stochastic network traffic.

Chapter 3

Coflow Scheduling for time-sensitive applications

This chapter of the thesis is based on the revised text of our paper [37]. However, some portions of the chapter contain verbatim content, for example, figures and tables used in that very paper:

A. Hasnain and H. Karl. Coflow scheduling with performance guarantees for data center applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 850–856, 2020

In this chapter, we show shortcomings of the Varys [22] heuristic to meet coflow deadlines. Specifically, we propose a new heuristic and solve an online linear program for better coflow scheduling that admits more coflows while meeting their deadlines.

My major contribution in this paper was to design, implement, and evaluate the performance of both the new heuristic and the online linear program. Firstly, I sketched a counterexample to admit more coflows for time-sensitive applications that have to respond within certain deadlines. I then implemented the coflow scheduling algorithms and evaluated their performance through the trace-driven flow simulator. In addition, I compared results of both algorithms with the state-of-the-art Varys heuristic on different performance metrics, for example, percentage of successful coflow admissions.

The remainder of the chapter is organized as follows. It will first introduce the problem in Section 3.1 and then describe the network model before giving a motivating example in Section 3.2. In Section 3.3, we briefly talk about the shortcomings of existing approaches. After that, the problem

of admission control and rate allocation is formulated as an online linear programming problem in Section 3.4; the proposed solution finds *minimum* required data rates to complete data transmission of coflows within their deadlines. In addition, the proposed heuristic is described in Section 3.5. Both these scheduling algorithms are non-work-conserving and lead to unused resources. Therefore, the unused resources are allocated by a post-processing algorithm to ensure work-conservation (Section 3.6). The algorithms are evaluated through trace-driven simulation and the results are presented in Section 3.7. The chapter ends with a few concluding remarks in Section 3.8.

3.1 Problem Overview

In this chapter, we study the coflow-with-deadlines scheduling problem and propose a new heuristic (NH) and solve an online linear program (OLP) to optimize for guaranteed coflow completions within their deadlines. We develop a new heuristic that admits more coflows than contemporary schemes [22] without violating the deadlines of admitted coflows. Specifically, for a new coflow request, the heuristic iterates over the earliest completion times of admitted coflows (earlier than the new coflow's deadline) to find possible data rates for its flows to finish within that deadline. When a coflow terminates, data rates are reallocated to ongoing flows. This heuristic is run on arrivals and departures of (co)flows. We also present a new optimization algorithm OLP that admits even more coflows than the heuristic but runs slower. The optimization algorithm *maximizes* coflow admissions by *minimizing the maximum data rates used by active coflows as long as the coflows finish within deadline*. The optimization algorithm is formulated as an online linear programming problem; it is also solved at (co)flows arrival and departures using the Gurobi solver [2].

Predictable performace for coflows with deadlines requires *guaranteed completion* so we expect our scheduler to satisfy this property. Additionally, the solution can be *work-conserving*, that is, it avoids underutilizing the network and distributes unused resources among admitted, active coflows. Both the OLP and the heuristic guarantee timely completion of admitted coflows and

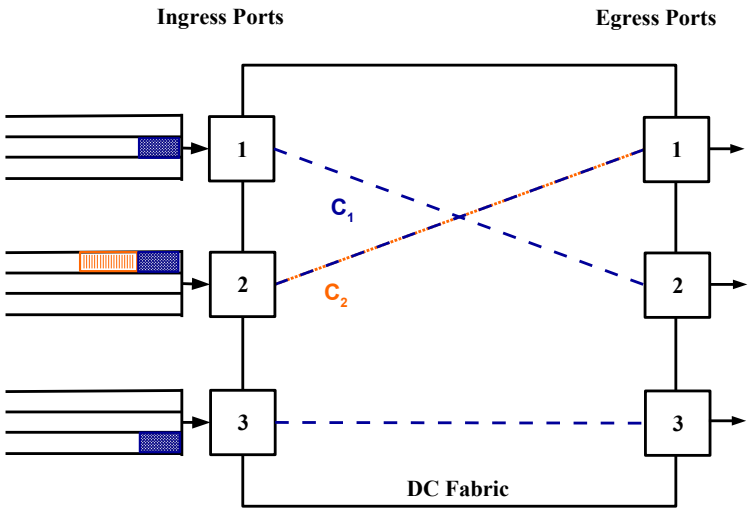

Figure 3.1: Coflow scheduling on a 3*3 datacenter fabric (Based on [37] ©2020 IEEE)

achieve high network utilization through work conservation (Algorithm 3); the heuristic, however, will reject more coflows than the OLP.

3.2 Model definitions

3.2.1 Network Model

We abstract the datacenter network as a non-blocking switch [6, 48, 10, 27] that interconnects M servers to the switch, as shown in Fig. 3.1. Now assume that each server $m \in M$ is connected to ingress and egress ports through a link with capacity K_m. Here, a flow $f_c^{i,j}$ from the set of flows F_c in coflow $c \in C$ transfers data volume v_f from its ingress port P_i^{in} to egress port P_j^{out} via the switch (see Table 2.1 for notation). For simplicity, these flows are organized in virtual output queues [71] on ingress ports. A flow $f_c^{i,j}$ is assigned a data rate r_f by the coflow scheduling algorithm. The data rate r_f is constrained by the available data rate on its uplink and downlink.

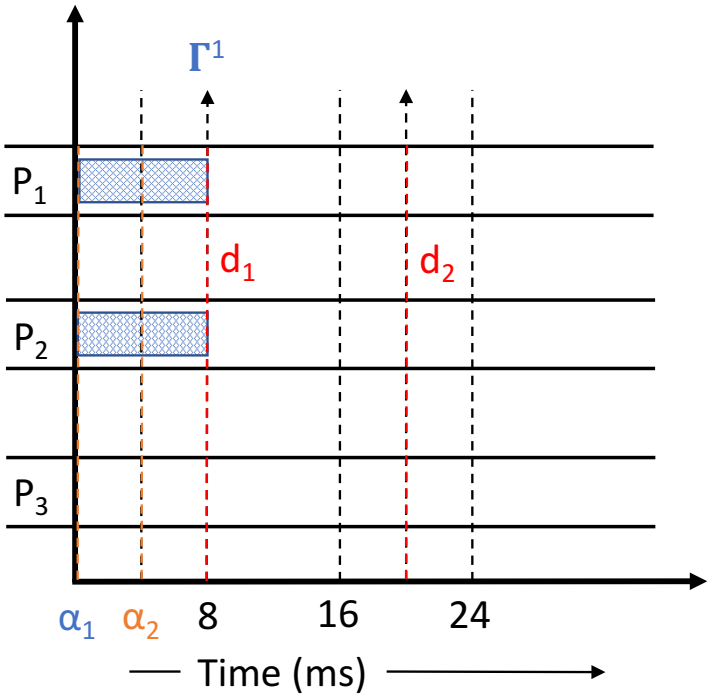

Figure 3.2: Varys coflow scheduling scheme for coflows in Fig. 3.1 admits only c_1 (Based on [37] ©2020 IEEE)

3.2.2 Motivating Example

Consider the example in Fig. 3.1 with two coflows c_1 (blue) and c_2 (orange). These coflows arrive at different times, that is, $\alpha_1 = 0\,\text{ms}$ and $\alpha_2 = 4\,\text{ms}$, with individual deadlines $d_1 = 8\,\text{ms}$ and $d_2 = 20\,\text{ms}$, respectively. Each coflow has a set of flows $f_c^{i,j} \in F_c$ on multiple links. The coflow c_1 has three flows (blue) of data size $v_1^{1,2} = 1\,\text{MB}$, $v_1^{2,1} = 1\,\text{MB}$, and $v_1^{3,3} = 1\,\text{MB}$, while the coflow c_2 has a single flow (orange) of data size $v_2^{2,1} = 1.5\,\text{MB}$. All flows on ingress ports are organized by destination on the non-blocking switch. Each port connects a server to the switch through a 1 Gbps link.

Varys [22] (Fig. 3.2) *admits coflow* c_1 at time 0 ms and allocates 1 Gbps data

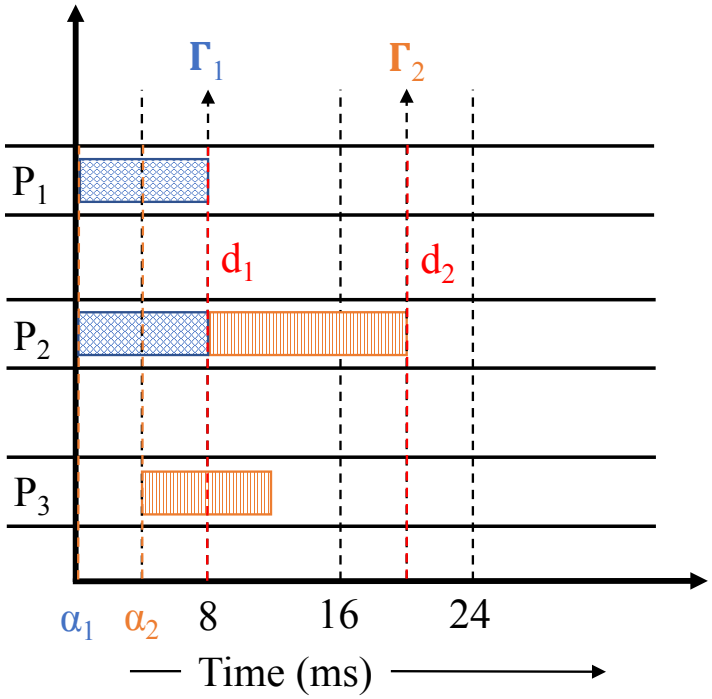

Figure 3.3: Optimal schedule for coflows in Fig. 3.1 admits both c_1 and c_2 (Based on [37] ©2020 IEEE)

rate to finish all flows by the deadline (8 ms). However, it *rejects coflow* c_2 at time $\alpha_2 = 4$ ms because coflow c_1 will miss its deadline on admission of coflow c_2, which requires at least 0.6 Gbps from the link capacity K_2 to finish by the deadline (20 ms). In contrast, the optimal schedule (Fig. 3.3) admits the coflow c_2 at $\alpha_2 = 4$ ms because it foresees the coflow completion Γ_1 at 8 ms and can commit a data rate $r_f = 1$ Gbps, at which the flow $f_2^{2,1}$ can start sending data from time 8 ms to finish by the deadline (20 ms). Although flow $f_2^{2,1}$ receives no service (that is, zero data rate) between two decision intervals α_2 and Γ_1, it is not starved for arbitrarily long times. Therefore, the optimal schedule avoids starvation and guarantees timely completion after coflow admissions.

3.3 Related Work

Coflow schedulers for deadline coflows: Literature has many proposals of coflow scheduling [19] for performance objectives like minimizing average coflow completion time [3, 22, 84] or meeting coflow deadlines [109, 108, 63, 22]. Our heuristic for coflows with deadlines differs from Varys [22] in that it finishes non-bottleneck flows in a coflow as soon as possible to avoid starvation. Chronos [63] combines priority scheduling with limited multiplexing to meet coflow deadlines and mixCoflow [109, 108] reduces footprint of resources used by deadline coflows but both approaches does not foresee and use released resources on completion of active coflows before the deadline of new coflow.
Flow schedulers: Deadline-aware flow scheduling in networks has been extensively studied [6, 43, 105] but it is unsuitable (or sub-optimal) for coflow admissions. For example, PDQ [43] approximates earliest deadline first algorithm for optimal flow scheduling but it is sub-optimal for coflow scheduling to meet deadlines [109, 3, 108, 63, 22]. Therefore, lack of knowledge about presence of coflows can degrade performance of data-parallel applications.

3.4 Optimization problem formulation

3.4.1 Overview

The coflow scheduling problem is formulated as an online linear programming (OLP) problem. It is solved whenever, at time t_e of the scheduling event $e \in [1, \cdots, E]$, one of the currently active coflows $\bar{\mathbb{C}}_e \subseteq \mathbb{C}$ (or its flow) departs after completing its data transmission or a new coflow arrives at time $\alpha_c, c \in C, c \notin \mathbb{C}$. In the first case, the data rates allocated to existing flows might be increased. In the second case, if there is a *feasible* solution to the problem, the obtained data rates $r_f, \bar{c} \in \bar{\mathbb{C}}_e \cup \{c\}$ are allocated to admitted coflows, otherwise the coflow c is rejected (and data rates already allocated to admitted flows are not changed). We emphasize that the resulting rates are not necessarily work-conserving; the output of this optimization problem is post-processed by the work-conserving algorithm, which is described in Section 3.6.

We assume that, as input, the OLP only has information about the already admitted, active coflows $\bar{\mathbb{C}}_e \subseteq \mathbb{C}$ plus the newly arriving coflow; future coflows' arrivals are unknown. Each new coflow request $c \in C, c \notin \mathbb{C}$ has an associated deadline d_c and a set of flows F_c. All flows F_c in coflow c share same arrival time α_c and deadline d_c. Each flow $f_c^{i,j} \in F_c$ in coflow c has a source i, destination j, and data volume v_f, all of which is known upon coflow arrival. All flows in active coflows $\bar{\mathbb{C}}_e$ are independent of their sibling flows in the same coflow and they are backlogged over the non-blocking switch, which abstracts out a datacenter network (Section 3.2.1).

3.4.2 Input

- Set of active coflows $\bar{\mathbb{C}}_e \subseteq \mathbb{C}$
- A new coflow request $c \in C, c \notin \mathbb{C}$ with an arrival time α_c, a deadline d_c, and a set of flows F_c
- Each flow $f_c^{i,j} \in F_c$ has a source i, destination j, and remaining data volume $\bar{v}_f$

- Each server $m \in M$ is connected to the non-blocking switch via a link of capacity K_m
- Current time t_e of the coflow departure or arrival event e (for simplicity, we assume that these events do not happen at exactly the same time; if they did, we process the coflow departure first)

3.4.3 Decision variables

For each flow $f_{\bar{c}}^{i,j} \in F_{\bar{c}}$ in coflow $\bar{c} \in \bar{\mathbb{C}}_e \cup \{c\}$, a data rate r_f is assigned to flow $f_{\bar{c}}^{i,j}$ by the coflow scheduling algorithm. It has a positive real value.

$$\forall \bar{c} \in \bar{\mathbb{C}}_e \cup \{c\}, \forall f_{\bar{c}}^{i,j} \in F_{\bar{c}}, i, j \in M : r_f \in \mathbb{R}^+ \tag{3.1}$$

3.4.4 Constraints

Link capacity constraints ensure that the total resource allocation (that is, data rate r_f) to flows is no more than the link capacity. Here, the constraint (3.2) ensures that the total data rates allocated to flows $f_{\bar{c}}^{i,j}$ on each uplink from source i to the switch must be less or equal to the uplink capacity K_i. Similarly, the constraint (3.3) ensures that the total data rates allocated to flows $f_{\bar{c}}^{i,j}$ on each downlink from the switch to destination j must not exceed the downlink capacity K_j.

$$\forall i \in M : \sum_{\bar{c} \in \bar{\mathbb{C}}_e \cup \{c\}} \sum_{j \in M} r_f \leq K_i \tag{3.2}$$

$$\forall j \in M : \sum_{\bar{c} \in \bar{\mathbb{C}}_e \cup \{c\}} \sum_{i \in M} r_f \leq K_j \tag{3.3}$$

Deadline constraints guarantee that the admitted coflows meet their deadlines. Constraint (3.4) ensures that each flow $f_{\bar{c}}^{i,j} \in F_{\bar{c}}$ in coflow $\bar{c} \in \bar{\mathbb{C}}_e \cup \{c\}$ is assigned, at least, minimum data rates r_f such that the remaining data volume $\bar{v}_f$ of flow $f_{\bar{c}}^{i,j}$ can be transmitted within its deadline.

$$\forall \bar{c} \in \bar{\mathbb{C}}_e \cup \{c\}, \forall f_{\bar{c}}^{i,j} \in F_{\bar{c}}, i,j \in M : r_f(d_{\bar{c}} - t_e) \geq \bar{v}_f \tag{3.4}$$

3.4.5 Objective functions

The objective (3.5) is to minimize the maximum data rates used by coflows (that is, both new and active coflows) as long as all of these coflows finish within deadline.

$$\min \quad \max_{i,j \in M} \sum_{\bar{c} \in \bar{\mathbb{C}}_e \cup \{c\}} r_f \tag{3.5}$$

3.5 Heuristic

We also propose a new heuristic (NH) for coflow scheduling that runs faster than the OLP but admits slightly fewer coflows. Like the OLP, the heuristic makes a decision to either admit or reject a coflow c on its arrival α_c and redistribute dates rates on departure of either a coflow $\alpha_c + \Gamma_c$ or one of its flows. It admits a new coflow $c \in C, c \notin \mathbb{C}$ only if it can meet its deadline without violating deadlines of already admitted coflows.

The algorithm 1, first, filter outs the active coflows $S \subseteq \bar{\mathbb{C}}_e$ based on their completion times, such that the completion time of an active coflow in S is less than or equal to coflow c's deadline, that is, $\Gamma_{S[index]} \leq d_c$ (line 1 in Algorithm 1). It, then, iterates (in earliest-completion-first order) over filtered coflows S to find a minimum data rate r_f (Algorithm 2) at each departure time $t = \alpha_{S[index]} + \Gamma_{S[index]}$ such that the new coflow c can finish within its deadline $\Gamma_c \leq d_c$ (line 8 in Algorithm 1). It calculates Γ_c from the remaining resources, only after providing minimum data rates to active coflows (line 6 of Algorithm 1) to avoid deadline violation. The Γ_c is equivalent to $t - \alpha_c + \Gamma_c^{'}$ where t is the departure time of a coflow in S and $\Gamma_c^{'}$ is the minimum time required to finish data transmission within deadline d_c, only after a flow $f_c^{i,j}$ has started sending data at time $t \geq \alpha_c$ with data rate

Algorithm 1 Coflow scheduling to guarantee deadline

Input: active coflows $\bar{\mathbb{C}}_e$ and new coflow, c

S = getCoflowsByB2CT($\bar{\mathbb{C}}_e, c, t_e$) {filter out competing coflows, which share links with c}
$t = t_e$
$index = 0$
$\mathbb{A} = \bar{\mathbb{C}}_e$
while true **do**
  allocDR($\mathbb{A}, t_e$) {refer Algorithm 2};
  $\Gamma_c = t - \alpha_c + \Gamma_c^{'}$;
  if $\Gamma_c \leq d_c$ **then**
    $c.start\_time = t$ {start sending data at time t};
    $\mathbb{C}^{'}$ = Enqueue c in $\bar{\mathbb{C}}_e$
    allocDR($\mathbb{C}^{'}, t$) {refer Algorithm 2};
    Distribute unused data rate to $\mathbb{C}^{'}$ {refer Algorithm 3};

    return true
  end if
  if $index < |S|$ **then**
    $t = \alpha_{S[index]} + \Gamma_{S[index]}$
    $remove(\mathbb{A}, S[index])$ {remove coflow $S[index]$ from $\mathbb{A}$}
    $index+ = 1$
  else
    Distribute unused data rate to $\bar{\mathbb{C}}_e$ {refer Algorithm 3}

    return false
  end if
end while

Algorithm 2 Allocate data rate to flows

Input: active coflows $\bar{\mathbb{C}}_e, time$
Output: data rates of flows

1: **for** $c \in \bar{\mathbb{C}}_e$ **do**
2: **for** $f_c^{i,j} \in F_c$ **do**
3: $r_f = \dfrac{\bar{v}_f}{\alpha_c + \Gamma_c - time}$
4: Update remaining data rates $Rem(P_i^{in})$ and $Rem(P_j^{out})$
5: **end for**
6: **end for**

r_f (that is, $\Gamma_c^{'} = \max(max_i \frac{\sum_j v_f}{min(Rem(P_i^{in}), Req(P_i^{in}))}$, $max_j \frac{\sum_i v_f}{min(Rem(P_j^{out}), Req(P_j^{out}))})$). Here, $Rem(\cdot)$ and $Req(\cdot)$ are the remaining and required data rates, respectively, on one of the links used by coflow c (line 7 in Algorithm 1). The unused resources are distributed to active coflows using Algorithm 3 (lines 12 and 21 in Algorithm 1).

3.6 Work-Conserving Resource Allocation

Work-conserving algorithm avoids idle yet usable network resources, lowers coflow completion time, and increases coflow admissions. We implement a new work-conserving algorithm, as shown in Algorithm 3. It distributes unused resources to active coflows by NH, which only allocate minimum data rates required to meet the deadline of coflows.

The algorithm first reorders admitted, active coflows in earliest-completion-first order (line 1 in Algorithm 3), which is defined as the earliest completion time of active coflows. It then performs a backfilling pass (line 5 in Algorithm 3) to distribute remaining data rates to reordered coflows, $\mathbb{C}^{'}$. Here, $Rem(\cdot)$ is the remaining data rate on ingress link i or egress link j while $Count(j)$ is the total number of flows on egress link j. The distribution of

Algorithm 3 Work-Conserving Algorithm

Input: active coflows, $\bar{\mathbb{C}}_e$

1: Reordered $\mathbb{C}^{'}$ by $\alpha_c + d_c - t_e$
2: **for** $c \in \mathbb{C}^{'}$ **do**
3: $num = Count(j)$
4: **for** $f_c^{i,j} \in F_c$ **do**
5: $r_f += min(Rem(i), Rem(j))/num$;
6: Update remaining data rates of links
7: $num -= 1$
8: **end for**
9: **end for**

unused resources to flows is constrained by the remaining capacity of both ingress and egress links.

The key insight here is that the coflows with the earliest completion time are prioritized among active coflows for allocation of remaining unused, usable resources. This insight enables us to reduce CCTs and admit more coflows. Since all active coflows are allocated minimum resources before distribution of unused resources, no active coflow is starved indefinitely from prioritizing shorter coflows (based on earliest completion time) in work-conserving algorithm.

3.7 Evaluation

We evaluated both *OLP* and *heuristic* through trace-driven simulation on a machine equipped with a 16-core CPU (Intel Xeon E5-2695) and 128 GB total memory. Specifically, the experiments were run on a guest VM of this machine, where the guest VM was allocated 4 CPUs and 8 GB total memory.

The highlights are:

- Simulation results show that our OLP and NH admit 1.1× more coflows than Varys heuristic (cp. Section 3.7.2) on the original Facebook trace [20, 22]. Put another way, both algorithms reduce coflow rejections as much as 10×.
- Futher evaluation on custom traces with varying network load and number of coflows shows that the NH consistantly outperforms the Varys heuristic (cp. Section 3.7.2).

3.7.1 Methodology

Workload

Trace: To find concrete parameters for the workload, we use published data center traces of production traffic from a Facebook Map/Reduce cluster [20, 22]. The cluster connects 3000 servers on 150 racks, which are organized similarly to Fig. 3.1. The mappers or reducers in the same rack are combined into single rack-level mapper or reducer, respectively. The trace contains 526 coflows, where a coflow captures information about the arrival time, number of mappers and reducers with their network port, and the total data volume. Unfortunately, the trace is not detailed enough to specify the concrete data volume per flow, but only per reduce task (which generates multiple flows). We make the simple assumption here that a reduce task distributes its data equally over all of its flows.
Since the original Facebook trace is limited in certain workload characteristics (for example, lower network load and fixed number of coflows), we, additionally, generate custom traces using the coflow workload generator in Sincronia [3] and study performance of our heuristic on varying network load and number of coflows.
Deadline: Since this trace represents best-effort data shuffle, there is no notion of deadline available from this published trace. We consider coflows' completion time in an empty network as (very optimistic) base case and look at a *relative deadline factor (x)* to compute actual deadlines that are proportionally longer than this optimistic base case (since coflows are likely to share links, it is highly unlikely that all coflows could be admitted without any laxity in the deadlines). The simulation is, then, run with

different factors to evaluate impact of varying, longer deadlines on coflow admissions (cp. Section 3.7.2).

Metrics

Percentage of admitted, successfully completed coflows: The primary performance metric for deadline-sensitive coflows is the percentage of coflows that are admitted and met their deadlines.
Percentage of admitted coflows that missed their deadline: The secondary performance metric is the percentage of coflows that are admitted but missed their deadlines. We expect all actually admitted flows to meet their deadlines; this number should be 0.

Simulator setup

Packet-level simulators like ns-2 are not suitable for coflow scheduling because of high overhead [116]. Therefore, like [116, 22, 4], we developed a Python-based, flow-level, discrete-event simulator for this paper. Unlike a packet-driven simulator, where packet transmission is the natural supplier of events, a flow-level simulator is driven by the (co)flows *arrival and departure events*. Events in this simulator are the arrival and possible admission as well as termination of (co)flows. The simulator's model is the set of links and their nominal data rates; for each flow, its remaining data volume and its currently assigned data rate as well as the set of links it traverses. The simulator ensures that the sum of the data rates of all flows using a link is at most as large as the link's nominal data rate. Whenever a flow's data rate changes, its termination time is recalculated based on the remaining data volume.

Whenever a coflow arrives, the scheduler admits or rejects coflows using one of the algorithms above and determines the data rates of flows. The computed data rates are assigned to flows. Whenever one of the flows or an admitted coflow completes (or misses deadline), its resources are freed up. The unused resources are then distributed to active flows of already admitted coflows. This allows efficient simulations as only very few events

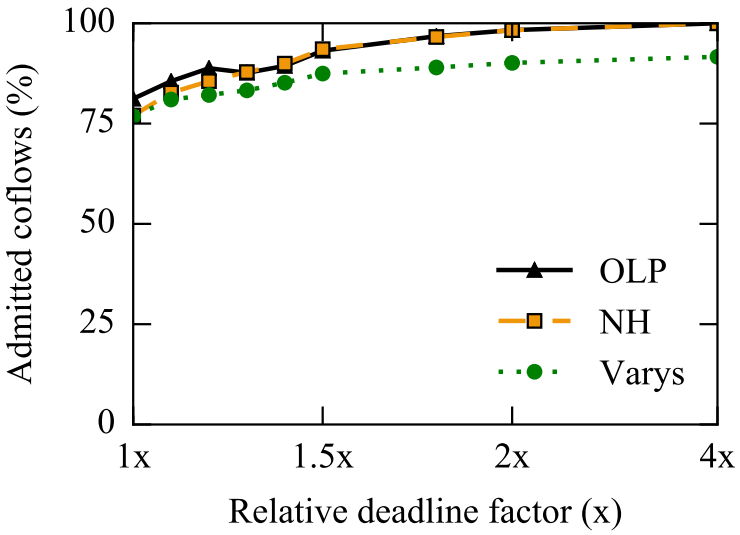

Figure 3.4: Percentage of admitted coflows by different scheduling algorithms (Based on [37] ©2020 IEEE)

per flow needs to be processed; on the downside, it is less accurate than a packet-level simulator, for example, ignoring TCP congestion control aspects.

3.7.2 Simulation Results

After running our OLP and NH for guaranteed coflow completion times, we found some interesting results on coflow admissions, miss rate, and computational time.

Percentage of admitted coflows

Both our OLP and NH admitted 1.1× more coflows than the state-of-the-art Varys heuristic, as shown in Fig. 3.4. In the default case ($x = 1$), 81.18 % coflows are admitted by the OLP algorithm and all of them met their

Table 3.1: Percentage of admitted coflows by OLP and NH more than Varys

Relative deadline factor (x)	Increase in coflow admissions (%)	
	OLP	NH
1.1x	6 %	2 %
1.2x	8 %	4 %
1.3x	7 %	7 %
1.8x	9 %	9 %
2x	9 %	9 %

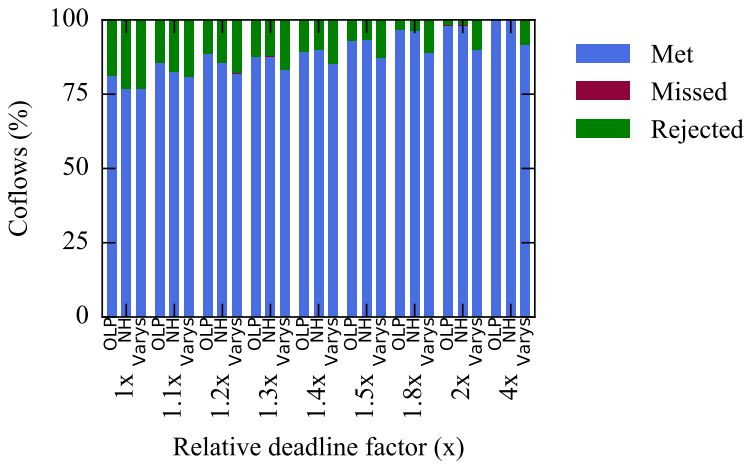

Figure 3.5: Percentage of coflows that meet deadline (Based on [37] ©2020 IEEE)

deadline as compared to 76.81 % coflows admitted by the Varys heuristic. We think that the increase in coflow admissions over the Varys heuristic (in comparison to 75 % admitted coflows quoted in [22]) is because of the decisions being made at (co)flow arrival and departure events in our implementation. Unlike the fixed, 10 s long decision intervals in [22], we expect our approach to give more precise results. Additionally, our variant of Varys used, like the other two algorithms, our new work-conserving algorithm (Section 3.6).

Recall that the default deadline of coflows is set to a coflow's completion time in an *empty* network (obviously making it impossible to achieve 100 % admission rate). Thus, we analyse the impact of slightly longer deadlines, namely *relative deadline factor (x)*, on coflow admissions. We found that both our OLP and heuristic NH admitted more coflows than Varys at different relative deadline factors and guaranteed completion within deadlines, as shown in Fig. 3.4 and Table 3.1. In addition, our heuristic NH performed competitively in comparison to the OLP algorithm.

Percentage of coflows that missed their deadline

Like Varys, and as expected, all of the admitted coflows in OLP and heuristic met their deadline, as shown in Fig. 3.5. For example, in the default case ($x = 1$), the OLP in our implementation admitted 81.18 % coflows and all of them met the deadline.

Network load

Since the original Facebook trace has low network load, we further analyse the performance of our heuristic NH under varying network loads. For instance, we generate custom traces with same number of coflows (526) but two different network loads (that is, 0.9 and 0.5). The result (in Fig. 3.7) shows that the performance of our heuristic improves as the network load decreases from 0.9 to 0.5. The higher coflow admissions at lower network loads is expected because coflows, in general, have lower competition for resources among themselves. Importantly, NH admits more coflows than

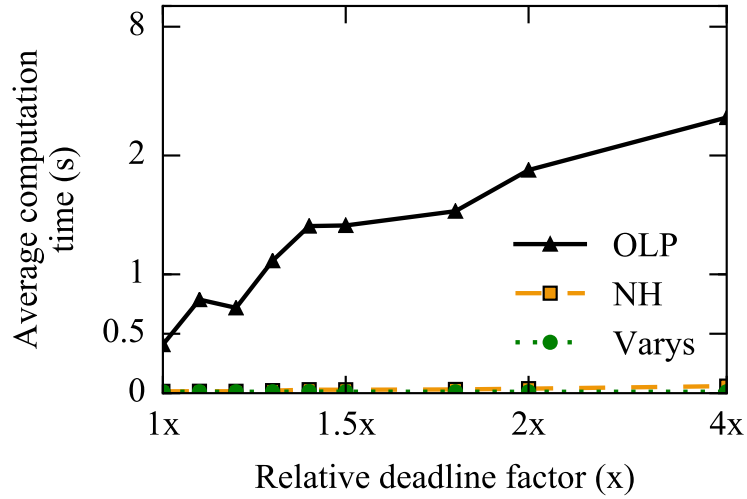

Figure 3.6: Average computation time of different scheduling algorithms (Based on [37] ©2020 IEEE)

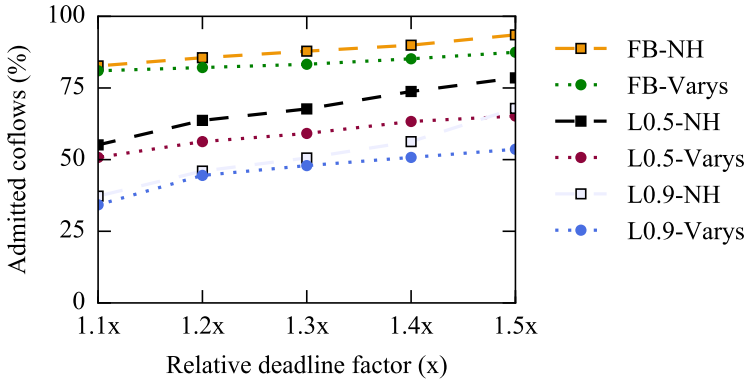

Figure 3.7: Impact of varying network load on coflow admissions (Based on [37] ©2020 IEEE)

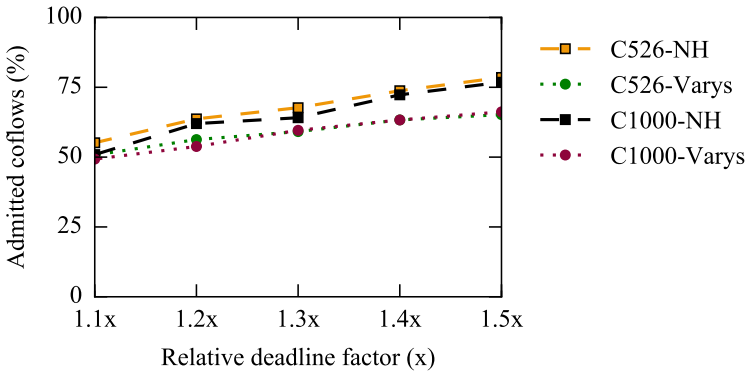

Figure 3.8: Impact of increase in number of coflows on coflow admissions (Based on [37] ©2020 IEEE)

Varys in all three network loads (that is, FB, 0.5, and 0.9) at different relative deadline factors (x). Since the greater factor value provides more slack time to coflows for completion, coflow schedulers admit more coflows at higher factors, for example, 1.5x than 1.1x.

Impact of more coflows

Similarly, we perform additional analysis of our heuristic against increase in number of coflows. For example, the result (in Fig. 3.8) uses two custom traces of different sizes (that is, 526 and 1000 coflows) with same 0.5 network load. It shows that the contention for network resources increases with the increase in number of coflows. Importantly, our heuristic NH consistantly admits more coflows than the Varys heuristic [22] in both custom traces.

3.8 Concluding Remarks

In this chapter, we have presented two algorithms to increase coflow admissions while meeting their deadlines. Specifically, both algorithms have admitted 1.1× more coflows than the state-of-the-art Varys heuristic in

large-scale trace-driven simulations. In addition, our heuristic runs faster yet performs competitively with the OLP algorithm.

However, coflow heuristics, in general, are susceptible to stochastic coflow arrivals, that is, they admit less coflows for the desired performance objective to maximize coflow admissions while meeting their deadlines. For instance, a large coflow (based on either completion time, size, width or length) impedes admission of upcoming, smaller coflows. From next chapter, we further elaborate the problem and propose its solutions.

Chapter 4

Learning Flow Scheduling

This chapter of the thesis is based on the revised text of our conference paper; therefore, some portions of the chapter contain verbatim content, for example, figures and tables used in the paper:

Asif Hasnain and Holger Karl. Learning flow scheduling. In *2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–6, 2021

In this chapter, we study the impact of stochastic network traffic on flow scheduling heuristics and argue that these heuristics are often susceptible to stochastic flow arrivals in a network. Specifically, we propose a new flow scheduler, namely learning flow scheduling (LFS), which learns flow scheduling policies and adapts scheduling decisions according to network changes.

My main contribution in this chapter was to design, implement, and evaluate the LFS scheduler for learning flow scheduling policies. Firstly, I sketched a counterexample to show that non-preemptive flow schedulers like D^3 [105] are susceptible to stochastic network traffic, that is, they admit less flows than an optimal schedule (Section 4.1) for the desired performance objective to *maximize flow admissions while meeting their deadlines*. Then, I formulated the flow scheduling problem as a Markov decision process (MDP) and designed the state space, the action space, and the reward function. The problem is solved using deep reinforcement learning (DRL) in which the flow scheduler learns to make decisions by interacting with the environment. For this, I implemented the flow-level simulator as an environment and

a training algorithm to learn a scheduling policy for the specified higher-level performance objective. In addition, I evaluated the performance of the trained LFS scheduler through trace-driven simulation and compared its results with those of the greedy heuristics.

The remainder of the chapter is organized as follows. The first Section 4.1 of the chapter introduces the problem with a motivating example. In the next Section 4.2, relevant literature is analysed. As a first step to learn a flow scheduling policy, we describe the model in Section 4.3.1. The LFS scheduling problem is formulated as an MDP; specifically, the average reward formulation for continuous flow scheduling tasks is described in Section 4.3.2. The LFS scheduler uses a policy gradient algorithm for end-to-end training in a network environment (Section 4.3.3). The proposed LFS scheduler is evaluated on custom traces, which are generated by sampling flow information (that is, arrival time, size, and deadline) from different distributions. Our results (in Section 4.4) show that the trained LFS scheduler outperforms greedy heuristics under varying network loads. Specifically, the LFS scheduler learns a policy that admits more smaller flows than long flows for the specified performance objective (that is, *maximize* the number of flow admissions while meeting deadlines). Moreover, it is flexible enough to quickly adapt to various performance objectives by using different rewards. In addition, the LFS is compared with greedy heuristics under varying network load. In Section 4.5, the chapter concludes with a few closing remarks.

4.1 Introduction

In most non-trivial scenarios, flow scheduling [13, 6, 43, 4] is NP-hard [30]. Hence, flow scheduling is often performed using heuristics, which are optimized for a specific workload. These heuristics are developed by manually identifying features in network structure [8] or flow arrival patterns, which is a time-consuming activity with long turn-around time. This problem is aggravated when workload changes render such hand-crafted heuristics no longer useful.

Table 4.1: Competing flows on a link

Flows	Arrival time	Size	Deadline
f_1	0	4	7
f_2	1	1	1
f_3	4	2	2

To address this challenge, in this paper, we attempt to *automate network resource management*, specifically, flow scheduling for deadline-sensitive flows using LFS. Like many input-driven applications [69, 70, 67, 87], LFS learns flow scheduling policies through reinforcement learning (RL). RL is well-suited for such learning because it automates learning Markov structure in flows through end-to-end training on network states. The scheduling agent is based on a neural network [7, 36], which is trained to learn a scheduling policy by directly interacting with the environment. We focus here on the flow admissions aspect. That means that the policy takes an action to admit or reject new flows, arriving online in different network states. On each action, the environment produces a reward as feedback for the policy to know how well it is doing on flow admissions. The rewards are based on a workload-specific performance objective (that is, *maximize* the number of flow admissions and meet deadlines for time-sensitive datacenter applications, for example, web search or social networking). The flow scheduler, after a flow has been admitted, assumes no preemption and an admitted flow is assigned a constant data rate from the time a flow starts executing till its completion. The data rate assigned to flows is computed by a greedy heuristic within the environment.

One of several challenges in training a policy is learning to make decisions on stochastic flow arrivals to a network. The stochastic flow arrivals make it difficult for a policy to learn from rewards because the successive arrival of smaller flows in an input can produce higher rewards on admission than large flows for this objective. Consider, for instance, three competing

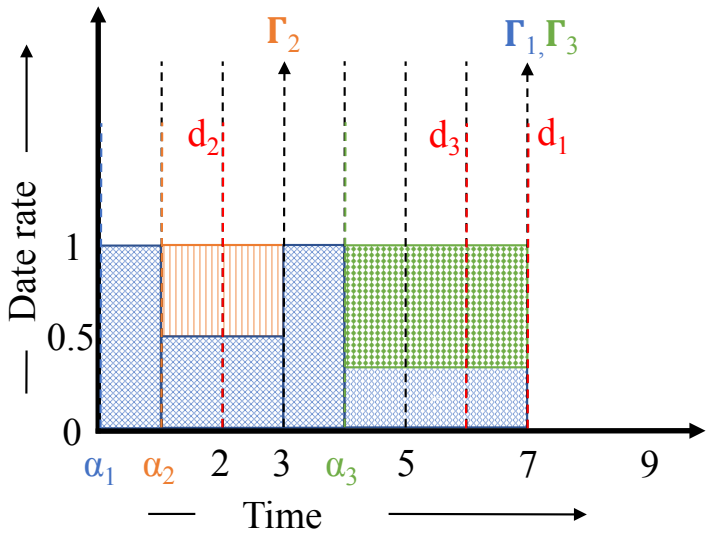

Figure 4.1: D^3 schedule (Based on [39] ©2021 IEEE)

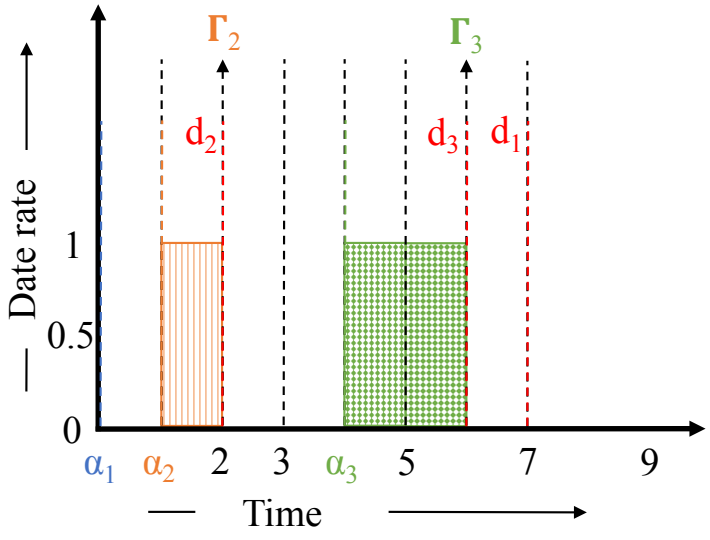

Figure 4.2: Optimal schedule (Based on [39] ©2021 IEEE)

flows in Table 4.1, that is, f_1, f_2, and f_3 (see Table 4.2 for notation). These flows have different arrival times, individual sizes and deadlines. They are competing for resources on a single link with total data rate of 1 unit. When flow f_1 (blue in Fig. 4.1) arrives first at time 0, the *non-preemptive* D^3 [105] scheduler assigns, at least, minimum data rate to flow f_1 to finish within its deadline (7). However, D^3 *hogs* the link's resource by admitting flow f_1, which has a large deadline. Consequently, two flows f_2 (orange) and f_3 (green) fail to meet their deadlines because both flows require, at least, 1/1 and 2/2 units of link resource, respectively, to finish within their deadlines. But the remaining data rates at time 1 and 4, after minimum allocation to active flow f_1, are 1/2 and 2/3 units, respectively. Unlike the myopic D^3 scheme, the optimal schedule (in Fig. 4.2) *rejects* large flow f_1 at time 0 because it can admit more, smaller flows, that is, f_2 and f_3, for the desired performance objective (that is, *maximize* the number of flows while meeting their deadlines). It commits, at least, minimum data rates to two flows, that is, $r_2 = 1$ and $r_3 = 1$, from their arrival times to finish within the deadlines. Such an optimal schedule can be achieved if LFS scheduler effectively learns flow arrivals pattern in network states.

4.2 Related Work

Flow scheduling with deadlines: D^3 [105] is a non-preemptive scheduler that serves flows in order of their arrivals. However, it hogs (or blocks) upcoming smaller flows — based on their short deadlines — by admitting flows with longer deadlines [43]. In addition, D^2TCP [96] reduces the number of missed flow deadlines; however, under high traffic load, it misses higher number of flow deadlines [113]. In contrast, the LFS scheduler learns to reject large flows and admit small flows for the specified performance of maximizing flow admissions. It differs from deadline-aware flow schedulers like PDQ [43], which approximates earliest deadline first algorithm, because it is a non-preemptive scheduler and avoids starvation of flows. In a recent proposal [31], an RL-based flow scheduler is trained to compute data rates of deadline flows but it uses a Q-learning lookup table, which is not a scalable approach.

Flow scheduling with and without deadlines: Karuna [13] concurrently schedules mixed flows (that is, flows with and without deadlines) to optimize for two performance objectives, that is, maximize flow admissions while meeting deadlines and minimize average FCT, respectively. Since it gives priority to deadline flows, non-deadline flows have slightly higher average FCT [98]. Unlike Karuna [13], Aemon [98] is information-agnostic and prioritizes non-deadline flows over deadline flows to reduce average FCT. In contrast, we focus only on a single performance objective (to maximize flow admissions while meeting with deadlines) and leave comparison of both approaches with the LFS scheduler for future work.
Flow scheduling without deadlines: PIAS [9] utilizes priority queues to implement multi-level feedback queue (MLFQ). Specifically, it demotes a flow from higher priority queues to lower priority queues based on the data volume a flow has sent. PIAS effectively emulates SJF flow scheduling without prior information of flow data volume. Similarly, NUMFabric [74] requires support of programmable priority queues in switches for resource allocation and EPN [62] leverages priority queues in commodity switches to reduce average FCT. In addition, RL-based flow scheduler AuTO [14] automates traffic optimization in datacenter networks but, unlike LFS scheduler, it only optimizes for average FCT using strict priority queueing.

4.3 Design

4.3.1 Model definition

We consider a single network link l and, like prior work [6], assume that the information about a new flow $l_f \in F$ is known at arrival time α_f (see Table 4.2 for notation). That information includes data volume v_f and relative deadline d_f of the flow. If an arriving flow f is admitted, its assigned data rate r_f is calculated using a greedy heuristic and it is constrained by the data rate of link K_l. The heuristic simply divides flow size v_f by the remaining time to deadline d_f to get the data rate r_f. It is kept simple to learn the scheduling policy for higher flow admissions (Section 4.3.2). We further assume that active flows $\bar{\mathbb{F}}_e$ are not preempted and they continuously

Table 4.2: Notation for flow and system model

F	Set of arrived, online flows, where F corresponds to F_c in Table 2.1
$\mathbb{F}$	Set of admitted flows $\mathbb{F} \subseteq F$, which comprises active and completed flows
$\bar{\mathbb{F}}_e$	Set of active flows $\bar{\mathbb{F}}_e \subseteq \mathbb{F}$ at time t_e, not including a flow arriving at t_e
l_f	A new flow f request on link l (the event e causing that new flow will be clear from context)
α_f	Arrival time of flow f
d_f	Relative deadline of flow f
$\alpha_f + d_f$	Absolute deadline of flow f
K_l	Total available data rate of link l

receive link resource, that is, data rate, from the time a flow starts execution till completion.

4.3.2 RL model

The problem is formulated as a discrete-time Markov decision process (MDP) and it is solved using a policy gradient algorithm of deep RL, where a flow scheduling agent interacts with a single-link l environment. The scheduling agent can fully observe the state s_e at time t_e, that is, information about active flows $\bar{\mathbb{F}}_e$ on link l and the new flow request f is available. A flow departure is processed within the environment and not made visible to the RL agent; if one of the active flows $\bar{\mathbb{F}}_e$ completes, it is removed from the network. On each scheduling event e, the scheduling agent takes an action a_e in state s_e, collects a reward R_e from the environment, and shifts to the next state s_{e+1}, where the next scheduling event$e+1$ occurs at time t_{e+1}. Specifically, the state transition is assumed to satisfy the Markov property,

Table 4.3: Notation for learning model

Q	Number of different flow arrival sequences as defined in Table 4.2
F_k	The kth set of flow arrival sequences, where $k \in \{1, \cdots, Q\}$ and F_k corresponds to F in Table 4.2
E_k	Number of scheduling events in F_k, where E_k corresponds to E in Table 2.1
λ	Arrival rate of a flow arrival sequence F_k
τ	Episode length
N	Number of different sample trajectories
s_e	Observed state at time t_e
$a_e \in \{0, 1\}$	Action taken at time t_e
R_e	Reward (or penalty) received on action a_e in state s_e
$\bar{R}_e$	Time-average reward at time t_e
$R_e - \bar{R}_e$	*Differential* reward
G_e	*Differential* return (sum of *differential* rewards from state s_e to the terminal state s_E)
$\pi_\theta(a_e \mid s_e)$	Policy network, learnt by the actor
θ	Parameters of the policy network $\pi_\theta(\cdot)$
β_θ	Learning rate (or step size) of policy network $\pi_\theta(\cdot)$
$V_{v_k}(s_e)$	State-value function for the flow arrival sequence F_k (critic)
v_k	Parameters of the state-value network $V_{v_k}(\cdot)$
β_v	Learning rate of state-value network $V_{v_k}(\cdot)$
δ_e	Error in estimation of *differential* return G_e
$\mathbb{P}(s_{e+1} \mid s_e, a_e)$	State transition probability function
$J(\theta)$	Performance objective for policy network $\pi_\theta(\cdot)$
$\nabla_\theta J(\theta)$	Policy gradient

that is, the new state s_{e+1} depends only on the current state s_e and the action a_e taken at time step t_e.

Since the problem space is large and continuous (the interation between flow scheduling agent and link environment goes on forever), the scheduling agent uses a neural network [36] to learn the policy $\pi_\theta(a_e|s_e)$, where θ are parameters of a *policy network*. The policy $\pi_\theta(a_e|s_e)$ is defined as the probability of taking action a_e in state s_e with parameters θ. After each action a_e, the environment provides a *differential* reward $R_e \leftarrow R_e - \bar{R}_e$ (Section 4.3.2) to the scheduling agent, where $\bar{R}_e$ is the time-average reward at time t_e. The scheduling agent uses the reward as signal to improve the policy $\pi_\theta(a_e|s_e)$. The reward is based on a higher-level performance objective $J(\theta)$ to *maximize* the number of admitted flows while meeting their deadlines.

The scheduling agent trains the policy network $\pi_\theta(a_e|s_e)$ through the *REINFORCE algorithm with baseline* (sometimes, also called *Monte-Carlo (MC) actor critic*) [104]. Although the algorithm is unbiased, it has high variance in policy gradient. The variance is caused by single-sample estimate and stochastic flow arrivals, which impede effective learning of a scheduling policy. The variance is usually reduced by subtracting a state-value function (critic) as baseline [101] from the actual *differential* return, where the state-value function estimates the *differential* return.

However, a single state-value function as baseline is proven ineffective [70] to estimate *differential* return in the presence of different flow arrivals because training on different flow arrivals adds noise to the reward and makes it difficult to estimate *differential* return using a single state-value function. One way [70] to effectively estimate *differential* return, with different flow arrivals, is to train a separate state-value function $V_{v_k}(s_e)$ for each flow arrivals sequence $F_k, k \in \{1, \cdots, Q\}$, where v_k are parameters of a state-value network and Q is the total number of flow arrival sequences. These multiple state-value functions act as critic to *evaluate* the scheduling policy $\pi_\theta(a_e|s_e)$ and provide feedback to the policy network (which is also called an actor).

Objective: The high-level performance objective for scheduling policy is to *maximize* the number of flow admissions while meeting their deadlines.

The objective is defined by an average-reward formulation because flow scheduling is a continuous task (Section 4.3.2). Specifically, the environment gives more reward for flows with smaller flow completion time (FCT) than larger FCT to achieve higher flow admissions.

State space: It represents the fully observed state s_e at time t_e of a particular scheduling event e, that is, a flow arrival. The state information is a flat feature vector of active flows $\bar{\mathbb{F}}_e \subseteq \mathbb{F}$ and the new flow request f on link l, where the maximum number of concurrent, active flows is limited to 50 (Section 4.4.2). The feature vector is passed as an input to the policy network for learning flow structures.

- Each active flow $f \in \bar{\mathbb{F}}_e$ has
  - its remaining data volume $\bar{v}_f$
  - its remaining time to deadline $\alpha_f + d_f - t_e$
  - its assigned data rate r_f
- For data rates of active flows on link l, it always holds that $\sum_{f \in \bar{\mathbb{F}}_e} r_f \leq K_l$.
- A new flow request f has an arrival time α_f, a relative deadline d_f, and a data volume v_f

Action space: It is a discrete set $a_e \in \{0, 1\}$, where the actions $a_e = 1$ and $a_e = 0$ represent the decision to either admit or reject the flow request f in state s_e, respectively. The decision is taken by the flow scheduling agent.

Actor: The actor directly learns a *softmax paramerterized policy* $\pi_\theta(a_e|s_e)$, which outputs probability distribution over all actions A in state s_e with parameters θ. The action $a_e \in A$, to admit or reject a new flow in state s_e, is then sampled from action probabilities using Gumbel-Softmax distribution [46]. The actor updates policy parameters θ via gradient descent in the direction (that is, gradient) suggested by the critic (Section 4.3.2). It receives a feedback from critic, on the performance of its current scheduling policy, in the form of an estimated error δ_e (sometimes, also called an *advantage*). The actor uses the estimated error δ_e to update action probabilities such that it reaches high-valued states with more flow admissions and attempts to keep error δ_e positive (that is, collect better-than-time-average reward). Specifically, it computes the policy gradient $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_e|s_e)\delta_e$ to update policy parameters such that $\theta \leftarrow \theta + \beta_\theta \nabla_\theta J(\theta)$, where β_θ is the constant learning rate (or step size) for policy parameters θ. The policy gradient $\nabla_\theta J(\theta)$ on performance objective $J(\theta)$ increases probability of taking action

a_e in state s_e if the action (for example, to admit a flow) resulted in higher *differential* return than the estimated return by critic.

Critic: The critic *evaluates* the scheduling policy using multiple state-value networks as baseline. The state-value network $V_{v_k}(s_e)$ estimates the *differential* return G_e in a particular flow arrival sequence F_k, where $G_e = R_e - \bar{R}_e + R_{e+1} - \bar{R}_{e+1} + \cdots$ is the sum of *differential* rewards from state s_e to the terminal state s_E and F_k is the kth set of flow arrivals in Q sequences, that is, $k \in \{1, \cdots, Q\}$. Intuitively, the critic has a separate state-value function $V_{v_k}(s_e)$ for each flow arrival sequence F_k to reduce variance of policy gradient from training on different flow arrival sequences. It criticizes the actor's action, based on actor's policy, by sending an error δ_e to actor. The critic computes error δ_e from the *differential* return G_e (as per actor's policy) and its estimation of value of (being in) current state $V_{v_k}(s_e)$. It is given by $\delta_e = G_e - V_{v_k}(s_e)$, where a positive error δ_e means that the actor's action was good (so should be repeated) because it led to better-than-time-average reward. On the contrary, a negative error reflects worse-than-time-average reward, which means the actor should avoid this, bad action. The state-value network $V_{v_k}(s_e)$ uses *differential* return G_e from policy to improve its predictive accuracy. It does so by reducing the magnitude of loss in estimated return close to zero, where the loss function is mean squared error (MSE).

Average reward: Since the flow scheduling problem is a continuous task, an average reward is better suited than the total reward and it maximizes $\lim_{\tau\to\infty} 1/\tau \sum_{e=0}^{\tau} R_e$ [91], where τ is a training episode length. Specifically, the network environment rewards the scheduling agent with a *differential* reward $R_e \leftarrow R_e - \bar{R}_e$, where $\bar{R}_e$ is the moving time-average reward at time t_e from all previous scheduling events of current and previous training episodes. The scheduling agent receives a reward for every action a_e in state s_e, where the reward function is designed as follows:

- We call an action (by scheduling policy) to admit a new flow f a *true positive* (TP) decision if it is indeed possible to assign enough rate to the flow f to meet its deadline. This can be tested immediately by checking flow deadline, volume, and currently available data rate. A TP decision for a new flow f, in the confusion matrix, returns higher reward for a flow with smaller FCT than a flow with large FCT, that is, $R_e = 1/\Gamma_f + (1/\Gamma_f * 1/|\bar{\mathbb{F}}_e|)$, where $|\bar{\mathbb{F}}_e|$ is the number of active flows at time t_e.

- A *false negative* (FN) decision means that the scheduling policy could have admitted the new flow f in current state s_e but did not. It might be a correct action after enough learning, for example, to reject a flow with large FCT, for this performance objective $J(\theta)$. A FN decision produces a penalty $R_e = -1/\Gamma_f$, where the actor is penalized more for rejecting a flow with smaller FCT than large FCT.
- An action is considered a *true negative* (TN) decision if the scheduling policy has learnt to correctly reject a new flow f if there is not enough link resource so that the flow cannot meet its deadline. A TN decision has zero reward $R_e = 0$.
- The action to admit a new flow f is called *false positive* (FP) decision if the flow cannot actually be assigned sufficient rate to meet its deadline. A FP decision for a new flow f is penalized with $R_e = -(\Gamma_f + |\bar{\mathbb{F}}_e|)$.

4.3.3 Training algorithm

The scheduling policy is trained using algorithm 4. Since the initial scheduling policy is assumed poor, the earlier training episodes (or epochs) are terminated stochastically at time τ to help learning on online flow arrivals (line 4). However, the episode lengths are gradually increased during training (line 15). In each episode, the training algorithm rolls out multiple trajectories $j \in [1, \cdots, N]$ on current scheduling policy $\pi_\theta(a_e|s_e)$ (line 6), computes return from *differential* rewards (line 8), and estimates error in predicting return (line 9). Based on the estimated error and return, it updates parameters of the policy and critic networks, respectively (line 10–14).

4.4 Evaluation

We evaluated *LFS* scheduler through a flow-level simulator as an environment on a machine with a 16-core CPU (Intel Xeon E5-2695) and 128 GB total memory. The highlights are:

- The RL-based flow scheduler learns to optimize the specified performance objective.

Algorithm 4 Monte-Carlo Actor Critic Algorithm for Training

Input: Policy network $\pi_\theta(\cdot)$ and state-value networks $\{V_{v_1}(\cdot), \cdots, V_{v_Q}(\cdot)\}$

$\theta, \{v_1, \cdots, v_Q\}$ {Initialize parameters of policy and state-value networks with Glorot uniform initializer}
Initialize learning rates $\beta_\delta, \beta_v > 0$ and the time-average reward $\bar{R}_e = 0$
for each episode **do**
  Sample an episode length τ from a geometric distribution
  Sample a new set of flows F for each of the flow arrival sequences $F_k, k \in \{1, \cdots, Q\}$
  Rollout multiple trajectories $j \in [1, \cdots, N]$ on current policy $\pi_\theta(a_e|s_e) \sim \{s_1, a_1, R_1, \cdots, s_E, a_E, R_E\}$ until $t_e \leq \tau$
  $\bar{R}_e \leftarrow \bar{R}_e + 1/(N \cdot E) \sum_{j=1}^{N} \sum_{e=1}^{E} R_e$; {Update time-average reward}
  Calculate *differential* return in each state, that is, $G_e = \sum_{e'=e}^{E} R_{e'} - \bar{R}_{e'}$
  $\delta_e = G_e - V_{v_k}(s_e)$ {Calculate the error δ_e in estimation}
  for $j \in [1, \cdots, N]$ **do**
    $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_e|s_e)\delta_e$; {Compute policy gradient using the error δ_e}
    $V_{v_k} \leftarrow V_{v_k} + \beta_v \nabla_{V_{v_k}} V_{v_k}(s_e) G_e$; {Update parameters V_{v_k} of the state-value network $V_{v_k}(\cdot)$}
  end for
  $\theta \leftarrow \theta + \beta_\theta \nabla_\theta J(\theta)$; {Update parameters θ of policy network}
  $\tau \leftarrow \tau + \epsilon$
end for

- It admits, on average, 1.05× (or 5 %) more flows than the greedy algorithm on an unseen test dataset.
- It consistently outperforms the competing schemes under varying network load (Section 4.4.2).

4.4.1 Methodology

Workload

Like prior work [6], we assume that the flow information is known at arrival and it includes data volume and deadline — relative to the arrival time. The flow interarrival time α_f is sampled from the exponential distribution, where the flow arrival rate is set to $\lambda = 1$. The flow arrival rate λ is varied during testing to create *network loads* and compare performance of the LFS flow scheduler with competing schemes (Section 4.4.2). The LFS flow scheduler is, however, trained on a single flow arrival rate λ. Since datacenter traffic has a long-tailed distribution [11, 78, 5], that is, most of the flows are small but the majority of the data is transmitted by a few large flows, the flow size v_f is sampled from the Pareto distribution, where the pareto shape and scale is set to 2.0 and 100.0, respectively. In addition, the flow deadline d_f is a uniformly drawn value between 1 and 4 seconds because most flows in datacenter traffic last less than a few seconds [78].

Environment

Unlike flow simulator in Chapter 3, the new simulator is only driven by flow *arrival and departure events* and it is wrapped in an environment to interact with the flow scheduling agent. Specifically, the environment implements the gym interface [12] and whenever a new flow arrives α_f, the flow scheduler (that is, LFS or a competing algorithm) either admits or rejects the flow. The data rate r_f for the flow is computed within the environment using a greedy heuristic. The heuristic simply divides flow size v_f by the deadline d_f to get the data rate r_f. It is not work-conserving, that is, the excessive data rate on link l is not distributed, after minimum resource allocation, to neither the newly admitted nor to already active flows. Whenever one of the

active flows completes, the simulator removes the flow from the network. The incomplete, active flows continue executing at the same, constant data rate.

Scheduling agent

The scheduling agent consists of a policy network and multiple state-value networks. For neural networks, we use multilayer perceptrons (MLPs) because MLPs are suitable for classification problems. Although the selection of hyperparameters for effective learning of MLPs is quite challenging, the scheduling agent was able to learn a reasonable policy with the following set of values. Specifically, each MLPs network has 2 hidden layers, where the two hidden layers comprise 200 and 128 neurons each and their activation function is set to ReLU [75]. These networks use Adam [53] optimizer to update their parameters, where the learning rates β_θ and β_v for parameters of policy and state-value networks are set to 7×10^{-3} and 7×10^{-3}, respectively. In addition, the entropy value for *exploring* the scheduling policy is intialized to 1. However, it is gradually decayed during training of the policy network, that is, it decays with value 1×10^{-3} until the minimum entropy value 1×10^{-4}.

Metrics

We primarily measure the flow admissions by various scheduling schemes. In addition, we measure rewards and flow sizes to answer the following questions:

- Does the LFS scheduler learn a policy to optimize the performance objective (Section 4.3.2)?
- What is the behaviour of trained LFS scheduler under varying *network load*?
- Do the admitted flows *meet their deadline*? We expect that all admitted flows finish within their deadline.
- Why does the LFS scheduler admit more flows?

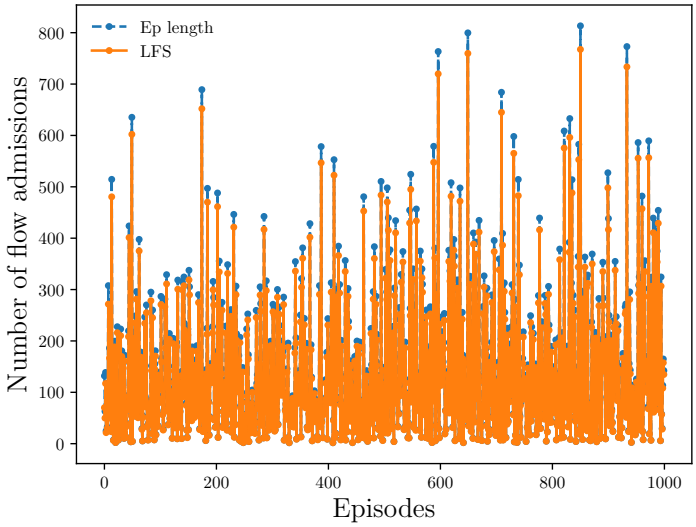

Figure 4.3: Total flow admissions during training of LFS scheduler (Based on [39] ©2021 IEEE)

4.4.2 Simulation results

Training of the LFS scheduler

The LFS scheduler is trained using algorithm 4. The algorithm runs 1000 episodes and, in each training episode, 1000 flows are generated to train the LFS scheduler. Each episode is terminated at the maximum time τ, which is sampled from a geometric distribution. The reset probability of episode length $p = 1 \times 10^{-2}$ decays (with value 4×10^{-6}) until the minimum value $p = 5 \times 10^{-8}$ during training. In each episode, the training algorithm rollouts 6 parallel trajectories on the current policy and the *differental* reward is enabled by default. In addition, the flow arrival rate λ is set to 1 and the maximum number of concurrent, active flows is configured to 50.

The results show that the LFS scheduler learns a reasonable policy to admit flows. Specifically, the total number of flow admissions and and the average reward received during training of the LFS scheduler are shown

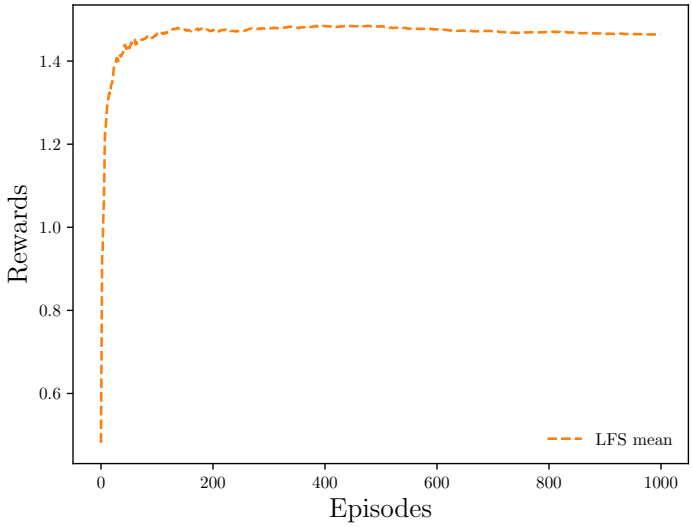

Figure 4.4: Average reward during training of LFS scheduler (Based on [39] ©2021 IEEE)

in Fig. 4.3 and 4.4, respectively. As expected, the average reward increases with every training episode. However, after approximately 150 episodes, the average reward stops increasing, which indicates that the LFS scheduler has converged to a scheduling policy.

Network load

The trained LFS scheduler is compared with a greedy scheduling algorithm and its variants. The greedy variants schedule flows based on a value drawn from binomial distribution with different success probabilities, that is, $p \in \{0.95, 0.9, 0.8, 0.7\}$. Since the arrival time α_f, data volume v_f, and deadline d_f of flows are samples of different distributions, the test episodes are passed different seeds from training episodes to produce unseen, new samples. The arrival rate λ of flows is varied to produce a wide range of *network loads*. For example, Fig. 4.5-4.10 shows flow admissions by various scheduling schemes at different flow arrival rates, that is, $\lambda \in \{1, 2, 5, 10, 15, 20\}$. The result shows that the LFS scheduler admits more flows than the greedy schedulers,

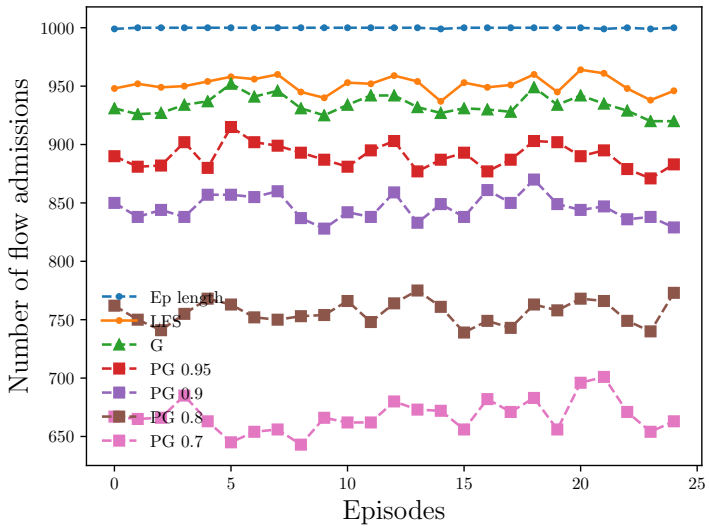

Figure 4.5: Flow admissions by various scheduling schemes under different network loads for example, flow arrival rate $\lambda = 1$ (Based on [39] ©2021 IEEE)

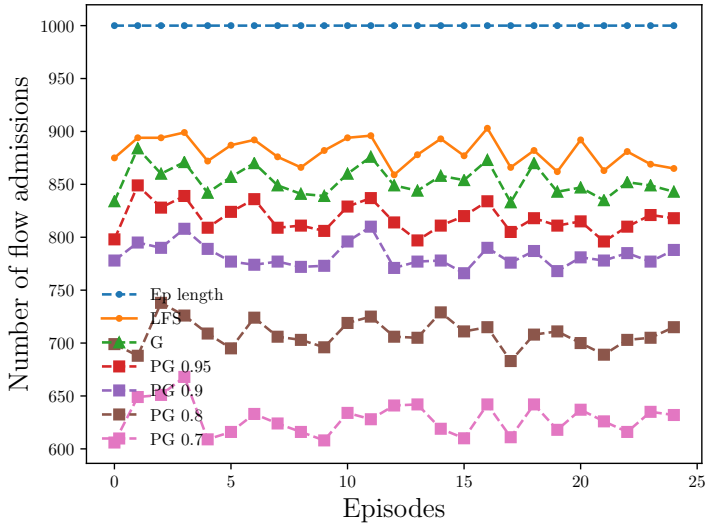

Figure 4.6: Flow admissions at flow arrival rate $\lambda = 2$ (Based on [39] ©2021 IEEE)

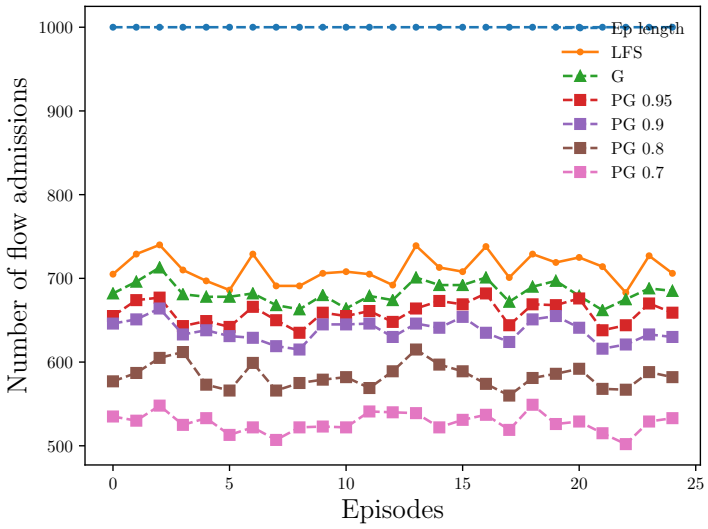

Figure 4.7: Flow admissions at flow arrival rate $\lambda = 5$ (Based on [39] ©2021 IEEE)

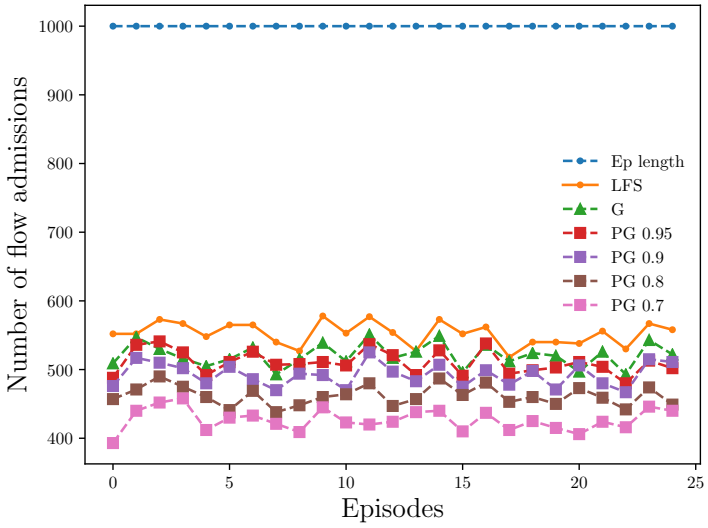

Figure 4.8: Flow admissions at flow arrival rate $\lambda = 10$ (Based on [39] ©2021 IEEE)

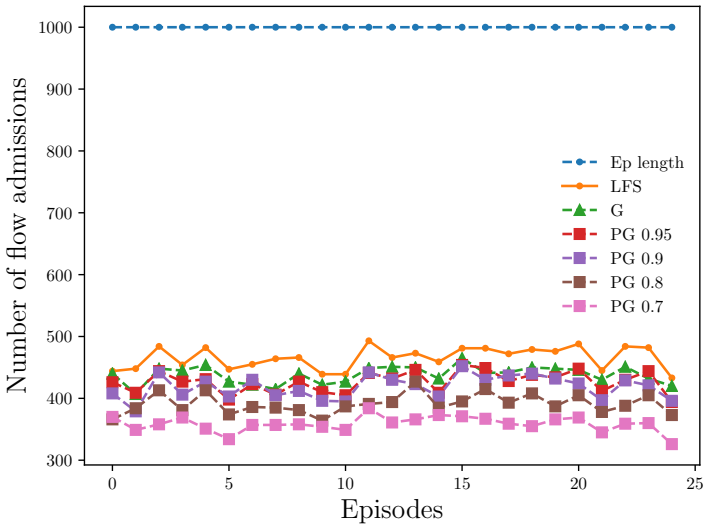

Figure 4.9: Flow admissions at flow arrival rate $\lambda = 15$ (Based on [39] ©2021 IEEE)

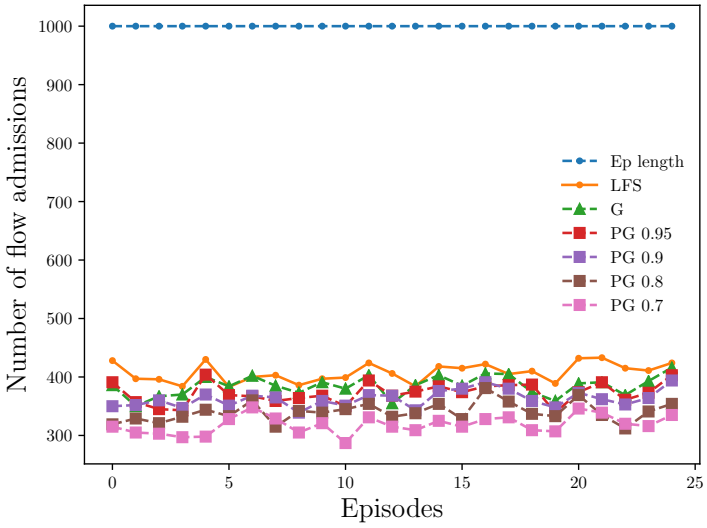

Figure 4.10: Flow admissions at flow arrival rate $\lambda = 20$ (Based on [39] ©2021 IEEE)

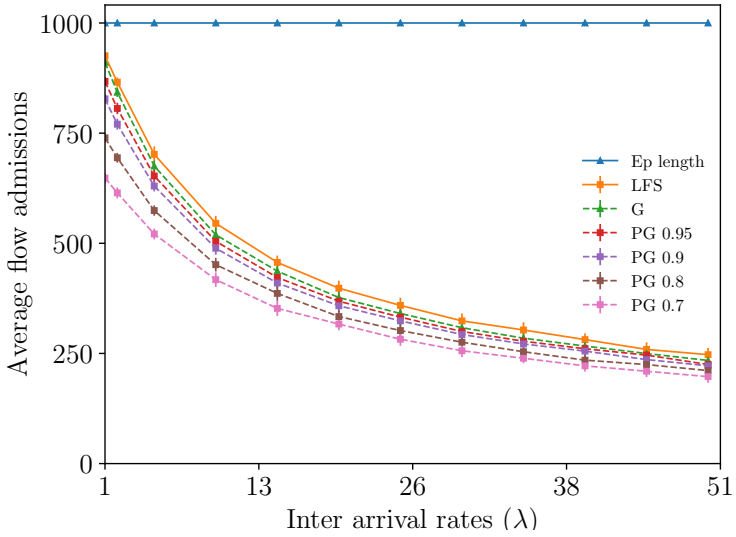

Figure 4.11: Average flow admissions under different network loads (Based on [39] ©2021 IEEE)

even under high network loads. In addition, Fig. 4.11 shows average flow admissions over many test episodes at different network loads. As expected, we see higher flow admissions under low network loads because it is likely that multiple flows are concurrently active. Each datapoint in Fig. 4.11 is an averaged value over 25 test episodes at a particular flow arrival rate λ. The dataset in each of the test episodes is unseen. We found that, for different network loads, the LFS scheduler admitted, on average, 1.05× (or 5 %) more flows than the best greedy scheduler G.

Number of flows that met their deadline

Since all schedulers are non-preemptive schedulers, admitted flows continue to receive link resources, that is, data rates, from the time they start executing till their completion. During testing, we found that all flows met their deadline as per our expectation.

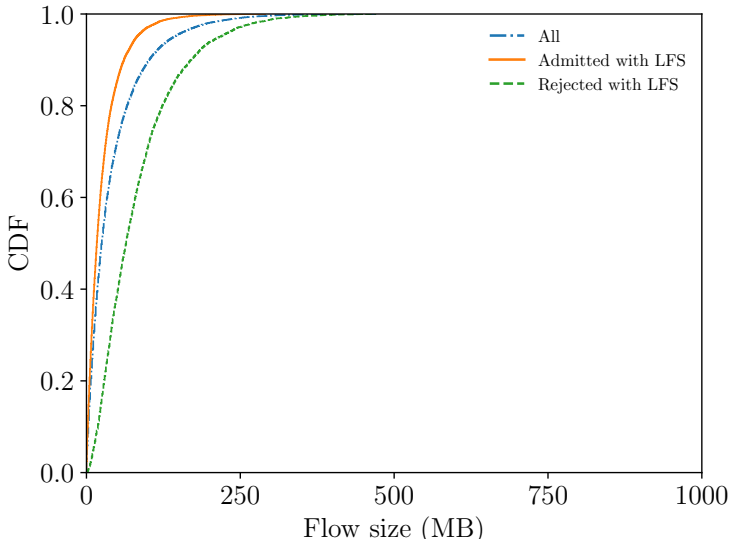

Figure 4.12: CDF of flow sizes of admitted and rejected flows by the LFS scheduler (Based on [39] ©2021 IEEE)

Performance Gain

We evaluate the gains of the LFS scheduler on a particular network load, in which the flow arrival rate is set to $\lambda = 5$. In the test, we record the size of both admitted and rejected flows and plot the cummulative distribution function (CDF). The CDF of flow sizes showed that the trained LFS scheduler mostly admitted smaller flows in the test. This is evident in Fig. 4.12, which implies that the policy priortized smaller flows over large flows for the designated performance objective and, if necessary, rejected large flows to admit not-yet-arrived smaller flows.

4.5 Concluding Remarks

In this chapter, we have demonstrated that the LFS scheduler automatically learns flow structures using deep RL and outperformed the greedy

flow scheduling heuristics under varying network load. In addition, it is practically feasible to quickly adapt to different performance objectives by simply retraining the scheduling policy on the redesigned reward function. For example, with reward $R_e = -|\bar{\mathbb{F}}_e|$, LFS can learn a scheduling policy to *minimize the average flow completion time* or it can *maximize the network utilization* with reward $R_e = \sum_{f \in \bar{\mathbb{F}}_e} r_f$ at each timestep.

In the next chapter, we argue that the LFS scheduler can be generalized to a network with coflows and a coflow scheduler can learn scheduling policies for the specified high-level performance objective.

Chapter 5

Learning Coflow Admissions

This chapter of the thesis is based on the revised text of our workshop paper; therefore, some portions of the chapter contain verbatim content, for example, figures and tables used in the paper:

Asif Hasnain and Holger Karl. Learning coflow admissions. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2021

In this chapter, we study the impact of stochastic network traffic on successful coflow admissions in existing heuristics [37, 22]. Specifically, we propose a new coflow scheduler, namely learning coflow scheduling (LCS), to *maximize coflow admissions while meeting their deadlines*.

My main contribution in this paper was to design, implement, and evaluate performance of the LCS coflow scheduler. Firstly, I formulated the coflow scheduling problem as a Markov decision process (MDP) and designed the necessary state space, action space, and the reward function. The problem is then solved using deep reinforcement learning (DRL) where the coflow scheduling agent directly interacts with the environment. For this purpose, I implemented both the scheduling agent and the flow-level simulator as environment. Moreover, I evaluated the performance of trained LCS through large-scale trace-driven simulation and compared its results with the state-of-the-art heuristic.

Starting from Section 5.1, the chapter first gives a quick overview and motivation of the problem, going beyond the high-level description of the introductory chatpers. It then discusses related literature work in Section 5.2. Both the application and the network model are defined in Section 5.3. In

Section 5.4, the coflow scheduling is formulated as a MDP. In addition, the design and implementation of the LCS coflow scheduler is covered in Section 5.4. The experimental evaluation of performance is then discussed in Section 5.5. Finally, the chapter ends with a few concluding remarks in Section 5.6.

5.1 Motivation

Typically, coflow scheduling is performed by carefully crafted heuristics [37, 59, 22], which assign network resources to the coflow's constituent flows. These heuristics optimize for different performance objectives like *minimizing* average CCTs or *meeting* coflow deadlines – where a coflow is only completed once all its constituting flows are completed within that deadline. Although they exploit communication patterns (or features) inside the network, coflow heuristics are susceptible to stochastic network traffic [37, 22], that is, they admit fewer coflows and often do not perform well for the specified performance objective, as shown in Fig. 5.1. In addition, a coflow heuristic written for one performance objective does not perform well on another performance objective (for a different workload) [22] and, practically, developing a universal coflow scheduler to reply majority of coflow scheduling algorithms is difficult yet a desired solution [72]. Therefore, in this chapter, we question the common practice of writing handcrafted coflow heuristics and give a proposal to replace them with a coflow scheduler that *can automatically learn patterns in network traffic and adapt scheduling decisions.*

Specifically, we consider scheduling for deadline-sensitive coflows without designing conventional heuristics. In the previous Chapter 4, we looked at the simpler setting to learn flow scheduling on a single link, where an RL-based flow scheduler learns to admit more flows than a greedy flow scheduling heuristic. In this chapter, we argue that such learning can be generalized to the big-switch network model [37, 22, 27] with coflows. We propose a new framework, namely Learning Coflow Scheduling (LCS), to learn admission policies to *maximize* coflow admissions while *meeting* their deadlines. The task is to either admit or reject an arriving coflow. If there are

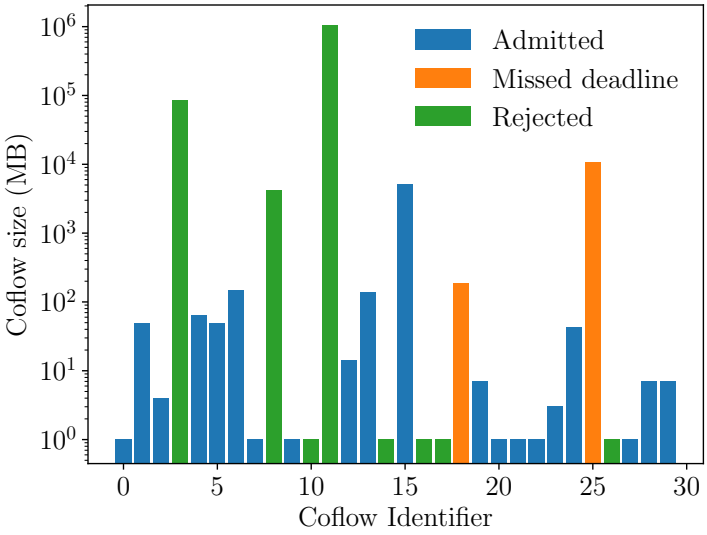

Figure 5.1: Varys admits and finishes 20 coflows within their deadlines (Based on [38] ©2021 IEEE)

insufficient resources for an admitted coflow (where coflows compete for limited link resources), the admitted coflow will miss its deadline. Hence, coflow admissions should not be done greedily or by any other obvious heuristic. Once a coflow finishes or exceeds its deadline, its resources are released and redistributed to other coflows competing for the same links.

We use a DRL approach to admit coflows; the data rate is assigned to admitted (co-)flows by the agent's environment using our heuristic from Chapter 3. DRL is well suited for learning such policies because it allows a coflow scheduling agent to directly interact with the network environment. It automates learning of key coflow patterns through end-to-end training on the observed network states such that the coflow scheduling agent can adapt decisions based on the feedback (reward) from the network environment. We will reward successfully completed coflows and (severely) punish coflows missing their deadlines. If a coflow is rejected, the environment produces a reward value of zero for the scheduling agent.

Learning an online admission policy (to maximize coflow admissions) is

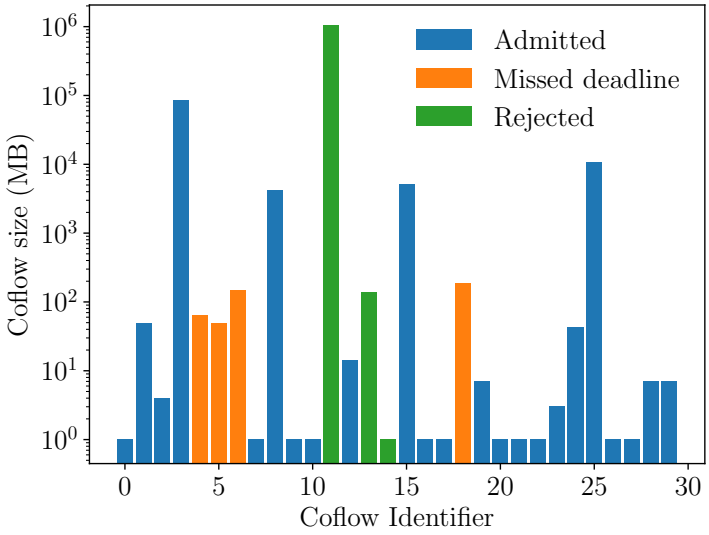

Figure 5.2: the RL-based scheduling agent LCS admits as much as 23 coflows, which finish within their deadlines (Based on [38] ©2021 IEEE)

challenging as coflows arrive stochastically and the flow volumes come from an unknown distribution, for example, the cache workload [78], the data mining workload[33], the Hadoop workload, or the web search workload [5], as shown in Fig. 2.1. Consider, for example, the first 30 coflows from the Facebook trace [22]. In this example, deadlines are set to the time the coflow would need if it were executed alone in an empty network. The well-known Varys coflow scheduling scheme – our reference case – admits and successfully completes only 20 coflows out of total 30 coflows (Fig. 5.1). On the other hand, the trained coflow scheduler, LCS, admits more coflows (that is, 23) than Varys while still *meeting* their deadlines (Fig. 5.2). Such a better scheduling order for coflows can be achieved for different workloads if the coflow scheduling agent learns policies from coflow arrival patterns in different network states.

5.2 Related Work

Although there are a few RL-based proposals for coflow scheduling [15, 97, 90], they only minimize average CCT (Chapter 2). In addition, there are many non-ML proposals [108, 63, 22] to *maximize coflow admissions while meeting their deadlines* (as discussed in Chapter 2 and 3), we do not find any literature work that employs ML techniques to optimize for this performance objective.

5.3 Model

5.3.1 Application Model

In this chapter, we consider the performance objective to *maximize the number of admitted coflows that meet their deadlines*. In general, a data-parallel application makes a request to either admit or reject a coflow $c \in C$ (Table 2.1). The coflow request consists of multiple flows F_c, an arrival time α_c, and a relative deadline d_c. Each individual flow $f_c^{i,j}$ has a source i, destination j, and an individual data volume v_f. We assume that the information about flows F_c within a coflow $c \in C$ is available on their arrival.

5.3.2 Network Model

Similar to the network model in Section3.2.1, we abstract out the datacenter network [115] as a non-blocking "big switch" [22, 27]. At any time t_e, multiple coflows are competing for resources on multiple links via their constituting flows. A flow's data rate r_f may vary during execution as new coflow requests arrive or an active coflow leaves the network upon completion Γ_c. We assume that a coflow can have at most one flow between any (source, destination) pair (multiple flows would simply be aggregated into one flow). We assume that the coflow scheduler is non-preemptive, that is, any active coflow continues to receive resources (that is, data rate) as assigned by the heuristic [37]. Since network resource utilization is

Table 5.1: Notation for learning model

Q	Number of different coflow sequences as defined in Table 2.1
C_k	The kth set of coflow arrivals, where $k \in \{1, \cdots, Q\}$ and C_k corresponds to C in Table 2.1
E_k	Number of scheduling events in C_k, where E_k corresponds to E in Table 2.1
τ	Episode length
N	Number of different sample trajectories
s_e	Fully observed state at time t_e
$a_e \in \{0, 1\}$	Action taken in state s_e
R_e	Reward (or penalty) received on action a_e in state s_e
$\bar{R}_e$	Time-average reward at time t_e
$R_e - \bar{R}_e$	*Differential* reward
G_e	*Differential* return (sum of *differential* rewards from state s_e to the terminal state s_E)
s_e^j	State s_e^j in trajectory j
a_e^j	Action a_e^j in trajectory j
R_e^j	Reward R_e^j in trajectory j
$\pi_\theta(a_e \mid s_e)$	Policy network
θ	Parameters of the policy network $\pi_\theta(\cdot)$
β_θ	Learning rate (or step size) of policy network $\pi_\theta(\cdot)$
$\mathbb{P}(s_{e+1} \mid s_e, a_e)$	State transition probability function
$J(\theta)$	Performance objective for policy network $\pi_\theta(\cdot)$
$\nabla_\theta J(\theta)$	Policy gradient

important for reducing CCTs, we use a work-conserving scheme to distribute remaining resources (that is, data rate) after minimum resource allocation.

5.4 Design & Implementation

5.4.1 RL model

The coflow scheduling problem is modeled as a discrete-time markov decision process (MDP), which is defined as a sequence of state, action, and reward, that is, (s_e, a_e, R_e), where e is a scheduling event (Table 5.1). Specifically, the state transition is assumed to satisfy the Markov property, that is, the probability of the current state s_e only depends on the action a_e taken in the previous state s_{e-1}.

The MDP is solved using DRL where the coflow scheduling agent learns by directly interacting with the environment.

State space: The fully observed network state s_e at a coflow arrival event at time t_e is a flat feature matrix (that is, no feature reduction by, for example, Principal Component Analysis [50]) describing both new and active coflows $\bar{\mathbb{C}}_e$. The feature matrix has three dimensions, which are indexed by the source, destination, and active coflow number (we reuse coflow numbers once a coflow is finished to limit the size of this matrix). Specifically, an entry in the feature matrix for a flow $f_c^{i,j}$ describes the following flow features (for both new and active coflows):

- The flow's *remaining* data volume $\bar{v}_f \in \mathbb{R}_{\geq 0}$. For new flows, its remaining data volume of course equals its total data volume $v_f \in \mathbb{R}_{\geq 0}$.
- The coflow's deadline $d_f \in \mathbb{R}_{>0}$

The number of concurrent, active coflows $|\bar{\mathbb{C}}_e|$, at any time t_e, is limited by a maximum value. This is justifiable as the number of concurrently active coflows $|\bar{\mathbb{C}}_e|$ in the network can vary; datacenters, however, have reported 10s of concurrently active coflows [78] at most. Since the number of active flows (and, in general, coflows) is variable, the feature matrix has zero padding for combinations of source, destination and coflow index that currently do not exist.

Action space: The action space $a_e \in \{0, 1\}$ is discrete, with only actions $a_e = 1$ and $a_e = 0$ representing the decision to either admit or reject a new coflow in state s_e. The scheduling decisions for the new coflows are taken by the policy network (Section 5.4.1).
Rate allocation and reacting to flow/coflow completion (FCT/CCT) events are handled by the network environment (Section 5.5.1).
Reward function: Since coflow scheduling is a continuous task, an average reward is better suited than a total reward and it maximizes $\lim_{\tau \to \infty} 1/\tau \sum_{e=0}^{\tau} R_e$ [91, Ch. 10], where τ is an episode length. Specifically, the network environment gives a *differential* reward $R_e \leftarrow R_e - \bar{R}_e$ [91, Ch. 10.3, Ch. 13.6] to the scheduling agent, where $\bar{R}_e$ is the time-average reward at time t_e. Since the high-level objective is to *maximize* coflow admissions while *meeting* their deadlines, the reward function is formulated as follows.
On *admitting a new coflow* $c \in C$, the network environment produces the reward $R_e = (1/d_c + 1/\sum_{f \in F_c} v_f + 1/|F_c| + 1/(\max\{v_1, \cdots, v_{|F_c|}\}))$ for the scheduling agent. The fine-grained reward R_e consists of key coflow features, that is, the deadline d_c, the total volume $\sum_{f \in F_c} v_f$, the number of constituent flows $|F_c|$, and the volume of the largest flow $\max\{v_1, \cdots, v_{|F_c|}\}$. The reciprocal of these feature values indicates that we prefer a coflow with a short deadline, low total volume, as few flows as possible, and low volume for the largest flow to *maximize* coflow admissions. If the network environment did not provide a fine-grained reward (like above), the scheduling agent would find it difficult to learn to optimize the desired objective, that is, *maximize* coflow admissions. For instance, we noticed that a coarse-grained reward R_e like +1 does not produce a desired policy.
If an active coflow fails to meet its deadline, the network environment penalizes the scheduling agent with $R_e = -(1/d_c + 1/\sum_{f \in F_c} v_f + 1/|F_c| + 1/(\max\{v_1, \cdots, v_{|F_c|}\}))$. The penalty R_e consists of the same coflow features; however, here the value is negated. The penalty ensures that the scheduling agent does not converge to a greedy policy in which the coflow scheduling agent admits all coflows, irrespective of whether a coflow can meet its deadline or not. If the penalty is scaled up, we noticed that it impacts overall coflow admissions, which are being *maximized*. For example, when the penalty R_e is scaled up by multiplying it with a factor $(-1, -\infty)$ to punish it severly for missing a coflow deadline, the scheduling agent often reduced overall successful coflow admissions.

Policy network: Since the state space is large and continuous, learning a coflow scheduling policy was quite challenging. LCS uses a policy gradient algorithm [104] to directly learn a coflow scheduling policy $\pi_\theta(\cdot)$. The coflow scheduling policy $\pi_\theta(a_e|s_e)$ is defined as the probability of taking action a_e in state s_e, where θ are the network parameters. The actual action $a_e \in \{0, 1\}$, to admit or reject a new coflow in state s_e, is then sampled from action probabilities using the Gumbel-Softmax distribution [46]. Typically, policy gradient methods learn by performing gradient descent on the neural network parameters θ using the loss from *differential* return in training. If the action produces better-than-time-average *differential* return (that is, more successful coflow admissions), the scheduling agent increases the probability of taking action a_e in state s_e, otherwise, it decreases the probability (of taking action a_e in state s_e). Here, the policy gradient $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_e|s_e)$ provides the direction to the policy network in parameter space.

5.4.2 Training Algorithm

The policy network is trained using the Monte Carlo [40] policy gradient algorithm 5. Since the initial scheduling policy is assumed to be poor, the earlier training episodes are terminated stochastically to help learning on online coflow arrivals (line 3). In each episode, the training algorithm rolls out multiple trajectories $j \in [1, \cdots, N]$ on current scheduling policy $\pi_\theta(a_e|s_e)$ (line 5) and computes return (line 7) from *differential* rewards. Based on the *differential* return, a baseline [101] is computed to estimate the average *differential* return (line 8). The baseline is unique for every set of coflows C to reduce variance in policy gradient from learning on different sets of coflow arrival sequences $C_k, k \in \{1, \cdots, Q\}$. It is then subtracted from the *differential* return (in a particular episode) to estimate the quality of return in comparison to the average *differential* return. Specifically, it helps to reduce variance in the policy gradient from rewards (line 10).

Based on the better-than-time-average or worse-than-time-average *differential* return, the training algorithm updates network parameters θ of the policy network via gradient descent (line 12), where β_θ is the learning rate (or step size) for the network parameters θ. Since the full-batch gradient $\nabla_\theta J(\theta)$ update of the loss from all samples, that is, sequence of (s_e, a_e, R_e), of an

Algorithm 5 Monte Carlo policy gradient algorithm for training

Input: Policy network $\pi_\theta(\cdot)$

Initialize network parameters θ, learning rate $\beta_\theta > 0$, and the time-average reward $\bar{R}_e = 0$ {θ are initialized with the Glorot uniform initializer}
for each episode **do**
  Sample an episode length τ
  Take a set of coflows C in coflow arrival sequences $C_k, k \in \{1, \cdots, Q\}$
  Rollout multiple trajectories $j \in [1, \cdots, N]$ on current policy $\pi_\theta(\cdot) \sim \{s_1^j, a_1^j, R_1^j, \cdots, s_E^j, a_E^j, R_E^j\}$ until τ
  $\bar{R}_e \leftarrow \bar{R}_e + 1/(N \cdot E) \sum_{j=1}^{N} \sum_{e=1}^{E} R_e^j$;
  Calculate *differential* return $G_e^j = \sum_{e'=e}^{E} R_{e'}^j - \bar{R}_{e'}$
  $b_e = 1/N \sum_{j=1}^{N} G_e^j$ {Compute baseline}
  for $j \in [1, \cdots, N]$ **do**
    $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_e|s_e)(G_e^j - b_e)$;
  end for
  $\theta \leftarrow \theta + \beta_\theta \nabla_\theta J(\theta)$; {Update network parameters θ}
  $\tau \leftarrow \tau + \epsilon$
end for

episode is memory-intensive (because of large state space), the network parameters θ are updated using mini-batching.

Although the scheduling agent is trained using the episodic algorithm, it should work in any production environments where coflows scheduling is a continuous task.

5.5 Experimental Evaluation

We evaluated LCS using a flow-level simulator as an environment with a production workload from Facebook [20, 22]. Our experiment results answer the following questions: (1) Does LCS learn a reasonable coflow scheduling policy to *maximize* coflow admissions while *meeting* coflow deadlines? (2) How does LCS scheduler perform compared with the heuristics?

5.5.1 Methodology

Workload

Similar to the workload in Chapter 3, we use a production trace from Facebook [20, 22] for our experimental evaluation. Specifically, we use the relative deadline factor to provide some slack for the data tranfers (Section 3.7.1).However, Varys [22] uses slightly different notion of deadlines: given a factor between 0.1 and 10, the actual deadline results from multiplying the coflow duration with a random variate from a uniform distribution $U(1, 1 + x)$. For fair comparison, we use our notion of deadlines for both coflow schedulers.

Network Environment

In the network environment (as shown in Fig. 5.3), we developed a custom-tailored flow simulator to represent the network resources and the coflows

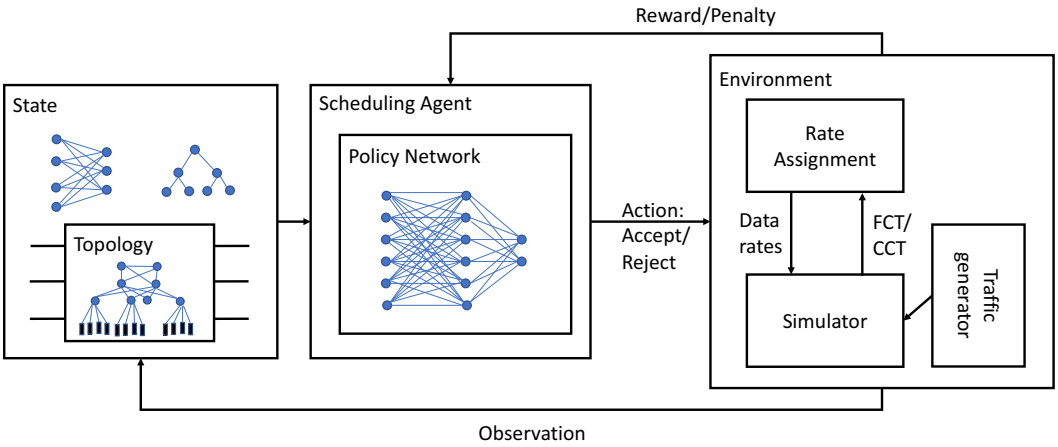

Figure 5.3: Scheduling agent with environment (Based on [38] ©2021 IEEE)

using them. In addition, we complement the flow simulator by a rate assignment component that deals with the actual resource allocation decisions, once the RL scheduling agent has admitted a coflow.

Simulator: Similar to the flow simulator in Chapter 3, the simulator in the network environment is driven by the scheduling decisions on coflow arrival and coflow/flow departure events. However, when a coflow arrives, the coflow scheduling agent only makes decision to either admit or reject the coflow request and then delegates control to the rate assignment component. In addition, whenever a flow or all flows of a coflow departs, the simulator delivers the corresponding events to the rate assignment component. The rate assigment component is then free to reassign those resources as it sees fits (and to inform the simulator about changes in rate allocation). If a coflow misses the deadline, all its remaining flows are dropped from the network and the occupied resources are released.

Rate allocation in the environment: While the LCS agent decides all admissions, the flow rates are computed and assigned within the environment. These rates of active flows are computed by our (deterministic, non-ML based) heuristic [37] (in Chapter 3). The heuristic works by iterating over the earliest completion times of active coflows (earlier than the new coflow's deadline) to find possible data rates such that the constituent flows (of the new coflow) finish within their deadline. We use it in a work-conserving

way, that is, when flows complete or coflows are dropped, we reassign the freed-up resources to other flows competing for those links.
Simulator input: We drive the simulator from production traces as made available by, for example, Facebook [22].

LCS

We build the LCS admission agent using a multilayered perceptrons (MLPs) neural network [79, 103] because MLPs are good function approximators and classifiers. Since the state space in our problem is large and continuous, finding the right number of hidden layers and neurons is challenging because a large policy network is computationally and memory-intensive. After trying different numbers of layers and neurons, we are able to learn a reasonable coflow scheduling policy with a small neural network. Specifically, there are two hidden layers in the network, with 32 and 16 neurons, respectively, and their activation function is set to ReLU [75].

The policy network uses Adam optimizer [53] to update its parameters, where the learning rate β_θ for network parameters is set to 10^{-4}. The gradients of the neural network are updated in mini-batches and the batch size during training of the trace is set to 128. In addition, the entropy cofficient for *exploring* the coflow scheduling policy is intialized to 0.01; it is gradually decayed with value 10^{-3} until the minimum value 10^{-4} during training of the policy network. The policy network outputs action probabilities, for which an action is sampled using the Gumbel-Softmax distribution [46].

Since the input feature matrix has zero padding for a fixed length, LCS passes input through a *masking* layer to inform the model of zero padding so that it ignores these invariant values while making scheduling decisions. With masking turned off, the agent finds it difficult to learn a coflow scheduling policy. In fact, after a few training episodes, it learned to greedily admit or reject all coflows, depending on the reward function. Since the keras *Flatten* layer in policy network does not support masking, we have extended it to pass input mask through the policy network.

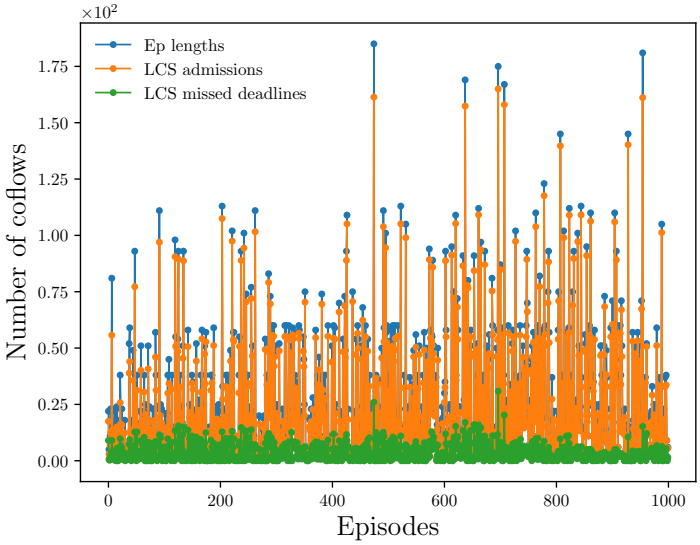

Figure 5.4: Total coflow admissions and missed deadlines during training of LCS scheduler, where the deadline factor is set to 1.0 (Based on [38] ©2021 IEEE)

5.5.2 Simulation Results

Training

The LCS scheduler is trained using the algorithm 5. The algorithm runs 1000 training episodes and, in each episode, it rollouts 8 parallel trajectories, that is, sequences of (s_e, a_e, R_e). The trajectories are stochastically terminated after the maximum time τ to limit scheduling to short set of coflows because the initial coflow scheduling policy is poor (for example, from random initialization of network parameters) and training algorithm can run into problems like exploding and vanishing policy gradient. The episode lengths are gradually increased during training to make challenging decisions for large set of coflows. Specifically, the reset probability of episode length $p = 5 \cdot 10^{-5}$ decays (with value $4 \cdot 10^{-10}$) until the minimum value $p = 5 \cdot 10^{-8}$ during training of the policy network. The *differential* reward is enabled to reduce noise in rewards and the maximum number of concurrent, active coflows is set to 10 for fixed input to the MLP networks. The results in Fig. 5.4

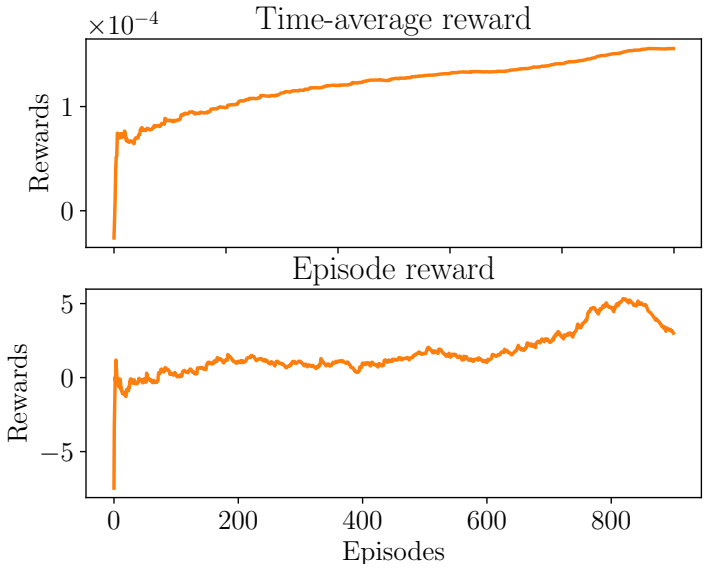

Figure 5.5: Time-average rewards $\bar{R}$ and episode rewards R received during training of LCS scheduler (Based on [38] ©2021 IEEE)

show that the LCS scheduler learns a reasonable policy to admit coflows while meeting their deadlines. Specifically, the percentage of *successful* coflow admissions increases during training.

The Fig. 5.5 plots both the time-average reward and per-episode mean reward received during the training of the LCS scheduler. As expected, the time-average reward increases during training; however, after approximately 900 episodes, the time-average reward stops increasing, which indicates that the policy network has converaged to a coflow scheduling policy.

Comparison

We compare our LCS scheduler with the Varys [22] heuristic, which admits coflows if it can allocate minimum resources (that is, data rates) from remaining resources to meet coflow deadlines. Here, we use the *relative deadline factor* to compare both coflow schedulers. The result in Fig. 5.6 shows that the LCS scheduler successfully admits more coflows than Varys [22]. Each

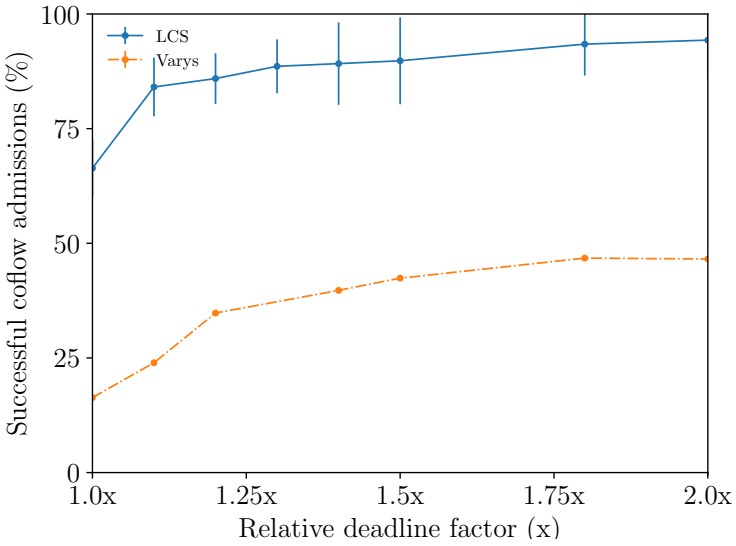

Figure 5.6: Percentage of successful coflow admissions by the two coflow schedulers under different network load (Based on [38] ©2021 IEEE)

data point of LCS in Fig. 5.6 is the average over successful coflow admissions in 25 different test episodes (Fig. 5.7). The percentage of successful coflow admissions by Varys are surprisingly lower than the successful coflow admissions reported in the paper. When we re-run Varys with their notion of coflow deadlines, the percentage of successful coflow admissons were still less than the reported results. For example, for $x = 1$, Varys successfully admitted 45.44 % of the total coflows compared to 75 % [22] – even though they have a lenient criteria for considering a successful coflow admission, that is, if $\Gamma_c - d_c < 100$ holds, the coflow $c \in C$ has met its deadline. If we consider 75 % successful coflow admissions for $x = 1$ from Varys, LCS still has higher successful coflow admissions (that is, 94.3 % at the relative deadline factor 2.0).

Fig. 5.7 plots multiple test episodes at different *deadline factors*, varied from 1.0 to 2.0. In each test episode, a different seed is used. We see variance in successful coflow admissions because the actions are sampled from a distribution (that is, Gumbel-Softmax [46]). As expected, the percentage of successful coflow admissions increases with the increase in deadline

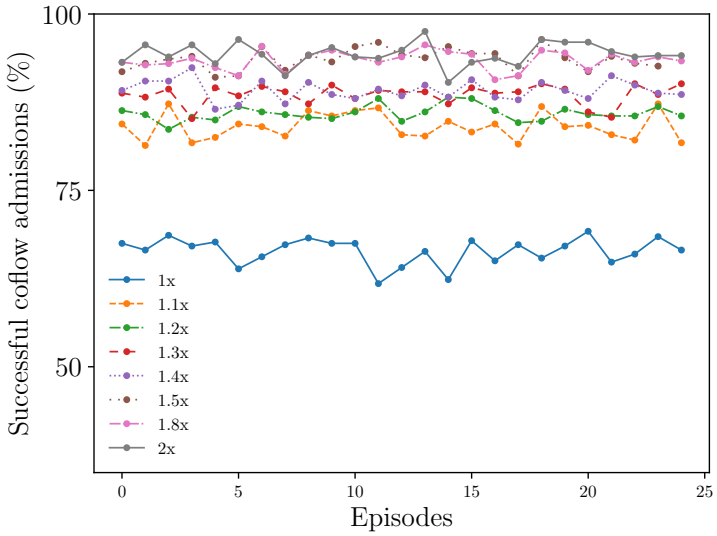

Figure 5.7: Percentage of successful coflow admissions by LCS at different relative deadline factors (Based on [38] ©2021 IEEE)

factors.

5.6 Concluding Remarks

In this chapter, we presented LCS, a DRL-based admission scheme for coflows with deadlines. Specifically, LCS learned a scheduling policy for the specified high-level performance objective to *maximize* coflow admissions while meeting their deadlines. It outperformed the competing heuristic Varys on a relevant production workload.

Chapter 6

Conclusion

Can a coflow scheduler automatically learn online scheduling policies to improve network performance of data-parallel applications? This dissertation primarily answers this question by demonstrating the LCS scheduler. Specifically, the LCS scheduler achieves the desired performance objective — to maximize coflow admissions while meeting their deadlines — using reinforcement learning. The presented RL techniques enable us to make informed decisions in the presence of stochastic coflow arrivals in data traffic. Besides, we have presented a new coflow heuristic and an RL-based flow scheduler for the same performance objective and outperformed competing schedulers. In addition, we have evaluated all schedulers through large-scale trace-driven simulation on production traces.

6.1 Future Work

This dissertation has demonstrated that flow and coflow schedulers can effectively learn scheduling policies using reinforcement learning. In this chapter, we revisit the design choices of the LFS and LCS scheduler to point out their limitations and highlight some of the interesting research directions for future work.

Learning Information-agnostic Coflow Scheduling: Like many existing proposals [114, 22], our approaches assume that schedulers have prior knowledge (for example, number of flows, flow sizes) at arrival time. However, in some applications [94, 44], prior knowledge of a coflow is unknown

for which various information-agnostic coflow schedulers [102, 59, 20] are proposed. However, automatically learning these coflow scheduling policies using RL [15, 97] is an interesting avenue for future research.

Joint Optimization of Job and Coflow Scheduling: While we consider only single-stage coflows in this dissertation, coflows can have dependencies in a multi-stage job DAG [20], that is, a coflow starts-after or finishes-before other coflows. Therefore, DRL-based coflow scheduling policies with dependency constraints can be further explored (for instance, DeepWeave [90]).

Distributed Coflow Scheduling and Routing: Although we abstract out the data center network as a non-blocking switch in our approaches, integrating coflow routing with scheduling has been an interesting research topic [85, 57, 116]. In fact, CoRBA [85] has recently formulated and solved the problem to reduce the average CCT as a mixed-integer nonlinear program. In addition, RAPIER [116] has earlier shown that the average CCT can be reduced by integrating coflow scheduling and routing; however, their approach makes centralized decisions using global network information, which is not a scalable approach in large networks. Therefore, learning to make distributed decisions using multiple RL agents in a large network is an interesting problem for future research.

In-network Coflow Scheduling: With the recent advances in programmable packet scheduling [88], providing in-network support to coflow scheduling at end-hosts is another interesting research topic. For instance, pCoflow [41] has shown that such integration is effective to reduce average CCT.

Theoratical Evaluation: Although the theoratical analysis of coflow scheduling to minimize average CCT has been extensively studied [3, 84, 52], the theoretical investigation of coflow scheduling with deadline constraints is an open research topic for future work. In addition, a near-optimal information-agnostic coflow scheduler can be designed as future work.

References

[1] Apache Hadoop. http://hadoop.apache.org/. Date last accessed 02-June-2021.

[2] Gurobi Solver. http://www.gurobi.com/. Date last accessed 02-June-2021.

[3] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 16–29, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.

[7] Md Zahangir Alom, Tarek M. Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Mahmudul Hasan, Brian C. Van Essen, Abdul A. S. Awwal, and Vijayan K. Asari. A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8(3), 2019.

[8] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), May 2020.

[9] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 455–468, USA, 2015. USENIX Association.

[10] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.

[11] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[13] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 174–187, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 191–205, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Tianba Chen, Wei Li, YuKang Sun, and Yunchun Li. M-drl: Deep reinforcement learning based coflow traffic scheduler with mlfq threshold adaption. *International Journal of Parallel Programming*, May 2021.

[16] Mosharaf Chowdhury. *Coflow: A Networking Abstraction for Distributed Data-Parallel Applications*. PhD dissertation, University of California, Berkeley, 2015.

[17] Mosharaf Chowdhury, Samir Khuller, Manish Purohit, Sheng Yang, and Jie You. Near optimal coflow scheduling in networks. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 123–134, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 407–424, USA, 2016. USENIX Association.

[19] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 31–36, New York, NY, USA, 2012. ACM.

[20] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 393–406, New York, NY, USA, 2015. ACM.

[21] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 98–109, New York, NY, USA, 2011. Association for Computing Machinery.

[22] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

[23] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. volume 41, pages 205–220, 10 2007.

[27] N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merive. A flexible model for resource management in virtual private networks. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 95–108, New York, NY, USA, 1999. ACM.

[28] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 99–112, New York, NY, USA, 2012. Association for Computing Machinery.

[29] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135. PMLR, 06–11 Aug 2017.

[30] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[31] D. Ghosal, S. Shukla, A. Sim, A. V. Thakur, and K. Wu. A reinforcement learning based network scheduler for deadline-driven data transfers. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.

[32] G. R. Ghosal, D. Ghosal, A. Sim, A. V. Thakur, and K. Wu. A deep deterministic policy gradient based network scheduler for deadline-driven data transfers. In *2020 IFIP Networking Conference (Networking)*, pages 253–261, 2020.

[33] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[34] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, May 2015. USENIX Association.

[35] Tuomas Haarnoja, Aurick Zhou, P. Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

[36] Martin T. Hagan, Howard B. Demuth, and Mark Beale. *Neural Network Design*. PWS Publishing Co., USA, 1997.

[37] A. Hasnain and H. Karl. Coflow scheduling with performance guarantees for data center applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 850–856, 2020.

[38] Asif Hasnain and Holger Karl. Learning coflow admissions. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2021.

[39] Asif Hasnain and Holger Karl. Learning flow scheduling. In *2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–6, 2021.

[40] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 04 1970.

[41] Cristian Hernandez Benet, Andreas J. Kassler, Gianni Antichi, Theophilus A. Benson, and Gergely Pongracz. Providing In-network Support to Coflow Scheduling. *arXiv e-prints*, page arXiv:2007.02624, July 2020.

[42] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[43] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 127–138, New York, NY, USA, 2012. ACM.

[44] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 59–72, New York, NY, USA, 2007. Association for Computing Machinery.

[45] Akshay Jajoo, Rohan Gandhi, and Y. Charlie Hu. Graviton: Twisting space and time to speed-up coflows. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[46] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. 2017.

[47] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015*

ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 435–448, New York, NY, USA, 2015. Association for Computing Machinery.

[48] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. Eyeq: Practical network performance isolation at the edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 297–311, Lombard, IL, 2013. USENIX.

[49] Huiling Jiang, Qing Li, Yong Jiang, GengBiao Shen, Richard Sinnott, Chen Tian, and Mingwei Xu. When machine learning meets congestion control: A survey and comparison. *Computer Networks*, 192:108033, 2021.

[50] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[51] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Int. Res.*, 4(1):237–285, May 1996.

[52] S. Khuller, Jingling Li, Pascal Sturmfels, Kevin Sun, and Prayaag Venkat. Select and permute: An improved online framework for scheduling to minimize weighted completion time. *ArXiv*, abs/1704.06677, 2018.

[53] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[54] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–18, 2021.

[55] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[56] Wenxin Li, Xu Yuan, Wenyu Qu, Heng Qi, Xiaobo Zhou, Sheng Chen, and Renhai Xu. Efficient coflow transmission for distributed stream processing. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, page 1319–1328. IEEE Press, 2020.

[57] Yupeng Li, Shaofeng H.-C. Jiang, Haisheng Tan, Chenzi Zhang, Guihai Chen, Jipeng Zhou, and Francis C. M. Lau. Efficient online coflow routing and scheduling. In *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '16, page 161–170, New York, NY, USA, 2016. Association for Computing Machinery.

[58] Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.

[59] L. Liu, H. Xu, C. Gao, and P. Wang. Bottleneck-aware coflow scheduling without prior knowledge. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 50–55, 2020.

[60] Wai-xi Liu. Intelligent routing based on deep reinforcement learning in software-defined data-center networks. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2019.

[61] Jinjie Lu, Waixi Liu, Yinghao Zhu, Sen Ling, Zhitao Chen, and Jiaqi Zeng. Scheduling mix-flow in sd-dcn based on deep reinforcement learning with private link. In *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*, pages 395–401, 2020.

[62] Yuanwei Lu, Guo Chen, Larry Luo, Kun Tan, Yongqiang Xiong, Xiaoliang Wang, and Enhong Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[63] S. Ma, J. Jiang, B. Li, and B. Li. Chronos: Meeting coflow deadlines in data center networks. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2016.

[64] Yiqing Ma, Han Tian, Xudong Liao, Junxue Zhang, Weiyan Wang, Kai Chen, and Xin Jin. Multi-objective congestion control. *arXiv preprint arXiv:2107.01427*, 2021.

[65] Sridhar Mahadevan and Georgios Theocharous. Optimizing production manufacturing using reinforcement learning. In *Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference*, page 372–377. AAAI Press, 1998.

[66] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[67] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[68] Hongzi Mao, S. Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, M. Alizadeh, and E. Bakshy. Real-world video adaptation with reinforcement learning. *ArXiv*, abs/2008.12858, 2020.

[69] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[70] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. In *International Conference on Learning Representations*, 2019.

[71] 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, Aug 1999.

[72] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 501–521, Santa Clara, CA, March 2016. USENIX Association.

[73] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[74] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 188–201, New York, NY, USA, 2016. Association for Computing Machinery.

[75] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, Madison, WI, USA, 2010. Omnipress.

[76] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.

[77] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery.

[78] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. Association for Computing Machinery.

[79] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[80] Stefan Schneider, Ramin Khalili, Adnan Manzoor, Haydar Qarawlus, Rafael Schellenberg, Holger Karl, and Artur Hecker. Self-learning multi-objective service coordination using deep reinforcement learning. *IEEE Transactions on Network and Service Management*, pages 1–1, 2021.

[81] Stefan Schneider, Adnan Manzoor, Haydar Qarawlus, Rafael Schellenberg, Holger Karl, Ramin Khalili, and Artur Hecker. Self-driving network and service coordination using deep reinforcement learning. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–9, 2020.

[82] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1889–1897. JMLR.org, 2015.

[83] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 07 2017.

[84] M. Shafiee and J. Ghaderi. An improved bound for minimizing the total weighted completion time of coflows in datacenters. *IEEE/ACM Transactions on Networking*, 26(4):1674–1687, Aug 2018.

[85] Li Shi, Yang Liu, Junwei Zhang, and Thomas Robertazzi. Coflow scheduling in data centers: Routing and bandwidth allocation. *IEEE Transactions on Parallel and Distributed Systems*, 32(11):2661–2675, 2021.

[86] Yang Shi, Jiawei Fei, Mei Wen, Qun Huang, and Nan Wu. Metaflow: A better traffic abstraction for distributed applications. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1123–1130, 2019.

[87] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, L Robert Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[88] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[89] Penghao Sun, Zehua Guo, Sen Liu, Julong Lan, Junchao Wang, and Yuxiang Hu. Smartfct: Improving power-efficiency for data center networks with deep reinforcement learning. *Computer Networks*, 179:107255, 2020.

[90] Penghao Sun, Zehua Guo, Junchao Wang, Junfei Li, Julong Lan, and Yuxiang Hu. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 3314–3320. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[91] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[92] Bingchuan Tian, Chen Tian, Haipeng Dai, and Bingquan Wang. Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 864–872, 2018.

[93] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy

Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 147–156, New York, NY, USA, 2014. Association for Computing Machinery.

[94] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 565–580, Boston, MA, February 2019. USENIX Association.

[95] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 185–191, New York, NY, USA, 2017. Association for Computing Machinery.

[96] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 115–126, New York, NY, USA, 2012. Association for Computing Machinery.

[97] S. Wang, S. Wang, R. Huo, T. Huang, J. Liu, and Y. Liu. Deepaalo: Auto-adjusting demotion thresholds for information-agnostic coflow scheduling. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1123–1128, 2020.

[98] Wei Wang, Shiyao Ma, Bo Li, and Baochun Li. Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[99] Bernd Waschneck, André Reichstaller, Lenz Belzner, Thomas Altenmüller, Thomas Bauernhansl, Alexander Knapp, and Andreas Kyek. Optimization of global production scheduling with deep reinforcement learning. *Procedia CIRP*, 72:1264–1269, 2018. 51st CIRP Conference on Manufacturing Systems.

[100] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[101] Lex Weaver and Nigel Tao. The optimal reward baseline for gradient-based reinforcement learning. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, UAI'01, pages 538–545, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[102] Zhe Wei, Songtao Guo, Guiyan Liu, and Yuanyuan Yang. Coflow scheduling with unknown prior information in data center networks. In *ICC 2021 - IEEE International Conference on Communications*, pages 1–6, 2021.

[103] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science. Thesis (Ph. D.). Appl. Math. Harvard University*. PhD thesis, 01 1974.

[104] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992.

[105] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 50–61, New York, NY, USA, 2011. ACM.

[106] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 123–134, New York, NY, USA, 2013. Association for Computing Machinery.

[107] Wai xi Liu, Jun Cai, Qing Chun Chen, and Yu Wang. Drl-r: Deep reinforcement learning approach for intelligent routing in software-defined data-center networks. *Journal of Network and Computer Applications*, 177:102865, 2021.

[108] Renhai Xu, Wenxin Li, Keqiu Li, and Xiaobo Zhou. Shaping deadline coflows to accelerate non-deadline coflows. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–6, 2018.

[109] Renhai Xu, Wenxin Li, Keqiu Li, Xiaobo Zhou, and Heng Qi. Scheduling mix-coflows in datacenter networks. *IEEE Transactions on Network and Service Management*, 18(2):2002–2015, 2021.

[110] Zhiyuan Xu, Jian Tang, Jingsong Meng, Weiyi Zhang, Yanzhi Wang, Chi Harold Liu, and Dejun Yang. Experience-driven networking: A deep reinforcement learning based approach. pages 1871–1879, 04 2018.

[111] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

[112] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[113] Han Zhang, Xingang Shi, Xia Yin, Fengyuan Ren, and Zhiliang Wang. More load, more differentiation — a design principle for deadline-aware congestion control. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 127–135, 2015.

[114] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 160–173, New York, NY, USA, 2016. Association for Computing Machinery.

[115] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding lifecycle management complexity of datacenter topologies. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 235–254, USA, 2019. USENIX Association.

[116] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*, pages 424–432, 2015.

[117] Qihua Zhou, Peng Li, Kun Wang, Deze Zeng, Song Guo, and Minyi Guo. Swallow: Joint online scheduling and coflow compression in datacenter networks. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 505–514, 2018.

[118] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. Snc-meister: Admitting more tenants with tail latency slos. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 374–387, New York, NY, USA, 2016. Association for Computing Machinery.

[119] Timothy Zhu, Michael Kozuch, and Mor Harchol-Balter. Workload-compactor: reducing datacenter cost while providing tail latency slo guarantees. pages 598–610, 09 2017.

[120] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.