EXPLOITING THE SPATIAL DIMENSION OF BIG DATA JOBS FOR EFFICIENT CLUSTER JOB SCHEDULING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Akshay Jajoo

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Y. Charlie Hu, Chair

School of Electrical and Computer Engineering

Dr. Sonia Fahmy

Department of Computer Science

Dr. Chunyi Peng

Department of Computer Science

Dr. Pedro Fonseca

Department of Computer Science

Approved by:

Dr. Clifton W. Bingham

Graduate Committee Chair, Department of Computer Science,

Purdue University

To my parents, brother, and *Narayan*

ACKNOWLEDGMENTS

One of the most important things I have learned, by practical experience and observations, is that no ONE does anything. Everything is always done by many (or maybe all). It is just one or a few people who get labeled as the doer. The same is the case with this thesis. Though it will go out in my name, it is a result of the direct or indirect efforts of many. I want to mention a few of the major contributors here.

First and foremost, I want to express my reverence and gratitude towards my advisor Prof. Y. Charlie Hu. He not only focused on my technical learning but also paid close attention to improve my communication and collaboratively working skills. He took painstaking efforts to teach me the importance of hard work, planning, and time management. His traits like dedication, fighting till the last breath, and never compromising with the quality of work always inspired me. This work would certainly not have been possible without him.

During the course of my Ph.D. research, I intensively collaborated with Prof. Xiaojun Lin and Dr. Rohan Gandhi. I learned a lot through the intense and though provoking discussions with them. Working with them was a great learning experience. Their easy availability for help made my entire Ph.D. journey very smooth. Thanks to Prof. Cheng-Kok Koh for insightful discussions.

I also want to thank my advisory committee members, Prof. Sonia Fahmy, Prof. Chunyi Peng, and Prof. Pedro Fonseca, for their insightful comments and all the help during the course of this work.

I realized that during the entire Ph.D. phase, my lab (DSNL) was my first home. My collegues at DSNL made my whole journey full of fun. A big thanks to all of them: Jiayi Meng, Sibendu Paul, Vaibhav Jain, Qiang Xu, Ahmed Alawneh, Pranab Dash, Xiaomeng (Mona) Chen, Rohan Gandhi, Abhilash Jindal, Ning Ding, Vishwanath Singh.

I also want to thank all my course instructors. Grad courses at Purdue are a fun way of learning advanced computer science. Thanks to prof. Buster Dunsmore, his understanding nature made it very easy to work with him in managing undergrad software engineering projects for two semesters without impacting my research.

My interactions with my friends in the Computer Science Department in and outside the academic life made my life at Purdue smooth, and I learned many things from them without putting any extra effort. A few of them are known as Ashish Jain, Aritra Bose, Ben Harsha, Devesh Kumar Singh, Prashant Ravi, Shamiek Mangipudi, Shivaram Gopal, Sruthi Panchapakesan, and Venkata Gandikota aka GV.

I gained significant practical experience during my internship at Google in the summer of 2017, thanks to my host team NetArch. Also, thanks to Shreyas N. Basarge, Pranav Hegde, Pratik Chirania, and Jignesh Borisa for being my local buddies there.

Thanks to all the administrative staff of CS, ECE, Grad School, and Purdue for making all the administrative tasks smooth and hassle-free. Thanks for sending paychecks and reimbursements on time :). Thanks to ECN for providing excellent IT support. Thanks to Ashlee Messersmith for patiently helping with all the formatting needs of this document.

Some of my friends were available unconditionally at times of critical needs, and they deserve a special mention. I want to thank Shivaram Gopal, Esha Bemra, Ben Harsha, Abhas Deva, Tanuja Bemra, Anand Bemra, Tarutal Ghosh Mondal, Akash Ashapure, and Ashutosh Srivastav.

Several other people were important part of my life out of school, and it was their presence that kept me energetic. Asish Ghoshal, Akash Ashapure, B. Gayatri Mandava, Deepika Sharma, Jawad Raheel, Ksheer Sagar, Shivaram Gopal, Sneha Jha, Tarutal Ghosh Mondal, Tejasvi Parupudi, Vasudevan Iyer, Vidit Shah. A big thanks to Aayush Ankit, Akshay Jain, Kinshukh Gupta, Mohit Jain, and Tahir Patel.

During these five years of Ph.D. I have shared houses with several people. I prefer calling them homemates instead of housemates. Without them, it would have felt

like staying at a hotel for five long years. Their company made my stay fun and comfortable and enabled me to focus on my research. Thanks to Vaibhav Jain, Prateek Jain, Prateek Shukla, Vandith Pamuru, Navin Modi, Ram Kartheek, Shivaram Gopal, Ashutosh Srivastav, and Vijayaraghavan Sundararajan.

Alan Yang was a great local friend who was available at times when I needed him. I want to thank my friends in India for their constant love, support, and encouragement during the course of this work, in particular, I want to mention Shantanu Srivastav, Sourabh Agarwal, and my IITG friend's group MKM.

Prof. Amit Varma, and Prof. Ganesh Subbarayan, were my teachers outside of my academic life. I learned many life values by interacting with them. I want to express my reverence towards them. They made my non-academic life at Purdue also a great learning experience. Interacting with Prof. Dharmendra Saraswat, and Prof. Kashchandra G Raghothama in several activities were fun and gave a homely feeling. A big thanks to Prof. Brijesh K. Srivastava and Prof. Seema Mattoo for being supportive in my non-academic life.

I want to express my reverence and gratitude towards my parent Mrs. Lalita Jajoo (Maa) and Mr. Akhilesh Kr. Jajoo (Bhaiji). They made me foundationally strong right from childhood and always motivating me to think big and work hard. I thank them for instilling values like honesty, self-dependence, and respect for all. It is their hard work, sacrifices, and dedication in my upbringing that has enabled me to achieve whatever I have. I also want to express my reverence and gratitude towards my elder brother, Mr. Abhishek Jajoo (bhaiya), for being my goto person and my mentor not only throughout my academic career but life in general. He made sure that when I am involved with my Ph.D. research, other necessary aspects of my life and career don't get sidetracked. His support made me remain carefree and focus on my research. Thanks to my aunty Mrs. Indu Jajoo my sister-in-law Mrs. Tripti Jajoo and my cousins Mrs. Chitra Chandak and Mr. Shashikant Jajoo, for their constant support and encouragement. Thanks to dear Priyal for being a supportive company

and importantly for not troubling me to the best of her abilities. Thanks to all my family members for their love and encouragement.

Forgive me if I missed someone. I want to end my acknowledgement by quoting the following verse from the Bhagawad Gita:

kAyena vAchA manasendriyairvA
buddhyAtmanA vA prakRute svabhAvAt |
karomi yadyat sakalaM parasmai
nArAyaNAyeti samarpayAmi ||

The verse means the following: Whatever I do with the Body, Speech, Mind, or the Sense Organs, either by discrimination of the Intellect or by the deeper feelings of the Heart, or by the existing Tendencies of the Mind. I do them all without ownership, and I surrender them at the feet of *Sri Narayana* (the all-pervading supreme lord).

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABBREVIATIONS

CCT	Coflow Completion Time
JCT	Job Completion Time
SJF	Shortest Job First
SRTF	Shortest Remaining Time First
LAS	Least Attained Service

ABSTRACT

Jajoo, Akshay Ph.D., Purdue University, December 2020. Exploiting the Spatial Dimension of Big Data Jobs for Efficient Cluster Job Scheduling. Major Professor: Y. Charlie Hu.

With the growing business impact of distributed big data analytics jobs, it has become crucial to optimize their execution and resource consumption. In most cases, such jobs consist of multiple sub-entities called tasks and are executed online in a large shared distributed computing system. The ability to accurately estimate runtime properties and coordinate execution of sub-entities of a job allows a scheduler to efficiently schedule jobs for optimal scheduling.

This thesis presents the first study that highlights *spatial dimension*, an inherent property of distributed jobs, and underscores its importance in efficient cluster job scheduling. We develop two new classes of *spatial dimension* based algorithms to address the two primary challenges of cluster scheduling.

First, we propose, validate, and design two complete systems that employ learning algorithms exploiting *spatial dimension*. We demonstrate high similarity in runtime properties between sub-entities of the same job by detailed trace analysis on four different industrial cluster traces. We identify design challenges and propose principles for a sampling based learning system for two examples, first for a coflow scheduler, and second for a cluster job scheduler.

We also propose, design, and demonstrate the effectiveness of new multi-task scheduling algorithms based on effective synchronization across the spatial dimension. We underline and validate by experimental analysis the importance of synchronization between sub-entities (flows, tasks) of a distributed entity (coflow, data analytics jobs) for its efficient execution. We also highlight that by not considering sibling sub-

entities when scheduling something it may also lead to sub-optimal overall cluster performance. We propose, design, and implement a full coflow scheduler based on these assertions.

1 INTRODUCTION

Increasing digitization has led to the generation of an unprecedently large volume of data. Additionally, the rapid evolvement of cloud computing has ushered a new dimension in computing capabilities. Leveraging these two developments computer scientists are building a variety of, generic and diverse, cloud based distributed applications to rapidly process enormous amounts of data and draw meaningful insights from them. Applications of this type are popularly known as big data analytics applications. They could be log analysis [1,2], machine learning [3–5], SQL queries [6,7] and many others.

Diverse fields ranging from public policy to DNA analysis are employing big data analytics. A recent estimate by Statista shows that big data analytics is a $56 billion market currently and is predicted to be a $103 billion market by 2027 [8]. Principles of distributed computing are playing an important role in big data analytics [9], and the distributed big data analytics applications are the most widely used format. With their growing business impact, researchers naturally started focusing on optimizing its execution. Significant work has been done and is going on for optimizing their runtime and resource consumption [10–17]. In this thesis, we propose a new and practical approach to learn runtime properties and scheduling distributed analytics jobs.

1.1 Distributed Big Data Analytics Jobs

1.1.1 Overview

A large amount of digital data is being generated at an increasingly fast rate. According to a report by statista in May 2020, 26 Zettabytes (ZB)[1] of data was generated worldwide in 2017. That is expected to be 59 ZB in 2020 and, projections show that it will be 149 ZB by 2024. Table 1.1 shows the number of activities that happened per minute for different internet services in 2017 and 2020 [18]. Given such a huge amount of data which is anticipated to grow at an exponential rate (so data will also become obsolete in a short span), fast processing time becomes crucial to be able to draw any useful insights from the data. Supercomputers can be one obvious solution for that. However, they are highly expensive, and distributed computing has been a successful alternative to it [19, 20]. Engineers, intuitively, started using multiple machines to speed up data processing for long. In 2000, Seisint Incorporation (now LexisNexis Risk Solutions) developed a distributed platform for data processing, the HPCC (High-Performance Computing Cluster) Systems [21]. It remained private until 2011. The HPCC system automatically partitions, distributes, stores and delivers structured, semi-structured, and unstructured data across multiple commodity servers. However, the big breakthrough in distributed big data analytics happened in 2004 when Google proposed a formal framework, MapReduce, for distributed data analytics [1]. Several distributed data analytics applications like Apache Hadoop [2], Apache Hive [6], Spark [22] and, Apache Tez [7] have been developed inspired by this framework. A typical independent entity in these applications is called a *job* that consists of several sub-units called *tasks*. These tasks generally run in two parallel phases *map* and *reduce*. A *job* usually splits the input data-set into independent chunks that are first processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the *maps*, and inputs them to the *reduce tasks*. The final output is stored in filesystems by the *reduce tasks* [1, 23].

[1] 1 ZB = 10^{15} Megabyte

Table 1.1.: Amount of Internet activities per minute in 2017 and anticipated to be generated in 2020 [18].

Activities	**2020**	**2017**
Emails sent	200 million	150 million
Google searches	4.2 million	3.8 million
Tweets constructed	480,000	448,800
Instagram images uploaded	60,00	66,00
YouTube videos viewed	4.7 million	4.2 million
Facebook new users	400	360

1.1.2 Distributed Nature and Communication Phase

The distributed big data analytics jobs run on distributed computing systems. Distributed computing systems are networks of a large number of attached nodes or entities connected through a fast local network [24]. The use of network communication is extensive in the complete execution of these jobs, where the primary use is for transmitting data between different phases of the job [1,25]. Hence, efficient network performance is also crucial for job performance.

1.2 Scheduling in Shared Clusters

1.2.1 Challenges

Clusters are being shared among multiple users to execute a variety of distributed jobs. Such jobs typically arrive online and compete for shared resources. To best exploit the cluster and to ensure that jobs also meet their service level objectives, efficient job scheduling is essential. Since jobs arrive online, their runtime characteristics are not known a priori. This lack of information makes it challenging for the scheduler to determine the right order for running the jobs that maximize resource utilization and meets the application service level objective (SLOs). Additionally, jobs have different SLOs.

For some it is necessary to respect deadlines. For some others, faster completion or minimizing the use of a particular resource may be the requirement. Such a diverse set of objectives pose further challenges to effective job scheduling [10, 12, 14, 26–29].

1.2.2 Learning in Cluster Scheduling

An effective way to tackle the challenges of cluster scheduling is to learn the runtime characteristics of pending jobs, as accurately estimating job runtime characteristics allows the scheduler to exploit offline scheduling algorithms that are known to be optimal, *e.g.,* Shortest Job First for minimizing the average completion time.

Indeed, there has been a large amount of work [12–16, 29–31] on learning job runtime characteristics online (real-time) to facilitate cluster job scheduling.

In essence, all of the previous online learning algorithms learn job runtime characteristics from observing historical executions. In practice, however, as shown in several previous works [10, 12, 14, 30] learning from history is not very accurate. Our analysis of two production cluster traces has also validated it (details in chapter 4).

Another class of learning techniques employed is Least-Attained-Service (LAS) [32] based. Which is essentially a "try and miss" approach to approximate SJF and have very high learning overhead (details in chapter 3). Its variants like CLAS, D-CLAS [11] are used in coflow scheduling and Kairos [33] in cluster job scheduling.

1.2.3 Synchronizations in Cluster Scheduling

In typical distributed data analytics jobs, there are multiple phases like a Map-Reduce job has three phases - map, shuffle (communication) and, reduce [1]. Each of these phases has multiple entities like multiple tasks in the map and reduce phase and multiple flows in the shuffle phase. However, all of these components are working towards the same goal, *i.e.,* job completion. And for job completion, all components of all of its phases should finish. Also, the phases may have a dependency on the previous ones like, in principle, in map-reduce job's shuffle cannot begin till the map

finishes, and reduce cannot begin till shuffle finishes. This makes synchronization across phases as well as within phases important for efficient execution.

For instance, by speeding up the communication (shuffle) stage, where the data is transferred between compute nodes, we can speed up the job. However, improving network level metrics such as flow completion time may not translate into improvements at the application level metrics (such as job completion time). The coflow abstraction [25] was proposed to bridge such a gap. The abstraction captures the collective network requirements of applications, which is then used to improve the network level performance that directly translates into application performance improvements.

Additionally, most of the existing works have directly translated the algorithms designed for single CPU systems to the domain of distributed systems [11]. They have just made some simple changes for adaptations and have not harnessed the opportunity which arises from the distributed nature of such jobs. When taking into account the distributed nature of jobs, it is evident that SJF is not optimal in the first place. Intuitively, in a single-CPU job scheduling of N jobs, scheduling any job to run first will block the same number of other jobs, $N-1$. In scheduling a distributed job (entity) across multiple machines, however, since different jobs (entities) can have different numbers of tasks (sub-parts) distributed at different machines, scheduling a different job first can block a different number of other jobs (at the tasks where its machines lie). We denote this degree of competition as *contention*. In other words, the waiting time for other jobs will depend on the duration as well as the *contention* of the jobs. Hence it is important to consider this aspect of distributed jobs too. Heuristics like Smallest Effective Bottleneck First (SEBF) [34] have only partly attempted to address this issue.

1.3 Thesis

1.3.1 Statement

Exploiting the spatial dimension of big data jobs can improve cluster job scheduling as it enables developing effective online learning of job runtime properties and better syncrohnization of job tasks.

1.3.2 Key Idea

The core idea behind distributed systems is to partition any given work (job) into smaller parts (tasks) such that the outcome of all the parts when put together is the desired outcome of the job. It is very natural to do the partition such that the distribution of workload is equal across tasks. Several works have been motivated solely to achieve the goal of equal partitioning like Late [35], SkewReduce [36] and LEEN [37]. Analysis using cluster scheduling trace from 2Sigma [38] (figure 1.3) and network-flows trace from Facebook [39] (figure 1.2) supports this assertion.

The above described nature of a distributed job or entity can be summarized as follows: A distributed job or entity has many constituent sub-entities which are similar and are working towards one common goal. We abstract this collection of sub-entities as *spatial dimension* (see figure 1.1). The observation of this *spatial dimension* leads us to the following two ideas: (1) The *spatial dimension* can be exploited to efficiently learn information about jobs online as sub-entities can reveal information about each other ; (2) Synchronizing execution of sub-entities can make scheduling efficient.

1.3.3 Contributions

Leveraging the observation of *spatial dimension*, an inherent property of distributed jobs, we have developed a new class of efficient and practical scheduling algorithms for distributed systems.

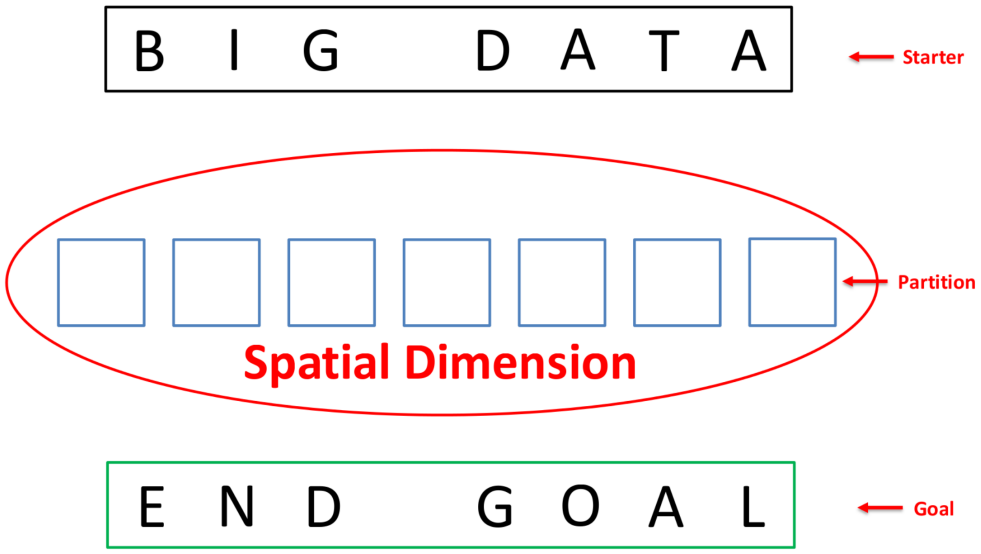


Figure 1.1.: The *spatial dimension* in distributed jobs originates from data partitioning and distributed processing. This figure illustrates the idea.

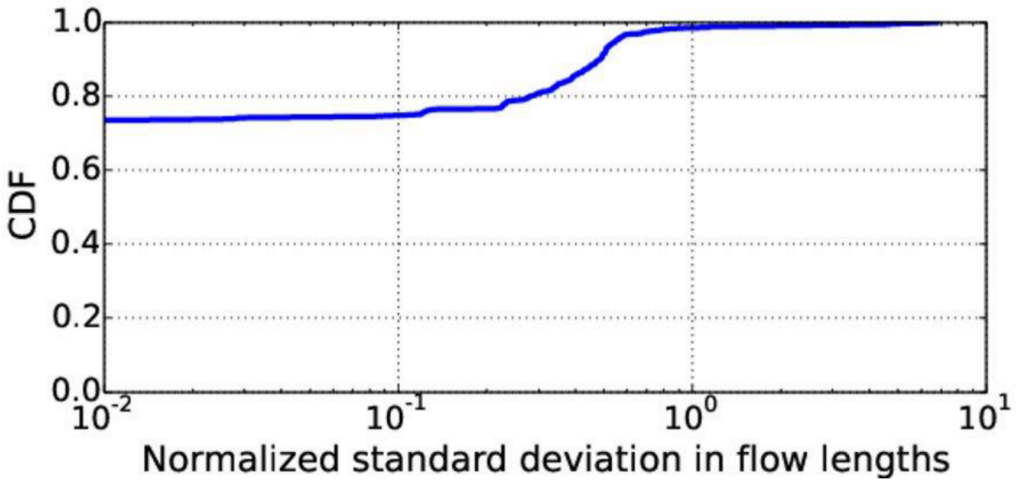


Figure 1.2.: CDF of normalized standard deviations in the lengths of flows of the communication phase of a distributed job across the jobs in a publicly available trace from the Facebook cluster [39].

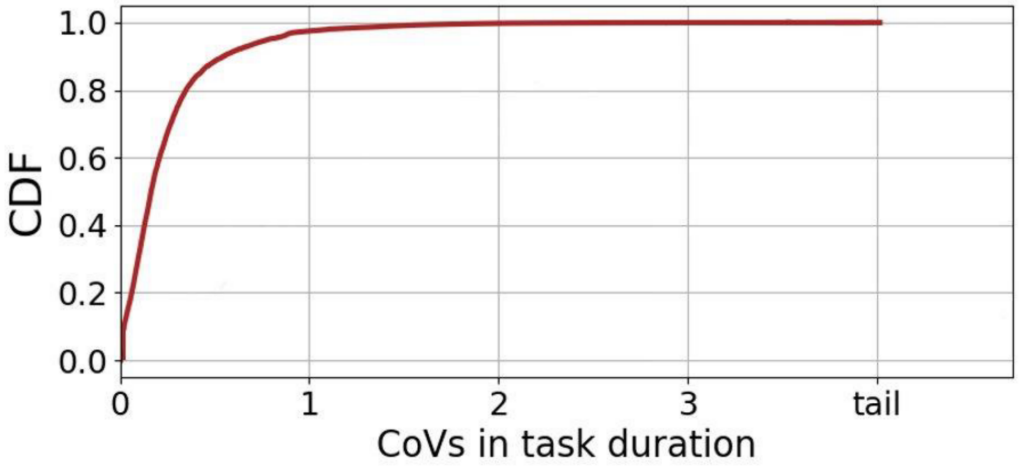


Figure 1.3.: CDF of the coefficient of variation (CoV) in the duration of tasks of the same distributed jobs. The figure shows the curve for cluster scheduling trace from 2Sigma [38]

As the first demonstration of our approach, we developed such types of algorithms for scheduling network flows in data centers. We have also designed a novel online (real-time) learning technique that exploits the spatial dimension of distributed jobs. Our technique predicts with 4.23× (5.02×) more average accuracy than the state-of-the-art history based predictor when tested on industrial cluster traces from 2Sigma (Google). Based on this learning technique we developed a cluster job scheduler that improves average job completion time by 1.65×.

The following are the two main contributions in this thesis:

1.3.3.1 Contribution 1: *A New Class of Online Learning Algorithms based on Sampling the Spatial Dimension.*

We propose, validate, and design two complete systems that employ sampling based learning. We validate our claim of high similarity in runtime properties between sub-entities of the same job by detailed trace analysis on four different industrial

cluster traces. We also present quantitative and experimental analysis for this. We propose design challenges and principles for a sampling based learning system for two examples, first for a coflow [25] scheduler and second a cluster job scheduler. To the best of our knowledge, we are the first ones to propose, validate, design and implement sampling based learning systems.

1.3.3.2 Contribution 2: *New Multi-Task Scheduling Algorithms by Synchronizing the Spatial Dimension.*

We underline and validate by experimental analysis the importance of synchronization between sub-entities (flows, tasks) of a distributed entity (coflow [25], data analytics jobs) for its efficient execution. We also highlight that by not considering sibling sub-entities when scheduling something it may also lead to sub-optimal overall cluster performance. We propose, design, and implement a full coflow [25] scheduler based on these assertions.

In this thesis, we further describe our fully designed systems for efficient coflow [25] scheduling of big data jobs and efficient cluster job scheduling.

2 Saath

This chapter is based on a publication in *Proceedings of CoNEXT'17* by ACM [40]. Following is its DOI: https://doi.org/10.1145/3143361.3143364 .

In this chapter, we discuss Saath, an online coflow scheduler that improves coflow scheduling by explicitly synchronizing intra-coflow and inter-coflow scheduling across spatial dimension.

2.1 Introduction

In analytics at scale, speeding up the communication stage directly helps to speed up the analytics jobs. In such settings, network-level metrics such as flow completion time (FCT) do not necessarily improve application-level metrics such as job completion time [11, 25, 41]. The coflow abstraction [25] is proposed to capture the network requirements of data-intensive applications so that improving network-level performance directly improves application-level performance.

In particular, a coflow consists of multiple concurrent flows within an application that are semantically synchronized; the application cannot make progress until all flows in a coflow have completed. Since in compute clusters, each job may consist of one or more coflows, and multiple jobs share the network fabric, it raises the *coflow scheduling problem* with the objective of minimizing the overall Coflow Completion Time (CCT) (NP-hard [34, 42]).

State-of-the-art coflow schedulers such as Aalo [11] in essence apply the classic online approximate Shortest-Job-First (SJF) algorithm using priority queues, where shorter coflows finish in high priority queues, and longer coflows do not finish in high priority queues, and are moved to and will finish in low priority queues.

Since a coflow has many flows distributed at many network ports, Aalo approximates the online SJF, designed for a single CPU, in a distributed setting. It uses a global coordinator to sort the coflows to the logical priority queues based on the progress made (total bytes sent); the flows of a coflow are assigned to the same priority queue at all network ports. At each port, the local scheduler applies a FIFO policy to schedule flows in each priority queue.

We make a key observation that this way of dividing the coflow scheduling task fundamentally does not take into account the *spatial dimension* of coflows scheduling, *i.e.,once assigned to the priority queues, the individual flows of a coflow are scheduled without any coordination until the coflow switches the queue or its flows finish.* Such lack of coordination in turn leads to two problems that negatively impact the quality of the scheduling algorithm.

Out-of-sync problem: First, the flows of a coflow at different ports can get scheduled at different times, which we refer to as the *out-of-sync* problem. Since the CCT is determined by the flow that completes the last, the flows that completed earlier did not help the CCT, but unnecessarily blocked or delayed the flows of some other coflows in their respective local ports, affecting the CCT of those coflows. Our evaluation using a production cluster trace shows that the out-of-sync problem is prevalent and severe (§2.2.2): over 20% of coflows with equal-length flows experience over 39% normalized deviation in FCT.

Contention-Oblivion problem: Second, when taking into account the spatial dimension of coflows, we observe that SJF (based on the total bytes of coflows) is not optimal in the first place. Intuitively, in a single-CPU job scheduling of N jobs, scheduling any job to run first will block the same number of other jobs, $N - 1$. In scheduling coflows across ports, however, since different coflows have different numbers of flows distributed at the ports, scheduling a different coflow (its flows) first can block a different number of other coflows (at the ports where its flows lie). We denote this degree of competition as *coflow contention*. In other words, the waiting time of other coflows will depend on the duration as well as the *contention* of the

coflow across its ports. Both SJF and Smallest Effective Bottleneck First (SEBF) [34] only consider coflow duration, and ignore the contention which can result in poor CCT.

In this chapter, we propose a new online coflow scheduling algorithm Saath.[1] Like Aalo, Saath is an online coflow scheduler that does not require apriori knowledge of coflows. Unlike Aalo, Saath explicitly takes into account the spatial dimension in scheduling coflows to overcome the out-of-sync and contention-oblivion drawbacks of prior coflow scheduling algorithms. Saath employs three key ideas. First, it mitigates the out-of-sync problem by scheduling coflows using an *all-or-none* policy, where all the flows of a coflow are scheduled simultaneously. Second, to decide on which coflow to schedule first following all-or-none, Saath implements contention-aware coflow scheduling. As the coflow durations are not known apriori, Saath adopts the same priority queue structure as Aalo and starts all coflows from the highest priority queue on their arrival. Instead of FIFO, Saath schedules coflows from the same queue using *Least Contention First* (LCoF), where the contention due to one coflow is computed as the number of other coflows blocked on its ports when the coflow is scheduled. LCoF prioritizes the coflows of less contention to reduce the total waiting time. Saath further uses coflow deadlines to avoid starvation. In contrast, Aalo [11] uses FIFO for online coflow scheduling, and other scheduling policies in Varys [34] (including SEBF) are offline and require apriori knowledge about coflow sizes.

Third, we observe that using the total bytes sent to sort coflows to priority queues ignores the spatial dimension and worsens the out-of-sync problem. When some flows of a coflow are scheduled due to out-of-sync, that coflow will take *longer* to reach the total-bytes queue threshold, which leads to other coflows being blocked on those ports for longer durations, worsening their CCT. Saath addresses this problem by using a *per-flow* queue threshold, where when at least one flow crosses its share of the queue threshold, the entire coflow moves to the next lower priority queue.

[1] Saath implies a sense of togetherness in Hindi.

Additionally, Saath handles several practical challenges in scheduling coflows in compute clusters in the presence of dynamics, including stragglers, skew and failures.

We implemented and evaluated Saath using a 150-node prototype deployed on a testbed in Microsoft Azure, and large-scale simulations using two traces from production clusters. Our evaluation shows that, in simulation, compared to Aalo, Saath reduces the CCT in median case by 1.53× and 1.42× (P90 = 4.5× and 37×) for the two traces while avoiding starvation. Importantly, this CCT reduction translates into a reduction in the job completion time in testbed experiments by 1.46× on average (P90 = 1.86×).

In summary, contributions presented in this chapter are following:

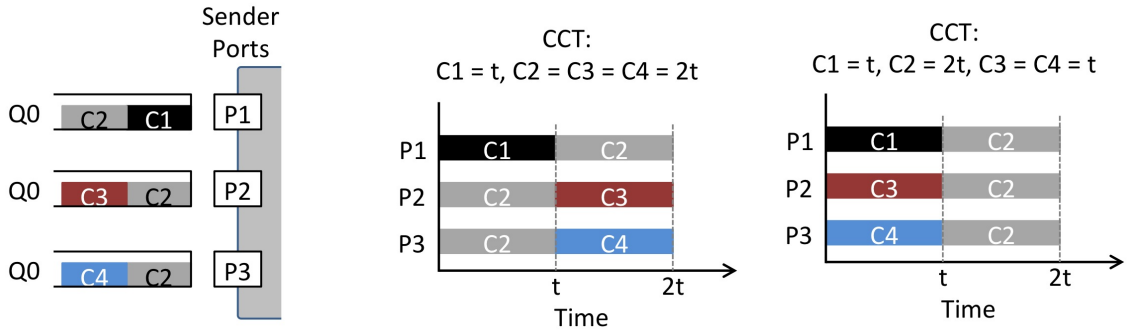
- Using a production datacenter trace from Facebook, we show the prevalence of the *out-of-sync* problem in existing coflow scheduler Aalo, where over 20% of coflows with equal-length flows experience over 39% normalized deviation in FCT.
- We show that the SJF (and also Shortest-Remaining-Time-First) scheduling policies are not optimal in coflow scheduling as they ignore contention across the parallel ports when scheduling coflows.
- We present the design, implementation and evaluation of Saath that explicitly exploits the *spatial dimension* of coflows to address the limitations of the prior art, and show the new design reduces the median (P90) CCT by 1.53× (4.5×) and 1.42× (37×) for two production cluster traces.

2.2 Prior-art and its Drawbacks

In this section, we briefly discuss the limitations of the Aalo scheduler which Saath overcomes.

2.2.1 Aalo Scheduler

Aalo [11] was proposed to schedule coflows online without any prior knowledge. Aalo approximates SCF using: (1) discrete priority queues, and (2) transitioning the



(a) Setup (b) Aalo (FIFO) (c) Optimal

Figure 2.1.: The out-of-sync problem in SAATH. The arrival time for coflow $C_1 < C_2 < C_3 < C_4$. The individual CCTs in Aalo (average=1.75·t) and optimal case (average=1.25·t) are denoted in Fig 2.1(b) and 2.1(c).

coflows across the queues using the total bytes sent so far by a coflow. In particular, Aalo starts all coflows in the highest priority queue and gradually moves them to the lower priority queue as the coflows send more data and exceed the per-queue thresholds. This design choice facilitates the completion of shorter coflows as known longer coflows move to lower priority queues, making room for potentially shorter coflows in the higher priority queues.

To implement the above online approximate SCF in a distributed setting, Aalo uses a global coordinator to assign coflows to logical priority queues. At each network port, the individual local ports then act *independently* in scheduling flows in its local priority queues, *e.g.,* by enumerating flows from the highest to lowest priority queues and using FIFO to order the flows in the same queue. In doing so, Aalo is oblivious to the *spatial dimension*, *i.e.,* it does not coordinate the flows of a coflow across different ports, which leads to two performance drawbacks.

2.2.2 Drawback 1: Out-of-Sync Problem

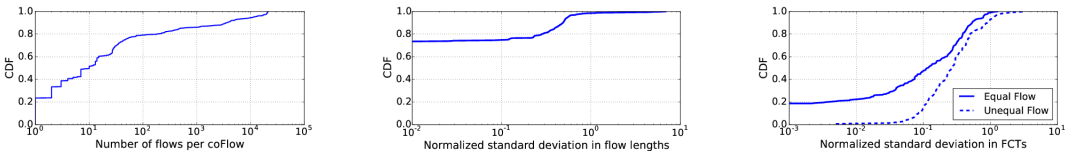
As individual ports locally have flows of different coflows, FIFO can result in the *out-of-sync problem*, *i.e.,* flows of a coflow are scheduled at different times at different

ports, as shown in the example in Fig. 2.1. The out-of-sync problem can substantially worsen the overall CCT in two ways:

1. Since the CCT depends on the completion time of the bottleneck (slowest) flow of the coflow, even if non-bottleneck flows of a coflow finish earlier, doing so does not improve the CCT of that coflow. Instead, such scheduling could block other potentially shorter coflows at those ports, and hence worsen their CCT (Fig. 2.1).

2. Aalo uses the *total bytes* sent so far to move coflows down the priority queues which further worsens the above problem. When only a subset of the flows of a coflow are scheduled, it would take longer to reach the same total-bytes queue-crossing threshold compared to when all the flows are scheduled. Hence, the scheduled flows occupy their ports for longer time, which does not improve their CCT, yet may worsen the CCT of other coflows that otherwise could have been scheduled.

To understand the extent of the out-of-sync problem, we analyze the variance of the flow completion time of each coflow under Aalo, using a trace from Facebook clusters [39]. First, Fig. 2.2(a) plots the distribution of the number of flows per coflow, and Fig. 2.2(b) plots the distribution of the standard deviation of flow lengths per coflow, normalized by its average flow length. We see that in the FB trace, 23% of the coflows have a single flow, 50% have multiple, equal-length flows, and the remaining 27% have multiple, unequal-length flows. We then plot the standard deviation of FCT of each of the multi-flow coflows, normalized by the average FCT of its flows. We note that the flows of a coflow can be of uneven length, which can contribute to uneven FCT. To isolate this factor, in Fig. 2.2(c), we separately show this distribution for coflows with equal and unequal flow lengths (excluding single-flow coflows). We see that the out-of-sync problem under Aalo is severe: the FCT of 50% (20%) of the equal-flow-length coflows have over 12% (39%) normalized deviation,



(a) Distribution of coflow width (b) Normalized standard deviation of flow lengths (c) Normalized standard deviation in FCTs for Aalo

Figure 2.2.: The out-of-sync problem in Aalo. (a) Distribution of number of flows in a coflow. (b) Distribution of standard deviation of flow lengths normalized by the average flow length, per coflow. (c) Distribution of normalized standard deviation of FCTs for multi-flow coflows under Aalo. In (c), we have excluded the coflows with single flows (23%).

and of the coflows with multiple, uneven-length flows, 50% (20%) have over 27% (50%) normalized deviation in FCT.

2.2.3 Drawback 2: SJF is Sub-optimal for Coflows

Assuming that the flows of each coflow are now scheduled in synchrony, the coordinator still needs to decide which coflows should go first to reduce the overall CCT. SCF derived from SJF has been a de-facto policy [11, 41]. We observe that SCF based on the total bytes sent by coflows is not optimal in coflow scheduling even in the (ideal) offline settings when the coflow sizes are known apriori. Similarly, even the Shortest-Remaining-Time-First (SRTF) which improves SJF by allowing preemption is not optimal even when coflow sizes are known apriori. The key reason is that these scheduling policies are designed for scheduling jobs serially on a single work engine. They are oblivious to the spatial dimension of coflows, *i.e.,* different flows of a coflow may be scheduled concurrently and contend with different numbers of other coflows (empirically proven in Appendix). Intuitively, two coflows C1 and C2 with durations t_1 and t_2 may block k_1 and k_2 other coflows when their flows are scheduled across individual ports. For example, in Fig. 2.1, k_1=1, k_2=3, k_3=k_4=1. Thus, the increase of the total waiting time of other coflows when scheduling C1 and C2 would be $t_1 \cdot k_1$ and $t_2 \cdot k_2$, respectively. SJF and SRTF only consider t_1 and t_2, and miss out the k_1

and k_2 factors, which can result in higher total waiting time for the rest of coflows and thus sub-optimal CCT.

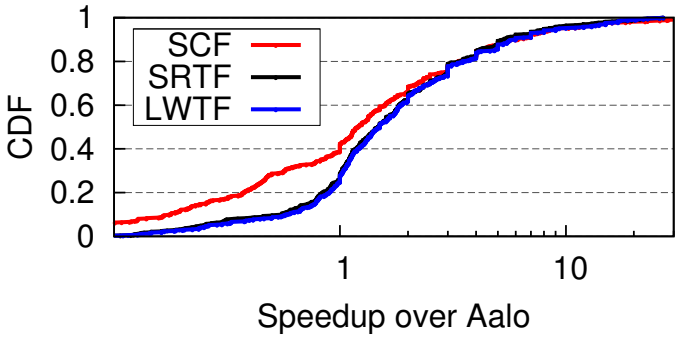
As a quick evidence that SJF is not optimal for coflow scheduling, we compare it with a *Least-Waiting-Time-First (LWTF)* policy. In LWTF, the coflows are sorted based on the increase in the total waiting time of other coflows, *i.e.,* $t \cdot k$. We then compare the improvement of the CCT of individual coflows as well as the overall CCT under LWTF, SCF and SRTF over Aalo in the ideal offline settings where the coflow sizes are known, using the FB trace. Fig. 2.3 shows LWTF outperforms SRTF and SCF, suggesting SCF and SRTF are not optimal, and considering *contention* when scheduling coflows leads to better CCT.

2.3 Key Ideas

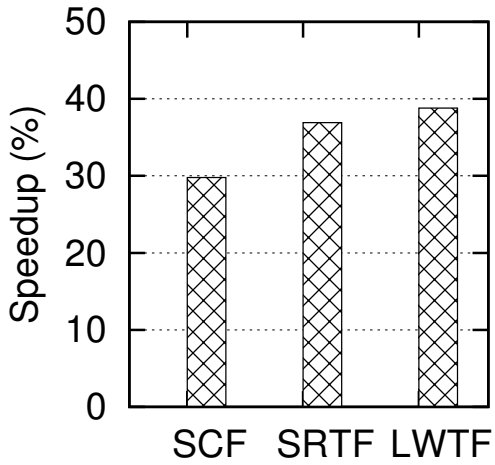
To address the two limitations of Aalo, we propose a new online coflow scheduler called Saath that explicitly takes into account the *spatial dimension* of coflows, *i.e.,* the flows of each coflow across different network ports. Specifically, Saath directly tackles the two limitations of Aalo: (1) the out-of-sync problem is mitigated by scheduling all flows of a coflow together; (2) the *contention among coflows across the ports* is explicitly considered in scheduling coflows. In the following, we detail on these core ideas that shape the Saath design.

(1) All-or-none: The first key idea in Saath is to schedule the coflows using an *all-or-none* policy, *i.e.,* either all the flows of a coflow are scheduled together, or none. This design choice effectively alleviates the out-of-sync problem in Aalo, as the ports that used to schedule a subset of flows of a coflow early can now delay scheduling them, without potentially inflating the CCT of that coflow, since its CCT depends on the completion of its last flow. The scheduling slots at those ports can be used for some other coflows, potentially improving their CCT.

Our key insight is that, in the context of conventional flow scheduling, typically the FCT of one flow cannot be improved without degrading the FCT of another flow [43].



(a) Speedup in CCT

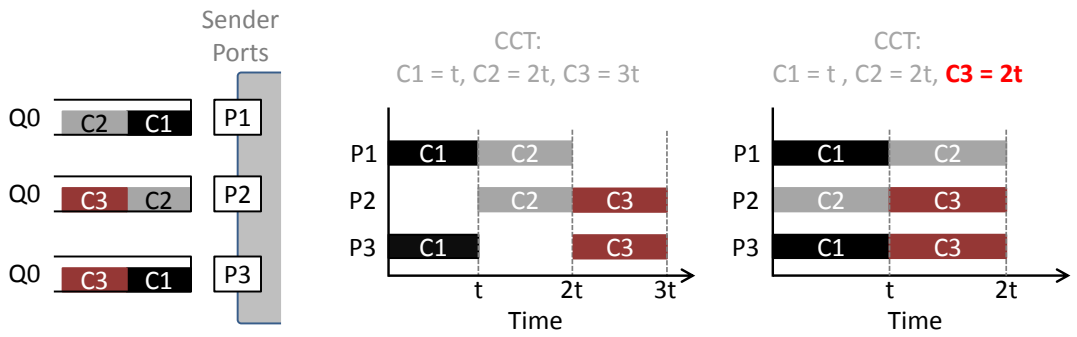


(b) Overall CCT

Figure 2.3.: Comparing CCT speedup using SCF, SRTF and LWTF over Aalo assuming flow statistics are known. The curves for SRTF and LWTF overlap in (a). Overall CCT in (b) is the average CCT for all coflows.

However, this is not true in the context of coflows, as the CCT of a coflow comprising of many flows depends on the completion of the last flow, and thus a delay in the earlier finishing flows of a coflow should not inflate its CCT but could improve the CCT of other coflows. In doing so, the CCT of one coflow can be improved without worsening the CCT of other coflows.

However, all-or-none alone can potentially result in poor port utilization because it requires *all* ports of a coflow to be available when scheduling; if not all ports needed by a coflow are available, they may be all sitting idle as shown in the example



(a) Setup (b) All-or-none (c) All-or-none + Work-conservation

Figure 2.4.: Unused ports in all-or-none can elongate CCT as in (b), with average CCT = 2·t. (c) Work-conservation can speedup coflows (average CCT = 1.67·t).

in Fig. 2.4(b). Saath carefully designs the work conservation scheme to schedule additional flows at ports that are otherwise left idle, as shown in Fig. 2.4(c) (§2.4.2).

One may observe that, as shown in Fig. 2.4(c), applying work conservation appears to break away from all-or-none. We argue that it does not re-create the out-of-sync limitation in Aalo. Recall that the out-of-sync limitation in Aalo was caused due to scheduling a coflow at a time slot that otherwise could have been used for a potentially shorter coflow. In Saath, work-conservation schedules a coflow in an otherwise *empty* time slot, which does not push back other coflows. Instead it will only speed up the coflows.

We note that if the flow lengths in a coflow are skewed, all-or-none may not finish all the flows of a coflow together. Since Saath is an online coflow scheduler, it does not know the flow lengths beforehand. As a result, in some cases, it may end up delaying scheduling a longer flow to align with other flows, which may delay completion of that flow and worsen the CCT of the coflow. Our evaluation (§2.6.2) shows that such cases are rare, and overall all-or-none improves CCT.

(2) Faster coflow-queue transition: Since the flow durations are not known apriori, like Aalo, Saath uses the priority queue structure to approximate the general

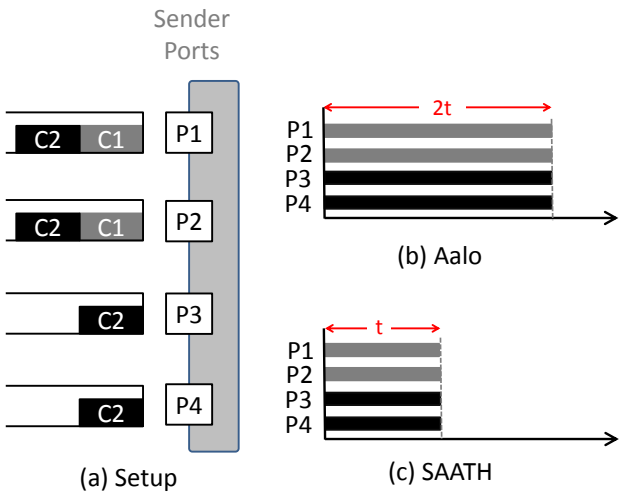


Figure 2.5.: Fast queue transition in SAATH. (a) Coflow organization. (b) Transition for C1 and C2 in Aalo. Assume the queue threshold is $bandwidth \cdot 4t$. C2 takes $2t$ time units to reach the threshold as 2 ports (out of 4) are sending data. (c) Fast queue transition in SAATH. The per-flow queue threshold for C2 is $bandwidth \cdot t$ as there are 4 flows, and it takes t time units to reach the threshold.

notion of Shortest Coflow First, by helping shorter coflows finish (early) in high priority queues.

The key challenge in priority queue-based design is to quickly determine the right queue of the coflow, so that the time that longer coflows contend with the shorter coflows is minimized. Like Aalo, SAATH starts all the coflows from the highest priority queue. Unlike Aalo, SAATH uses *per-flow queue thresholds*. When an individual flow of a coflow reaches its fair share of the queue threshold before others, *e.g.,* from work conservation, we move the *entire coflow* to the next lower priority queue.

In essence, if a coflow is expected to cross the queue threshold, using per-flow queue thresholds effectively speeds up such queue transition as shown in Fig. 2.5, where a coflow is transitioned to the next queue in time t instead of $2 \cdot t$ as in Aalo. Such faster queue transition has an immediate benefit: it frees the ports where the remaining flows of that coflow are falling behind *sooner*, *i.e.,* by moving them to the

next lower priority queues at their corresponding ports, so that other high priority coflows could be scheduled sooner, potentially improving their CCT.

In Saath, we calculate the fair share threshold by simply splitting the queue threshold equally among all the flows of a coflow. More sophisticated ways can be used in clusters with skewed flow duration distribution.

(3) Least-Contention-First policy within a queue: Once the coflows are assigned to the priority queues, the next challenge is to order and schedule the coflows from the same queue. In Saath, we propose the Least-Contention-First (LCoF) policy, where the *contention* of a coflow is calculated as the number of other coflows blocked when that coflow is scheduled at all of its ports.[2] Under LCoF, all the coflows in each queue are sorted according to the increasing order of contention, and the scheduler scans the sorted list from each queue, starting from the highest priority queue, and schedules the coflow that competes against the least number of other coflows, as long as there is enough port bandwidth remaining. In essence, by scheduling coflows in the LCoF manner, Saath allows more coflows (who have less contention) to be scheduled in parallel in conforming to all-or-none and hence more coflows to finish earlier. Our evaluation results (Fig. 2.10) confirm that Saath gains significant improvement by use of LCoF.

In summary, Saath improves the CCT of the coflows from the same priority queue using LCoF and all-or-none, and accelerates the coflow queue transition using per-flow queue thresholds to further improve the overall CCT.

2.4 Online Scheduler Design

In addition to the three key ideas for improving CCT, Saath also needs to (1) provide starvation-free guarantee for continuous progress, as LCoF can indefinitely

[2]We note the contention thus defined is an approximation to the impact scheduling that coflow does to the overall CCT, which should be weighted by the remaining flow lengths of the coflow, which however is not known.

delay scheduling a coflow that always has higher contention than other coflows, and (2) speed up coflows during cluster dynamics such as node failures and stragglers.

In this section, we present the detailed Saath design to overcome these challenges. The key design features in Saath are summarized as follows:

1. *All-or-none*: mitigates the out-of-sync problem;
2. *Per-flow queue threshold*: speeds up queue transition;
3. *LCoF*: orders coflows within a queue in a contention-aware manner;
4. *Work-conservation*: improves port utilization and the overall CCT;
5. *Handling cluster dynamics*: speeds up the flows of a coflow due to dynamics such as failures and stragglers by moving the coflow back to higher priority queues;
6. *Starvation-free:* provides starvation-free guarantees.

2.4.1 Saath Architecture

Fig. 2.6 shows the Saath architecture. The key components are the global coordinator and local agents running at the individual ports. A computing framework such as Hadoop or Spark first registers (removes) the coflows when a job arrives (finishes). At every fixed scheduling interval, the global coordinator computes the schedule for all the ports based on the coflow information from the framework and flow statistics sent by the local agents (which update the global coordinator at each scheduling interval, details in §2.4.2). The coordinator then pushes the schedule back to the local agents. Local agents maintain the priority queues and use them to schedule coflows. They continue to follow the current schedule until a new schedule is received from the global coordinator.

Saath uses the same queue structure as Aalo, and has the same parameter settings . In Saath, there are N queues, Q_0 to Q_{N-1}, with each queue having lower queue threshold Q_q^{lo} and higher threshold Q_q^{hi}, and $\mathrm{Q}_0^{lo} = 0$, $\mathrm{Q}_{N-1}^{hi} = \infty$, $\mathrm{Q}_{q+1}^{lo} = \mathrm{Q}_q^{hi}$. Saath uses exponentially growing queue thresholds, *i.e.,* $\mathrm{Q}_{q+1}^{hi} = \mathrm{E} \cdot \mathrm{Q}_q^{hi}$.

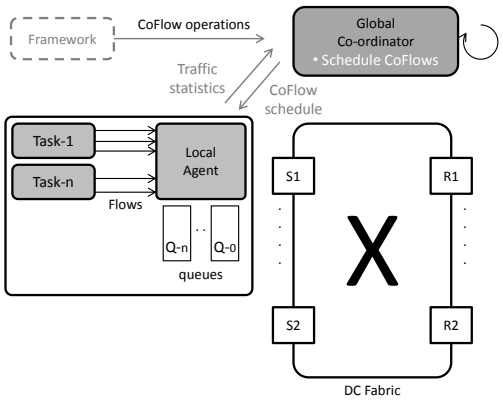


Figure 2.6.: SAATH architecture.

2.4.2 SAATH Scheduler

Fig. 2.7 shows the scheduling algorithm used by the global coordinator to periodically compute the schedule to minimize the CCT using all-or-none, per-flow queue threshold, and LCoF, and to provide starvation-free guarantee.

Input: The input to the algorithm includes (1) the set of coflows (C), (2) traffic sent by the longest flow of every coflow ($t_{c,f}$), (3) starvation-free deadline (d_c), (4) the ports used by individual coflows ($p_{c,p}$), (5) Total capacity (bandwidth) available at p^{th} port (B_p).

We calculate k_c, the number of coflows *contending* with the c-th coflow across all the ports. This is used in implementing the LCoF policy.

Output: *f.rate*, *i.e.,* the bandwidth assigned to each coflow at each port.

Objective: Minimize the average CCT.

D1. Overall algorithm: (1) First, the coordinator determines the queue of the coflows based on the maximum data sent by any flow of a coflow, *i.e.,* $m_c = max(\forall_{f \in f_c}, t_{c,f})$ and per-flow threshold (see D3, D4) (line 2). (2) Next, it sorts the coflows, starting from the highest priority queue to the lower priority queues. (3) Within each queue, it sorts the coflows using LCoF, *i.e.,* based on their k_c values (line 3:4). (4) It then scrolls through coflows one by one, and if all the ports of a

```
1: procedure SCHEDULE((coflows C))
2:     AssignQueue(C)                                   ▷ Assign queues
3:     for q in Q do
4:         L = SortLCoF(C, q)                           ▷ Sort using LCoF
5:         missed = {}
6:         for CoFlow c in L do
7:             if AllOrNoneSchedule(c) then
8:                 rate = min. available rate for all flows
9:                 for flow f in c do f.rate = rate
10:                UpdateAvailableBandwidthForSenders(c)
11:                UpdateAvailableBandwidthForReceivers(c)
12:            else
13:                missed.add(c)
14:        WorkConservation(missed)
15: procedure ASSIGNQUEUE((C))
16:    for c in C do
17:        c.queue = GetQueue(c.width,c.maxFlowLength)
18: procedure WORKCONSERVATION(C^m)
19:    for c in C^m do
20:        for f in c do
21:            f.rate = min(f.sender.remainBW, f.receiver.remainBW)
22:        UpdateAvailableBandwidthForSenders(c)
23:        UpdateAvailableBandwidthForReceivers(c)
```

Figure 2.7.: SAATH scheduling algorithm.

coflow (sender and receiver) have available bandwidth (line 7), the coflow is scheduled. SAATH assigns the bandwidth as discussed in D2 below, based on which the port allocated bandwidth is incremented (line 9, 10). If any of the ports are unavailable, the coordinator skips that coflow and moves to the next coflow. (5) The algorithm terminates when all coflows are scanned or all bandwidth is exhausted by work conservation (see D4 below).

D2. Assigning flow bandwidth: As in MADD [34], SAATH assigns equal rates (bandwidth) at the ports as there is no benefit in speeding-up flows at certain ports when the CCT depends on the slowest flow. At a port, we use max-min fairness to schedule the individual flows of a coflow (to different receivers). Hence, the rate

of the slowest flow is assigned to all the flows in the coflow, and the port-allocated bandwidths at the coordinator are incremented accordingly.

D3. Determining coflow queue: Similar to Aalo, SAATH uses exponentially growing queue thresholds. To realize faster queue transition, we divide the queue threshold (Q_q^{hi}) equally among all the flows (flow count = N_c) of a coflow. For example, when a queue threshold is 200MB, a coflow with 100 flows has a per-flow queue threshold of 2MB. SAATH assigns coflow to a queue based on the maximum data sent by any of its flows, using Eq. (2.1):

$$\frac{Q_{q-1}^{hi}}{N_c} \leq m_c \leq \frac{Q_q^{hi}}{N_c} \tag{2.1}$$

D4. Work conservation: When following the all-or-none policy, it is possible that some of the ports do not have flows scheduled (§2.3); these ports can be used to schedule coflows outside all-or-none, triggering work conservation (line 14, 18-23). In work conservation, the coflows are scheduled based on the ordered list of the unscheduled coflows.

D5. Starvation Avoidance: Recall that FIFO provides starvation-free guarantee as every flow in a queue is guaranteed forward progress [11]. Such guarantees are not offered by LCoF. To avoid starvation, the coordinator sets a *deadline* for each coflow. Importantly, this deadline is derived based on FIFO. Whenever a coflow arrives in a queue, a fresh deadline is set for it. For that, the coordinator first generates FIFO ordering at all ports by enumerating all the coflows in that queue. If there are C_q coflows in the queue, and t is the minimum time a coflow needs to spend in the queue based on the queue threshold, the deadline for the new coflow for that queue is set to $d \cdot C_q \cdot t$, where d is a constant (d = 2 in our prototype §3.9). SAATH then prioritizes the coflows that reach their deadlines. Essentially, SAATH provides the same deadline guarantee (within a factor of d) as a FIFO based scheduler.

2.4.3 Handling Cluster Dynamics

Compute clusters in datacenters frequently undergo a variety of dynamics including node failures and network congestion. Moreover, even individual jobs may experience stragglers and data skew, multiple stages and waves. In this section, we detail on how Saath adapts to such dynamics to reduce their impact on the CCT.

Improving tail due to failures, stragglers, skew: Cluster dynamics such as node failures and stragglers can delay some flows of a coflow, which can result in poor CCT as CCT depends on the completion of the last flow. We observe in such cases, some flows of the coflow may have already finished. In such cases, we heuristically make use of the flow length of the completed flows to approximate the SRTF policy to potentially speed up such coflows, as follows: (1) the coordinator estimates the length of unfinished flows of a coflow using the median flow length of its currently finished flows (f_e). (2) It estimates the remaining flow lengths for straggling/restarted flows $f_i^{rem} = f_e - f_i$, where f_i is the flow length so far for the i-th unfinished flow. (3) It estimates the remaining time of a coflow as $m_c = max(f_i^{rem})$ since the CCT depends on the last flow, and uses m_c to re-assign the coflow to a queue using Eq. 2.1.

The intuition behind the optimization is that, once some of the flows finish, we no longer need to use the priority queue thresholds to estimate flow lengths – we can simply use m_c as above. The benefit of this approximated SRTF policy is that Saath can move up a coflow from *low to high priority queues* when its flows start to finish; as the remaining flows send more data, f_i increases and thus f_i^{rem} decreases. Moving the coflow to a higher queue will accelerate its completion, while following SRTF. We note that calculating f_e as the median of the finished flows is a heuristic; more sophisticated schemes such as Cedar [44] can be used to estimate flow lengths, which we leave as future work.

In contrast, Aalo does not move the coflow to the higher priority queues even when fewer and fewer flows are pending, because coflow is assigned to a queue based

on the total bytes sent so far, which only grows as the flows of a coflow send more data.

Scheduling Multi-stage DAG and multiple waves: Often times, a single analytics query consists of multiple co-dependent stages, where each stage has multiple jobs. Such queries are represented as a Directed Acyclic Graph (DAG) to capture the dependencies (e.g., Hive [6] on Hadoop or Spark). The DAG representation is available before the start of the query while the scheduler builds the query plans. In Saath, instead of having one coflow for every *job* in a stage, we have one coflow for every *stage*. This optimization helps Saath to slow down some of the fast jobs in one stage without affecting the overall completion time, as the completion of the DAG stage depends on the completion of the slowest job in that stage.

Similarly, a single MapReduce job may have its map and reduce tasks scheduled in multiple waves, where a single wave only has a subset of the map or reduce tasks. We represent such cases again as a DAG, where a single wave is represented as a single coflow, and the DAG consists of serialized stages, each with one single coflow. In such cases, the goal of DAG scheduling is the same as the coflow scheduling, and the same coflow scheduling design can be used.

Un-availability of the data: Another important challenge is that the data may not always be available in the communication stage as the computing frameworks often *pipeline the compute and communication stages* [45], *i.e.,* the subset of the data is sent from one phase to another as soon as it becomes available, without waiting for the completion of the whole stage. In such frameworks, not all flow data is always available [45] due to some slow or skewed computation. If the coordinator schedules a coflow when some of its data is not available, that time slot is wasted.

To address this problem, in Saath, the ports first accumulate enough data on each of the flows of the coflow for one δ, *i.e.,* the interval at which local agents coordinate with the coordinator, and explicitly notify the coordinator when such data is available. This information is piggybacked in the flow statistics sent periodically

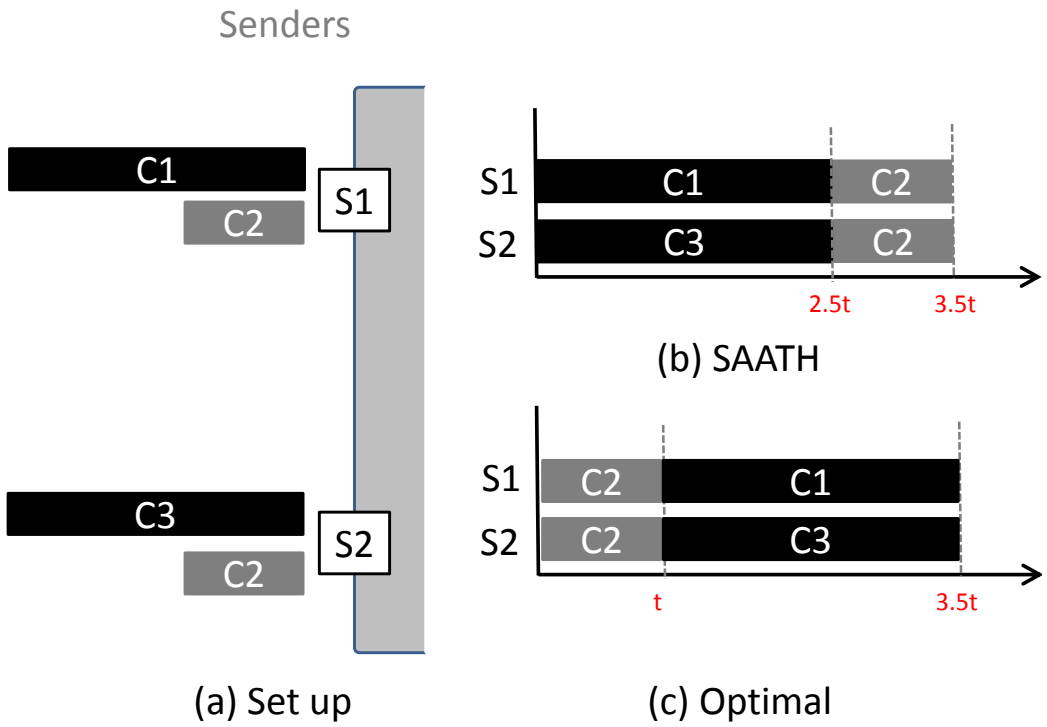


Figure 2.8.: LCoF limitations. (a) shows the setup, coflow durations, (b) and (c) show the coflow progress in SAATH and optimal. The average CCT in (b) is $\frac{2.5+2.5+3.5}{3} = 2.83$, and in (c) is $\frac{1.0+3.5+3.5}{3} = 2.66$.

and thus has minimal overhead. The coordinator only schedules the coflows that have enough data to send.

2.4.4 LCoF Limitation

Although LCoF substantially outperforms other scheduling policies (§3.9, §3.10), there are rare cases where LCoF performs worse. The key reason is that LCoF schedules coflows based on the contention; if there are coflows that have less contention but are longer in size, scheduling such coflows using LCoF would be sub-optimal as shown by the example in Fig. 2.8. However, our trace shows that such coflows only constitute a minor fraction of the total coflows (§2.6.2, bin-2 in Fig. 2.11 and

Fig. 2.12), and hence their impact is dwarfed by the improvements on other coflows from using LCoF.

2.5 Implementation

We implemented Saath consisting of the global coordinator and local agents (Fig. 2.6) in 5.2 KLoC in C++.

Coordinator: The coordinator schedules the coflows based on the operations received from the framework and traffic statistics from the local agents. The key implementation challenge for the coordinator is that it needs to be fast in computing and updating the schedules. The Saath coordinator is multi-threaded and is optimized for speed using a variety of techniques including pipelining, process affinity, and concurrency whenever possible.

Conceptually, the coordinator computes new schedules in fixed intervals *e.g.,* the time required to send 1MB at a port, which is 8ms with our setting. In practice, due to the delay in computing and propagating the schedules, the coordinator and local agents work in a pipelined manner. In each interval, the coordinator computes a new schedule consisting of the coflow order and flow rates, based on the flow stats received during the previous interval, and pushes them to local agents right away. How local agents react is described below.

Since the coordinator makes scheduling decisions on the latest flow stats received from the local agents, it is stateless, which makes it easy for the coordinator to recover from failures. When the coordinator fails, new deadlines are calculated for each coflow.

Local agents: Upon receiving a new schedule from the coordinator, each local agent schedules the flows accordingly, *i.e.,* they comply to the previous schedule until a new schedule is received. In addition, the local agents periodically, at the same frequency at which the coordinator calculates new schedules, send the relevant coflow statistics, including per-flow bytes sent so far and which flows finished in this

interval, to the coordinator. To intercept the packets from the flows, local agents require the compute frameworks to replace `datasend(), datarecv()` APIs with the corresponding Saath APIs, which incurs very small overhead. Lastly, the local agents are optimized for low CPU and memory overhead (evaluated in §2.7.3), enabling them to fit well in the cloud settings [46].

Coflow operations: The global coordinator runs independently from, and is not coupled to, any compute framework, which makes it general enough to be used with any framework. It provides RESTful APIs to the frameworks for coflow operations: (a) `register()` for registering a new coflow when it enters, (b) `deregister()` for removing a coflow when it exits, and (c) `update()` for updating coflow status whenever there is a change in the coflow structure, particularly during task migration and restarts after node failures.

2.6 Simulation

We evaluated Saath using a 150-node testbed cluster in Azure that replays the Hive/MapReduce trace from Facebook (FB). In addition, we evaluate Saath using large-scale simulations using traces from production clusters of Facebook and a large online service provider (OSP). The FB trace is for 150 ports and is publicly available at [39]. The OSP trace is from a Microsoft cluster and has O(1000) jobs collected from O(100) ports.[3] The highlights of these evaluation are:

- Saath significantly improves the overall CCT. In simulation using the FB trace, the CCT is improved by 1.53× in the median case (P90 = 4.50×). For the OSP trace, the improvements in CCT are 1.42× in the median case (P90 = 37.2×).
- In testbed experiments, compared to Aalo, Saath improves job completion time by 1.46× on average (P90 = 1.86×);
- The Saath prototype is fast and has small memory and CPU footprints;

[3]We cannot specify the exact numbers for proprietary reasons, which are also excluded from Fig. 2.2 in §4.2.

- The breakdown of the improvement justifies the effectiveness of LCoF, all-or-none, and faster queue transition design ideas.

We present detailed simulation results in this section, and the testbed evaluation of our prototype in §3.10.

Setup: In replaying the traces (FB and OSP), we maintain the same flow lengths and flow ports. The *default parameters* in the experiments are: starting queue threshold (Q_0^{hi}) is 10MB, exponential growth factor (E) is 10, the number of queues (K) is set to 10, and the new schedule calculation interval δ is set to 8ms. Our simulated cluster uses the same number of nodes (network ports) and link capacities as per the trace. We assume full bisection bandwidth supporting 1 Gbps/port is available, and congestion can happen only at the network ports. The simulator is written in 4 KLOC in C++.

We compare Saath against two start-of-the-art coflow schedulers, Aalo and Varys, which are open-sourced [47]. All the experiments use the above default parameters including K, E, S, unless otherwise stated. Since Varys does not use multiple queues, there is no use of queueing parameters for Varys related experiments.

2.6.1 CCT Improvements

We first compare the speedup of Saath over other scheduling policies. We define the *speedup* using Saath as the ratio of the CCT under other policy to the CCT under Saath for individual coflows. The results are shown in Fig. 2.9. The Y-axis denotes the median speedup, and error bars denote the 10-th and 90-th percentile speedups. We show the results for the FB and OSP traces. The key observation is that Saath improves the CCT over Aalo by 1.53× (median) and 4.5× (P90) for the FB trace, and 1.42× (median) and 37.2× (P90) for the OSP trace. Interestingly, Saath achieves the speedup close to that of SEBF in Varys [34] even though SEBF runs offline and assumes the coflow sizes are known *apriori*, whereas Saath runs online without apriori coflow knowledge.

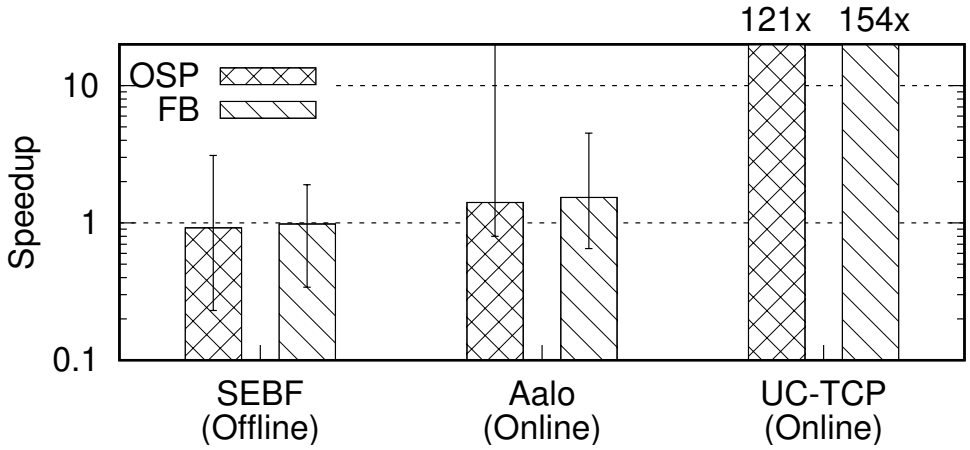


Figure 2.9.: Speedup using Saath over other scheduling policies. Saath achieves speedup of 154× and 121× (median) over UC-TCP for two traces.

The higher speedup at P90 for the OSP trace over the FB trace is attributed to larger improvement to the CCT of small and narrow coflows using all-or-none and LCoF (§2.6.2). We observe that the ports are busier (*i.e.,* having more coflows queued at individual ports) for the OSP trace than the FB trace, which when coupled with FIFO in Aalo, amplifies the waiting time for short and narrow coflows in the OSP trace. In contrast, LCoF facilitates such coflows, resulting in dramatic reduction in their waiting time.

We also compare Saath against an un-cordinated coflow scheduler (UC-TCP) under which individual ports independently schedule the arriving coflows without any global coordinator. In UC-TCP, there are no queues, and all the flows are scheduled upon arrival as per TCP. Lack of coordination, coupled with lack of priority queues severely hampers the CCT in UC-TCP. Saath achieves a median speedup of 154× and 121× over UC-TCP in the FB and OSP traces, respectively.

These results show that Saath is effective in accelerating the coflows compared to Aalo, and is close in performance compared to Varys which assumes prior knowledge of coflow lengths, and achieves high speedups compared to other un-coordinated scheduler.

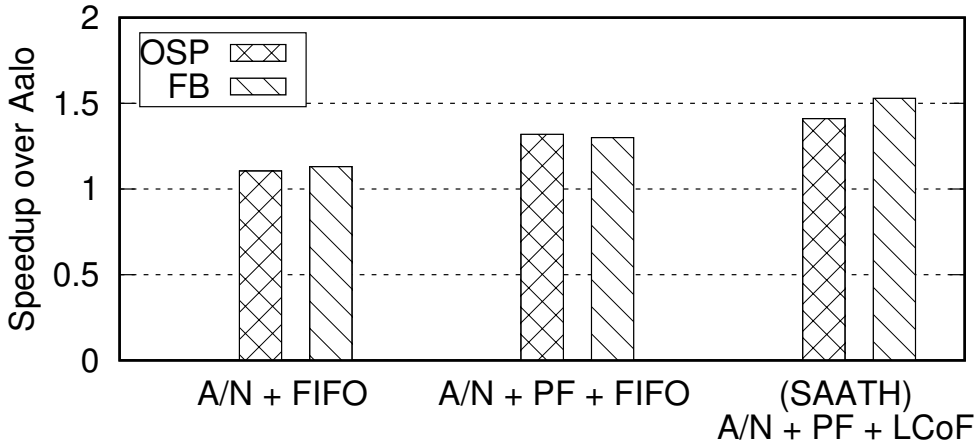


Figure 2.10.: Saath speedup breakdown across three complimentary design ideas. Y-axis shows the median speedup. Abbreviations: (1) A/N: all-or-none, (2) PF: per-flow queue threshold, (3) LCoF: Least-Contention-First.

Table 2.1.: Bins based on total coflow size and width.

	width $\leq$ 10	width $>$ 10
size $\leq$ 100MB	bin-1	bin-2
size $>$ 100MB	bin-3	bin-4

2.6.2 Impact of Design Components

In this experiment, we evaluate the impact of the individual design components on the speedup in CCT over Aalo. The results are shown in Fig. 2.10. To better understand the impact, we also show the CCT improvement grouped into different bins based on their width and size of the coflows (Table 2.1) in Fig. 2.11 and Fig. 2.12. We make the following key observations.

First, only using all-or-none (A/N) and FIFO, *i.e.,* without LCoF and per-flow queue threshold (P/F), the speedup over Aalo using FB (OSP) trace is 1.13× (1.1×) in median case and 3.05× (7.2×) at P90. Fig. 2.11 and Fig. 2.12 show that all-or-none is effective for small, thin coflows, *i.e.,* coflows with fewer flows, as the probability of finding all the ports available is higher. For other bins, the benefits are lower due

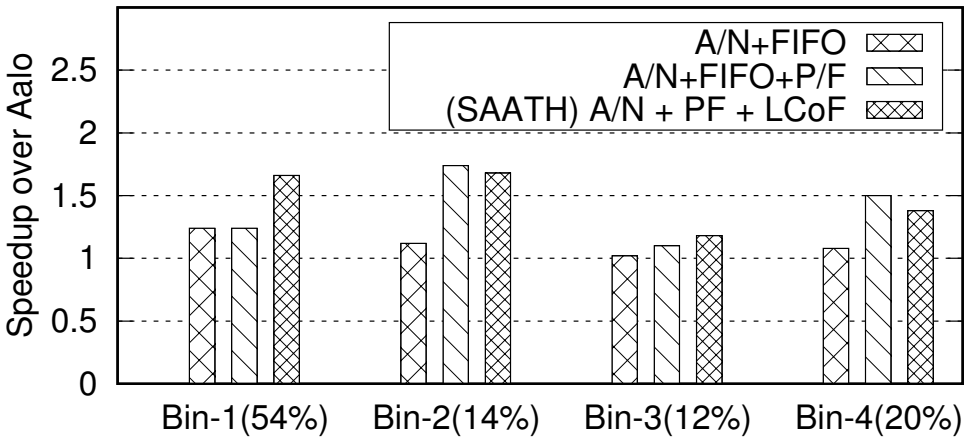


Figure 2.11.: Saath speedup breakdown into bins based on size and width shown in table 2.1 for FB trace. The numbers in x-label denote fraction of all coflows in that bin. Y-axis shows the median speedup.

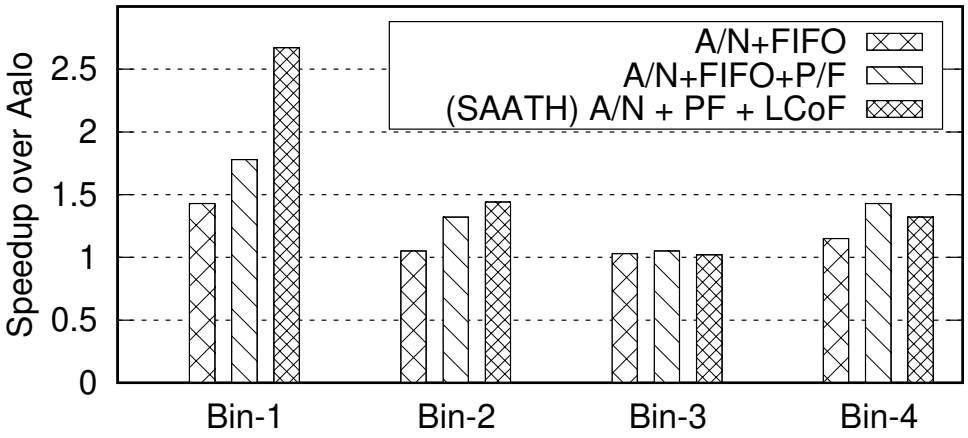


Figure 2.12.: Saath speedup breakdown into bins based on size and width shown in table 2.1 for OSP trace. We omit the distribution of coflows in individual bins for proprietary reasons. Y-axis shows the median speedup.

to the use of FIFO, where a wide coflow causes Head-of-Line (HoL) blocking other potentially short coflows.

Second, while all-or-none only addresses one limitation of the out-of-sync problem, using per-flow queue thresholds (P/F) addresses the second limitation of out-of-sync problem by quickly jumping the queues (§2.3). As a result, A/N+P/F improves

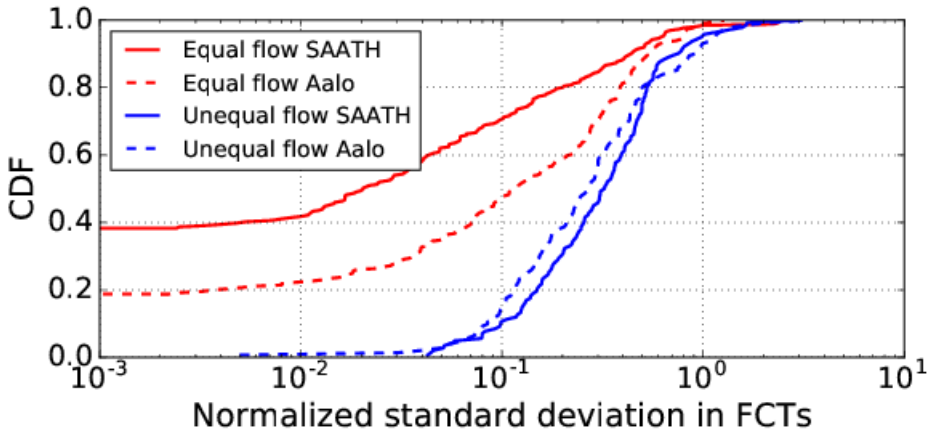
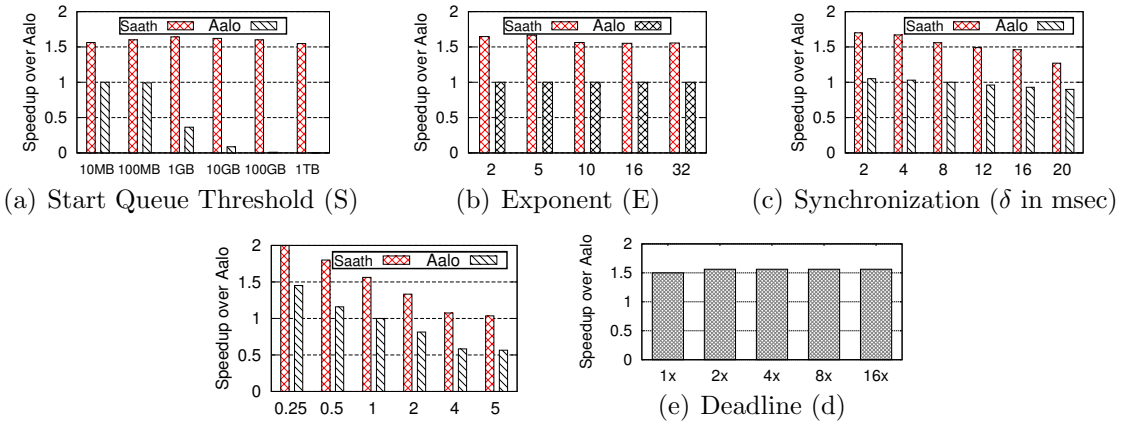


Figure 2.13.: Normalized standard deviation of FCTs of multi-flow coflows, under Saath and Aalo using FB trace. We have excluded the coflows with width = 1 (23.5%).

the speedup over Aalo in the FB (OSP) trace to 1.3× (1.32×) in the median case, and 3.83× (13×) at P90.

We again zoom into the improvement in using P/F on coflows in different bins, shown in Fig2.11 and Fig. 2.12. P/F is highly effective for coflows in bins 2 and 4, which are wider (width > 10). The larger numbers of flows in these wider coflows increase the chance of at least one flow crossing the per-flow queue threshold and thus move the coflows to the next queues faster.

Third, we replace FIFO with LCoF and retain A/N and P/F from previous experiment. This combines all the three complimentary ideas in Saath, and is labeled as Saath in Fig. 2.10. We see that using LCoF achieves a median speedup over Aalo of 1.53× (P90=4.5×) for the FB trace, and of 1.42× (P90=37×) for the OSP trace. This is primarily because LCoF schedules coflows using Least Contention First and reduces the HoL blocking in FIFO. As shown in Fig. 2.11 and Fig. 2.12, LCoF improves the CCT of coflows in all bins. Particularly, it substantially benefits *short and thin* coflows (bin-1), as HoL blocking due to FIFO blocks these coflows the most,



(a) Start Queue Threshold (S) (b) Exponent (E) (c) Synchronization (δ in msec)

(d) Arrival time scaling (A) (e) Deadline (d)

Figure 2.14.: SAATH sensitivity analysis.

without significantly impacting the coflows in other bins. This shows that LCoF on top of all-or-none is effective.

Lastly, Fig. 2.13 shows the CDF of the standard deviation of FCTs of individual coflows with more than one flow under SAATH and Aalo for the FB trace. We show results separately for coflows with equal and unequal flow length. We see that SAATH significantly reduces the variation in FCTs: 40% of coflows with equal flow lengths finished their flows at the same time, as opposed to 20% in Aalo, and 71% of them had normalized FCT deviation under 10%, compared to 47% in Aalo. We note that SAATH does not completely eliminate the out-of-sync problem because of work conservation (§2.3). We do not show the results for the OSP trace for proprietary reasons.

2.6.3 Sensitivity Analysis

We next evaluate the sensitivity of SAATH and Aalo to various design parameters. Due to space limitation, we only show the results for the FB trace. The results for the OSP trace are similar.

Start queue threshold (S): In this experiment, we vary queue threshold of the starting (highest priority) queue, which controls how long coflows stays in the starting

queue. Fig. 2.14(a) shows that Aalo is highly sensitive to S. This is because as S grows, more coflows stay in the highest priority queue, and Aalo performs worse due to HoL blocking under FIFO, which is addressed by LCoF in Saath. In contrast, Saath is relatively insensitive to S, precisely because LCoF alone addresses the HoL blocking weakness of FIFO.

Multiplication factor (E): In this experiment, we vary the queue threshold growth factor E from 2 to 32. Recall that the queue thresholds are computed as $Q_q^{hi} = Q_{q-1}^{hi} \cdot E$. Thus, as E grows, the number of queues decreases. As shown in Fig. 2.14(b), Saath and Aalo are both insensitive to E.

Synchronization interval (δ): Recall that the global coordinator calculates a new schedule every δ interval. In this experiment, we vary δ and measure its impact on the CCT. Fig. 2.14(c) shows as δ increases, the speedup in Aalo and Saath both diminish. As shown in §2.7.3, Saath comfortably finishes calculating each new schedule within 8 msec even during busy periods (with an average of 0.57 msec and P90 of 2.85 msec). Thus when δ increases, the CCT increases because the ports may finish the current scheduled flows and become idle before receiving a new schedule from the coordinator. This shows that in general shorter scheduling intervals help to keep all the ports busy which in turn requires the global coordinator to be able to calculate schedules quickly.

Coflow arrival time (contention): In this experiment, we vary the arrival time (A) between the coflows to vary the coflow *contention*. The x-axis in Fig. 2.14(d) shows the factor by which the arrival times are sped up. For example, $A = 0.5$ denotes that coflows arrive $0.5\times$ faster ($2\times$ slower), whereas $A = 4$ denotes that coflows arrive $4\times$ faster. The y-axis shows the speedup compared to the default Aalo, *i.e.,* Aalo with $A = 1$. Fig. 2.14(d) shows that as A increases, the overall speedup in both Saath and Aalo decreases. This is expected because increasing A causes more contention and the coflows are queued up longer increasing their CCT under both schemes.

More importantly, when we increase A, the speedup of Saath over Aalo increases, from $1.53\times$ to $1.9\times$, showing the higher the contention, the more Saath outperforms Aalo, using the LCoF policy.

Coflow deadline (d): Recall that LCoF by default does not provide the starvation-free guarantees, and can starve the coflows with high contention. To avoid starvation, Saath assigns each coflow a deadline of $d \cdot C_q \cdot t$ (D5 in §2.4.2), where $C_q \cdot t$ denotes the estimated deadline based on current coflows if scheduled under FIFO. In this experiment, we measure the impact of d on the CCT speedup, where d is varied from 1 to 16. Fig. 2.14(e) shows that Saath is insensitive to d, and comfortably schedules the coflows within the deadline. Even when the deadlines are as per FIFO (d = 1), Saath can achieve a median speedup of $1.5\times$ over Aalo. A small drop in the speedup at d = 1 is because starvation-free scheduling occasionally kicks in, and the coflows that passed the deadline are forced to be scheduled, which would not have been under LCoF only, resulting in worse CCT. At higher values of d (> 2), Saath has more freedom to re-order the coflows to facilitate their CCT without violating the deadlines.

2.7 Testbed Evaluation

Testbed setup: Similar to simulations, our testbed evaluation keeps the same job arrival times, flow lengths and flow ports in trace replay. All the experiments use the default parameter values of K, E, S, δ. For the testbed experiments, we rerun the trace on Spark-like framework on a 150-node cluster in Microsoft Azure [48]. We use the FB trace as it has a cluster size similar to that of our testbed. The coordinator runs on a Standard F4s VM (4 cores and 8GB memory) based on the 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor. Each local agent runs on a D2v2 VM (2 cores and 7GB memory) based on the same processor and with 1 Gbps network connection.

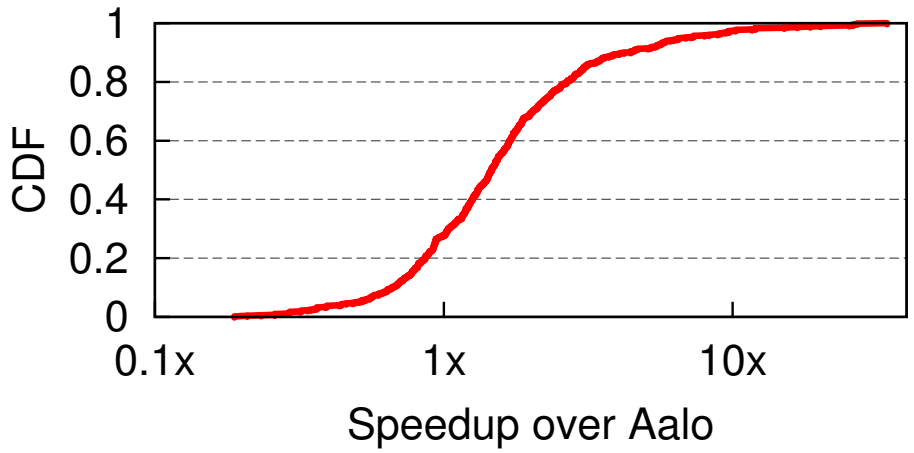


Figure 2.15.: [Testbed] Speedup in CCT in Saath.

In testbed evaluation, we compare Saath against Aalo. The primary evaluation metric is the speedup in CCT. We also compare the speedup in job completion time and Saath scheduling overheads.

2.7.1 Improvement in CCT

In this experiment, we measure the speedup in Saath compared to Aalo. Fig.2.15 shows that the ratio of CCT under Saath over that under Aalo ranges between 0.09-12.15×, with an average of 1.88× and a median of 1.43× compared to under Aalo, which is close to the reduction observed in the simulation experiments. Although Saath improves the CCT for the majority of CoFlows (>70%), it slows down some of the CoFlows. These CoFlows are favored by FIFO as they arrived early. The same CoFlows would be pushed back by Saath if they observe high contention. Additionally, the starvation avoidance rarely kicked in ($< 1\%$) even for $d = 2$. This experiment shows that Saath is effective in improving CCT in real settings.

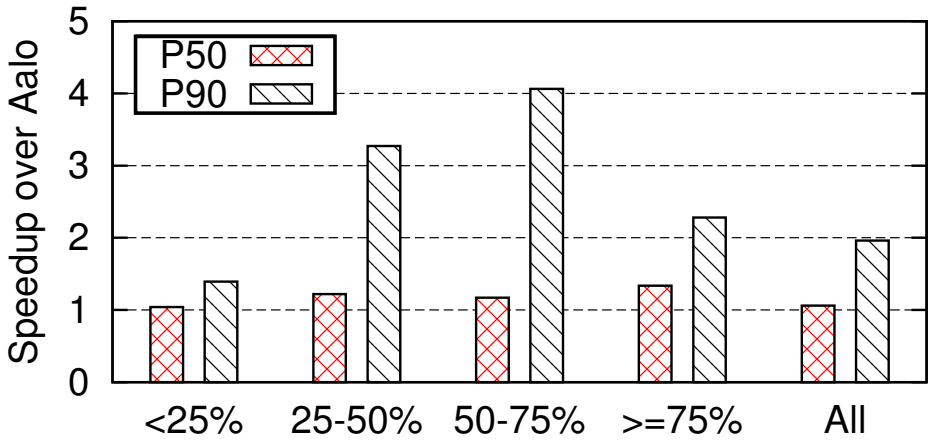


Figure 2.16.: [Testbed] Speedup in job completion time using Saath over Aalo. X-axis shows the fraction of total job time spent in shuffle phase.

2.7.2 Job Completion Time

Next, we evaluate how the improvements in CCT affects the job completion time. In data clusters, different jobs spend different fractions of their total job time in data shuffle. In this experiment, the fraction of time that the jobs spent in the shuffle phase follows the same distribution used in Aalo [11]. Fig. 2.16 shows that Saath substantially speeds up the job completion time of the shuffle-heavy jobs (shuffle fraction $\geq$50%) by 1.83× on average (P50 = 1.24× and P90 = 2.81×). Additionally, across all jobs, Saath reduces the job completion time by 1.42× on average (P50 = 1.07× and P90 = 1.98×). This shows that the benefits in improving CCT translates into better job completion time. As expected, the improvement in job completion time is smaller than the improvements in CCT because job completion time depends on time spent in both compute and communication (shuffle) stages, and Saath improves only the communication stage.

Table 2.2.: [Testbed] Resource usage in Saath and Aalo.

		Saath		Aalo	
		Average	P90	Average	P90
Global co-ordinator	CPU (%)	37.8	42.7	33.5	35.5
	Memory (MB)	229	284	267	374
	Total time (LCoF / All-or-none) (msec)	0.57 (0.02 / 0.24)	2.85 (0.03 / 0.7)	0.1	0.2
Local node	CPU (%)	5.6	5.7	5.5	5.7
	Memory (MB)	1.68	1.7	1.75	1.78

2.7.3 Scheduling Overhead

We next evaluate the overheads in Saath at the coordinator and the local agents. Table 2.2 shows the overheads in terms of CPU and memory utilization for both Saath and Aalo. We measure the overheads in two cases: (1) Average: the average utilization during the entire execution of the trace, (2) Busy: the 90-th percentile utilization indicating the performance during busy periods when a large number of CoFlows arrive. As shown in Table 2.2, Saath has a very small overhead at the local nodes, where the CPU and memory utilization is minimal even during busy times. The global coordinator also uses the server resources economically – compared to Aalo, overall Saath incurs 4.3% increase in average CPU utilization. Finally, the scheduling latency is overall small, although higher than Aalo due to all-or-none, LCoF and per-flow scheduling. The time it takes the coordinator to calculate new schedules is 0.57 msec on average and 2.85 msec at P90.

We also break down the computation time at the coordinator in Saath into the time spent in ordering CoFlows (using per-flow thresholds and LCoF), scheduling using all-or-none, and the rest which is for assigning rates for work conservation. Table 2.2 shows that most of the computation time is spent on assigning rates for work conservation; ordering the CoFlows using LCoF accounts for less than half of the schedule compute time.

In summary, our overhead evaluation shows that the cost of the Saath scheduling algorithm is moderate, and that the CCT improvement in Saath outweighs its costs.

2.8 Summary

In this chapter, we have shown that the prior-art coflow scheduler Aalo suffers from the out-of-sync problem and uses a simple FIFO intra-queue scheduling policy which ignores coflow contention across ports. We present Saath that addresses the limitations in Aalo by exploiting the spatial dimension in scheduling coflows. Saath uses all-or-none to schedule all flows of a coflow together, and the Least-Contention-First policy to decide on the order of coflows from the same priority queue. Our evaluation using a 150-node testbed in Microsoft Azure and large scale simulations using traces from two production clusters shows that compared to Aalo, Saath reduces the CCT by 1.53× and 1.42× in median, and 4.50× and 37× at the 90-th percentile for two traces.

Appendix A: SJF for Coflows is Sub-optimal

Fig. 2.17 illustrates that SJF is *not optimal* even when all the coflows arrive at the same time, and their durations are known apriori. When an i-th coflow is scheduled, the increase in the waiting time of other coflows is $t_i \cdot k_i$, where t_i is the duration coflow i is scheduled, and k_i is the contention, *i.e.,* the number of other coflows blocked. In coflows, k_i is non-uniform as different ports have different coflows and differen coflows reside at different numbers of ports. However, SJF only considers t_i, and is agnostic to k_i, and thus results in higher waiting time as shown in Fig. 2.17. In this example, $k_1 = 2, k_2 = k_3 = 1$, and t_1, t_2, t_3 are shown in the figure. SJF schedules coflow C1 first as it is the shortest. However, C1 blocks the other two coflows, increasing the total waiting time by $2 \cdot 5t$, which leads to sub-optimal average CCT.

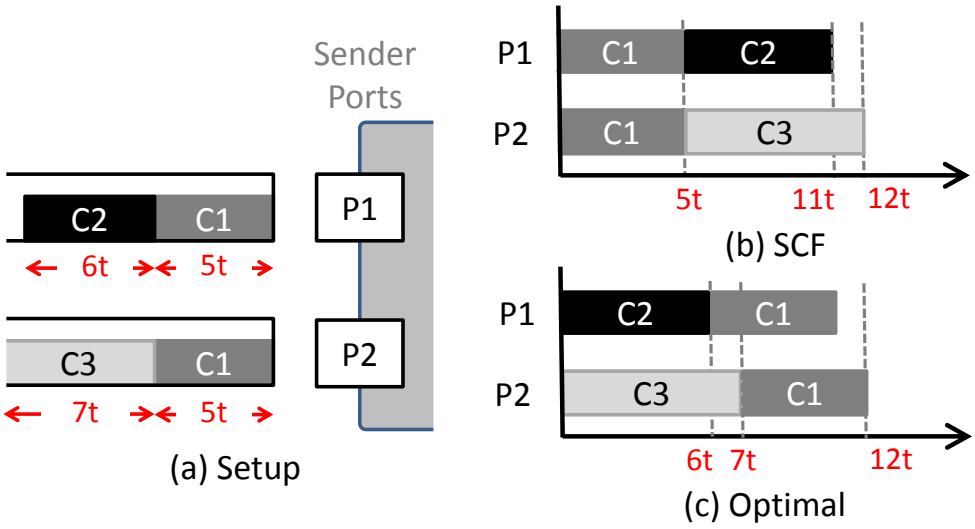


Figure 2.17.: SJF is sub-optimal. (a) shows the setup and coflow durations, whereas (b) and (c) show the coflow progress. The average CCT in (b) is $\frac{5+11+12}{3} = 9.3$, and in (c) is $\frac{12+6+7}{3} = 8.3$.

3 Philae

This chapter is based on a publication in *2019 Annual Technical Conference* by USENIX Association [49].
Following is its DOI: https://www.usenix.org/conference/atc19/presentation/jajoo

In Chapter 2, we show that by synchronizing in space, we can significantly improve the average coflow completion time. In this chapter, we focus on learning coflow sizes online, another crucial aspect of online coflow scheduling, by exploiting the spatial dimension, where the coflow size is the total size of its constituent flows.

3.1 Motivation and Problem Statement

In big data analytics jobs, speeding up the communication stage where the data is transferred between compute nodes is important to speed up the jobs. However, improving network level metrics such as flow completion time may not translate into improvements at the application level metrics such as job completion time. The coflow abstraction [25] was proposed to bridge such a gap. The abstraction captures the collective network requirements of applications, as reduced coflow completion time (CCT) can directly lead to faster job completion time [41, 50].

Scheduling coflows in non-clairvoyant settings, *i.e.,* without any apriori knowledge, is a daunting task. Ideally, if the coflow sizes (bytes to be transfered) are known apriori, then one can apply variations of the classic Shortest-Job-First (SJF) algorithm (*e.g.,* [34]).

There have been a number of efforts on network designs for coflows [34, 51, 52] that assume complete prior knowledge of coflow sizes. However, in many practical settings, coflow characteristics are not known a priori. For example, multi-stage jobs

pipeline data from one stage to the next as soon as the data is generated, which makes it difficult to know the size of each flow [53, 54]. A recent study [53] shows various other reasons why it is not very plausible to learn flow sizes from applications, for example, learning flow sizes from applications requires changing either the network stack or the applications.

The major challenge in developing an effective non-clairvoyant coflow scheduling scheme has centered around how to learn the coflow sizes online quickly and accurately, as once the coflow sizes (bytes to be transferred) can be estimated, one can apply variations of the classic Shortest-Job-First (SJF) algorithm such as Shortest Coflow First [34] or apply an LP solver (*e.g.,* [52]).

3.2 Existing Approach for Learning Coflow Sizes and its Limitations

State-of-the-art online non-clairvoyant schedulers such as Saath [40], Gravtion [55] and Aalo [11] in essence learn coflow sizes and approximate SJF using discrete priority queues, where all newly arriving coflows start from the highest priority queue, and move to lower priority queue as they send more data (without finishing), *i.e.,* cross the per-queue thresholds. In this way, the smaller coflows finish in high priority queues, while the larger coflows gradually move to the lower priority queues where they finish after smaller coflows.

To realize the above idea in scheduling coflows which have flows at many network ports, *i.e.,* in a distributed setting, Aalo uses a global coordinator to assign coflows to logical priority queues, and uses the total bytes sent by all flows of a coflow as its logical "length" in moving coflows across the queues. The logical priority queues are mapped to local priority queues at each port, and the individual local ports then schedule the flows in its local priority queues, *e.g.,* by enumerating flows from the highest to lowest priority queues and using FIFO to order the flows within each queue.

In essence, Aalo learns coflow sizes by actually scheduling the coflow, a "try and miss" approach to approximate SJF. As coflow sizes are not known, in each queue,

Aalo schedules each coflow for a fixed amount of data (try). If the coflow does not finish (miss), it is demoted to a lower priority queue. Afterwards, such a coflow will no longer block coflows in higher priority queues.

Using multiple priority queues to learn the relative coflow sizes of coflows this way, however, negatively affects the average CCT and the scalability of the coordinator:

(1) Intrinsic queue-transit overhead: Every coflow that Aalo transits through the queues before reaching its final queue worsens the average CCT because during transitions, such a coflow effectively *blocks other shorter* coflows in the earlier queues it went through, which would have been scheduled before this coflow starts in a perfect SJF.

(2) Overhead due to inadvertent round-robin: Although Aalo attempts to approximate SJF, it inadvertently ends up doing *round-robin* for coflows of similar sizes as it moves them across queues. Aalo assigns a *fixed threshold of data transfer* for each coflow in each queue. Assume there are "N" coflows in a queue that do not finish in that queue. Aalo schedules one coflow (chosen using FIFO) and demotes it to a lower priority queue when the coflow reaches the data threshold. At that point, the next coflow from the same queue is scheduled, which joins the previous coflow at a lower priority queue after exhausting its quantum, and this cycle continues as coflows of similar sizes move through the queues. Effectively, these coflows experience the round-robin scheduling which is known to have the worst average CCT [56], when jobs are of similar sizes.

(3) Limited scalability from frequent updates from local ports: To support the try-and-error style learning, the coordinator requires frequent updates from all local ports of the bytes sent for each coflow in order to move coflows across multiple queues timely. This results in high load on the central coordinator from receiving frequent updates and calculating and sending new rate allocations, which limits the scalability of the overall approach.

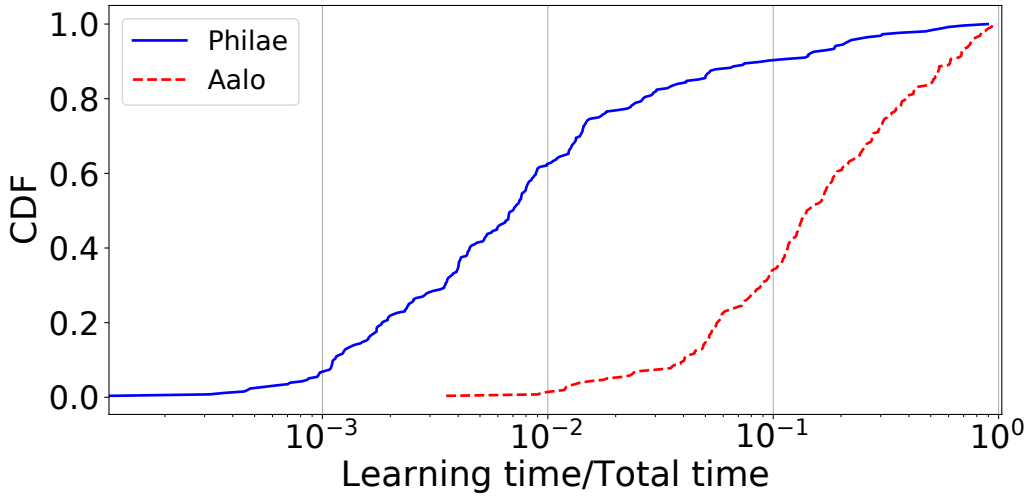


Figure 3.1.: CDF of learning overhead per coflow, *i.e.,* the time to reach the correct priority queue as a fraction of CCT, excluding coflows directly scheduled by PHILAE or finish in Aalo's first queue.

Empirical measurement We quantify the coflow size *learning overhead* of Aalo, defined as the portion of the bytes of a coflow that has been transferred (or the fraction of its CCT spent in doing so) before reaching its correct queue, using a trace from Facebook clusters [39] (see detailed methodology in §3.9). Figure 3.1 shows that 40% of the coflows that moved beyond the initial queue reached the correct priority queue after spending more than 20% of their CCT moving across early queues.

3.3 Background and Problem Statement

We start with a brief review of the coflow abstraction and the need for non-clairvoyant coflow scheduling. We then state the network model and problem formulation.

Coflow abstraction In data-parallel applications such as Hadoop [2] and Spark [22], the job completion time heavily depends on the completion time of the communication stage [50, 57]. The coflow abstraction [25] was proposed to speed up the communication stage to improve application performance. A coflow is defined as a set of flows

between several nodes that accomplish a common task. For example, in map-reduce jobs, the set of all flows from all map to all reduce tasks in a single job forms a typical coflow. The coflow completion time (CCT) is defined as the time duration between when the first flow arrives and the last flow completes. In such applications, improving CCT is more important than improving individual flows' completion time (FCT) for improving the application performance [11, 34, 40, 41, 55].

Non-clairvoyant coflows Data-parallel directed acyclic graphs (DAGs) typically have multiple stages which are represented as multiple coflows with dependencies between them. Recent systems (*e.g.,* [**?**, 54, 58, 59]) employ optimizations that pipeline the consecutive computation stages which removes the barrier at the end of each coflow, making knowing flow sizes of each coflow beforehand difficult. Thus in this chapter, we focus on *non-clairvoyant* coflow scheduling which do not assume knowledge about coflow characteristics such as flow sizes upon coflow arrival.

Non-blocking network fabric We assume the same non-blocking network fabric model in recent network designs for coflows [11, 34, 40, 52, 55], where the datacenter network fabric is abstracted as a single non-blocking switch that interconnects all the servers, and each server (computing node) is abstracted as a network port that sends and receives flows. In such a model, the ports, *i.e.,* server uplinks and downlinks, are the only source of contention as the network core is assumed to be able to sustain all traffic injected into the network. We note that the abstraction is to simplify our description and analysis, and is not required or enforced in our evaluation.

Problem statement Our goal is to *develop an efficient non-clairvoyant coflow scheduler that optimizes the communication performance, in particular the average CCT, of data-intensive applications without prior knowledge, while guaranteeing starvation freedom and work conservation and being resilient to the network dynamics.* The problem of non-clairvoyant coflow scheduling is NP-hard because coflow scheduling even assuming all coflows arrive at time 0 and their size are known in advance is

already NP-hard [34]. Thus practical non-clairvoyant coflow schedulers are approximation algorithms. Our approach is to dynamically prioritize coflows by efficiently learning their flow sizes online.

3.4 Key Idea

Our new non-clairvoyant coflow scheduler design, Philae, is based on a key observation about coflows that a coflow has a *spatial dimension*, *i.e.,* it typically consists of many flows. We thus propose to explicitly learn coflow sizes online by using *sampling*, a highly effective technique used in large-scale surveys [60]. In particular, Philae preschedules sampled flows, called *pilot flows*, of each coflow and uses their measured sizes to estimate the coflow size. It then resorts to SJF or variations using the estimated coflow sizes.

Developing a complete non-clairvoyant coflow scheduler based on the simple sampling idea raises three questions:

(1) Why is sampling more efficient than the priority-queue-based coflow size learning? Would scheduling the remaining flows after sampled pilot flows are completed adversely affect the coflow completion time?

(2) Will sampling be effective in the presence of skew of flow sizes?

(3) How to design the complete scheduler architecture?

We answer the first two questions below, and present the complete architecture design in §3.5.

3.4.1 Why is Sampling more Efficient?

Scheduling pilot flows first before the rest of the flows can potentially incur two sources of overhead. First, scheduling pilot flows of a newly arriving coflow consumes port bandwidth which can delay other coflows (with already estimated sizes). However, compared to the multi-queue based approach, the overhead is much smaller for two reasons: (1) Philae schedules only a small subset of the flows (*e.g.,* fewer than

1% for coflows with many flows). (2) Since the CCT of a coflow depends on the completion of its last flow, some of its earlier finishing flows could be delayed without affecting the CCT. Philae exploits this observation and schedules pilot flows on the least-busy ports to increase the odds that it only affects earlier finishing flows of other coflows.

Second, scheduling pilot flows first may elongate the CCT of the newly arriving coflow itself whose other flows cannot start until the pilot flows finish. This is again typically insignificant for two reasons: (1) A coflow (*e.g.,* from a MapReduce job) typically consists of flows from all sending ports to all receiving ports. Conceptually, pre-scheduling one out of multiple flows from each sender may not delay the coflow progress at that port, because all flows at that port have to be sent anyway. (2) Coflow scheduling is of high relevance in a busy cluster (when there is a backlog of coflows in the network), in which case the CCT of coflow is expected to be much higher than if it were the only coflow in the network, and hence the piloting overhead is further dwarfed by a coflow's actual CCT.

3.4.2 Why is Sampling Effective in the Presence of Skew?

The flow sizes within a coflow may vary (*skew*). Intuitively, if the skew across flow sizes is small, sampling even a small number of pilot flows will be sufficient to yield an accurate estimate. Interestingly, even if the skew across flow sizes is large, our experiment indicates that sampling is still highly effective. In the following, we give both the intuition and theoretical underpinning for why sampling is effective.

Consider, for example, two coflows and the simple setting where both coflows share the same set of ports. In order to improve the average CCT, we wish to schedule the shorter coflow ahead of the longer coflow. If the total sizes of the two coflows are very different, then even a moderate amount of estimation error of the coflow sizes will not alter their ordering. On the other hand, if the total sizes of the two coflows are close to each other, then indeed the estimation errors will likely alter their ordering.

However, in this case since their sizes are not very different anyway, switching the order of these two coflows will not significantly affect the average CCT.

Analytic results. To illustrate the above effect, we show that the gap between the CCT based on sampling and assuming perfect knowledge is small, even under general flow size distributions. Specifically, coflows C_1 and C_2 have cn_1 and cn_2 flows, respectively. Here, we assume that n_1 and n_2 are fixed constants. Thus, by taking c to be larger, we will be able to consider wider coflows. Assume that each flow of C_1 (correspondingly, C_2) has a size that is distributed within a bounded interval $[a_1, b_1]$ ($[a_2, b_2]$) with mean μ_1 (μ_2), *i.i.d.* across flows. However, the exact distributions can be arbitrary. Let T^c be the total completion time when the exact flow sizes are known in advance. Let $\tilde{T}^c$ be the average CCT by sampling m_1 and m_2 flows from C_1 and C_2, respectively. Without loss of generality, we assume that $n_2\mu_2 \geq n_1\mu_1$. Then, using Hoeffding's Inequality, we can show that,

$$\lim_{c\to\infty} \frac{\tilde{T}^c - T^c}{T^c} \leq 4\exp\left[-\frac{2(n_2\mu_2 - n_1\mu_1)^2}{\left(\frac{n_2(b_2-a_2)}{\sqrt{m_2}} + \frac{n_1(b_1-a_1)}{\sqrt{m_1}}\right)^2}\right] \frac{n_2\mu_2 - n_1\mu_1}{n_2\mu_2 + 2n_1\mu_1} \tag{3.1}$$

(Note that here we have used the fact that, since both coflows share the same set of ports and c is large, the CCT is asymptotically proportional to the coflow size.)

Equation (3.1) can be interpreted as follows. First, due to the first exponential term, the relative gap between $\tilde{T}^c$ and T^c decreases as $b_1 - a_1$ and $b_2 - a_2$ decrease. In other words, as the skew of each coflow decreases, sampling becomes more effective. Second, when $b_1 - a_1$ and $b_2 - a_2$ are fixed, if $n_2\mu_2 - n_1\mu_1$ is large (i.e., the two coflow sizes are very different), the value of the exponential function will be small. On the other hand, if $n_2\mu_2 - n_1\mu_1$ is close to zero (i.e., the two coflow sizes are close to each other), the numerator on the second term on the right hand side will be small. In both cases, the relative gap between $\tilde{T}^c$ and T^c will also be small, which is consistent with the intuition explained earlier. The largest gap occurs when $n_2\mu_2 - n_1\mu_1$ is on the same order as $\frac{n_2(b_2-a_2)}{\sqrt{m_2}} + \frac{n_1(b_1-a_1)}{\sqrt{m_1}}$. Finally, although these analytical results assume

that both coflows share the same set of ports, similar conclusions on the impact of estimation errors due to sampling also apply under more general settings.

The above analytical results suggest that, when c is large, the relative performance gap for CCT is a function of the number of pilot flows sampled for each coflow, but is independent of the total number of flows in each coflow. In practice, large coflows will dominate the total CCT in the system. Thus, these results partly explain that, while in our experiments the number of pilot flows is never larger than 1% of the total number of flows, the performance of our proposed approach is already very good.

Finally, the above analytical results do not directly tell us how to choose the number of pilot flows, which likely depends on the probability distribution of the flow size. In practice, we do not know such distribution ahead of time. Further, while choosing a larger number of pilot flows reduces the estimation errors, it also incurs higher overhead and delay. Therefore, our design (§3.5) needs to have practical solutions that carefully address these issues.

3.5 Philae Design

In this section, we present the detailed design of Philae, which addresses three design challenges: (1) *Coflow size estimation:* How to choose and schedule the pilot flows for each newly arriving coflow? (2) *Starvation avoidance:* How to schedule coflows after size estimation using variations of SJF that avoid starvation? (3) *Coflow scheduling:* How to schedule among all the coflows with estimated sizes?

3.5.1 Philae architecture

Fig. 3.2 shows the Philae architecture. Philae models the entire datacenter as a single big-switch with each computing node as an individual port. The scheduling task in Philae is divided among (1) a central coordinator, and (2) local agents that run on individual ports. A computing framework such as Spark [61] first registers (removes) a coflow when a job arrives (finishes). Upon a new coflow arrival, old

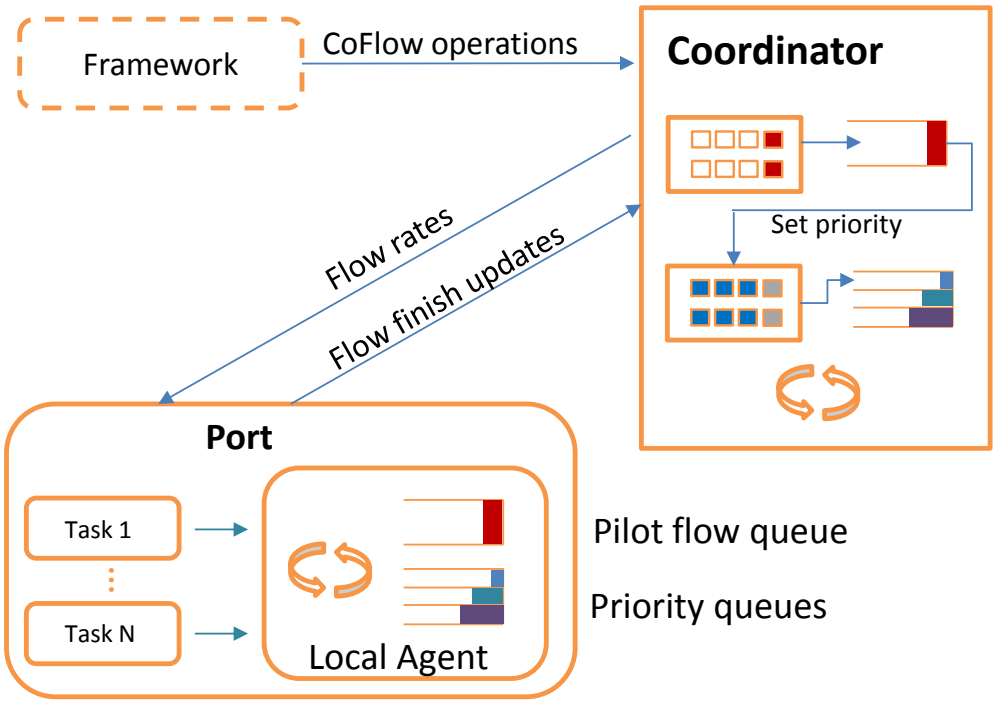


Figure 3.2.: PHILAE architecture.

coflow completion, or pilot flow completion, the coordinator calculates a new coflow schedule, which includes (1) coflows that are to be scheduled in the next time slot, and (2) flow rates for the individual flows of a coflow, and pushes this information to the local agents which use this information to allocate their bandwidth. The local agents will follow the current schedule until they receive a new schedule.

3.5.2 Sampling pilot flows

As discussed in §3.4, PHILAE estimates the size of a coflow online by actually scheduling a subset of its flows (*pilot flows*) at their ports. We do not schedule the flows of a coflow other than the pilot flows until the completion of the pilot flows in order to avoid unnecessary extra blocking of other potentially shorter coflows.

How many pilot flows? When a new coflow arrives, PHILAE first needs to determine the number of pilot flows. As discussed at the end of §3.4, the number of pilot flows affects the trade-off between the coflow size estimation accuracy and scheduling overhead. For coflows with skewed flow sizes, accurately estimating the total coflow size potentially requires sampling the sizes of many pilot flows. However, scheduling pilot flows has associated overhead, *i.e.,* if the coflow turns out to be a large coflow and should have been scheduled to run later under SJF.

We explore several design options for choosing the number of pilot flow. Two natural design choices are using a constant number of pilot flows or a fixed fraction of the total number of flows of a coflow. In addition, we observe that typical coflows consist of flows between a set of senders (*e.g.,* mappers) and a set of receivers (*e.g.,* reducers) [1]. We thus include a third design choice of a fixed fraction of sending ports. This design also spreads the pilot flows to avoid having multiple pilot flows contending for the same sending ports. We empirically found that (§3.9.2) limiting the pilot flows to 5% to 10% of the number of its sending ports (*e.g.,* mappers in a MapReduce coflow) strikes a good balance between estimation accuracy and overhead. We note the total number of flows sampled in this case is still under 1%.

Finally, we *estimate the total coflow size as* $S = f_i \cdot N$, where N is the number of flows in a coflow, and f_i is the average size of the sampled pilot flows.

Which flows to probe? Second, PHILAE needs to decide which ports to schedule the chosen number of probe flows for a coflow. For this, we use a simple heuristic where, upon the arrival of a new coflow, we select the ports for its pilot flows that are least busy, *i.e.,* having pilot flows from the least number of other coflows. PHILAE starts with the least busy sending port and iterates over receiving ports starting with the least busy receiving port and assigns the flow if it exists. It then updates the statistics for the number of pilot flows scheduled at each port and repeats the above process. Such a choice will likely delay fewer coflows when the pilot flows are scheduled and hence reduce the elongation on their CCT. We note that such an online heuristic

may not be optimal; more sophisticated algorithms can be derived by picking ports for multiple coflows together. However, we make this design choice for its simplicity and low time complexity to ensure that the coordinator makes fast decisions.

How to schedule pilot flows? In Philae, we prioritize the pilot flows of a new coflow over existing flows to accelerate learning the size of the new coflow. In particular, at each port, pilot flows have high priority over non-pilot flows. If there are multiple outstanding pilot flows (of different coflows) at a port, Philae schedules them in the FIFO order.

3.5.3 Coflow scheduling with starvation avoidance

Once the sizes of coflows are learned, we can apply variations of the SJF policy to schedule them. However, it is well known that such policies can lead to starvation.

There are many ways to mitigate the starvation issue. However, a subtlety arises where *even slight difference in how starvation is addressed can result in different performance*. For example, the multiple priority queues in Aalo has the benefit of ensuring progress of all coflows, but assigning different time-quanta to different priority queues can result in different average CCT for the same workload. To ensure the fairness of performance comparison with Aalo, we need to ensure that both Philae and Aalo provide the same level of starvation freedom (or progress measure).

For this reason, in discussions and experiments mentioned in this report, we inherit the multiple priority queue structure from Aalo for coflow scheduling. As in Aalo, Philae sorts the coflows among multiple priority queues. In particular, Philae uses N queues, Q_0 to Q_{N-1}, with each queue having lower queue threshold Q_q^{lo} and higher threshold Q_q^{hi}, where $\mathrm{Q}_0^{lo} = 0$, $\mathrm{Q}_{N-1}^{hi} = \infty$, $\mathrm{Q}_{q+1}^{lo} = \mathrm{Q}_q^{hi}$, and the queue thresholds grow exponentially, *i.e.,* $\mathrm{Q}_{q+1}^{hi} = \mathrm{E} \cdot \mathrm{Q}_q^{hi}$.

The overall coflow scheduling in Philae works as follows. After the coflow size is estimated using pilot flows, Philae assigns the coflow to the priority queue using inter-coflow policies discussed in §3.5.4. Within a queue, we use FIFO to schedule

coflows. Lastly, we use weighted sharing of network bandwidth among the queues, where a priority queue receives a network bandwidth based on its priority. As in Aalo, the weights decrease exponentially with decrease in the priority of the queues.

Using FIFO within the priority queue and weighted fair sharing among the queues together ensure the same starvation freedom and thus meaningful performance comparison between PHILAE and Aalo [11].

3.5.4 Inter-coflow scheduling policies

In PHILAE, we explore four different scheduling policies based on different combinations of coflow size and contention, two size-based policies (*A, B*) as in Aalo, a contention-based, similar to the intra-queue policy used in Saath [40] (*C*), and a new contention-and-length-based policy (*D*):

(A) Smallest job first: Coflows are sorted based on coflow size ($l \cdot n$).

(B) Smallest remaining data first: Coflows are sorted based on remaining data ($l \cdot n - d$).

(C) Least contention first: Coflows are sorted based on their contention (c).

(D) Least length-weighted total-port contention first: Coflows are sorted based on the sum of port-wise contention times estimated flow length $\sum_p c^p \cdot l$.

We use the following parameters of a coflow to define the metrics in scheduling algorithms: (1) average flow length (l) from piloting, (2) number of flows (n), (3) number of sender and receiver ports (s, r), (4) total amount of data sent so far (d), (5) contention (c), defined as the number of other coflows sharing any ports with the given coflow, and (6) port-wise contention (c^p), defined as the number of other coflows blocked at a given port p.

PHILAE uses Policy D by default, as it results in the least average CCT (§3.9). For all policies, we continue to use the priority-queue based scheduling, and the algorithms only differ in what metric they use in assigning coflows to the priority queues. In contrast, Aalo does not handle inter-coflow contention, and uses the total bytes sent so far (d) to move coflows across multiple priority queues.

3.5.5 Rate Allocation

Once the scheduling order of the coflows is determined, we need to determine the rates for the individual flows at each port. First, since we want to quickly finish the pilot flow, at any port that has pilot flows, PHILAE assigns the *entire* port bandwidth to the pilot flows. For the remaining ports, as discussed in §3.5.3, across multiple queues, PHILAE assigns weighted shares of the port bandwidth, by assigning them varying numbers of scheduling intervals according to the weights assigned to each priority queues.

Second, at each scheduling interval, PHILAE assigns rates for the flows of the coflow in the head of the FIFO queue as follows. It assigns equal rates at all the ports containing its flows as there is no benefit in speeding-up its flows at certain ports when its CCT depends on the slowest flow. At each port, we could use max-min fairness to schedule the individual flows of the coflow (to different receivers), and then assign the rate of the slowest flow to all the flows in the coflow. Afterwards, the port-allocated bandwidths are incremented accordingly at the coordinator, which then allocates rates for the next coflow in the same FIFO queue, and so on.

Though the above max-min approach has the advantage of minimizing bandwidth wastage, it slows down the coordinator which has to iterate over many flows. In our experiments, we used a simple scheme where we assign the *entire* bandwidth at the sender and receiver ports to one flow of the coflow at the head of the FIFO queue at a time. We found that this simple scheme has very marginal effect on CCTs but makes the rate assignment process considerably faster.

3.5.6 Additional design issues

Thin coflow bypass Recall that, in PHILAE, when a new coflow arrives, PHILAE only schedules its pilot flows. All other flows of that coflow are delayed until the pilot flows finish and coflow size is known. However, such a design choice can inadvertently

lead to higher CCTs for coflows, particularly for thin coflows, *e.g.,* a two-flow coflow would end up serializing scheduling its two flows, one for the piloting purpose.

To avoid CCT degradations for thin coflows, we schedule all flows of a coflow if its width is under a threshold (set to 7 in Philae; §3.9.6 provides sensitivity analysis for thresholds).

Failure tolerance and recovery Cluster dynamics such as stragglers or node failure can delay some of the flows of a coflow or start new flows, increasing their CCT. The Philae design automatically self-adjusts to speed up coflows that are affected by cluster dynamics using the following mechanisms: (1) It adjusts the coflow size as the amount of data left by the coflow, which is essentially the difference between the size calculated using pilot flows and amount of data already sent. (2) It calculates contention only on the ports that have unfinished flows.

Work Conservation By default, Philae schedules non-pilot flows of a coflow only after all its pilot flows are over. This can lead to some ports being idle where the non-pilot flows are waiting for the pilot flows to finish. In such cases, Philae schedules non-pilot flows of coflows which are still in the sampling phase at those ports. In work conservation, the coflows are scheduled in the FIFO order of arrival of coflows.

3.6 Scalability Analysis

Compared to learning coflow sizes using priority queues (PQ-based) [11,40], learning coflow sizes by sampling Philae not only reduces the learning overhead as discussed in §3.4.1 and shown in §3.9.2, but also significantly reduces the amount of interactions between the coordinator and local agents and thus makes the coordinator highly scalable, as summarized in Table 3.1.

First, *PQ-based learning requires much more frequent update from local agents.* PQ-based learning estimates coflow sizes by incrementally moving coflows across priority queues according to the data sent by them so far. As such, the scheduler needs

Table 3.1.: Comparison of frequency of interactions between the coordinator and local agents.

	Update of data sent	Update of flow completion	Rate calculation
PHILAE	No	Yes	Event triggered
Aalo	Periodic (δ)	Yes	Periodic (δ)

frequent updates (every δ ms) of data sent per coflow from the local agents. In contrast, PHILAE directly estimates a coflow's size upon the completion of all its pilot flows. The only updates PHILAE needs from the local agents are about the flow completion which is needed for updating contentions and removing flows from active consideration..

Second, *PQ-based learning results in much more frequent rate allocation.* In sampling-based approach, since coflow sizes are estimated only once, coflows are re-ordered only upon coflow completion or arrival events or in the case of contention based policies only when contention changes, which is triggered by completion of all the flows of a coflow at a port. In contrast, in PQ-based learning, at every δ interval, coflow data sent are updated and coflow priority may get updated, which will trigger new rate assignment.

Our scalability experiments in §3.10.3 confirms that PHILAE achieves much higher scalability than Aalo.

3.7 Implementation

We implemented both PHILAE and Aalo scheduling policies in the same framework consisting of the global coordinator and local agents (Fig. 3.2), in 5.2 KLoC in C++.

Coordinator: The coordinator schedules the coflows based on the operations received from the registering framework. The key implementation challenge for the coordinator is that it needs to be fast in computing and updating the schedules. The PHILAE coordinator is optimized for speed using a variety of techniques including pipelining, process affinity, and concurrency whenever possible.

Local agents: The local agents update the global coordinator only upon completion of a flow, along with its length if it is a pilot flow. Local agents schedule the coflows based on the last schedule received from the coordinator. They comply to the last schedule until a new schedule is received. To intercept the packets from the flows, local agents require the compute framework to replace `datasend(), datarecv()` APIs with the corresponding Philae APIs, which incurs very small overhead.

Coflow operations: The global coordinator runs independently from, and is not coupled to, any compute framework, which makes it general enough to be used with any framework. It provides RESTful APIs to the frameworks for coflow operations: (a) `register()` for registering a new coflow when it enters, (b) `deregister()` for removing a coflow when it exits, and (c) `update()` for updating coflow status whenever there is a change in the coflow structure, *e.g.,* during task migration and restarts after node failures.

Table 3.2.: Performance improvement over Aalo for varying pilot flow selection schemes.

	Constant	Proportional to number of senders					Proportional to number of flows	
	2	5%	10%	20%	50%	100%	1%	10%
Avg. error	13.21%	6.14%	5.42%	4.94%	5.53%	4.25%	4.15%	2.90%
Avg. CCT	1.27x	1.51x	1.45x	1.50x	1.50x	1.50x	1.43x	0.49x
P50 speedup	1.75x	1.78x	1.76x	1.71x	1.52x	1.40x	1.33x	0.69x
P90 speedup	9.00x	9.58x	9.00x	9.15x	8.33x	8.45x	8.23x	8.23x

3.8 Evaluation Highlights

We evaluated Philae using a 150-node and a 900-node testbed cluster in Azure and using large scale simulations by utilizing a publicly available Hive/MapReduce trace collected from a 3000-machine, 150-rack Facebook production cluster [39] and multiple derived traces with varying degrees of flow size skew to measure Philae's robustness to skew.

- **Facebook (FB) trace**: The trace contains 150 ports and 526 ($> 7 \times 10^5$ flows) coflows, that are extracted from Hive/MapReduce jobs from a Facebook production cluster. Each coflow consists of pair-wise flows between a set of senders and a set of receivers.

Due to the lack of other publicly available coflow trace[1], we derived three additional traces using the original Facebook trace in order to more thoroughly evaluate PHILAE under varying coflow size skew:

- **Low-skew-filtered**: Starting with the FB trace, we filtered out coflows that have skew (*max flow length*/*min flow length*) less than a constant k. We generated five traces in this class with $k = 1, 2, 3, 4, 5$. The filtered traces have 142, 100, 65, 51 and 43 coflows, respectively.

- **Mantri-like**: Starting with the FB trace, we adjusted the sizes of the flows sent by the mappers, keeping the total reducer data the same as given in the original trace, to match the skew of a large Microsoft production cluster trace as described in Mantri [57]. In particular, the sizes are adjusted so that the coefficients of variation across mapper data are about 0.34 in the 50^{th} percentile case and 3.1 in the 90^{th} percentile case. This trace has the same numbers of coflows and ports as the FB trace.

- **Wide-coflows-only**: We filtered out all the coflows in the FB trace with the total number of flows ≤ 7, the default thin coflow bypass threshold (thinLimit) in PHILAE. The filtered trace has 269 coflows spreading over 150 ports.

The primary performance metrics used in the evaluation are CCT or CCT speedup, defined as the ratio of a CCT under other baseline algorithms and under PHILAE, piloting overhead, and coflow size estimation accuracy.

The highlights of our evaluation results are:

[1]A challenge that has also been faced by previous work on coflow scheduling such as [11, 51, 55, 62].

(1) PHILAE significantly improves the CCTs. In simulation using the FB trace, the average CCT is improved by 1.51× over the prior art, Aalo. Individual CCT speedups are 1.78× in the median case (P90 = 9.58×). For the Mantri-like trace, the average CCT is improved by 1.36× and individual CCT speedups are 1.75× in the median case (P90 = 12.0×).

(2) The CCT improvement mainly stems from the reduction in the learning overhead (in terms of latency and amount of data sent) in determining the right queue for the coflows. Compared to Aalo, median reduction in the absolute latency in finding the right queue for coflows in PHILAE is 19.0×, and in absolute amount of data sent is 20.0× (§3.9.2).

(3) PHILAE improvements are consistent when varying the skew among the flow sizes in a coflow (§3.9.5).

(4) PHILAE improvements are consistent when varying its parameters (§3.9.6).

(5) The PHILAE coordinator is much more scalable than that of Aalo (§3.10.3).

3.9 Simulation

We present detailed simulation results in this section, and the testbed evaluation of our prototype in §3.10.

Experimental setup: Our simulated cluster uses the same number of nodes (sending and receiving network ports) as in the trace. As in [11], we assume full bisection bandwidth is available, and congestion can happen only at network ports.

The *default parameters* for Aalo and PHILAE in the experiments are: starting queue threshold (Q_0^{hi}) is 10MB, exponential threshold growth factor (E) is 10, number of queues (K) is set to 10, the weights assigned to individual priority queues decrease exponentially by a factor of 10, and the new schedule calculation interval δ is set to 8ms for the 150-node cluster [2], the default suggested in its publicly available simulator [11]. In PHILAE, a new schedule is calculated on demand, upon arrival

[2] 8ms is the time to send 1MB of data.

of a new coflow, completion of a coflow, or completion of all pilot flows of a coflow. Finally, in Philae the threshold for thinLimit (T) is set to 7, the number of pilot flows assigned to wide coflows are $max(1, 0.05 \cdot S)$, where S is the number of senders, and the default inter-coflow scheduling policy in Philae is Least length-weighted total-port contention.

3.9.1 Pilot Flow Selection Policies

We start by evaluating the impact of different policies in choosing the pilot flows for a coflow in Philae. Table 3.2 summarizes the improvement in average CCT of Philae over Aalo and average error in size estimation normalized to the actual coflow size, when varying the pilot flow selection policy while keeping other parameters as the default in Philae, using the FB trace.

Unsurprisingly, the estimation accuracy increases when increasing the number of pilot flows across the three selection schemes: constant, fraction of senders, and fraction of total flows. However, as the number of pilot flows increases (over the range of parameter choices), the CCT speedup (P50 and P90 of individual coflow CCT speedups) decreases. This is because the benefit from size estimation accuracy improvement from using additional pilot flows does not offset the added overhead from completing the additional pilot flows and the delay they incur to other coflows.

We find sampling 5% of the number of senders per coflow strikes a good trade-off between piloting overhead and size estimation accuracy leading to the best CCT reduction. We thus set it $(0.05 \cdot S)$ as the default pilot flow selection policy.

3.9.2 Piloting Overhead and Accuracy

Next, using the default pilot selection policy, we evaluate Philae's effectiveness in estimating coflow sizes by sampling pilot flows. Fig. 4.3 shows a scatter plot of the actual coflow size vs. estimated size from running Philae under the default settings. We observe that Philae coflow's size estimation is highly accurate except

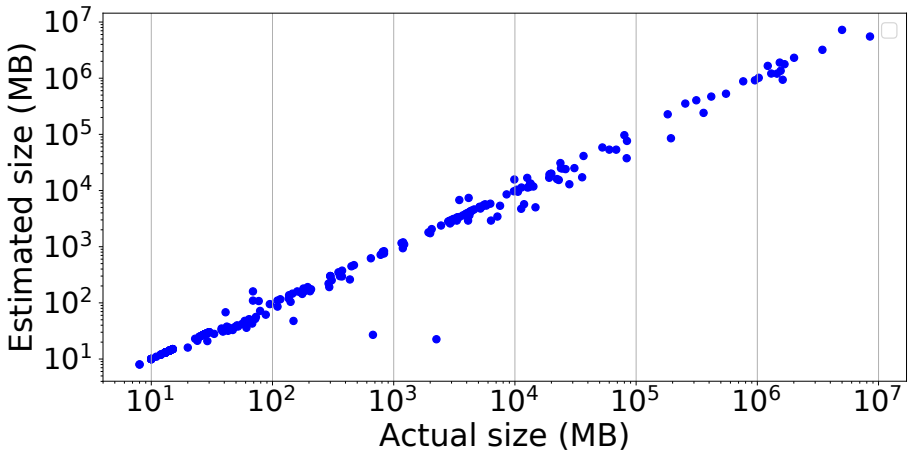


Figure 3.3.: PHILAE coflow size learning accuracy. Coflows that did not go through the piloting phase (48%) are not shown.

for a few outliers. Overall, the average and standard deviation of relative estimation error are 0.06 and 0.15, respectively, and for the top 99% and 95% coflows (in terms of estimation accuracy), the average (standard deviation) of relative error are only 0.05 (0.12) and 0.03 (0.07) respectively. Interestingly, a few coflows experience large estimation errors, and we found they all have very high skew in their flow lengths; the mean standard deviation in flow lengths, normalized by the average length, of the bottom 1% (in terms of accuracy) ranges between 4.6 and 6.8.

Fig. 3.1 shows the cost of estimating the correct queue for each coflow in PHILAE and Aalo, measured as the time in learning the coflow size as a fraction of the coflow's CCT in PHILAE and Aalo. We see that under PHILAE, about 63% of the coflows spent less than 1% of their CCT in the learning phase, while under Aalo, 63% coflows reached the correct priority queue after spending up to 22% of their CCT moving across other queues. Compared to Aalo, PHILAE in the median case sends 20× less data in determining the right queue and reduces the latency in determining the right queue by 19×.

3.9.3 Inter-Coflow Scheduling Policies

Philae differs from Aalo in two ways: online size estimation and inter-flow scheduling policy. Here, we evaluate the effectiveness of the four inter-coflow scheduling policies of Philae discussed in §3.5.4, keeping the remaining parameters as the default. Such evaluation allows us to decouple the contribution of sampling-based learning from the effect of scheduling policy difference.

Table 3.3 shows the CCT Improvement of Philae under the four inter-flow scheduling policies over Aalo. We make the following observations.

First, Philae with the purely sized-based policy, **Smallest job first (A)**, which uses the same inter-queue and intra-queue scheduling policy as Aalo and only differs from Aalo in coflow size estimation, reduces the average CCT (P50) of Aalo by 1.40x (1.48x).

In contrast, the default Philae uses **Least length-weighted total-port contention (D)**, which uses the sum of size-weighted port contention to assign coflows to priority queues, and slightly outperforms the size-based policy A; it reduces the average CCT (P50) of Aalo by 1.51x (1.78x). This is because it captures the diversity of contention at different ports, which happens often in real distributed settings, and at the same time accounts for the coflow size by using length-weighted sum of the port-wise contention. The above results for policy A and policy D indicate that the primary improvement in Philae comes from its sampling-based coflow size estimation scheme.

Shortest remaining time first (B) performs similarly as smallest job first. This is because the preemptive nature of SRTF will kick in only on arrival of new coflows. Also, although SRTF is advantageous for small coflows, since Philae already schedules thin coflows at high priority, many thin and thus small coflows are anyways being scheduled at high priority under both policies A and B, and as a result they perform similarly.

Table 3.3.: CCT speedup in PHILAE under different inter-coflow scheduling policies (§3.5.4) over Aalo.

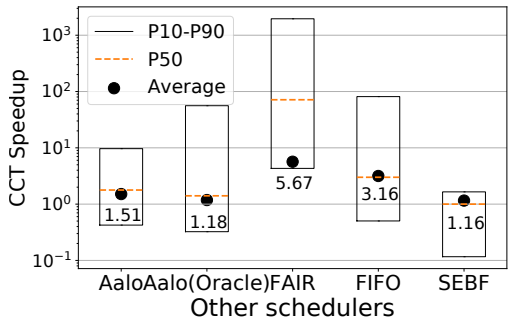
Priority estimation metric	P50	P90	Avg. CCT
Estimated size (A)	1.48x	8.27x	1.40x
Remaining size (B)	1.54x	8.34x	1.37x
Global Contention (C)	0.75x	8.26x	0.13x
Length-weighted total-port contention (D) (PHILAE)	1.78x	9.58x	1.51x

Finally, **Least contention first (C)** performs poorly. This is because contention for a coflow is defined as the unique number of other coflows that share ports, and as a result such a policy completely ignores the size (length) of the coflows.

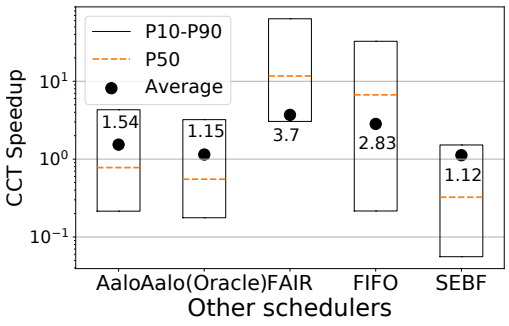
3.9.4 Average CCT improvement

We now compare the CCT speedups of PHILAE against 5 well-known coflow scheduling policies: (1) Aalo, (2) Aalo-Oracle, which is an oracle version of Aalo where the scheduler knows the final queue of a coflow upon its arrival time and directly starts the coflow from that queue, (3) SEBF in Varys [34] which assumes the knowledge of coflow sizes apriori and uses the Shortest Effective Bottleneck First policy, where the coflow whose slowest flow will finish first is scheduled first. (4) FIFO, which is a single queue FIFO based coflow scheduler, and (5) FAIR, which uses per-flow fair sharing. We do not include Saath [40] in the comparison as it does not provide the same liveliness guarantees as PHILAE which as discussed in §3.5.3 can obscure the comparison result. All experiments use the default parameters discussed in the setup, including K, E, S, unless otherwise stated. The results are shown in Fig. 3.4(a). We make the following observations.

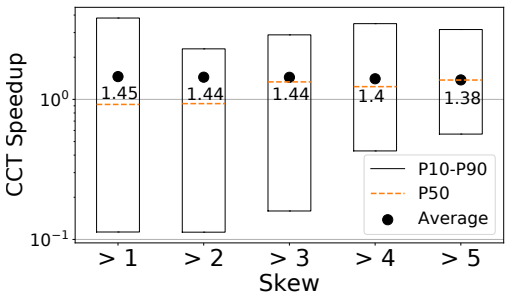
First, we compare CCT under PHILAE against under Aalo-Oracle, where Aalo-Oracle starts all coflows at the correct priority queues (*i.e.,* no learning overhead). PHILAE improves the average CCT by 1.18× and P50 CCT by 1.40×, respectively.



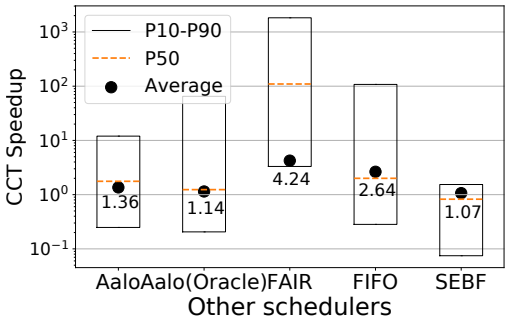
(a) Using original FB trace.



(b) Using Wide-coflows-only trace.



(c) Using 5 Low-skew-filtered traces.



(d) Using the Mantri-like trace.

Figure 3.4.: CCT speedup using PHILAE compared to using other coflow schedulers on different traces. In Fig. 3.4(c), the x-axis denotes the minimum skew in the 5 Low-skew-filtered traces.

Since Aalo-Oracle pays no overhead for coflow size estimation, its worse performance suggests that using length-weighted total-port contention in assigning coflows to the priority queues in PHILAE outperforms Aalo's size-based, contention-oblivious policy in assigning coflows to the queues.

Second, PHILAE improves the average CCT over Aalo by 1.51× (median) and P50 by 1.78. The significant *additional* improvement on top of the gain over Aalo-Oracle comes from fast and accurate estimation of the right queues for the coflows (Fig. 3.1).

Third, PHILAE, which requires no coflow size knowledge a priori, achieves comparable performance as SEBF [34]; it reduces the average CCT by 1.16×. Again this is because its total-port contention policy outperforms the contention-oblivious SEBF.

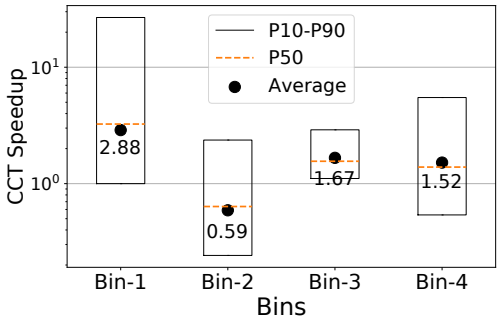


Figure 3.5.: Performance breakdown into bins shown in Table 3.4.

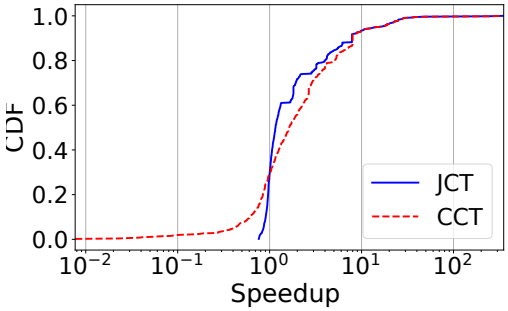


Figure 3.6.: [Testbed] Distribution of speedup in CCT and JCT in PHILAE using the FB trace.

Finally, PHILAE significantly outperforms the single-queue FIFO-based coflow scheduler, with a median (P90) CCT speedup of 3.00 (77.96)× and average CCT speedup of 3.16×, and the un-coordinated flow-level fair-share scheduler, with a median (P90) CCT speedup of 70.82× (1947×) and average CCT speedup of 5.66×.

To gain insight into how different coflows are affected by PHILAE over Aalo, we group the coflows in the trace into four bins defined in Table 3.4, and show in Fig. 3.5 the CCT speedups for each bin. We see that PHILAE improves CCT for all coflows in bin 1 and 3 and for large fraction in bin-4. Most of the underperforming coflows fall in bin-2. Coflows in bin-2 have width > 7 and size < 100MB, *i.e.,* the flows are short but wide. Because the width exceeds the thinLimit, PHILAE schedules the pilot flows to estimate the coflow size first (§3.5). Thus, although the remaining flows are short, they get delayed until the completion of the pilot flows, which results in CCT increase.

Finally, since thin coflows benefit from PHILAE's scheme of bypassing probing for thin coflows, we also compare PHILAE with other schemes using the Wide-coflows-only trace which consists of all coflows wider than the default thinLimit (7) in PHILAE. Fig. 3.4(b) shows that PHILAE continues to perform well, reducing the average CCT by 1.54×, 1.15×, and 1.12× over Aalo, Aalo-Oracle, and SEBF, respectively.

Table 3.4.: Bins based on total coflow size and width (number of flows). The numbers in brackets denote the fraction of coflows in that bin.

	width $\leq$ 7(thin)	width $>$ 7(wide)
size $\leq$ 100MB (small)	bin-1 (44.3%)	bin-2 (24.1%)
size $>$ 100MB (large)	bin-3 (4.5%)	bin-4 (27.1%)

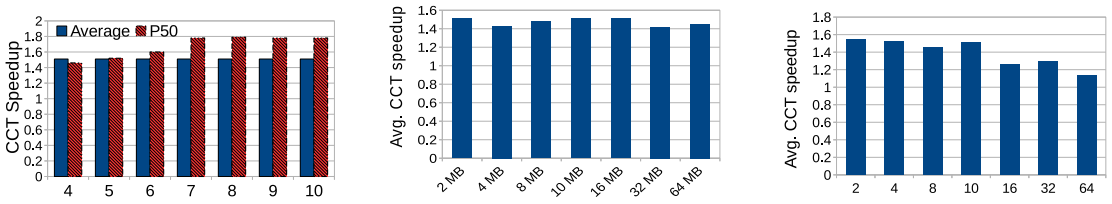
3.9.5 Robustness to Coflow Data Skew

Next, we evaluate Philae's robustness to flow size skew by comparing it against Aalo using traces with varying degrees of skew. First, we evaluate Philae using the Mantri-like trace. Fig. 3.4(d) shows that Philae consistently outperforms prior-art coflow schedulers. In particular, Philae reduces the average CCT by 1.36x compared to Aalo. Second, we evaluate Philae using the Low-skew-filtered traces which have low skew coflows filtered out. Fig. 3.4(c) shows that Philae performs better than Aalo even with highly skewed traces and reduces the average CCT by 1.45×, 1.44×, 1.44×, 1.40× and 1.38× for the five Low-skew-filtered traces containing coflows with skew of at least 1, 2, 3, 4 and 5, respectively.

3.9.6 Sensitivity Analysis

Compared to Aalo, Philae has only two additional paramaters: thinLimit and flow sampling rate. We already discussed the choice of sampling rate in §3.9.1. Below, we evaluate the sensitivity of Philae to thinLimit and other design parameters common to Aalo by varying one parameter at a time while keeping the rest as the default.

Thin coflow bypassing limit (T) In this experiment, we vary thinLimit (T) in Philae for bypassing coflows from the probing phase. The result in Fig. 3.7(a) shows that the average CCT remains almost the same as T increases. This is because the average CCT is dominated by wide and large coflows, which are not affected by thinLimit. However, the P50 speedup increases till $T = 7$ and tapers off after $T = 7$. The reason for the CCT improvement until $T = 7$ is that all flows of thin coflows



(a) Thin coflow bypass threshold (b) First queue capacity (Q_0^{hi}) (c) Exponent (E)

Figure 3.7.: [Simulation] PHILAE sensitivity analysis. We vary one parameter of PHILAE keeping rest same as default and compare it with Aalo.

(with width ≤ 7) are scheduled immediately upon arrival which improves their CCT, and the number of thin coflows is significant.

Start queue threshold (Q_0^{hi}) We next vary the threshold for the first priority queue from 2 MB to 64 MB. Fig. 3.7(b) shows the average CCT of PHILAE over Aalo. Overall, PHILAE is not very sensitive to the threshold of first priority queue and the CCT speedup over Aalo is within 8% of the default PHILAE (10 MB). The speedup appears to oscillate with a periodicity of 5x to 10x. For example, the speedups for 2 MB and 64 MB are close to that of the default (10 MB), while for 4 MB and 32 MB are lower. This can be explained by the impact of the first queue threshold on job segregation; with the default queue threshold growth factor of 10, every time the first queue threshold changes by close to 10x, the distribution of jobs across the queues become similar.

Multiplication factor (E) In this experiment, we vary the queue threshold growth factor from 2 to 64. Recall that the queue thresholds are computed as $Q_q^{hi} = Q_{q-1}^{hi} \cdot E$. Thus, as E grows, the number of queues decreases. As shown in Fig. 3.7(c), smaller queue threshold multiplication factor which leads to more queues performs better because of fine-grained priority segregation.

Table 3.5.: [Testbed] CCT improvement in PHILAE as compared to Aalo.

	P50	P90	Avg. CCT
FB Trace	1.63×	8.00×	1.50×
Wide-coflow-only	1.05×	2.14×	1.49×

3.10 Testbed Evaluation

Next, we deployed PHILAE in a 150-machine Azure cluster and a 900-machine cluster to evaluate its performance and scalability.

Testbed setup: We rerun the FB trace on a Spark-like framework on a 150-node cluster in Microsoft Azure [48]. The coordinator runs on a Standard DS15 v2 server with 20-core 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor and 140GB memory. The local agents run on D2v2 with the same processor as the coordinator with 2-core and 7GB memory. The machines on which local agents run have 1 Gbps network bandwidth. Similarly as in simulations, our testbed evaluation keeps the same flow lengths and flow ports in trace replay. All the experiments use default parameters K, E, S and the default pilot flow selection policy.

3.10.1 CCT Improvement

In this experiment, we measure CCT improvements of PHILAE compared to Aalo. Fig. 3.6 shows the CDF of the CCT speedup of individual coflows under PHILAE compared to under Aalo. The average CCT improvement is 1.50× which is similar to the results in the simulation experiments. We also observe 1.63× P50 speedup and 8.00× P90 speedup.

We also evaluated PHILAE using the Wide-coflow-only trace. Table 3.5 shows that PHILAE achieves 1.52× improvement in average CCT over Aalo, similar to that using the full FB trace. This is because the improvement in average CCT is dominated by large coflows, PHILAE is speeding up large coflows, and the Wide-coflow-only trace consists of mostly large coflows.

Table 3.6.: [Testbed] Average (standard deviation) coordinator CPU time (ms) per scheduling interval in 900-port runs. PHILAE did not have to calculate and send new rates in 66% of intervals, which contributes to its low average.

	Rate Calc.	New Rate Send	Update Recv.	Total
PHILAE	2.99 (5.35)	4.90 (11.25)	6.89 (17.78)	14.80 (28.84)
Aalo	4.28 (4.14)	17.65 (20.9)	10.97 (19.98)	32.90 (34.09)

Table 3.7.: [Testbed] Percentage of scheduling intervals where synchronization and rate calculation took more than δ for 150-port and $\delta'(= 6 \times \delta)$ for 900-port runs.

	150 ports	900 ports
PHILAE	1%	10%
Aalo	16%	37%

3.10.2 Job Completion Time

Next, we evaluate how the improvement in CCT affects the job completion time (JCT). In data clusters, different jobs spend different fractions of their total job time in data shuffle. In this experiment, we used 526 jobs, each corresponding to one coflow in the FB trace. The fraction of time that the jobs spent in the shuffle phase follows the same distribution used in Aalo [11], *i.e.,* 61% jobs spent less than 25% of their total time in shuffle, 13% jobs spent 25-49%, another 14% jobs spent 50-74%, and the remaining spent over 75% of their total time in shuffle. Fig. 3.6 shows the CDF of individual speedups in JCT. Across all jobs, PHILAE reduces the job completion time by 1.16× in the median case and 7.87× in the 90^{th} percentile. This shows that improved CCT translates into better job completion time. As expected, the improvement in job completion time is smaller than the improvement in CCT because job completion time depends on the time spent in both compute and shuffle (communication) stages, and PHILAE improves only the communication stage.

3.10.3 Scalability

Finally, we evaluate the scalability of PHILAE by comparing its performance with Aalo on a 900-node cluster. To drive the evaluation, we derive a 900-port trace by replicating the FB trace 6 times across ports, *i.e.,* we replicated each job 6 times, keeping the arrival time for each copy the same but assigning sending and receiving ports in increments of 150 (the cluster size for the original trace). We also increased the scheduling interval δ by 6 times to $\delta' = 6 \times \delta$.

PHILAE achieved 2.72× (9.78×) speedup in average (P90) CCT over Aalo. The higher speedup compared to the 150-node runs (1.50×) comes from higher scalability of PHILAE. In 900-node runs, Aalo was not able to finish receiving updates, calculating new rates and updating local agents of new rates within δ' in 37% of the intervals, whereas PHILAE only missed the deadline in 10% of the intervals. For 150-node runs these values are 16% for Aalo and 1% for PHILAE. The 21% increase in missed scheduling intervals in 900-node runs in Aalo resulted in local agents executing more frequently with outdated rates. As a result, PHILAE achieved even higher speedup in 900-node runs.

As discussed in§3.6, Aalo's poorer coordinator scalability comes from more frequent updates from local agents and more frequent rate allocation, which result in longer coordinator CPU time in each scheduling interval. Table 3.6 shows the average coordinator CPU usage per interval and its breakdown. We see that (1) on average PHILAE spends much less time than Aalo in receiving updates from local agents, because PHILAE does not need updates from local agents at every interval – on average in every scheduling interval PHILAE receives updates from 49 local agents whereas Aalo receives from 429 local agents, and (2) on average PHILAE spends much less time calculating new rates and send new rates. This is because rate calculation in PHILAE is triggered by events and PHILAE did not have to flush rates in 66% of the intervals.

3.11 Summary

State-of-the-art online coflow schedulers approximate the classic SJF by implicitly learning coflow sizes and pay a high penalty for large coflows. In this chapter we proposed the novel idea of sampling in the spatial dimension of coflows to explicitly and efficiently learn coflow sizes online to enable efficient online SJF scheduling. Our extensive simulation and testbed experiments have shown the new design offers significant performance improvement over prior art. Further, the sampling-in-spatial-dimension technique can be generalized to other distributed scheduling problems such as cluster job scheduling.

4 SLEARN

The success of PHILAE (Chapter 3) in effectively learning coflow sizes online motivated us to explore the broader applicability of sampling-based learning. We designed, SLEARN for predicting runtime properties like task durations of big data jobs online by online sampling. This chapter discusses SLEARN in detail.

4.1 Introduction

In big-data compute clusters, jobs arrive online and compete to share the cluster resources. In order to best utilize the cluster and to ensure that jobs also meet their service level objectives, efficient scheduling is essential. However, as jobs arrive online, their runtime characteristics are not known a priori. Due to this lack of information, it is challenging for the cluster scheduler to determine the right job execution order that optimizes scheduling metrics such as maximal resource utilization or application service level objectives.

An effective way to tackle the challenges of cluster scheduling is to learn the runtime characteristics of pending jobs, as accurately estimating job runtime characteristics allows the scheduler to exploit offline scheduling algorithms that are known to be optimal, *e.g.,* Shortest Job First (SJF) for minimizing the average completion time. Indeed, there have been a large amount of work [12–16, 29–31] on learning job runtime characteristics to facilitate cluster job scheduling.

In essence, all of the previous learning algorithms learn job runtime characteristics from observing historical executions of the same jobs, which execute the same code but process different sets of data, or of similar jobs, which have matching features such as the same application name, the same job name, or the same user who submitted the job.

The effectiveness of the above *history-based* learning schemes critically rely on two conditions to hold true: (1) The jobs are recurring; (2) The performance of the same or similar jobs will remain consistent over time.

In practice, however, the two conditions often do not hold true. First, many previous work have acknowledged that not all jobs are recurrent. For example, in the traces used in Corral [14] and Jockey [10], only 40% of the jobs are recurrent, and Morpheus [12] shows that only 60% of the jobs are recurrent. Second, even the authors of history-based prediction schemes such as 3Sigma [30] and Morpheus [12] strongly argued why runtime properties of jobs, even with the same input, will not remain consistent and will keep evolving. The primary reason is due to updates in cluster hardware, application software, and user scripts to execute the cluster jobs. Third, our own analysis of three production cluster traces (§4.4) have also shown that historical job runtime characteristics have considerable variations.

In this chapter, we explore an alternative approach to learning runtime properties of distributed jobs online to facilitate cluster job scheduling. The approach is motivated by the following key observations about distributed jobs running on shared clusters: (1) a job typically has a *spatial dimension*, *i.e.,* it typically consists of many tasks; and (2) the tasks (in the same phase) of a job typically execute the same code and process different chunks of similarly sized data [63, 64]. These observations suggest that if the scheduler first schedules a few sampled tasks of a job, known as pilot tasks, to run till finish, it can use the observed runtime properties of those tasks to accurately estimate those of the whole job. Effectively, such a *task-sampling-based* approach learns job properties in the spatial dimension. We denote the new learning scheme as SLearn, for "learning in space".

Intuitively, by using the execution of pilot tasks to predict the properties of other tasks, SLearn avoids the primary drawback of history-based learning techniques, *i.e.,* relying on jobs to be recurring and job properties to remain stationary over time. However, learning in space introduces two new challenges: (1) its estimation accuracy can be affected by the variations of task runtime properties, *i.e.,* task skew;

(2) delaying scheduling the remaining tasks of a job till the completion of sampled tasks may potentially hurt the job's completion time.

In this chapter, we perform a comprehensive comparative study of history-based learning (learning in time) and sampling-based learning (learning in space), to systematically answer the following questions:

1. Can learning in space be more accurate than learning in time?

2. If so, can delaying scheduling the remaining tasks of a job till the completion of sampled tasks be more than compensated by the improved accuracy and result in improved job performance, *e.g.,* completion time?

We answer the first question via quantitative analysis, and trace and experimental analysis based on three production job traces, including two public cluster traces from Google released in 2011 and 2019 [65, 66] and a private trace from 2Sigma [38]. We answer the second question by designing a generic scheduler that schedules jobs based on job runtime estimates to optimize a given performance metric, *e.g.,* average job completion time (JCT), and then plug into the scheduler different prediction schemes, in particular, learning in time and learning in space, to compare their effectiveness using three production job traces.

We summarize the major findings and contributions presented in this chapter as follows:

- Based on literature survey and analysis using three production cluster traces, we show that history is not a stable and accurate predictor for runtime characteristics of distributed jobs.

- We propose SLearn, a novel learning approach that uses sampling in the spatial dimension of jobs to learn job runtime properties online.

- Via quantitative, trace and experimental analysis, we demonstrate that SLearn can predict job runtime properties with much higher accuracy than history-based schemes. For the 2Sigma, Google 2011, and Google 2019 cluster traces,

the median prediction error are 18.98%, 13.68%, and 51.84% for SLearn but 36.57%, 21.39%, and 71.56% for the state-of-the-art history-based 3Sigma, respectively.

- We show that learning job runtime properties by sampling job tasks, although delays scheduling the remaining tasks of a job, can be more than compensated by the improved accuracy, and as a result reduces the average JCT. In particular, our extensive simulations and testbed experiments using a prototype on a 150-node cluster in Microsoft Azure show that compared to the prior-art history-based predictor, SLearn reduces the average JCT by 1.28×, 1.56×, and 1.32× for the extracted 2Sigma, Google 2011 and Google 2019 traces, respectively.

Despite these encouraging findings about learning in space, we envision several motivations for exploring combining history- and sampling-based learning. (1) For workloads with mixed recurring and first-time jobs, sampling-based learning can be applied to first-time jobs while history-based learning can be applied to recurring jobs. (2) History-based learning can be used to establish a prior distribution, and sampling-based approach can be used to refine the posterior distribution. Such a combination may potentially be more accurate than using either history or sampling alone. (3) Though not seen in the production traces used in our study, in case task-wise variation and job-wise variation fluctuate, adaptively switching between the two prediction schemes may also help. We plan to pursue these directions in our future work.

4.2 Background and Related Work

In this section, we provide a brief background on the cluster scheduling problem, review existing learning-based schedulers, and discuss their weaknesses.

4.2.1 Cluster Scheduling Problem

In both public and private clouds, clusters are typically shared among multiple users to execute diverse jobs. Such jobs typically arrive online and compete for shared resources. In order to best utilize the cluster and to ensure that jobs also meet their service level objectives (SLOs), efficient job scheduling is essential. Since jobs arrive online, their runtime characteristics are not known a priori. This lack of information makes it challenging for the scheduler to determine the right order for running the jobs that maximizes resource utilization and/or meets application SLOs. Additionally, jobs have different SLOs. For some meeting deadlines is important while for others faster completion or minimizing the use of networks is more important. Such a diverse set of objectives pose further challenges to effective job scheduling [10, 12, 14, 26–29].

4.2.2 Job Model

We consider big-data compute clusters running data-parallel frameworks such as Hadoop [2], Hive [6], Dryad [58], Scope [67], and Spark [22] that run simple MapReduce jobs [1] or more complex DAG-structured jobs, where each job processes a large amount of data. Each job consists of one or multiple stages, such as map or reduce, and each stage partitions the data into manageable chunks and runs many parallel tasks, each for processing one data chunk.

4.2.3 Existing Learning-based Schedulers

An effective way to tackle the challenges of cluster scheduling is to learn runtime characteristics of pending jobs. As such cluster schedulers using various learning methods have been proposed [12–15,27,30,31,68–71]. In essence, all previous learning schemes are *history-based*, *i.e.,* they learn job characteristics by observations made from the past job executions. In particular, existing learning approaches can be broadly categorized into the following groups, as summarized in Table 4.1.

Table 4.1.: Summary of selected previous work that use history-based learning techniques.

Name	Property estimated	Estimation technique	Learning frequency
Corral [14]	Job runtime	Offline model (not updated)	On arrival
DCOSR [13]	Memory elasticity profile	Offline model (not updated)	Scheduler dependent
Jockey [10]	Job runtime	Offline simulator	Periodic
3Sigma [30]	Job runtime history dist.	Offline model	On arrival

Learning offline models. Corral's prediction model is designed with the primary assumption that most jobs are recurring in nature, and additional assumptions such as the latency of each stage of a multi-stage job is proportional to the amount of data processed by it, which do not always hold true [14].

DCOSR [13] predicts the memory usage for data parallel compute jobs using an offline model built from a fixed number of profile runs that are specific to the framework and depend on the framework's properties. Any software update in the existing frameworks, addition of new framework or hardware update will require an update in profile.

For analytics jobs that perform the same computation periodically on different sets of data, Tetris [17] takes measurements from past executions of a job to estimate the requirements for the current execution.

Learning offline models with periodic updates. Jockey [10] periodically characterizes job progress at runtime, which along with a job's current resource allocation is used by an offline simulator to estimate the job's completion time and update the job's resource allocation. The running time estimates made by the simulator are based on performance statistics extracted from one or more previous runs of the recurring job. Jockey relies on job recurrences and cannot work with new jobs.

Learning from similar jobs. Instead of using execution history from the exact same jobs, JVuPredict [72] matches jobs on the basis of some common features such as application name, job name, the user who owns the job, and the resource requested by the job. Additionally, it uses multiple metrics, such as rolling average and median, in estimating the running time of a new job from such similar jobs.

3Sigma [30] extends JVuPredict [72] by introducing a new idea on prediction: instead of using point metrics to predict runtimes, it uses full distributions of relevant runtime histories. However, since it is impractical to maintain precise distributions for each feature value, it resorts to approximating distributions, which compromises the benefits of having full distributions.

4.2.4 Learning from History: Assumptions and Reality

Predicting job runtime characteristics from history information relies on the following two conditions to hold, which we argue may not be applicable to modern day clusters.

Condition 1: The jobs are recurring. Many previous work have acknowledged that not all jobs are recurrent. For example, the traces used in Corral [14] and Jockey [10] show that only 40% of the jobs are recurrent and Morpheus [12] shows that 60% of the jobs are recurrent.

Condition 2: The performance of the same or similar jobs will remain consistent over time. Previous works [10, 12, 14, 30] that exploited history-based prediction have considered jobs in one of the following two categories. (1) *Recurring jobs*: A job is re-scheduled to run on newly arriving data; (2) *Similar jobs:* A job has not been seen before but has some attributes in common with some jobs executed in the past [30, 72]. Many of the history-based approaches only predict for recurring jobs [10, 12, 14], while some others [30, 70–72] work for both categories of jobs.

However, even the authors of history-based prediction schemes such as 3Sigma [30] and Morpheus [12] strongly argued why runtime properties of jobs, even with the same input, will not remain consistent and will keep evolving. The primary reason is that updates in cluster hardware, application software, and user scripts to execute the cluster jobs affect the job runtime characteristics. They found that in a large Microsoft production cluster, within a one-month period, applications corresponding to more than 50% of the recurring jobs were updated. The source code changed by at least 10% for applications corresponding to 15-20% of the jobs. Additionally, over a one-year period, the proportion of two different types of machines in the cluster changed from 80/20 to 55/45. For a same production Spark job, there is a 40% difference between the running time observed on the two types of machines [12].

For these reasons, although the state-of-the-art history-based system 3Sigma [30] uses sophisticated prediction techniques, the predicted running time for more than 23% of the jobs have at least 100% error, and for many the prediction is off by an order of magnitude. In our analysis of three production cluster traces (see Figure 4.1), we observed similar levels of high variability in the runtime characteristics of the jobs with the same attributes.

4.3 SLearn – Learning in Space

In this chapter, we explore an alternative approach to learning job runtime properties online in order to facilitate cluster job scheduling. The approach is motivated by the following key observations about distributed jobs running in shared clusters: (1) a distributed job has a spatial dimension, *i.e.,* it typically consists of many tasks; (2) all the tasks in the same phase of a job typically execute the same code with the same settings [63, 64, 73], and differ in that they process different chunks of similarly sized data. Hence, it is likely that their runtime behavior will be statistically similar.

The above observations suggest that if the scheduler first schedules a few sampled tasks of a job to run till finish, it can use the observed runtime properties of those tasks

Table 4.2.: Comparison of learning in time and learning in space of job runtime properties.

	Applicability	Adapti-veness	Accuracy	Runtime overhead
Time	Recurring jobs	No/Yes	Depends	No
Space	New/Recurring jobs	Yes	Depends	Yes

to accurately estimate those of the whole job. In a modular design, such an online learning scheme can be decoupled from the cluster scheduler. In particular, upon a job arrival, the predictor first schedules sampled tasks of the job, called *pilot tasks*, till their completion, to learn the job runtime properties. The learned job properties are then fed into the cluster job scheduler, which can employ different scheduling polices to meet respective SLOs. Effectively, the new scheme learns job properties in the spatial dimension, *i.e.,learning in space*. We denote the new learning scheme as SLearn.

Intuitively, by using the execution of pilot tasks to predict the properties of other tasks, SLearn avoids the primary drawback of history-based learning techniques, *i.e.,* relying on job properties to remain stationary over time.

Learning in Time vs. Learning in Space Table 4.2 summarizes the pros and cons of the two learning approaches along four dimensions:

- **Applicability:** As discussed in §4.2.3, most history-based predictors cannot be used for the jobs of a new category or for categories for which the jobs are rarely executed. In contrast, learning in space has no such limitation; it can be applied to any new job.

- **Adaptiveness to change:** Further, history-based predictors assume job runtime properties persist over time, which often does not hold, as discussed in §4.2.4.

- **Accuracy:** The accuracy of the two approaches are directly affected by how they learn, *i.e.,* in space versus in time. The accuracy of history-based ap-

proaches is affected by how stable the job runtime properties persist over time, while that of sampling-based approach is affected by the variation of the task runtime properties, *i.e.,* the extent of task skew.

- **Runtime overhead:** The history-based approach has an inherent advantage of having very low to zero runtime overhead. It performs offline analysis of historical data to generate a prediction model. Afterwards there is almost no overhead in estimating runtime characteristics of newly arriving jobs. Variations of history-based predictors that use runtime feedback to update the prediction models may have some cost, but usually such systems are optimized to have low runtime update overhead. In contrast, sampling-based predictors do not have offline cost, but need to first run a few pilot tasks till completion before scheduling the remaining tasks. This may potentially delay the execution of non-sampled tasks.

The above qualitative comparison of the two learning approaches raises the following two questions: (1) Can learning in space be more accurate than learning in time? (2) If so, can delaying scheduling the remaining tasks of a job till the completion of sampled tasks be more than compensated by the improved accuracy, so that the overall job performance, *e.g.,* completion time, is improved? We answer the first question via quantitative, trace and experimental analysis in §4.4 and the second question via a case study of cluster job scheduling using the two types of predictors in §4.5.

4.4 Accuracy Analysis

In this section, we perform an in-depth study of the prediction accuracy of the two learning approaches: *learning in time* (history-based learning) and *learning in space* (task-sampling-based learning). Both approaches can potentially be used to learn different job properties for different optimization objectives. In this chapter, we focus

on job completion time because it is an important metric that has been intensively studied in recent work [11, 13, 14, 30, 33, 34, 74, 75].

We first derive analytical bounds on their prediction errors (§4.4.1). We then measure and compare the bounds in real traces from two production datacenters (§4.4.2). Finally, we experimentally compare the prediction accuracy of learning in space with a history-based predictor 3Sigma [30] in estimating the job runtimes (§4.4.3). We pick 3Sigma because it is a state-of-the-art history-based predictor that can learn for non-recurrent jobs.

4.4.1 Quantitative Comparison

We first present a theoretical analysis to compare the two approaches. We caution that here we use a highly-stylized model (e.g., two jobs and normal task-length distributions), which does not capture the possible complexity in real clusters, such as heavy parallelism across servers and highly-skewed task-length distributions. Nonetheless, it reveals important insights that help us understand in which regimes history-based schemes or sampling-based schemes will perform better. Consider a simple case of two jobs j_1 and j_2, where each job has n tasks. The size of each task of j_1 is known. Without loss of generality, let us assume that the task size of j_1 is 1. Thus, the total size of j_1 is n. The size of a task of j_2 is however unknown. Let x denote the average task size of j_2, and this its total size is nx. Clearly, if we knew x precisely, then we should have scheduled j_1 first if $x > 1$ and j_2 first if $x \leq 1$. However, suppose that we only know the following: (1) (Prior distribution:) x follows a normal distribution with mean μ and variance σ_o^2; (2) Given x, the size of a random task of the job follows a normal distribution with mean x and variance σ_1^2. Intuitively, σ_o^2 captures the variation of mean task-lengths *across* many *i.i.d.* copies of job j_2, *i.e.,* job-wise variation, while σ_1^2 captures the variation of task-lengths *within* a single run of job j_2, *i.e.,* task-wise variation.

We note that the parameters σ_o^2 and σ_1^2 are used below to understand the accuracy of both history-based and sampling-based predictors, whose goal is to estimate the

mean task-length x (and consequently the job runtime which equals x times the number of tasks) of a new copy of job j_2. The predictors themselves may not use the knowledge of σ_o^2 and σ_1^2. In practice, these parameters can be estimated offline from historical data. We will see soon that the performance of history-based schemes will mainly depend on σ_o^2, while the performance of sampling-based schemes will mainly depend on σ_1^2.

Towards this end, let us consider two options: (1) A history-based approach (§4.4.1.1) and (2) a sampling-based approach where we sample m tasks from j_2 (§4.4.1.2).

4.4.1.1 History-based Schemes

Since no samples of job j_2 are used, the best predictor for its mean task length is μ. In other words, the scheduling decision will be based on μ only. The difference between the true mean task length, x, and μ is simply captured by the job-wise variance σ_o^2.

4.4.1.2 Sampling-based Schemes

Suppose that we sample m tasks from j_2. Collect the sampled task lengths into a vector:

$$\vec{y} = (y_1, y_2, ..., y_m).$$

Then, based on our probabilistic model, we have

$$P\left(y_i|x\right) = \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{(y_i-x)^2}{2\sigma_1^2}}$$
$$P\left(\vec{y}|x\right) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{(y_i-x)^2}{2\sigma_1^2}}$$

We are interested in an estimator of x given $\vec{y}$. We have

$$P\left(x|\vec{y}\right) = \frac{P(\vec{y}|x)\cdot P(x)}{P(\vec{y})} = \frac{P(\vec{y}|x)\cdot P(x)}{\int_x P(\vec{y}|x)\cdot P(x)dx}$$

Table 4.3.: Summary of trace properties.

Trace	Arrival time	Resource requested	Resource usage	Indiv. task duration
2Sigma	Yes	Yes	No	Yes
Google 2011	Yes	Yes	Yes	Yes
Google 2019	Yes	Yes	Yes	Yes

$$= \frac{1}{\sqrt{2\pi}} \left[\frac{m}{\sigma_1^2} + \frac{1}{\sigma_o^2}\right]^{\frac{1}{2}} \cdot e^{-\left(\frac{m}{2\sigma_1^2}+\frac{1}{2\sigma_o^2}\right)\left(x - \frac{\sum_{i=1}^m \frac{1}{\sigma_1^2} y_i + \frac{1}{\sigma_o^2}\mu}{\frac{m}{\sigma_1^2}+\frac{1}{\sigma_o^2}}\right)},$$

where the last step follows from standard results on the posterior distribution with Gaussian priors (see, *e.g.,* [76]). In other words, conditioned on $\vec{y}$, x also follows a normal distribution with mean $= \frac{\sum_{i=1}^m \frac{1}{\sigma_1^2} y_i + \frac{1}{\sigma_o^2}\mu}{\frac{m}{\sigma_1^2}+\frac{1}{\sigma_o^2}}$ and variance $= \frac{1}{\frac{m}{\sigma_1^2}+\frac{1}{\sigma_o^2}}$.

Note that this represents the estimator quality using the information of both job-wise variations and task-wise variations. If the estimator is not informed of the job-wise variations, we can take $\sigma_o^2 \to +\infty$, and the conditional distribution of x given $\vec{y}$ becomes normal with mean $\frac{1}{m}\sum_{i=1}^m y_i$ and variance $\frac{\sigma_1^2}{m}$.

From here we can draw the following conclusions. First, whether history-based schemes or sampling-based schemes have better prediction accuracy for an unknown job depends on the relationship between job-wise variations σ_o^2 and the task-wise variation σ_1^2. If the job-wise variations are large but the task-wise variation is small, *i.e.,* $\sigma_o^2 >> \frac{\sigma_1^2}{m}$, then sampling-based schemes will have better prediction accuracy. Conversely, if the job-wise variations are small but the task-wise variation is large, *i.e.,* $\sigma_o^2 << \frac{\sigma_1^2}{m}$, then history-based schemes will have better prediction accuracy. Second, while the accuracy of history-based schemes is fixed at σ_o^2, the accuracy of sampling-based schemes improves as m increases. Thus, when we can afford the overhead of more samples, the sampling-based schemes become favorable. Our results from experimental data below will further confirm these intuitions.

4.4.2 Trace-based Variability Analysis

Our theoretical analysis in §4.4.1 provides insights on how the prediction accuracies of the two approaches depend on the variation of job run times across time and space. To understand how such variations fare against each other in practice, we next measure the actual variations in three production cluster traces.

Traces. Our first trace is provided by 2Sigma [38]. The cluster uses an internal proprietary job scheduler running on top of a Mesos cluster manager [77]. This trace was collected over a period of 7 months, in July 2016, and from 441 machines and contains approximately 0.4 million jobs [78]. Table 4.3 summarizes the information available in the traces that are used in our analysis.

We also include two publicly available traces from Google released in May 2011 and May 2019 [65, 66], collected from 1 and 8 Borg [79] cells over periods of 29 and 31 days, respectively. The machines in the clusters are highly heterogeneous, belonging to at least three different platforms that use different micro-architectures and/or memory technologies [80]. Further, according to [63], the machines in the same platform can have substantially different clock rates, memory speed, and core counts. Since the original Google 2019 trace has data from 8 different cells located in 8 different locations, and given that we already have two other traces from the US, we chose the batch tier of Cluster G in the Google 2019 trace, which is located in Singapore [73], as our third trace to diversify our trace collection.

We calculate the variations in task runtimes for each job across time and across space as follows.

Variation across time. To measure the variation in mean task runtime for a job across the history, we follow the following prediction mechanism defined in 3Sigma [30] to find similar jobs.

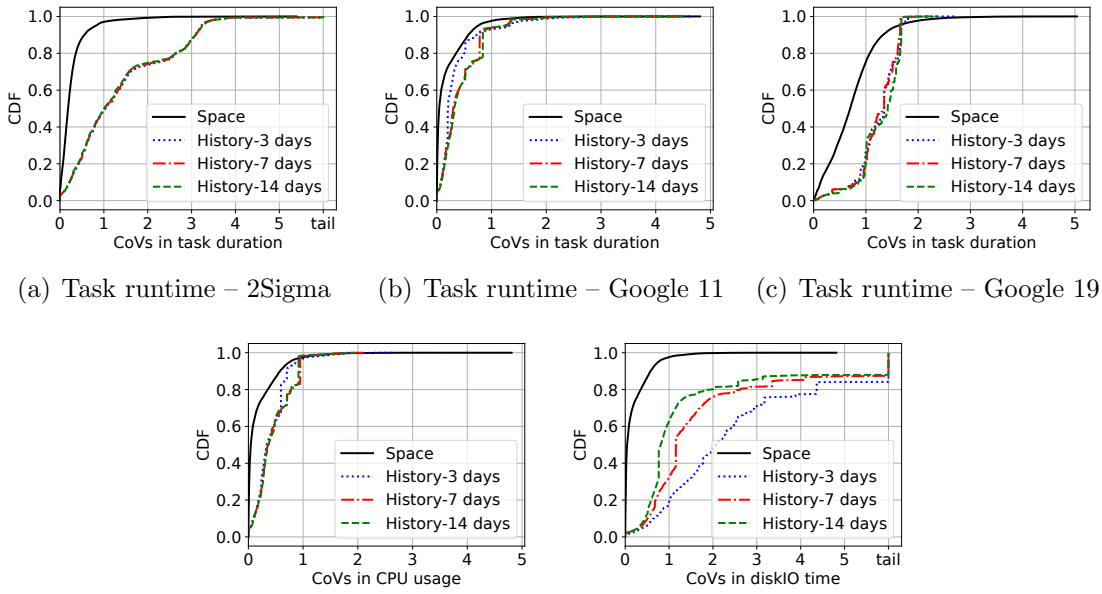
As discussed in §4.2.3, 3Sigma [30] uses multiple features to identify a job and predicts its runtime using the feature that gives the least prediction error in the past.

We include all six features used in 3Sigma: application name, job name, user name (the owner of the job), job submission time (day and hour), and resources requested by the job.

For each feature, we define the set of similar jobs as all the jobs executed in the history window (defined below) that had the same feature value. Next, we calculate the average task runtime of each job in the set. Then, we calculate the *Coefficient of Variation* (CoV) of the average task runtimes across all the jobs in the set. We repeat the above process for all the features. We then compare the CoV values thus calculated and pick the minimum CoV. Effectively, the above procedure selects the least possible variation across history.

Varying the history length in prediction across time. 3Sigma used the entire history for prediction. Intuitively, the length of the history affects the trade-off between the number of similar jobs and the staleness of the history information. For this reason, we optimized 3Sigma by finding and using the history length that gives the least variation. Specifically, we define the length of history based on a window size w, *i.e.,* the number of past consecutive days. In our analysis below, we vary w among 3, 7, and 14 for the three traces.

Variation across space. To measure the extent of variation across space, we look at the CoV (CoV $= \frac{\sigma}{\mu}$) in the task runtimes within a job. As shown in §4.4.1, the variance in the task runtime predicted from sampling is $\frac{\sigma_1^2}{m}$, where σ_1^2 is the variance in the runtimes across all the tasks within the job and m is the number of tasks sampled. Thus, we first estimate σ_1^2 from all tasks within the job. We then report the CoV of our task runtime prediction after sampling m tasks as $\frac{\sigma_1/\sqrt{m}}{\mu}$. Our complete scheduler design in §4.5.1 uses an adaptive sampling algorithm which mostly uses 3% for the three traces. Thus, for measuring the extent of variation across space here, we assume a 3% sampling ratio and plot $\frac{\sigma_1}{(\sqrt{0.03 \times numberOfTasksInJob}\ ) \times \mu}$.



(a) Task runtime – 2Sigma (b) Task runtime – Google 11 (c) Task runtime – Google 19

(d) CPU usage – Google 11 (e) Disk IO time – Google 11

Figure 4.1.: CDF of CoV of runtime properties across space and across time with varying history windows, using the 2Sigma, Google 2011 and Google 2019 traces. Single-task jobs are excluded from the analysis across space.

Variability comparison. For consistency, all analysis results here are for the same, shortest trace period that can be used for sliding-window-history based analysis, *e.g.,* the last 15 days under the 14-day window for the 29-day Google 2011 trace. (The analysis then varies the length of the sliding window in history-based learning.) Figure 4.2 visualizes the two CoVs for each of 70 randomly selected jobs from the 2Sigma trace in the order of their arrival, also using the best window size of 14 days. For clarity, we first show the result for 70 jobs extracted at random from the 2Sigma trace, plotted in the order of job arrival in Figure 4.2. For each job, we plot the following two values on the y-axis: (1) the CoV in average task runtime for the jobs in the job's history window, for the feature that gives the least CoV, using 30-day history window which was found to give the least variation across history as shown in Figure4.1(a); and (2) the CoV in task runtimes of the job. We see that for both traces, the variation across history is higher than the variation across tasks for more than 85% of the jobs.

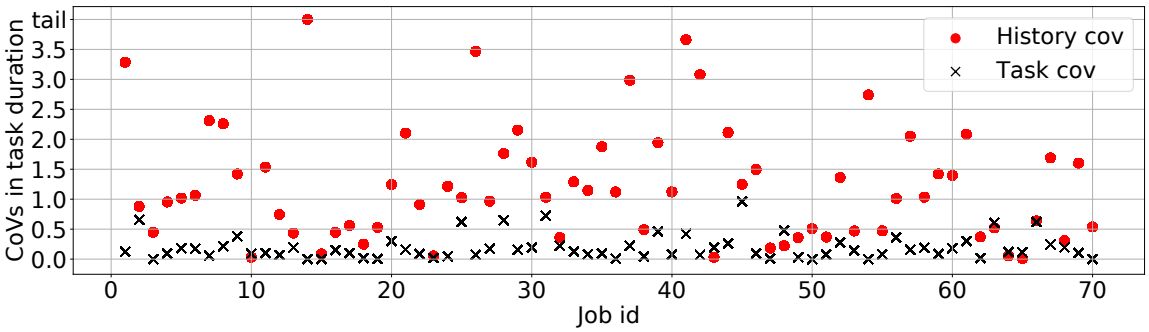


Figure 4.2.: CoVs across time and space for 70 jobs selected randomly from the 2Sigma trace. The x-axis represents job ids in the order of their arrival.

Fig. 4.1(a)–Fig. 4.1(c) show the CDFs of CoVs in task duration measured across space and across history for multiple history window sizes for the three traces. We see that in general using a shorter sliding window reduces the prediction error of 3Sigma, and the CoVs across tasks are moderately lower than the CoVs across history for the Google 2011 trace but significantly lower for 2Sigma and Google 2019 traces. For example, for the 2Sigma trace, the CoV across history is higher than the CoV across tasks for 85.40% of the jobs (not seen in Fig. 4.1(a) as jobs are ordered differently in different CDFs) and for more than 30% of the jobs, the CoV across history is at least 12.10× higher than the CoV across tasks.

Table 4.4 summarizes the results, where the CoVs across time correspond to the best history window size, *i.e.,* 3 days for both of the Google traces and 14 days for the 2Sigma trace. As shown in the table, the P50 (P90) CoV across history are 1.00 (3.10) for the 2Sigma trace, 0.20 (0.73) for the Google 2011 trace, and 1.35 (1.67) for the Google 2019 trace. In contrast, the P50 (P90) CoV value across the task duration of the same set of jobs is much lower, 0.18 (0.55) for the 2Sigma trace and 0.04 (0.58) for the Google 2011 trace. The P50 (P90) task duration CoV value for the Google 2019 trace, 0.70 (1.33), is higher compared to those of the other two traces, but is still much lower when compared to the CoV across history for the same trace, 1.35 (1.67).

Table 4.4.: CoV in task runtime across time and across space for the the 2Sigma, Google 2011, and Google 2019 traces.

Trace	CoV over Time		CoV over Space	
	P50	P90	P50	P90
2Sigma	1.00	3.10	0.18	0.55
Google 2011	0.20	0.73	0.04	0.58
Google 2019	1.35	1.67	0.70	1.33

The CoV values across tasks is much lower because the tasks of a job run the same code with the same flags, settings and priority, as mentioned in the trace schema released by Google [63, 73] and confirmed by 2Sigma engineers [64].

Fig. 4.1(d) and Fig. 4.1(e) further show the CDF of CoVs for CPU usage and Disk IO time for the Google 2011 trace (such resource usage is not available in the 2Sigma trace). The figures show that the variation in the values of these properties when sampled across space is also considerably lower compared to the variation observed over time.

4.4.3 Experimental Prediction Error Analysis

Recall from our analysis in §4.4.1 that lower task-wise variation than job-wise variation (§4.4.2) will translate into better prediction accuracy of sampling-based schemes over history-based schemes. While our analysis in §4.4.1 assumes normal distribution, we believe that a similar conclusion will hold in more general settings. To validate this, we next implement a sampling-based predictor SLearn, and experimentally compare it against a state-of-the-art history-based predictor 3Sigma [30] in estimating the job runtimes directly on production job traces.

Workload characteristics. Since the three production traces described in §4.4.2 are too large, as in 3Sigma [30], we extracted smaller traces for experiments using the procedure described below. We denote the extracted traces which consist of roughly 1250 jobs each as 2STrace, GTrace11 and GTrace19, respectively.

Since the history-based predictor 3Sigma needs a history trace, we followed the same process as in [30] to extract the training trace for 3Sigma and the execution trace for all predictors, in three steps. (1) We divided each original trace in chronological order in two halves. (2) We next selected the execution trace following the process below from the second half; these became 2STrace, GTrace11 and GTrace19, respectively. (3) We then selected jobs from the first half of each original trace that are feature-clustered with those jobs in the execution trace to form the "history" trace for 3Sigma.

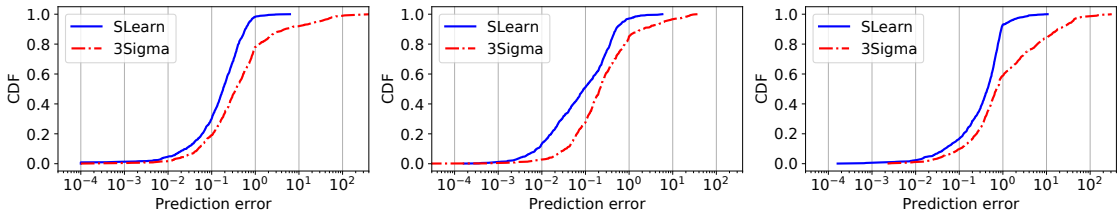
We extracted the execution trace from the above-mentioned second halves as follows. In extracting 2STrace, since the original cluster from where the 2Sigma trace was collected (441 nodes, each with 24 cores) is much larger than our experimental cluster (150 single-core nodes), we resized the job widths to preserve the job-width-to-cluster-size ratio by randomly dropping tasks and then randomly selected 1250 jobs with equal probability as the 2STrace. Similarly, since the Google traces do not have many wide jobs yet the original clusters are very wide, with 12.5K machines, we dropped jobs with more than 150 tasks, and randomly selected 1250 jobs with equal probability from the remaining jobs to create GTrace11 ad GTrace19, respectively.

Finally, for each extracted trace, we adjust the arrival time of the jobs so that the average cluster load matches that in the original trace [65, 66, 78]. Table 4.5 summarizes the workload per window of the extracted traces, where a window is defined as a 1000-second interval sliding by 100 seconds at a time, and the load per window is the total runtime of all the jobs arrived in that window, normalized by the total number of CPUs in the cluster times the window length, *i.e.,* 1000s. We see that for all three traces, the average system load is close to 1, though the load fluctuates over time, which is preserved by the random uniform job extraction.

Prediction mechanisms and experimental setups. We implement the 3Sigma predictor following its description in [30]. After learning the job runtime distribution (§4.4.2), it uses a utility function of the estimated job runtime associated with every

Table 4.5.: Statistics for system load per 1000s sliding window.

Trace	Average	P50	P90
2STrace	1.05	0.13	2.47
GTrace11	1.01	0.29	1.49
GTrace19	1.04	0.09	0.91



(a) 2STrace (b) GTrace11 (c) GTrace19

Figure 4.3.: Job runtime prediction accuracy.

job to derive its estimated runtime from the distribution, by integrating the utility function over the entire runtime distribution. Since our goal is to minimize the average JCT, we used a utility function that is inversely proportional to the square of runtime. We kept all the default settings we learned from the authors of 3Sigma [30].

As in §4.4.2, SLearn samples $max(1, 0.03 \cdot S)$ tasks per job, where S is the number of tasks in the job. We only show the results for wide jobs (with 3 or more tasks) as in the complete SLearn design (§4.5.1.1), only wide jobs go through the sampling phase.

Results. Fig. 4.3 shows the CDF of percentage error in the predicted job runtimes for the three traces. We see that SLearn has much better prediction accuracy than 3Sigma. For 2STrace, GTrace11, and GTrace19, the P50 prediction error are 18.30%, 9.15%, 21.39% for SLearn but 36.57%, 21.39%, 71.56% for 3Sigma, respectively, and the P90 prediction error are 58.66%, 49.95%, 92.25% for SLearn but 475.78%, 294.52%, 1927.51% for 3Sigma, respectively.

We observe that the predictor error for GTrace19 is higher than that for 2STrace and GTrace11 when we use either history-based or sampling-based prediction. This is because GTrace19 has higher CoV values for the average task runtimes, both across

the history and across the tasks of the same job (Table 4.4). This result also confirms that the prediction accuracy of both schemes are directly affected by the variance (§4.4.1).

4.5 Integrating Sampling-based Learning with Job Scheduling: A Case Study

In this section, we answer the second key question about the sampling-based learning: Can delaying scheduling the remaining tasks till completing the sampled tasks be compensated by the improved prediction accuracy? We answer it through extensive simulation and testbed experiments.

Our approach is to design a generic scheduler, denoted as GS, that schedules jobs based on job runtime estimates to optimize a given performance metric, average job completion time (JCT). We then plug into GS different prediction schemes to compare their end-to-end performance. In particular, we compare four predictors: (1) the sampling-based predictor SLearn, (2) the distribution based predictor proposed in 3Sigma [30], (3) a point estimate predictor, and (4) a LAS estimator. (5) an Oracle estimator, which always predicts with 100% accuracy. We further compare with a FIFO-based scheme, where the scheduler simply prioritizes jobs in the order of their arrival.

4.5.1 Scheduler and Predictor Design

4.5.1.1 Generic Scheduler GS

GS replaces the scheduling component of a cluster manager like YARN [81]. The key scheduling objective of GS is to minimize the average JCT. Additionally, GS aims to avoid starvation. so that all jobs can continually make progress.

The scheduling task in GS is divided into two phases, (1) job runtime estimation, and (2) efficient and starvation-free scheduling of jobs whose runtimes have been

estimated. We focus here on the scheduling mechanism and discuss the different job runtime estimators in the following sections.

Inter-job scheduling. Shortest job first (SJF) is known to be optimal in minimizing the average JCT when job execution depends on a single resource. Previous work has shown that scheduling distributed jobs even with prior knowledge is NP-hard (*e.g.,* [34]), and an effective online heuristic is to order the distributed jobs based on each job's total size [11]. In GS we use a similar heuristic; the jobs are ordered based on their total estimated runtime, *i.e., mean task runtime* $\times$ *number of tasks*.

Starvation avoidance. SJF is known to cause starvation to long jobs. Hence, in GS we adopt a well-known multi-level priority queue structure to avoid job starvation [11,32,82,83]. Once GS receives the runtime estimates of a job, it assigns the job to a priority queue based on its runtime. Within a queue, we use FIFO to schedule jobs. Across the queues, we use weighted sharing of resources, where a priority queue receives a resource share according to its priority.

In particular, GS uses N queues, Q_0 to Q_{N-1}, with each queue having a lower queue threshold Q_q^{lo} and a higher threshold Q_q^{hi} for job runtimes. We set $\mathrm{Q}_0^{lo} = 0$, $\mathrm{Q}_{N-1}^{hi} = \infty$, $\mathrm{Q}_{q+1}^{lo} = \mathrm{Q}_q^{hi}$. A queue with a lower index has a higher priority. GS uses exponentially growing queue thresholds, *i.e.,* $\mathrm{Q}_{q+1}^{hi} = \mathrm{E} \cdot \mathrm{Q}_q^{hi}$. To avoid any bias, we use the multiple priority queue structure with the same configuration when comparing different job runtime estimators.

Basic scheduling operation. GS keeps track of resources being used by each priority queue. It offers the next available resource to a queue such that the weighted sharing of resources among the queues for starvation avoidance is maintained. Resources offered to a queue are always offered to the job at the head of the queue.

4.5.1.2 SLearn

Since SLearn learns job runtimes online by sampling pilot tasks, it needs to interact with the scheduler. To seamlessly integrate SLearn with GS, we need to use one of the priority queues for scheduling sampled tasks. We denote it as the sampling queue.

Fast sampling. One design challenge is how to determine the priority for the sampling queue w.r.t. the other priority queues. On one hand, sampled tasks should be given high priority so that the job runtime estimation can finish quickly. On the other hand, the jobs whose runtimes have already been estimated should not be further delayed by learning new jobs. To balance the two factors, we use the second highest priority in GS as the sampling queue.

Handling thin jobs. Recall that in SLearn, when a new job arrives, SLearn only schedules its pilot tasks, and delay other tasks until the pilot tasks finish and the job runtime is estimated. Such a design choice can inadvertently lead to higher JCTs for thin jobs, *e.g.,* a two-task job would experience serialization of its two tasks. To avoid JCT degradations for thin jobs, we place a job directly in the highest priority queue if its width is under a threshold thinLimit.

Basic operations. Upon the arrival of a new job, the cluster manager asynchronously communicates the job's information to GS, which relays the information to SLearn. If the number of tasks in the job is under thinLimit, SLearn assigns it to the highest priority queue; otherwise, the job is assigned to the sampling queue, where a subset of its tasks (*pilot tasks*) will be scheduled to run. Once a job's runtime is estimated from sampling, it is placed in the priority queue corresponding to its runtime estimate where the rest of its tasks will be scheduled.

How many and which pilot tasks to schedule? When a new job arrives, SLearn first needs to determine the number of pilot tasks. Sampling more tasks can

give higher estimation accuracy, but also consumes more resources early on, which can potentially delay other jobs, if the job turns out to be a long job and should have been scheduled to run later under SJF. Further, we found the best sampling ratio appears to vary across difference traces. To balance the trade-off, we use an adaptive algorithm to dynamically determine the sampling ratio, as shown in Figure 4.4. The basic idea of the algorithm is to suggest a sampling ratio that has resulted in the lowest job completion time normalized by the job runtime based in the recent past. To achieve this, for every value in a defined range of possible sampling ratios (between 1% and 5%), it maintains a running score ($srScoreMap$), which is the average normalized JCT of T recently finished jobs that used the corresponding sampling ratio. In practice we found a T value of 100 works reasonably well. During system start-up, it tries sampling ratios of 2%, 3%, and 4% for the first $3T$ jobs (Line 2–7). It further tries sampling ratios of 1% and 5% if going down from 3% to 2% or going up from 3% to 4% reduces the normalized JCT. Afterward, for each new job, it uses the sampling ratio that has the lowest running score. Finally, upon each job completion, the score map is updated (Line 16–24).

Once the sampling ratio is chosen, SLearn selects pilot tasks for a job randomly.

How to estimate from sampled tasks? Several methods such as bootstrapping, statistical mean or median can be used to predict job properties from sampled tasks. In GS, we use empirical mean to predict the mean task runtime.

Work conservation. When the system load is low, some machines may be idle while the non-sampling tasks are waiting for the sampling tasks to finish. In such cases, SLearn schedules non-sampling tasks of jobs to run on otherwise idle machines. In work conservation, the jobs are scheduled in the FIFO order of their arrival.

```
1: procedure GetCurrentSamplingPercentage(Job j)
2:     if j in First T jobs then
3:         return 3
4:     else if j in Second T jobs then
5:         return 2
6:     else if j in Third T jobs then
7:         return 4
8:     minScore = getMinValue(srScoreMap)
9:     if minScore.SR == 2 then
10:        if 1.1*minScore.value < srScoreMap[3].value then
11:            return 1
12:    if minScore.SR == 4 then
13:        if srScoreMap[3].value > 1.1*minScore.value then
14:            return 5
15:    return minScore.SR
16: procedure UpdateScoreOnJobCompletion(Job j)
17:    sr = j.sr                          ▷ Get j's sampling ratio.
18:    normalizedJCT = j.jct              ▷ Get j's normalized JCT.
19:    UpdateScoresMap(sr, normalizedJCT)
20: procedure UpdateScoreMaps(sr, normalizedJCT)
21:    if Len(jobWiseSrScoresMap[sr])> T then
22:        Drop first element of jobWiseSrScoresMap[sr]
23:    jobWiseSrScoresMap[sr].append(normalizedJCT)
24:    srScoreMap[sr].value = mean(jobWiseSrScoresMap[sr])
```

Figure 4.4.: Adaptive sampling algorithm in SLearn.

4.5.1.3 Baseline Predictors and Policies

We compare SLearn's effectiveness against four different baseline predictors and two policies: **(1) 3Sigma:** as discussed in §4.4.3. **(2) 3SigmaTL:** same as 3Sigma but handles thin jobs in the same way as SLearn; they are directly placed in the highest priority queue. This is to isolate the effect of thin job handling. **(3)** Point-Est**:** same as 3Sigma, with the only difference being that, instead of integrating a utility function over the entire runtime history, it predicts a point estimate (median in our case) from the history. **(4) LAS:** The Least Attained Service [32] policy approximates SJF online without explicitly learning job sizes, and is most recently implemented in the Kairos [33] scheduler. LAS uses multiple priority queues and the priority is inversely proportional to the service attained so far, *i.e.,* the total execution

time so far. We use the sum of all the task execution time to be consistent with all the other schemes. **(5) FIFO:** The FIFO policy in YARN simply prioritizes jobs in the order of their arrival. Since FIFO is a starvation free policy, there is no need for multiple priority queues. **(6) ORACLE:** ORACLE is an ideal predictor that always predicts with 100% accuracy.

4.5.2 Experimental Results

We evaluated SLEARN's performance against the six baseline schemes discussed above by plugging them in GS and execute the 3 traces (2STrace, GTrace11, and GTrace19) on a 150-node testbed cluster in Azure and using large scale simulations.

4.5.2.1 Experimental Setup

Cluster setup. We implemented GS, SLEARN and baseline estimators with 11 KLOC of Java and python2. We used an open source java patch for Gridmix [84] and open source java implementation of NumericHistogram [85] for Hadoop. We used some parts from DSS, an open source job scheduling simulator [86], in simulation experiments.

We implemented a proxy scheduler wrapper that plugs into the resource manager of YARN [81] and conducted real cluster experiments on a 150-node cluster in MS Azure [48].

Following the methodology in recent work on cluster job scheduling [30, 70, 72], we model jobs as mapper-only jobs. We implemented a synthetic generator based on the Gridmix implementation to replay mapper-only jobs that follow the arrival time and task runtime from the input trace. The master runs on a standard DS15 v2 server with 20-core 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor and 140GB memory. The slaves run on D2v2 with the same processor with 2-core and 7GB memory.

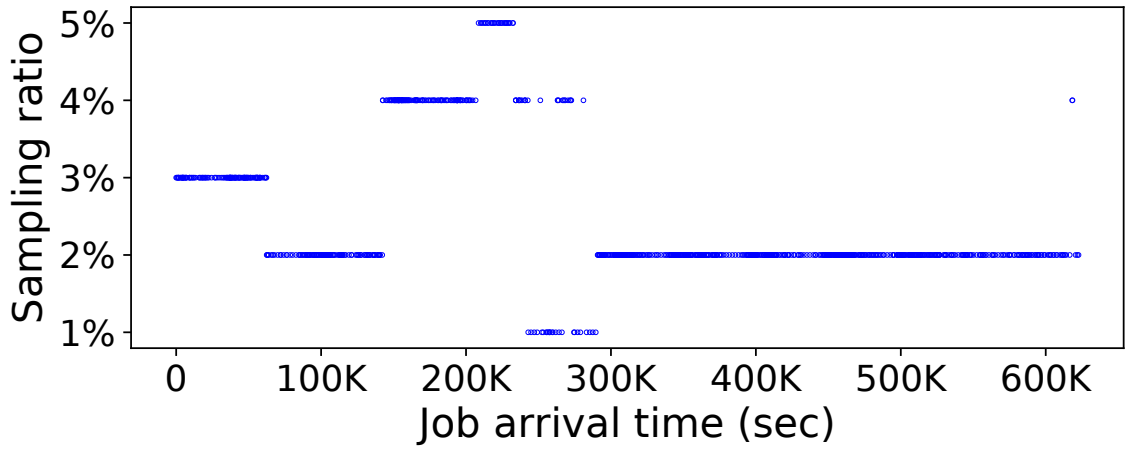
Parameters. The default parameters for priority queues in GS in the experiments are: starting queue threshold (Q_0^{hi}) is 10^6 ms, exponential threshold growth factor (E) is 10, number of queues (N) is set to 10, and the weights for time sharing assigned to individual priority queues decrease exponentially by a factor of 10. Previous work (*e.g.,* [11]) and our own evaluation have shown that the scheduling results are fairly insensitive to these configuration parameters. We omit their sensitivity study here due to page limit. SLearn chooses the number of pilot tasks for wide jobs using the adaptive algorithm described in §4.5.1.2 and the threshold for thin jobs is set to 3. We evaluate the effectiveness of adaptive sampling in §4.5.2.2 and the sensitivity to thinLimit in §4.5.2.8.

Performance metrics. We measure three performance metrics in the evaluation: JCT speedup, defined as the ratio of a JCT under a baseline scheme over under SLearn, the job runtime estimation accuracy, and job waiting time.

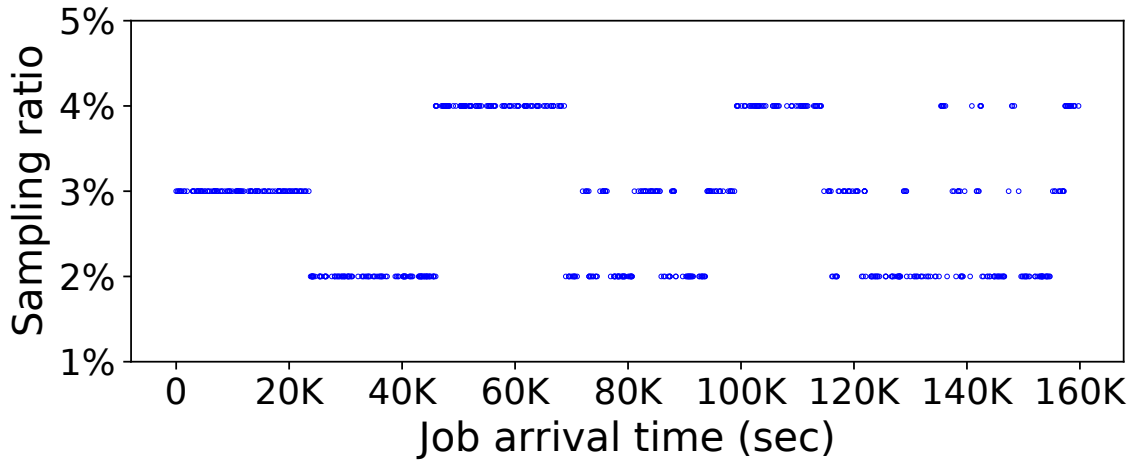
Workload. We used the same training data for history-based estimators and the test traces (2STrace, GTrace11 and GTrace19) as described in §4.4.3. We could not use the Mustang and Trinity traces released in [80] as they do not contain information about individual task runtimes.

4.5.2.2 Effectiveness of Adaptive Sampling

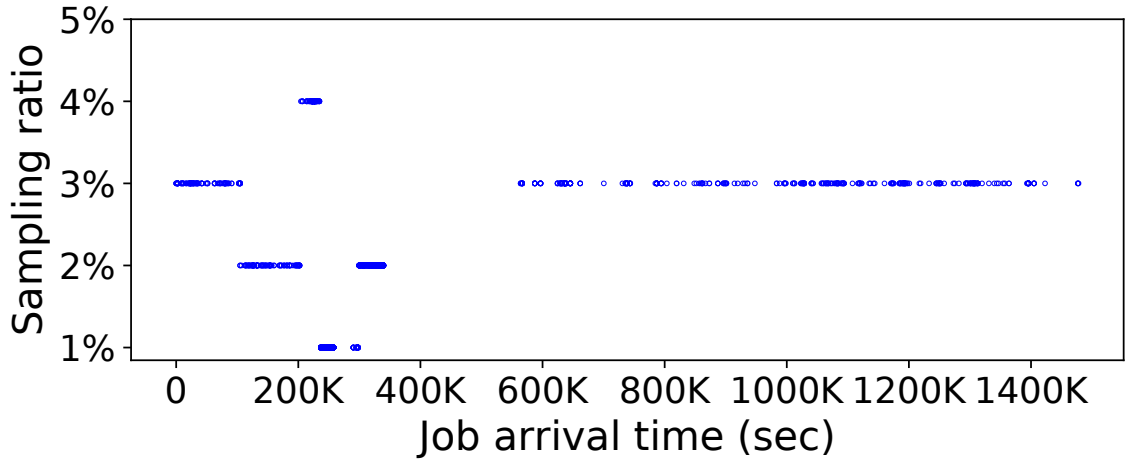
In this experiment, we evaluate the effectiveness of our adaptive algorithm for task sampling. Fig. 4.5 shows how the sampling ratio selected by the adaptive algorithm for each job varies between 1% and 5% over the duration of the three traces. We further compare average JCT speedup and P50 speedup under the adaptive algorithm with those under a fixed sampling ratio, ranging between 1% and 10%. Table 4.6 shows that the adaptive sampling algorithm leads to the best speedups for 2STrace and GTrace19 and is about only 1% worse than the best for GTrace11. Interestingly, we observe that no single sampling ratio works the best for all traces. Nonetheless, the



(a) 2STrace



(b) GTrace11



(c) GTrace19

Figure 4.5.: Sampling ratios selected by the adaptive sampling algorithm. The duration of initial $3T$ jobs appear varying due to uneven arrival times.

Table 4.6.: Performance improvement of SLearn over 3Sigma under adaptive sampling and fixed-ratio sampling.

	Fraction of tasks chosen as pilot tasks						
	1%	2%	3%	4%	5%	10%	Adap.
	2STrace						
P50 pred. error (%)	19.35	18.98	18.97	18.70	18.44	16.94	18.98
Avg. JCT speedup (×)	1.24	1.23	1.27	1.26	1.27	1.28	1.28
P50 speedup (×)	0.93	0.92	0.93	0.92	0.93	0.91	0.92
	GTrace11						
P50 pred. error (%)	14.38	14.04	13.62	13.11	12.69	9.09	13.68
Avg. JCT speedup (×)	1.52	1.55	1.54	1.56	1.58	1.51	1.56
P50 speedup (×)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	GTrace19						
P50 pred. error (%)	55.65	53.81	47.11	46.52	42.08	36.10	51.84
Avg. JCT speedup (×)	1.31	1.31	1.31	1.32	1.28	1.24	1.32
P50 speedup (×)	1.07	1.07	1.05	1.05	1.01	1.00	1.07

adaptive algorithm always chooses one that is the best or closest to the best in terms of JCT speedup. More importantly, we see that the adaptive algorithm does not always use the sampling ratio with the best prediction accuracy, which shows that it effectively balances the tradeoff between prediction accuracy and sampling overhead.

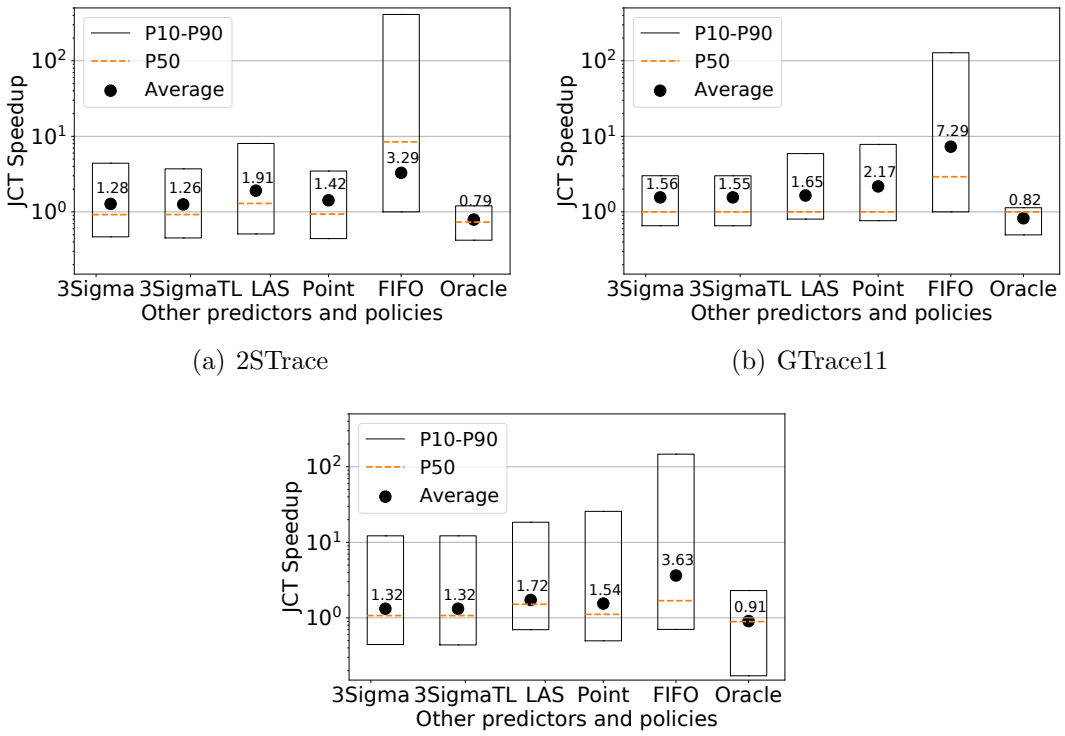
4.5.2.3 Prediction Accuracy

SLearn achieves significantly more accurate estimation of job runtime over 3Sigma – the details were already discussed in §4.4.3.

4.5.2.4 Average JCT Improvement

We now compare the JCT speedups achieved using SLearn over using the five baseline schemes defined in §4.5.1.3.

Fig. 4.6(a) shows the results for 2STrace. We make the following observations. (1) The JCT of Oracle serves as a lower-bound for all other schemes. Compared to Oracle, SLearn achieves an average and P50 speedups of 0.79× and 0.73×, respectively. This is because SLearn has some estimation error; it places 10.91% of



(a) 2STrace

(b) GTrace11

(c) GTrace19

Figure 4.6.: JCT speedup using SLearn as compared to other baseline schemes for the three traces.

wide jobs in the wrong queues, 3.54% in lower queues and 7.37% in higher queues. (2) SLearn improves the average JCT over 3Sigma by 1.28×. This significant improvement of SLearn comes from much higher prediction accuracy compared to 3Sigma (Fig. 4.3). (3) The improvement of SLearn over 3SigmaTL, 1.26×, is similar to that over 3Sigma, confirming thin job handling only played a small role in the performance difference of the two schemes. To illustrate SLearn's high prediction accuracy, we show in Table 4.7 the fraction of wide jobs that were placed in correct queues by SLearn and 3Sigma. We observe that SLearn consistently assigns more wide jobs to correct queues than 3Sigma for all three traces. (4) Compared to Point-Est, which uses a point estimate generated from historical data, SLearn improves the average JCT by 1.42×. Again, this is because SLearn estimates runtimes with higher accuracy. (5) Compared to LAS, SLearn achieves an average JCT speedup

Table 4.7.: Percentage of the wide jobs that had correct queue assignment.

Prediction Technique	SLearn	3Sigma
2STrace	89.09%	73.84%
GTrace11	86.45%	76.20%
GTrace19	73.96%	58.07%

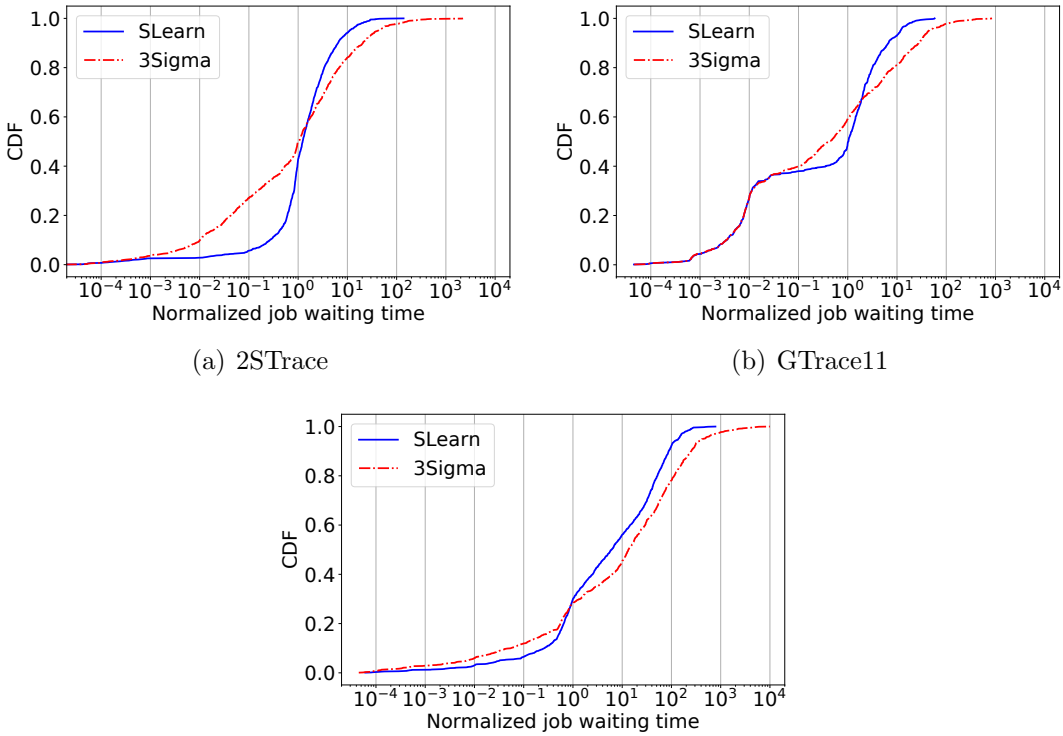
of 1.91× and P50 speedup of 1.29×. This is because LAS pays a heavy penalty in identifying the correct queues of jobs by moving them across the queues incrementally. (6) Lastly, compared with FIFO, SLearn achieves an average JCT speedup of 3.29× and P50 speedup of 8.45×.

Fig. 4.6(b) shows the results for GTrace11. Scheduling under SLearn again outperforms all other schemes. In particular, using SLearn improves the average JCT by 1.56× compared to using 3Sigma, 1.55× compared to using 3SigmaTL, 2.17× compared to using Point-Est, and 1.65× compared to using the LAS policy. Fig. 4.6(c) shows that scheduling under SLearn outperforms all other schemes for GTrace19 too. In particular, using SLearn improves the average JCT by 1.32×, 1.32×, 1.54×, and 1.72× compared to using 3Sigma, 3SigmaTL, Point-Est and the LAS policy, respectively.

In summary, our results above show that SLearn's higher estimation accuracy outweighs its runtime overhead from sampling, and as a result achieves much lower average job completion time than history-based predictors and the LAS policy for the three production workloads.

4.5.2.5 Impact of Sampling on Job Waiting Time

To gain insight into why sampling pilot tasks first under SLearn does not hurt the overall average JCT, we next measure and compare the *normalized waiting time* of jobs, calculated as the average waiting time of its tasks under the respective scheme, divided by the mean task length of the job.



(a) 2STrace

(b) GTrace11

(c) GTrace19

Figure 4.7.: CDF of waiting times for wide jobs

Fig. 4.7(b) shows the CDF of the normalized job waiting time under SLearn and 3Sigma. We see that the CDF curves can be divided into three segments. (1) The first segment, where both SLearn and 3Sigma have normalized waiting time (NWT) less than 0.04, covers 36.58% of the jobs, and 35.57% of the jobs are common. The jobs have almost identical NWT, much lower than 1 under both schemes. This happens because during low system load periods, *e.g.,* lower than 1, the scheduler will schedule all the tasks to run under both scheme; under SLearn it schedules non-sampled tasks of jobs to run before their sampled tasks complete due to work conservation. (2) The second segment, where both schemes have NWT between 0.04 and 1.90, covers 30.51% of the jobs, and 20.38% of the jobs are common. Out of these 20.38%, 29.81% have lower NWT under SLearn and 70.19% have lower NWT under 3Sigma. This happens because when the system load is moderate, the jobs experience longer

waiting time under SLearn than under 3Sigma because of sampling delay. (3) The third segment, where both schemes have NWT above 1.90, cover 32.91% of the jobs, and 24.68% of jobs are common. Out of these 24.68%, 83.08% have lower waiting time under SLearn and 16.92% under 3Sigma. This happens because when the system load is relatively high, although jobs incur the sampling delay under SLearn, they also experience queuing delay under 3Sigma, and the more accurate prediction of SLearn allows them to be scheduled following Shortest Job First more closely than under 3Sigma.

Fig. 4.7(a) and fig. 4.7(c) shows the results for the 2STrace and GTrace19. The plots for 2STrace are also similar to GTrace11, though the first segment here is relatively small. For GTrace19, the first segment is almost negligible. Also, the P50 NWT for SLearn in 2STrace is 1.17, and for GTrace19, it is 6.12. The NWT value for GTrace19 is significantly greater than 1. These differences in 2STrace and GTrace19 are because of the nature of the traces. GTrace11 has a mix of batch and non-batch jobs. However, the 2STrace and GTrace19 are purely batch job traces. Hence they are more bustier as compared to GTrace11. The P95 of load per window, with the definition of load the same as used in table 4.5, for GTrace11 is 3.35. Whereas, for GTrace19 and 2STrace, it is 5.27 and 10.68, respectively. The jobs which arrive in more bursty groups end up waiting for longer times.

In summary, as the system load fluctuates above and below 1 over time [65, 66], sampling pilot tasks first under SLearn does not hurt job completion time when the system load is low due to work conversation, and helps to reduce the average job completion time when the system load is high from more accurate job runtime prediction and hence more effective scheduling. A detailed timeline analysis of how the system load of the trace affects the relative job performance under the two predictors can be found in the appendix.

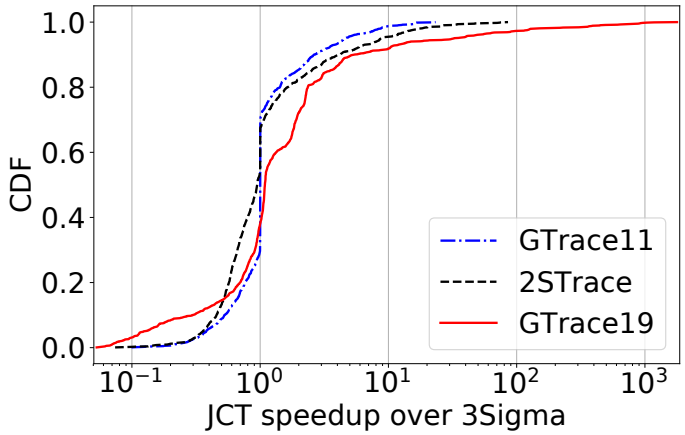


Figure 4.8.: [Testbed] CDF of speedup of SLearn over 3Sigma.

Table 4.8.: Breakdown of jobs based on total duration and width (number of tasks) for 2STrace. Shown in brackets are a bin's share in term of job count and total job runtime.

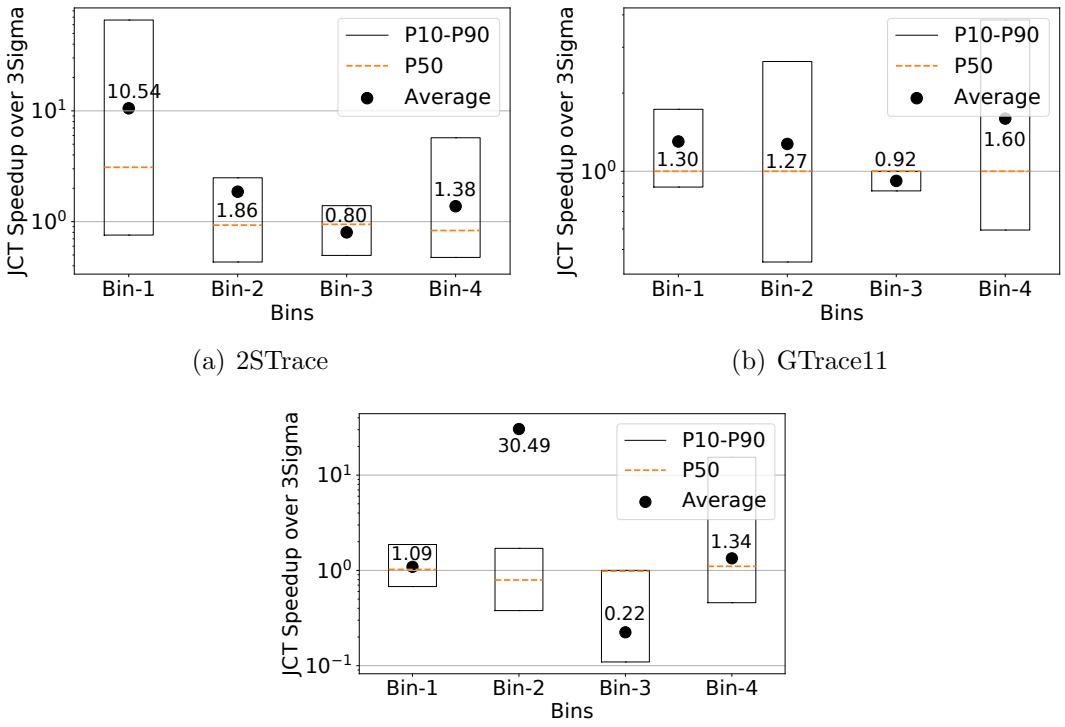
	width < 3 (thin)	width ≥ 3 (wide)
size $< 10^3 s$ (small)	bin-1 (4.55%, 0.01%)	bin-2 (28.73%, 0.06%)
size $\geq 10^3 s$ (large)	bin-3 (14.29%, 5.41%)	bin-4 (52.43%, 94.52%)

4.5.2.6 Testbed Experiments

We next perform end-to-end evaluation of SLearn and 3Sigma on our 150-node Azure cluster. Fig. 4.8 shows the CDF of JCT speedups using SLearn over 3Sigma using 2STrace, GTrace11 and GTrace19. SLearn's performance on the testbed is similar to that observed in the simulation. In particular, SLearn achieves average JCT speedups of 1.33×, 1.46×, and 1.25× over 3Sigma for the 2STrace, GTrace11, and GTrace19 traces, respectively.

4.5.2.7 Binning Analysis

To gain insight into how different jobs are affected by SLearn over 3Sigma, we divide the jobs into four bins in Table 4.8 for 2STrace and show the JCT speedups for each bin in Fig. 4.9(a).



(a) 2STrace

(b) GTrace11

(c) GTrace19

Figure 4.9.: Performance breakdown into the bins in Table 4.8, 4.9, and 4.10 respectively.

Table 4.9.: Breakdown of jobs based on total duration and width (number of tasks) for GTrace11. Shown in brackets are a bin's share in term of job count and total job runtime.

	width < 3 (thin)	width ≥ 3 (wide)
size $< 10^3 s$ (small)	bin-1 (35.25%, 0.66%)	bin-2 (8.00%, 0.26%)
size $\geq 10^3 s$ (large)	bin-3 (1.60%, 0.51%)	bin-4 (55.15%, 98.57%)

Table 4.10.: Breakdown of jobs based on total duration and width (number of tasks) for GTrace19. Shown in brackets are a bin's share in term of job count and total job runtime.

	width < 3 (thin)	width ≥ 3 (wide)
size $< 10^3 s$ (small)	bin-1 (0.80%, 0.00%)	bin-2 (3.74%, 0.01%)
size $\geq 10^3 s$ (large)	bin-3 (1.99%, 0.56%)	bin-4 (93.47%, 99.43%)

We make the following observations. (1) SLearn improves the JCT for 82.46% of the jobs in Bin-1 and the average JCT speedup for the bin is 10.54×. This happens because the jobs in this bin are thin and hence SLearn assigns them high priorities, which is also the right thing to do since these jobs are also small. (2) For bin-2, SLearn achieves an average JCT speedup of 1.86× from better prediction accuracy of SLearn. The speedups are lower than for bin-1 as the jobs have to undergo sampling. However, Bin-1 and Bin-2 make up only 0.01% and 0.06% of the total job runtime and thus have little impact on the overall JCT. (3) Bin-3, which has 14.29% of the jobs and accounts for 5.41% of the total job size, has a slowdown of 20.00%. The main reason is that SLearn treats thin jobs in the FIFO order, whereas 3Sigma schedules them based on predicted sizes. (4) Finally, Bin-4, which accounts for a majority of the job and total job size, has an average speedup of 1.38×, which contributes to the overall speedup of 1.28×. The job speedups come from more accurate job runtime estimation of SLearn over 3Sigma.

The results for the GTrace11, and GTrace19 are similar and are shown in Figure 4.9(b), and Figure 4.9(c) respectively. The bin size distributions for GTrace11, and GTrace19 is given in Table 4.9, and Table. 4.10.

4.5.2.8 Sensitivity to Thin Job Bypass

In this section, we evaluate SLearn's sensitivity to thinLimt. The results in Table 4.13 show that for GTrace11 and GTrace19, the average JCT speedup barely varies with thinLimit, but for 2STrace, there is a big dip when increasing thinLimit to four or five. This is because a significant number of jobs in 2STrace have width four, which causes the number of thin jobs to increase from 5.41% to 13.84% when increasing thinLimit from 3 to 4.

4.5.2.9 Intuitive Explanation of JCT Speedups in SLearn over 3Sigma

Finally, we provide an intuitive explanation for SLearn's improvement over 3Sigma. Figure 4.10 shows seven timeline values comparing SLearn and 3Sigma for the 2STrace as follows:

- The top curve shows the total workload arrived in the past 1000 seconds, in terms of execution duration. The values are plotted in steps of 1000 seconds along the x-axis. A unit along the y-axis corresponds to the workload that needs 1000 seconds of the entire cluster's compute capacity. Thus a workload of 1 in steady state implies no queue build-up under 100% utilization of the whole cluster.

- The next three curves show the *resistance* faced by newly arrived jobs under Oracle, 3Sigma and SLearn, respectively, where *resistance* for a job is defined as the amount of higher priority workload existing at the time of its arrival, including the remaining duration of the already scheduled tasks. A unit along the y-axis for these curves also corresponds to the workload that needs 1000 seconds of the entire cluster's compute capacity. For wide jobs (*i.e.,* with 3 or more tasks), under SLearn we show the *resistance* value corresponding to the moment when the job's size estimation is over and it has been placed in its estimated priority queue. The *resistance* values are plotted along the x-axis corresponding to each job's arrival time.

- The next two curves correspond to the percentage prediction error in 3Sigma and SLearn, respectively. They show signed error which are capped at 1000, *e.g.,* a value of -20 on error curves means the job was estimated to be 20% smaller and a value of 1000 means job was estimated at least 1000% larger. The values are plotted along the x-axis corresponding to each job's arrival time.

- The bottom curve shows the job speedup (positive values) or slowdown (negative values) of SLearn compared to 3Sigma, plotted along the x-axis corresponding

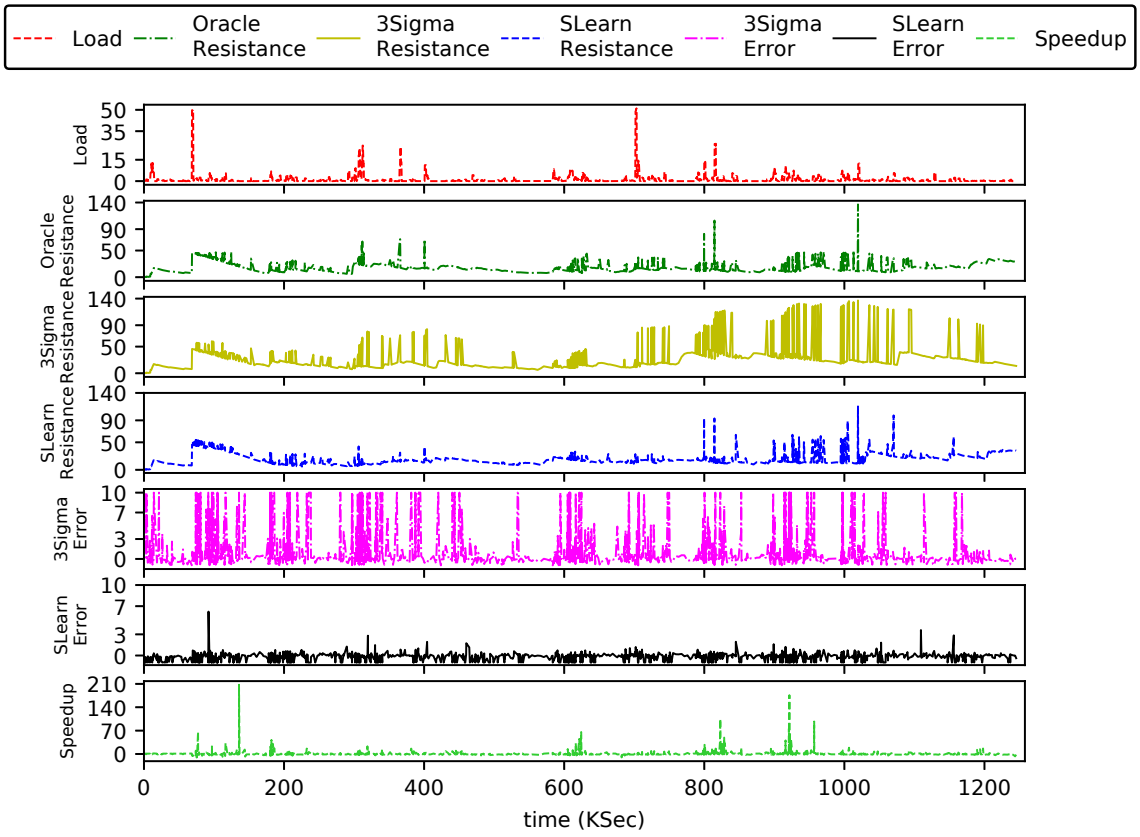


Figure 4.10.: Correlation between load, *resistance*, estimation error and speedup for 2STrace.

to each job's arrival time. Thus all values are either above 1, showing the speedups of jobs under SLearn over under 3Sigma, or below -1, showing the speedups of jobs under 3Sigma over under SLearn.

With the above definitions of the curves, we next discuss how these curves in Fig. 4.10 demonstrate provides insights to when and why SLearn outperforms 3Sigma.

- The speedup curve (bottom) shows the speedup under SLearn over under 3Sigma happens when the workload is high, *e.g.,* between 600s and 620s, and 800s to 840s. Conversely, when the workload is below 1, *e.g.,* between 400 and 600s, the two scheme perform similarly and there is no speedup of either scheme. In such cases, task sampling SLearn did not hurt jobs because non-

sampled tasks did not have to wait for completion of sampled tasks due to work conservation (§4.5.1.2).

- Intuitively, under any scheme, a job's completion time is roughly proportional to its own total runtime (which is independent of the scheduling) plus the *resistance* it sees upon arrival, because the *resistance* value indicates the amount of workload that needs to be scheduled before the arriving job gets to run.

- The *resistance* value, in turn, depends on the recently arrived workload and the prediction error and hence the scheduling decision for them.

- First, if more workload has arrived in the recent past, it is likely that a newly arrived job will face higher *resistance*. This is shown by the strong correlation between the load curve and the ORACLE resistance curve.

- Second, high runtime prediction error can lead to high *resistance*. When the job runtime is estimated by the predictor to be larger than its actual size, it may be misplaced in a lower priority queue. If the error is more than 1000% then the job will definitely be placed in a lower priority queue. In such cases, the job will likely face higher *resistance* than it would have with accurate estimation. Conversely, when the job runtime is underestimated, it may be placed in a higher priority queue. Though such a job will finish faster than otherwise, it will create more *resistance* for other jobs that are actually smaller than it and thus slow them down.

- The above impact of prediction error on *resistance* can be seen in Fig. 4.10. Since the prediction accuracy of SLEARN is high, it has less impact on the *resistance* and as a result its *resistance* (fourth curve) is very similar to that of ORACLE (second curve). In contrast, the *resistance* curve for 3Sigma (third curve) has many spikes, *e.g.,* between 800s and 1050s, which happen when the workload (top curve) is high and it has high positive prediction error (fifth curve).

Table 4.11.: Fraction of over-estimated and mis-placed jobs for 2STrace. Job performance in the third and seventh column is relative to the ORACLE.

	Overestimated jobs	Misplaced overestimated jobs	Slowed misplaced jobs	Average (P50) Positive error
3Sigma	59.78%	17.50%	12.19%	898.45 (48.00)%
SLEARN	43.75%	3.54 %	2.85%	30.65 (18.19)%

Table 4.12.: Fraction of under-estimated and mis-placed jobs for 2STrace. Job performance in the third and seventh column is relative to the ORACLE.

	Underestimated jobs	Misplaced under estimated jobs	Spedup misplaced jobs	Average (P50) Negative error
3Sigma	40.22%	8.65%	6.88%	-37.0 (-28.57)%
SLEARN	55.45%	7.37%	3.64%	-26.79 (-20.69)%

- Finally, we can see that where ever there is higher *resistance* under 3Sigma (third curve) compared to under SLEARN (fourth curve), *e.g.,* between 800s and 1000s, jobs experience speedups under SLEARN over under 3Sigma.

While the above explanation using Fig. 4.10 is based on the performance of SLEARN and 3Sigma relative to that of ORACLE, Table 4.11 and table 4.12, gives a direct comparison of the scheduling behavior of the jobs under the two schemes in terms of runtime overestimation/underestimation, prediction error, and the resulting misplacement to the priority queues. We see that a larger number jobs are misplaced under 3Sigma compared to SLEARN which led to the overall lower performance under 3Sigma.

In summary, whether a job finishes faster under SLEARN compared to 3Sigma depends on two factors: the recent workload and the runtime prediction error. Due to higher prediction error of 3Sigma compared to SLEARN, during high workload, jobs are more likely to be misplaced to the priorty queues and hence face higher *resistance*, which results in longer average completion time under 3Sigma.

Table 4.13.: Sensitivity analysis for thinLimit. Table shows average JCT speedup over 3Sigma.

thinLimit	2	3	4	5	6
2STrace	1.23x	1.28x	1.14x	0.97x	0.84x
GTrace11	1.54x	1.56x	1.55x	1.54x	1.53x
GTrace19	1.33x	1.32x	1.32x	1.30x	1.29x

4.6 Discussions and Future Work

Robustness to task skew. There are two factors that can potentially cause high variations in the runtime properties of tasks (*skew*) of a job: heterogeneity in cluster and computation skew. The three traces used in our analysis (§4.4) and experiments (§4.5) are from production datacenter traces. Out of them, the Google traces were collected from heterogeneous clusters, which already include task skew due to cluster heterogeneity, and all traces already include computation skew observed in real applications. Our analysis shows that for such real-world traces, sampling-based learning outperforms the state-of-the-art history-based learning scheme in terms of trace variability (§4.4.2), prediction accuracy (§4.4.3) as well as end-to-end performance (§4.5.2).

4.7 Summary

In this chapter, we performed a comparative study of task-sampling-based prediction and history-based prediction commonly used in the current cluster job schedulers. Our study answers two key questions: (1) Via quantitative, trace and experimental analysis, we showed that the task-sampling-based approach can predict job runtime properties with much higher accuracy than history-based schemes. (2) Via extensive simulations and testbed experiments of a generic cluster job scheduler, we showed that although sampling-based learning delays non-sampled tasks till completion of sampled tasks, such delay can be more than compensated by the improved accuracy over the prior-art history-based predictor, and as a result reduces the average JCT by

1.28×, 1.56×, and 1.32× for three production cluster traces. These results suggest task-sampling-based prediction offers a promising alternative to the history-based prediction in facilitating cluster job scheduling.

5 CONCLUSION AND FUTURE WORK

5.1 Conclusion

The ability to accurately estimate runtime properties of distributed entities (data analytics jobs, communication flows of the same job (coflows)) allows a cluster scheduler to effectively schedule them. State-of-the-art online cluster schedulers try to learn the properties using historical data or least-attained-service (LAS) based multiple priority queue techniques.

In this thesis, we performed a comparative study of sampling-based learning technique, which utilizes *spatial dimension*, against history-based and LAS-based multiple priority queue learning techniques. Our study highlights the following key points: (1) Via quantitative, trace, and experimental analysis, we demonstrate that the sampling-based approach can predict job runtime properties with much higher accuracy than history-based schemes and when compared to multiple priority queue techniques it is much faster. (2) Via extensive simulations and testbed experiments on a 150-node cluster in Microsoft Azure of a generic cluster job scheduler, we show delaying non-sampled tasks till completion of sampled tasks in sampling-based learning can be more than compensated by the improved accuracy over the prior-art history-based predictor. As a result of it the average JCT improves 1.44× and 1.40×, for two production cluster traces. (3) We also performed simulations and testbed experiments using the same azure cluster for coflow scheduling. Compared to existing LAS based schedulers we achieve 1.51× and 1.36× average CCT speedup on two different cluster traces. These results suggest sampling-based prediction offers a promising alternative to both history and LAS based learning mechanisms facilitating distributed cluster scheduling.

We also designed a coflow scheduler which focuses on better synchronization across *spatial dimension* of coflows. By merely carefully considering the *spatial dimension*, this designs achieves a median speedup in individual CCTs of 1.53× and 1.42× as compared to existing state-of-the-art schedulers.

Our work suggests consideration of *spatial dimension* in cluster scheduling can open up new promising options.

5.2 Future Work

Combining history and sampling-based learning. We envision a number of directions for the further study of learning runtime properties of distributed jobs for efficient cluster scheduling. There are several motivations for exploring combining history and sampling-based learning. (1) History-based learning can be used to establish a prior distribution, and sampling-based approach can be used to refine the posterior distribution. Such a combination may potentially be more accurate than using either history or sampling alone. For example, knowing the distribution of task lengths can help develop better max task length predictors. (2) Though not seen in the production traces used in our study, in case task-wise variation and job-wise variation fluctuate, adaptively switching between the two prediction schemes may help. (3) As shown in §4.5.2.5, performance of sampling based prediction and history based prediction also depend on system load. Under moderate load conditions history performs better whereas under high load sampling. So, this is another potential to merge the two learning approaches.

Learning for DAG jobs. For multi-phase DAG jobs, sampling-based prediction can be applied in each phase to optimize the performance of each phase. An interesting question is how to apply sampling if we wish to learn the runtime properties and optimize the performance of a multi-phase job as a whole (*e.g.,* [10, 75]). We expect that it may again be helpful to combine history-based prediction of the parameters

of future phases with sampling-based prediction of the current phase, which we plan to study in future work.

REFERENCES

[1] 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.

[2] Apache hadoop. http://hadoop.apache.org.

[3] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[4] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *Cidr*, volume 1, pages 2–1, 2013.

[5] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *2011 IEEE 27th International Conference on Data Engineering*, pages 231–242. IEEE, 2011.

[6] Apache hive. http://hive.apache.org.

[7] Apache tez. http://tez.apache.org.

[8] Big data market size revenue forecast worldwide from 2011 to 2027. https://www.statista.com/statistics/254266/global-big-data-market-forecast/.

[9] Sourav Mazumder, Robin Singh Bhadoria, and Ganesh Chandra (Eds.) Deka. *Distributed Computing In Big Data Analytics*. Springer, 2017.

[10] 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, pages 99–112, New York, NY, USA, 2012. ACM.

[11] 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.

[12] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI 16), pages 117–134, Savannah, GA, 2016. USENIX Association.

[13] Calin Iorgulescu, Florin Dinu, Aunn Raza, Wajih Ul Hassan, and Willy Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 97–109, Santa Clara, CA, 2017. USENIX Association.

[14] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[15] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.

[16] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 2:1–2:14, New York, NY, USA, 2014. ACM.

[17] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 455–466, New York, NY, USA, 2014. ACM.

[18] How much data is out there. https://www.nodegraph.se/how-much-data-is-on-the-internet/ Accessed 29th August 2020.

[19] Alba Amato. *On the Role of Distributed Computing in Big Data Analytics*, pages 1–10. Springer International Publishing, Cham, 2017.

[20] Gartner. Hype cycle for big data. In *On the role of Distributed Computing in Big Data Analytics*, volume 11. 2012.

[21] Hpcc systems. https://hpccsystems.com/.

[22] Apache spark. http://spark.apache.org.

[23] Apache hadoop map reduce tutorial. Accessed 29th August 2020. https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[24] Enis Afgan, Purushotham Bangalore, and Karolj Skala. Application information services for distributed computing environments. *Future Generation Computer Systems*, 27(2):173–181, 2011.

[25] 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.

[26] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[27] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 1–13, Philadelphia, PA, 2014. USENIX Association.

[28] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[29] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, Murali Chintalapati, and Dongmei Zhang. Improving service availability of cloud systems by predicting disk error. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 481–494, Boston, MA, 2018. USENIX Association.

[30] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 2:1–2:17, New York, NY, USA, 2018. ACM.

[31] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. Perforator: Eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 415–427, New York, NY, USA, 2016. ACM.

[32] Idris A. Rai, Guillaume Urvoy-Keller, and Ernst W. Biersack. Analysis of las scheduling for job size distributions with high variance. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 218–228, New York, NY, USA, 2003. ACM.

[33] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 135–148, New York, NY, USA, 2018. ACM.

[34] 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.

[35] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[36] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 75–86, New York, NY, USA, 2010. ACM.

[37] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 17–24, Washington, DC, USA, 2010. IEEE Computer Society.

[38] 2sigma hedge fund. www.twosigma.com.

[39] Coflow trace from facebook datacenter. https://github.com/coflow/coflow-benchmark.

[40] Akshay Jajoo, Rohan Gandhi, Y. Charlie Hu, and Cheng-Kok Koh. Saath: Speeding up coflows by exploiting the spatial dimension. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 439–450, New York, NY, USA, 2017. ACM.

[41] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 431–442, New York, NY, USA, 2014. ACM.

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

[43] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. pages 127–138, 2012.

[44] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold 'em or fold 'em?: Aggregation queries under performance variations. In *EuroSys*, 2016.

[45] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *ACM SIGCOMM*, 2012.

[46] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *NSDI*, 2013.

[47] Open-sourced aalo simulator. https://github.com/coflow/coflowsim.

[48] Microsoft azure. http://azure.microsoft.com.

[49] Akshay Jajoo, Y. Charlie Hu, and Xiaojun Lin. Your coflow has many flows: Sampling them for fun and speed. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 833–848, Renton, WA, 2019. USENIX Association.

[50] 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, pages 98–109, New York, NY, USA, 2011. ACM.

[51] Xin Sunny Huang, Xiaoye Steven Sun, and T.S. Eugene Ng. Sunflow: Efficient optical circuit scheduling for coflows. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 297–311, New York, NY, USA, 2016. ACM.

[52] 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. ACM.

[53] Vojislav Dukić, 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, 2019. USENIX Association.

[54] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[55] 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, 2016. USENIX Association.

[56] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Process Scheduling. Operating System Concepts*. John Wiley & Sons, 8 edition, 2010.

[57] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 265–278, Berkeley, CA, USA, 2010. USENIX Association.

[58] 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, pages 59–72, New York, NY, USA, 2007. ACM.

[59] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

[60] Stanley Lemeshow Paul S. Levy. *Sampling of Populations: Methods and Applications*. Wiley, 4 edition, Jun 2012.

[61] 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, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[62] 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, pages 160–173, New York, NY, USA, 2016. ACM.

[63] A document released by google containing schema and details of the cluster trace released by google. https://drive.google.com/open?id=0B5g07T_gRDg9Z0lsSTEtTWtpOW8.

[64] Personal communication with a 2sigma engineer regarding properties of the 2sigma trace used.

[65] Cluster trace from google - 2011. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

[66] Google cluster-usage traces, retrieved 21st july 2020. https://research.google/tools/datasets/google-cluster-workload-traces-2019/.

[67] Ronnie Chaiken, Bob Jenkins, Per-AAke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008. http://dx.doi.org/10.14778/1454159.1454166.

[68] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association.

[69] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times using historical information. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 122–142, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[70] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 121–134, New York, NY, USA, 2018. ACM.

[71] Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4):1 – 12, 2004.

[72] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A. Kozuch, and Gregory R. Ganger. Jamaisvu: Robust scheduling with auto-estimated job runtimes. In *Technical Report CMU-PDL-16-104*. Carnegie Mellon University, 2016.

[73] Google cluster-usage traces, retrieved 21st july 2020. https://drive.google.com/file/d/10r6cnJ5cJ89fPWCgj7j4LtLBqYN9RiI9/view.

[74] Zhe Huang, Bharath Balasubramanian, Michael Wang, Tian Lan, Mung Chiang, and Danny HK Tsang. Need for speed: Cora scheduler for optimizing completion-times in the cloud. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 891–899. IEEE, 2015.

[75] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, 2016. USENIX Association.

[76] Resutls on the posteriro distribution with gaussian priors. https://people.eecs.berkeley.edu/ jordan/courses/260-spring10/lectures/lecture5.pdf.

[77] 2sigma's proprietary job scheduler. https://www.twosigma.com/insights/article/cook-a-fair-preemptive-resource-scheduler-for-compute-clusters/.

[78] A private trace collected by 2sigma engineers from their clusters. www.twosigma.com.

[79] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[80] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 533–546, Boston, MA, 2018. USENIX Association.

[81] Apache hadoop yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[82] Edward G Coffman and Leonard Kleinrock. Feedback queueing models for time-shared systems. *Journal of the ACM (JACM)*, 15(4):549–576, 1968.

[83] Misja Nuyens and Adam Wierman. The foreground–background queue: a survey. *Performance evaluation*, 65(3-4):286–307, 2008.

[84] A patch for gridmix. https://issues.apache.org/jira/browse/YARN-2672.

[85] Hadoop patch for numeric histogram. https://issues.apache.org/jira/browse/YARN-2672.

[86] Dss scheduler. https://github.com/epfl-labos/DSS.

VITA

Akshay Jajoo is a Ph.D. candidate in school of Computer Science at Purdue University advised by Prof. Y. Charlie Hu. His research interests lie broadly in the domains of Distributed Systems, Data Center Networks and Computer Networks. Before joining Purdue he was at Indian Institute of Technology, Guwahati. He graduated from there in 2015 with a Bachelor of Technology degree in Computer Science and Engineering and was awarded the prestigious President Dr. Shankar Dayal Sharma Gold Medal for excellence in academics, extra-curricular activities and leadership roles.